Assignment of master’s thesis

Title: Implementation of Object Oriented Languages
Student: Bc. Rasul Seidagul
Supervisor: Ing. Petr Máj
Study program: Informatics
Branch / specialization: Web and Software Engineering
Department: Department of Software Engineering
Validity: until the end of summer semester 2022/2023

Instructions

Software development is often made easier by Object-Oriented Programming (OOP) that allows better code structure and reuse. In practice, implementing many of the object-oriented features imposes runtime costs in program performance (such as virtual dispatch) and in the complexity of language design (such as multiple inheritances). The objective of this thesis is to:

1 - evaluate OO features (virtual methods, full dynamic dispatch, multiple inheritances, traits, mixins, encapsulation, etc.) used in modern languages with respect to the problems they solve and the complexities language implementors and programmers face when using them.
2 - design an extension of the tinyC language used in the NI-GEN course that will support a reasonable subset of these features, called tinyC+
3 - implement a transpiler from tinyC+ to tinyC that will lower the OO features down the minimal tinyC feature set.

______________________________

Electronically approved by Ing. Michal Valenta, Ph.D. on 11 June 2021 in Prague.
Master’s thesis

IMPLEMENTATION OF OBJECT-ORIENTED LANGUAGES

Bc. Rasul Seidagul

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Petr Máj
June 23, 2022
Contents

Acknowledgments vi
Declaration vii
Abstract viii
Introduction 1

1 OOP languages and their features 3
  1.1 C++ Programming Language ........................... 5
      1.1.1 Class ........................................... 5
      1.1.2 Multiple Inheritance .......................... 8
      1.1.3 Virtual Methods ................................. 10
      1.1.4 Dynamic Cast .................................. 12
  1.2 C# Programming Language ............................ 14
      1.2.1 Similarities & Differences with C++ ........... 14
      1.2.2 Property and event member ..................... 15
      1.2.3 Single Inheritance ............................ 16
      1.2.4 Interfaces ..................................... 20
  1.3 Rust programming language .......................... 21
      1.3.1 Encapsulation .................................. 21
      1.3.2 Traits ......................................... 22
  1.4 JavaScript Programming Language ................... 26
      1.4.1 Prototype-based inheritance .................... 26
      1.4.2 Encapsulation, Abstraction and Polymorphism . 28

2 tinyC+ Language Design 31
  2.1 Rejected features .................................... 32
  2.2 Abstraction .......................................... 34
  2.3 Encapsulation ........................................ 36
  2.4 Inheritance ......................................... 39
  2.5 Polymorphism ........................................ 41

3 tinyC+ Language Implementation 47
  3.1 Name mangling ....................................... 47
  3.2 Class and its simple members ........................ 48
  3.3 Single inheritance and constructor reuse .......... 51
  3.4 Virtual methods ..................................... 53
  3.5 Interfaces .......................................... 57
  3.6 Object type casting .................................. 62

Conclusion 67
List of Figures

1.1 C++. Visualization of multiple inheritance graph. .......................... 9
1.2 C++. Visualization of type groups formed by multiple inheritance. .......... 11
1.3 C++. Visualization of single inheritance graph. .............................. 19

List of code listings

1.1 C++. Example of class declaration and definition. .......................... 5
1.2 C++. Example of constructor calls (class instantiations). .................. 7
1.3 C++. Example of multiple inheritance. ......................................... 8
1.4 C++. Example of call ambiguity caused by multiple inheritance. ............. 9
1.5 C++. Example of call ambiguity resolution. ................................... 10
1.6 C++. Example of virtual inheritance usage. ................................... 10
1.7 C++. Example of statically dispatched calls. ................................... 10
1.8 C++. Example of virtual method declaration and its override. .............. 11
1.9 C++. Example of abstract method and classes. ................................. 12
1.10 C++. Example of downcasting ..................................................... 13
1.11 C# . Example of property declaration. ......................................... 15
1.12 C# . Example of the single inheritance. ....................................... 17
1.13 C# . Example of the single inheritance. ....................................... 18
1.14 C# . Example of the single inheritance. ....................................... 18
1.15 C# . Example of the single inheritance. ....................................... 19
1.16 Rust. Encapsulation example. ..................................................... 21
1.17 Rust. Traits declaration and implementation example. ...................... 22
1.18 Rust. Traits usage example. ....................................................... 23
1.19 Rust. Trait default implementation example. .................................. 24
1.20 Rust. Trait default implementation addoption example. ..................... 24
1.21 Rust. Supertrait example. ......................................................... 24
1.22 Rust. Traits type combination example. ....................................... 25
1.23 JavaScript. Example of prototype usage in resolution of property access. 26
1.24 JavaScript. Example of object creation with prototype. .................... 27
1.25 JavaScript. Example of class declaration. ..................................... 28
1.26 JavaScript. Example of variable capture during construction. .............. 29
2.1 TinyC++. Example of class declaration. ....................................... 34
2.2 TinyC++. Class declaration in EBNF. ......................................... 34
2.3 TinyC++. Example of class declaration with access modifiers. .............. 36
2.4 TinyC++. Class declaration in EBNF. ......................................... 36
2.5 TinyC++. Example of class declaration with constructors. ................. 37
2.6 TinyC++. Class declaration in EBNF. ......................................... 38
I would like to thank my family and my brother who always believes in me and supported me morally during the development of this work. I would like to thank Petr Maj for all feedback and recommendations he has provided - it was pleasure to having him as my supervisor. Finally, I would also like to thank my opponent, Konrad Siek, for provided notes that helped me on my second attempt to finish this work.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Czech Republic, Prague on June 23, 2022
Abstrakt


Klíčová slova objektově orientované programování, transpiler, ni-gen, tinyC, tinyC+

Abstract

This work presents tinyC+, a language extension over a C-like language that supports Object-Oriented Programming. TinyC+ has been developed to aid the Code Generation course taught at FIT CTU. It serves as a demonstration of how the high-level OOP features are compiled. Therefore the primary design goals of tinyC+ were to provide minimal, yet functional object-oriented language so that its addition to the course would take very little explanation. TinyC+ is then transpiled into tinyC, a C-like language already used in the class. Therefore this thesis surveys the object-oriented features of several common programming languages, defines and justifies their minimal viable selection, and describes the transpiler implementation in detail.

Keywords object-oriented programming, transpiler, mie-gen, tinyC, tinyC+
Introduction

This work is about creating minimal OOP language as a supplement material for the course “Code Generators” of FIT at CTU. The language is called tinyC+ - an OOP extension to the existing imperative programming language called tinyC, a C-like programming language developed for educational purposes by Ing. Petr Maj. TinyC is a C-like imperative language, created to be a subject for students to practice and learn writing a compiler for it, which should generate CISC assembly code called “tiny86”. At the moment of writing, there is an emulator that executes tiny86 code; it can be found in the tiny-verse repository\[1\]. The tiny-verse repository was created by Ing. Petr Maj, and it is home to tinyC language compiler frontend, backend, tools, and many more. The language reference of tinyC can be found in the tiny-verse repository\[2\].

Nevertheless, why the resulting language focuses on OOP instead of other paradigms? With all pros and cons, what makes OOP standup is its popularity and widespread use. The paradigm itself is based on the natural desire of people to categorize the world into objects to simplify interaction with it. Nowadays, object-oriented programming is one of the most popular programming paradigms embodied in numerous programming languages like Java, C++, and Python. At the moment of writing, according to TIOBE Programming Community Index\[3\] on April 2022 - 7, out of the top 10 most “searchable”\[4\] programming languages support substantial object-oriented features or are an OOP language. According to the same source, what is even more impressive is that such OOP languages like Java, Python, Ruby, C++, and Objective-C - were the most searchable programming languages of the year (each in a different year starting from 2003).

Such a trend of interest in the OOP-based or OOP-associated languages is the motivation behind this work. Considering existing knowledge and advances in modern OOP languages, what would a minimal OOP language look like, and how can it be implemented? This work answers those questions.
Introduction

To summarize, the goal of this work is to create an OOP language that is easy to understand and dissect for students, which is why the implementation and design of the language are as simple as possible. The criteria of “minimalism” is described as the ability to address main OOP principles sufficiently. The work consists of three parts: an exploration of some of the existing OO features, design of the minimal OO feature set based on what has been explored, and implementation of the languages in the form of a transpiler program.

Apart from main goals, this work could also be seen as a recreation of the first stages of C++ development, where the language was initially developed as an extension called “C with classes” to the famous C programming language. So, people who study C++ history may find similarities as the design and implementation of tinyC+ are greatly inspired by it.

Additionally, while many programmers write their projects using OOP languages, not everyone understands how their code is executed by today’s machines and the costs of having those OO features. So, the lack of knowledge can be fulfilled by providing insight on how those features may be implemented in miniature.
Chapter 1

OOP languages and their features

In object-oriented programming model, a programmer writes program as composition of objects where execution of the program is represented in communication between objects. An object is data bundled with procedures that operate on it. A distinct property of an object as a feature is the *encapsulation* principle (1st principle of OOP). Encapsulation is a fact of deliberate exclusion of direct access to object’s data and internal (associated only with object’s data) procedures from outside code - concealing of object’s details. Along with encapsulation it is common to hear about information hiding where the terms are used interchangeably, however, it is accurate to say that encapsulation is application of the information hiding in concept of objects, where the information hiding is a broader term. Principles of the information hiding are: (a) it should not be necessary to consider *inessential details*; and (b) it should not be possible to access inessential details. In concept of software engineering, what a system entity (or the system itself) does and produces is what matters; where its implementation, execution, and/or internal structure are inessential details for the system’s user\(^5\).

This chapter evaluates different object-oriented features. An object-oriented feature is a programming language construct that exists to address OOP model. All features are viewed in the context of programming languages they are found in. Even though used languages are multiparadigm, meaning that direct support of OOP model is just a part of their design, the object-oriented part is often a cornerstone of their design. Nevertheless, the described languages are the most popular languages in the world up to date, and those are: Javascript, C++, and C#.

Besides encapsulation, there are three more principles, which are distinctive for modern OOP languages: abstraction, polymorphism, and inheritance. The evaluation of object-oriented features is based on how they follow or embody one or many of those principles.
In an OOP language the abstraction principle is about having patterns for generation and manipulation of objects. A great example of it is the class feature, which allows programmer to define a template that specifies properties and behaviour of an object type. Creating object “from” a class implicitly makes them of one type - all operations valid for one object is valid for all the others of the same class. Which by definition, creates a convinient way to control property and behaviour of many objects from single place - a class definition.

The polymorphism principle is about ability of an object to share multiple types. This, mainly, results in two distinct scenarios: (1) multiple objects may be substituted to operation that expects objects of the same type, (2) a single object may be substituted to operations that expects different type of object - as its definition satisfies all type requirements those operations expect. For ease of use, the (1) case will be refered as many-to-one and the (2) as one-to-many of object(s) in the context of polymorphism. Usually polymorphism is achieved with use of abstraction principle - objects of different types follows/implements the same type pattern/contract. While this concept grants power for programmer to freely write loosely-coupled code - it, most importantly, allows to modify codebases with the minimum amount of changes in comparison to code written using direct object types only; maintainability is the biggest benefit of polymorphism. In common applications of programming language design, the idea of the polymorphism principle is that a method (M) of an object (A) and object (B), which share the same type that houses the declaration of (M), execute differently based on the real type of (A) and (B).

The inheritance principle refers to ability for one entity to inherit properties of another one. An entity could be an object or the abstraction/template of the object (like class). Which is why, it is common for the inheritance principle be a natural extension of the applied abstraction principle in popular OOP languages. Inheritance allows to reduce code repetition and form type hierarchies, which also favors adoption of polymorphims as type groups are implicitly formed.

The following text is split into sections - each about languages with distinctive details, descriptions, and evaluation of their object-oriented features. For each discussed language, there is an explanation of what features and how satisfy those four OOP principles. The gathered knowledge is then used as basis for the design of tinyC+. Information about languages was taken from official websites available up to the date of this work release[6].
1.1 C++ Programming Language

C++ is a multi-paradigm general-purpose programming language. The language’s initial author is Bjarne Stroustrup, and the development was started at Bell Labs in 1979. Being a superset of C programming language - C++ is a language that provides both high-level and low-level programming. Due to high control over execution and memory, the language is commonly used to write performant and memory-efficient software, like operating systems and render engines. For performance reasons, many libraries used by other languages are written in C or C++. To date, the last version of the C++ standard was ratified and published by ISO in December 2020[7].

1.1.1 Class

The language is statically typed, so the specification of the object’s properties and behavior is defined at compile time. For this purpose, C++ has the class feature - a static template/specification of the object; and class instance represents an object. The typical C++ software is a procedural code that manages objects (class instances) and their interconnection to govern overall execution.

A class definition can consist of fields, methods, constructors, and destructors. For example, consider the following example of stack data structure as a class.

Code listing 1.1 C++. Example of class declaration and definition.

```cpp
#include <stdlib.h>

class Stack {

    // Fields
    private:
        int * source;
        size_t capacity;
        size_t count;

    public:
        // Default parameterless constructor
        Stack() {
            capacity = 0;
            count = 0;
            source = nullptr;
            std::cout << "empty\u001b[32mStack." << std::endl;
        }
```
// Constructor with parameter
Stack(int capacity) {
    this->count = 0;
    if (capacity > 0) {
        this->capacity = capacity;
        this->source = (int*) malloc(sizeof(int) * capacity);
    } else {
        this->capacity = 0;
        this->source = nullptr;
    }
    std::cout << "stack for " << capacity << "." << std::endl;
}

// Destructor
~Stack() {
    free(source);
}

// Methods
protected:
    void resize(size_t newCapacity) {
        capacity = newCapacity;
        source = (int*) realloc(source, newCapacity * sizeof(int));
    }

public:
    void push(int value) {
        if (capacity == count) resize(capacity * 2);
        source[count] = value;
        count++;
    }

    bool pop(int & result) {
        if (count == 0) return false;
        result = source[count];
        count--;
        if (capacity > count * 2) resize(capacity / 2);
        return true;
    }

    int get_count() {
        return count;
    }
}; // class Stack
The first feature that could be seen is access modifiers: public, private, and protected. They specify whether the class member (field or method) that follows it is available for access from everywhere, this class, and a derived class. In this case: the “push” method is callable from everywhere, the “source” field is visible inside the “Stack” scope only, and the “resize” can be called inside derived classes. A new access modifier overrides the previous one in the same scope.

Remark. The struct language construct adopted from C programming language changes its meaning in C++ and becomes an alias of class language construct. Everything about classes is true for structs - they are interchangeable. The only difference between them: the struct’s scope implicitly starts with public access. In contrast, the class’s scope - starts with private access to the following members.

The second class feature is constructors and destructors. In “Stack” class they are shown as methods with the same name: “Stack()” and “Stack()” respectfully. Constructor is a special part of a class, which is called to create a class instance. A constructor plays an important role in support of the encapsulation principle as it allows to set necessary data once at the moment of object creation and then keep data locked from outside influence; example of it seen in setup of “Stack.capacity”. By default class implicitly has a constructor with no parameters; in this case, the “Stack” default constructor has been explicitly declared and defined. Here are all class instantiation variations - the parameterless constructor is called in each case.

**Code listing 1.2** C++. Example of constructor calls (class instantiations).

```cpp
void foo() {
    // * Implicit
    Stack s0; // ..valid only if class has default or no constructor(s)
    // * Explicit
    Stack s1(); // call in declaration
    Stack s2 = Stack(); // call in assignment
    Stack * s3 = new Stack(); // call in assignment with heap allocation
    delete s3;
}
```

There is also a default destructor implicitly declared for all classes. However, in the “Stack” example, it has also been explicitly defined. While constructors are called at the place of assignment/declaration, for the destructor, it depends on where the object was declared. A destructor is called implicitly at the end of a scope where the destructor's class instance was declared. In “Stack” example, for “s0”, “s1”, “s2” the destructor is called right before “foo” function ends in reverse order of their creation.

An exception is heap-allocated object (in this case: “s3”) - its destructor never calls automat-
ically because multiple pointers can reference to the same object. For that, an explicit destructor call via “delete” language instruction must take place - it is responsibility of the programmer to properly delete object.

C++ is always criticized for being a memory-unsafe language. When a heap-allocated object was used, but its memory was never returned - the system will needlessly hold that memory locked for the rest of the software’s execution. The name for this issue is - a memory leak. The situation worsens when such objects are repeatedly created during execution. Eventually, the system will run out of memory and crash.

The risk of memory leaks makes the class feature, in combination with the constructor & destructor, a handful abstraction for controlling memory compared to how it is done in procedural languages like C. By allocating memory in the constructor and deallocating it in the destructor - the resource lifetime bounds to the object lifetime. The formalized and refined idea of such a programming technique is called RAII (Resource Acquisition Is Initialization). A simple application of it is demonstrated in the “source” allocation/deallocation via the constructors & destructor of the “Stack” example.

1.1.2 Multiple Inheritance

In C++, the inheritance principle of OOP is implemented in class-based inheritance. A class that inherits another is called derived class. A class that is inherited is called base/super class. The inheritance concludes in sharing data and operations of a base class with a derived class. C++ allows one class to inherit multiple others, giving the name for the technique - the multiple inheritance.

Consider the following example of multiple inheritance.

```cpp
struct Pet {
    std::string name;
    void feed() { std::cout << "Eats food." << std::endl; }
};
struct FlyingPet : Pet {
    void call() { std::cout << "Flies to owner." << std::endl; }
};
struct WalkingPet : Pet {
    void call() { std::cout << "Walks to owner." << std::endl; }
};
struct Parrot : FlyingPet, WalkingPet {
};
```

Code listing 1.3 C++. Example of multiple inheritance.
Here, “Parrot” is both flying and walking pet, “Dog” only walks, and “GoldFish” is just a pet with no additional characteristics. Their relation can be visualized as a graph; edges represent inheritance relations between classes, and edge direction from X to Y nodes is translatable to X inherits Y.

![Figure 1.1](image)

**Figure 1.1** C++. Visualization of multiple inheritance graph.

While multiple inheritance allows a programmer to form types of objects freely and consolidate standard code into a single place, however, it also introduces challenges. In the “pets” example, both the “WalkingPet” and the “FlyingPet” class declares the “call” method; when this method is called on a “Parrot” object - what version should be called: flying or walking pet? Such ambiguity is a well-known issue of multiple inheritance and is called the diamond problem. Besides that, data of the “Pet” class is duplicated in the “Parrot” class - a duplicate inheritance issue arises.

**Code listing 1.4** C++. Example of call ambiguity caused by multiple inheritance.

```cpp
void issuesOfMultipleInheritance() {
    Parrot p;
    p.name = "Dolly"; // compile error -> ambiguous access
    p.call(); // compile error -> ambiguous call
    GoldFish f;
    f.feed(); // prints: "Sucks and swallows food."
    Pet & fp = f;
    fp.feed(); // prints: "Eats food."
}
```

However, C++ provides ways to resolve ambiguity. For example, a programmer can specify which version of the method to call - this process is called scope resolution.
1.1.3 Virtual Methods

Multiple inheritance can be complex and unsafe, but it enhances software abstraction when used thoughtfully. A class by itself acts as a type for objects produced via class instantiation. Therefore, as one class inherits other classes, it also joins with its base classes into naturally formed type sets. Continuing pets example, pet type sets can be visualized as follows:

Type groups formed by multiple inheritance, by itself, does not satisfy the polymorphism principle of OOP. Even though the “GoldFish” can redefine the “feed” method, when it is called on a “GoldFish” object as it was type of the “Pet” - the “Pet.feed” version is used. This happens because, by default, a method in C++ is *statically dispatched* - its implementation version is defined at compile time.

To resolve the duplicate inheritance issue, a class can be inherited *virtually* - where only one copy of a base class member is inherited by a derived class.

Nevertheless, the main challenge is maintaining and extending such a codebase - it puts much responsibility on the language user. Which is why some languages like Java and C# supports only single inheritance.
In order to choose what method version needs to be used, it must happen at runtime time - a *dynamic dispatch* takes place. For that, in C++, a base class method must be declared as virtual, allowing a derived class to override the method - thus, changing behavior while preserving the same call interface. Therefore, a class that declares or inherits at least a single virtual method is polymorphic.

Continuing pets example: “Pet” and “GoldFish” can be rewritten as follows.

**Code listing 1.8** C++. Example of virtual method declaration and its override.

```cpp
struct Pet {
    virtual void feed() {
        std::cout << "Eats food." << std::endl;
    }
};
...
struct GoldFish : Pet {
    void feed() override {
        std::cout << "Sucks and swallows food." << std::endl;
    }
};
```

Now, both calls of the “feed” method in “dispatching_test” use the “GoldFish.feed” version and print “Sucks and swallows food.”. For dynamic dispatch to function, it requires supportive code and data to be generated. In C++, it is made via *virtual tables*: a special data structure...
that holds function pointers to implementations of virtual methods for the giving class. Therefore, a virtual method call inherently applies runtime cost compared to statically dispatched methods. Additionally, even though it is outside of language design, compilers cannot inline virtual methods, making their optimization less effective than inlinable non-virtual methods.

A virtual method can be declared without definition, such method is called abstract (or pure virtual). A class that contains an abstract method considers as abstract class. An abstract class cannot be instantiated but can be used as a reference or pointer type. The abstract method must be defined in a derived class; otherwise, the derived class becomes abstract too. Considering its properties, an abstract class is the purest polymorphic type that acts as a common target for type reduction. An abstract class with no data declared or inherited - is the closest representation of the interface concept, which languages like C# or Java have. Continuing the “pets” example - the syntaxis of an abstract method looks as follows.

Code listing 1.9 C++. Example of abstract method and classes.

```cpp
struct CanSwim { 
    virtual void swim() = 0; // an abstract "swim" method 
};
...
struct GoldFish : Pet, CanSwim {
    ...
    void swim() override { // definition of an abstract method 
    }
};
```

1.1.4 Dynamic Cast

To achieve full polymorphism, it is not enough to have object type casting from derived to base class - upcasting; downcasting (a reverse of upcasting) must be present too. An example of upcasting can be seen in the “dispatching_test” method example. In cooperation their usage allows free transition of an object between types in inheritance tree, with root being type of instantiation and branching going upward in inheritance relation.

For downcasting, C++ has a dedicated language operator: “dynamic_cast\(X_i(y)\)” where \(X\) is the target type of the cast, and \(y\) is an object. Cast target type and the subject must be of a pointer or reference type. Unlike upcasting, downcasting can fail when applied to a class instance that was not upcasted before from the given target type (the derived class).

The downcasting is an operation directly involved in the polymorphism process. Therefore,
only polymorphic classes are acceptable by the dynamic cast - the use of non-polymorphic classes produces a compile error. Continuing pets example, downcasting can looks as follows.

**Code listing 1.10 C++. Example of downcasting.**

```cpp
void func1(GoldFish & fish) {
    // upcasting from fish to pet
    Pet & pet = fish;
    pet.feed();
    func2(pet);
}

void func2(Pet & pet) {
    // downcasting from pet to fish
    GoldFish & fish = dynamic_cast<GoldFish&>(pet);
    fish.feed();
}
```

In summary, C++ is a programming language that fully addresses all four principles of OOP. The class feature, combined with other features, addresses the abstraction principle allowing a programmer to form a statically defined specification of objects. The access modifiers embody the encapsulation principle of OOP. Multiple inheritance allows complex relationships between classes that shares functionality, data, and appearance with each other. The methods virtualization feature turns type sets formed by multiple inheritance into polymorphic groups, those addressing the polymorphism principle. Overall, C++ is a great example of the interdependency of those OOP principles where one becomes fundament for the other enhancing the OOP style.
1.2 C# Programming Language

C# is a multi-paradigm general-purpose programming language focusing on OOP. Anders Hejlsberg designed it in 2000 at Microsoft. The language allows programmers to write secure and robust software that seamlessly runs and integrates into the .NET environment and infrastructure. C# has built-in garbage collection that automates memory management allowing developers to focus on what their software does, writing it in an abstract object-oriented way. The most recent language version is C# 10, which was released in 2021.

1.2.1 Similarities & Differences with C++

C# uses class-based object representation in contrast to prototype-based (like JavaScript). In C#, class declaration is a template for object creation that specifies data, open and closed data operations. A class declaration consists of multiple member types: fields, methods, constructors, properties, and events. Fields, methods, and constructors are similar to those in C++, with some exceptions, like data layout is not directly bound to field positions. C# classes mainly adopt the same design of classes in C++. The method virtualization syntax is more apparent than in C++: abstract methods are defined using the “abstract” keyword, and class can be made abstract using the same keyword in its declaration, without necessity for having a virtual method.

First differences appear at the concept level: C# was developed with the philosophy of everything being an object, which means that primitive data types like bytes and numbers are considered objects and behave respectfully. Everything inherits an implicit ancestor - “System.Object” (also can be defined using the “object” keyword). This approach makes language facilitate OOP programming, allowing a programmer to write consistent object-oriented code.

The class and struct language constructs share the same member types. However, unlike C++, C# classes and structs are not the same: they serve different purposes and differ conceptually. Classes are meant to create type hierarchies and organize codebase via inheritance and interface implementation. Structs are meant to specify raw or plain data types.

Notable differences between class and struct are:

- Class supports inheritance, where struct does not.
- Instantiation of a class is reference only, where for struct - is by value only.
- Instances of a class are passable via reference only. However, a struct instance can be passed as a value copy or reference (using the “ref” keyword).
Class is always allocated on the heap memory, while struct could be allocated both in stack and heap memory.

Class fields are implicitly defined with default values, where struct requires all fields definitions inside constructor when one is defined.

However, C# is not a copy nor descendant of C++. However, the following features are present in C++ as programming practices but not as standalone features as in C#. Those features either enhance or directly address OOP principles.

### 1.2.2 Property and event member

A notable addition for class and struct is the *property* member type. It is essentially a data proxy consisting of method pair: the first method supposes to set data - *setter*, and the second supposes to return data - *getter*. By default, the property implicitly points to a hidden field. However, when either setter or getter is defined - the property becomes just a set of two methods. The benefit of a property member is its interchangeability with a field member. For a programmer, a property behaves precisely like a field.

The C# example below shows how a property member appears as a read-only “area” field of the rectangle class, whereas inside the class acts as a method that computes area using “width” and “height”.

#### Code listing 1.11 C#. Example of property declaration.

```csharp
using System;

class Rectangle {
    public int width;
    public int height;
    public int area {
        get => width * height;
    }
}

class Program {
    // Prints: "Rectangle has area: 6:
    public static void Main(string[] args) {
        var r = new Rectangle();
        r.width = 2;
        r.height = 3;
        Console.WriteLine("Rectangle has area: "+ r.area);
    }
}
```
By design, field and property member features are interchangeable. It creates a data interface, allowing to change internal data representation (property definition) without changing code that uses the interface (like accesses from outside in the example above). This increases flexibility for software development and maintenance. A typical application of properties is data validation: when the outside code submits data for the setter - the setter can execute operations on the input data and apply a validated data version to the internal data. The property member feature directly addresses the encapsulation and enhances the abstraction principles of OOP.

The event event member is syntax sugar for special field declaration. The unseen declaration consists of an object that conceptually acts as a List-like object that can store delegate instances - similar to a vector of functors in C++. The idea of the event member is to act as a one-way communication channel between one-to-many objects. An object [A] could subscribe to the event of an object [B], but only [B] is able to invoke (initiate execution of) its event - meaning that only [B] could decides when to talk, while its subscribers patiently listen to it. This feature allows the creation of complex relations between objects where each could bind its method (or lambda function) to the event of another object, forming an execution network, which saves performance compared to an excessive spin check alternative.

The event member does not directly address any OOP principle. Instead, it mainly comes from different programming paradigms: event-driven, reactive, and observer-pattern. However, when an object [A] subscribes to an object [B] event - the [B] object has no idea of what and how many are listening to its event. Such anonymity naturally integrates and enhances the abstraction and encapsulation principle of OOP.

### 1.2.3 Single Inheritance

Unlike C++, C# class can inherit only a single other class. As mentioned in the C++ section, multiple inheritance introduces many challenges to programmers. Languages that came after C++ (like C# and Java) and part of the C-like syntax language family mostly rejected the concept of the multiple inheritance reducing it to single inheritance only in favor of lowering complexity and with it removing all associated semantical mistakes. While it might be seen as a limitation over the feature, the more sensible difference comes in a shift of the concept: multiple inheritance centers on code sharing, whereas single inheritance is more about extending code. Rather than combining classes, C# decided to extend them, forming a root-like class hierarchy. Derived classes in C# express specialized versions of their base class.
Using the previous “pets” examples of the C++ section, the following code demonstrates the
declaration of all features mentionable in this language section. The example is a bit different for
novelty and convenience but generally keeps the same behavior. For example, some object types
like “IFlyingPet” and “IWalkingPet” were transformed into interfaces - as multiple inheritanceis
not supported. First, let us declare the types used in the C# “pets” example.

**Code listing 1.12 C#. Example of the single inheritance.**

```csharp
using System;
using System.Collections.Generic;

public interface IHasName {
    string name {get;}
}

public interface ICanWalk : IHasName {
    void WalkTo(string place) {
        Console.WriteLine(name + " walks to " + place);
    }
}

public interface ICanFly : IHasName {
    void FlyTo(string place) {
        Console.WriteLine(name + " flies to " + place);
    }
}

public abstract class Entity : IHasName {
    public string name {get; set;}
}

public class Pet : Entity {
    public virtual void Eat(string food) {
        Console.WriteLine("Eats " + food);
    }
}
```

Aside from forming the framework for further specialized pet type declaration, the above
code demonstrates what roles the class and interface language feature take: classes responsible
for a main types formation, interfaces - for object type characterization (in this case: optional
abilities like fly or walk) or type requirement (like “IHasName”). The example also demonstrates
trait-like interface declaration where default implementation provides decent standard behavior
for the interface implementers.
Also, the “Entity” class demonstrates how in C#, a class can be declared abstract as a whole without requiring virtualizing one of its methods. It is a subtle difference mainly for ease of eye - impact on the language is the same. In this example, the entity type was marked as abstract deliberately to mark that it is not instantiable; instead, it is purely for declarative purpose.

Continuing example, the final pet types looks as follows.

**Code listing 1.13** C#. Example of the single inheritance.

```csharp
public class Parrot : Pet, ICanFly, ICanWalk { }
public class Dog : Pet, ICanWalk { }
public class GoldFish : Pet {
    public override void Eat(string food) {
        Console.WriteLine("Sucks and swallows " + food);
    }
}
```

To accommodate for the change in “ICanWalk” and “ICanFly” compared to “IWalkingPet” and “IFlyingPet” shown previously - the example will have an owner object that can call its pets.

**Code listing 1.14** C#. Example of the single inheritance.

```csharp
public class Owner : Entity {
    public void Call(Pet pet) {
        if (pet is ICanFly flyingPet) {
            flyingPet.FlyTo(name);
        } else if (pet is ICanWalk walkingPet) {
            walkingPet.WalkTo(name);
        } else {
            Console.WriteLine(pet.name + " does not respond to " + this.name + ", s call");
        }
    }
}
```

The “Owner.Call” method demonstrates ”is-a” relationship for interface types. The “x is Y y” language instruction checks whether the given object (x) is of type (Y) and, on success - declares variable (y), which is a casted (x), in the given scope (in this case: if statement). Besides “is” there is “as” cast operator, which is used as follows: “x as Y”, where the object (x) is of a nullable type. The expression result is the casted object(x) of the type (Y). In case object (x) is not of type (Y) - the “as” operator returns null. Both “as” and “is” cast operators are used for safe downcasting.
As for the formed inheritance relationships of classes in the “pets” example: each class extends its base; moving downwards - types become more specific at each inheritance level. Visualization of their inheritance relationship looks as follows:

![Figure 1.3 C++. Visualization of single inheritance graph.](image)

Finishing the “pets” example: the program might look like the execution of a polymorphic operation over a person’s pets collection. In the example below, the owner calls pets, and they eat one by one.

**Code listing 1.15 C#. Example of the single inheritance.**

```csharp
class Program {
    public static void Main(string[] args) {
        var owner = new Owner { name = "John Doe" };
        var pets = new List<Pet> {
            new Parrot { name = "ChiChi" },
            new Dog { name = "Rex" },
            new GoldFish { name = "Dorry" },
        };

        foreach (var pet in pets) {
            owner.Call(pet);
            pet.Eat("pet's yummies");
        }

        // Output is:
        // ChiChi flies to John Doe
        // Rex walks to John Doe
        // Dorry does not respond to John Doe's call
    }
}
```
1.2.4 Interfaces

Single inheritance falls short in giving as much polymorphism and abstraction as multiple inheritance does. For that, C# introduces interface: language construct that, when applied to a class or struct, ensures the existence of members declared by it on the target. It acts as an abstract layer between declaration and definition. Interfaces allow users to modularize a program into parts that could be modified or replaced without affecting dependent code, keeping implementation in isolation and forming a clear "type contract" a user could rely on.

An interface always represents a polymorphic type. Both "ICanFly" and "ICanWalk" have a default implementation. However, whether the pet uses the default or defines its own implementation is unknown at compile time. Therefore, interface members are virtual by default and do not need the "virtual" keyword.

Classes and structs can implement an arbitrary amount of distinct interfaces. Unlike class, an interface can inherit multiple other interfaces. Adding the interface default implementation feature in C# 8.0 made interfaces be like the traits language feature. How object-oriented programming benefits using traits is discussed further below in the section about Rust programming language.

Nevertheless, for most of C# history, interfaces played the role of abstract type contracts. An interface is an extensive part of the "is-a" relationship between objects, allowing to create type sets where objects do not need to belong to the same inheritance tree. This makes it easier for developers to write software detached from a particular solution. The use of interfaces decreases code coupling and simultaneously creates cohesive relations: where an interface acts as a type set for classes and structs that implement it. The interface language feature facilitates the abstraction principle in C#. It plays a significant role in developing and maintaining complex and large codebase.

In summary, many OOP features of the language are close in design and purpose to those in C++. Functionally C# can do everything C++ already does almost in the same way (with respect to a significant shift in memory management). Some features of C++ has more refined looks in C#, like the use of distinct keyword for declaring abstract classes & methods and "is & as" cast operators instead of "dynamic_cast". Nevertheless, a major shift was made in the concept of class inheritance: in C#, it is about a class extension, while in C++, it advances in class combination. To coup limitations of the single inheritance, the feature of interfaces addresses the need for unbound creation of abstract types. Same as C++, the language addresses all four main principles of OOP.
1.3 Rust programming language

Rust is a multi-paradigm, general-purpose programming language that competes with C and C++ in their respective domains. The distinctive feature of the language is safety without the expense of performance by providing “zero-cost” high-level abstractions and the implicit system of memory ownership. Compared to languages like C++ and Java, Rust is relatively new; the development started as early as 2006, with the first official release of version 1.0 in 2015.

Whether Rust is an OOP language or not is controversial. According to “The Rust Programming Language” book written by developers of Rust - the language could be considered as OOP depending on what OOP definition is used[8]. However, Rust satisfies the OOP definition of this work by addressing encapsulation and polymorphism principles.

1.3.1 Encapsulation

The encapsulation technique is controlled by the “pub” keyword, which grants access to struct’s methods and fields. By default, everything is private and cannot be accessed from outside of the given scope. Aside struct, encapsulation is also applied to a module - a collection of code (could be seen as advanced namespaces). Something similar could be seen in how C# allows specifying assembly-private access modifiers using the “internal” keyword.

Let us recreate the integer queue data structure to show encapsulation in work. Users should not have access to the internal state of the queue. Instead, users rely on the public interface, where available actions are: to enqueue, dequeue, and count elements.

Code listing 1.16 Rust. Encapsulation example.

```rust
struct Queue { list: Vec<i32> }
impl Queue {
    pub fn push(&mut self, value: i32) {
        self.list.insert(0, value);
    }

    pub fn pop(&mut self) -> Option<i32> {
        let count = self.list.len();
        if count > 0 {
            let r = self.list.remove(count - 1);
            Some(r)
        } else { None }
    }
}
```
### 1.3.2 Traits

The polymorphism principle is addressed via Rust’s *traits* feature. Traits are similar to interfaces and serve the same purpose of abstracting method calls via a public interface from their exact implementations. In Rust, traits in combination with generics allow for writing polymorphic code where an instance of each struct or enum that implements a given trait can be substituted.

In the following example, there is a declaration of trait `Animal` and its implementation for `Duck`, `Frog`, and `Fish` structs. Where each concrete type provides unique response for `Animal.sound` function call.

*Code listing 1.17 Rust. Traits declaration and implementation example.*

```rust
pub trait Animal {
    fn sound(&self);
}

struct Duck;
struct Frog;
struct Fish;

impl Animal for Duck {
    fn sound(&self) {
        println!("quack!");
    }
}
```
The notable feature of Rust traits - implementation is separated from the target (struct or enum). A trait can be implemented for structs from other files, modules, and crates (binary or library). However, one restriction for trait implementation is: that either trait itself is local or the implementer (struct or enum) is local - the programmer cannot implement an external trait to an external target within its crate codebase.

Continuing the previous example, the following code is a simple example of polymorphism where instances of Duck, Frog, and Fish are stored in one collection and then iterated to produce sound. For that, Rust has a “dyn” keyword, which specifies that the trait is dynamic and the underlying object can be of any type (in this case: Duck, Frog, or Fish); therefore, expect different behavior. This is a classic application of polymorphism where objects of multiple different types are reduced to one shared type (in this case: Animal).

Code listing 1.18 Rust. Traits usage example.

```rust
fn main() {
    let lake_animals : Vec<Box<dyn Animal>> = vec![
        Box::new(Duck{}),
        Box::new(Frog{}),
        Box::new(Fish{}),
    ];
    for animal in lake_animals {
        animal.sound();
    }
    // quack!
    // ribbit!
    // ...
}
```
Rust allows traits to have a default implementation to compensate for the lack of inheritance. For example, the previously shown `Animal` trait can have the following default implementation.

- **Code listing 1.19** Rust. Trait default implementation example.

```rust
trait Animal {
    fn sound() {
        println!("...");
    }
}
```

Now, every animal, by default, makes the sound “...” (nothing). Because `Fish` makes the same sound, its implementation can be simplified as follows.

- **Code listing 1.20** Rust. Trait default implementation adoption example.

```rust
impl Animal for Fish {};
```

By itself, the default implementation feature is limited to very simple code. However, Rust introduces the concept of `supertraits`. A supertrait is a trait that another trait requires to be implemented on the target struct/enum. The use of supertraits allows for a purely abstract (generalized) implementation for the given trait. Consider the following example:

- **Code listing 1.21** Rust. Supertrait example.

```rust
impl Functor {
    int compute(&self, Vec<int>);
};
impl Generator {
    Vec<int> generate(&self);
};
impl Instruction : Functor + Generator {
    int execute(&self) {
        let data = self.generate();
        let result = self.compute(data);
        result
    }
}
```

In the “Rust Programming Language” book, the author states that default trait implementation achieves the same goal of controlling and sharing behavior from one place. However, this addresses only one aspect of inheritance, which is a deliberate choice of Rust language developers.
The book author states that inheritance fell out of favor because of the risks it introduces. Therefore, Rust takes a different approach by using trait objects instead of inheritance in substitutable cases.

A novel feature of Rust, in comparison to similar but old solutions like in C# or Java, where an object can be cast to one type of interface at a time - Rust introduces the ability to combine traits using the + symbol in variable type declaration allowing to write more complex polymorphic types.

Following the previous animals example, Duck data can be extended with a new trait CanFly and a function like ask_to_fly can request a more specific argument type - an animal that can fly.

**Code listing 1.22** Rust. Traits type combination example.

```rust
trait CanFly {
    fn fly(&self);
};

impl CanFly for Duck {
    fn fly(&self) {
        println!("duck is flying away");
    }
}

fn ask_to_fly(animal: impl Animal + CanFly) {
    animal.fly();
}

fn main() {
    let duck = Duck{...};
    ask_to_fly(duck); // prints: duck is flying away
    ...
}
```

To summarize, in Rust, encapsulation is achieved by specifying the “pub” keyword to open access for members of the scope (struct or module) while, by default - access is denied. Polymorphism is addressed via traits - a type contract with a set of public functions (with or without definition) that an implementer (struct or enum) inherits and defines when wanted. Finally, the inheritance principle is not as powerful as in C++ or JavaScript; however, Rust does allow to form of an abstract, generalized version inheritable by all exact trait implementations by allowing traits to depend on others and use that assumption in its default implementation.
1.4 JavaScript Programming Language

JavaScript is a dynamically typed multiparadigm programming language. The language is the lingua-Franca of web development: every decent browser has a built-in JavaScript engine to run websites' client code. JavaScript is an interpreted language, like Python.

JavaScript adopts garbage collection as a memory management solution. An exclusive case is built-in language objects like “Object” or “Array”, which are not subject to garbage collection because they are used as default prototypes of newly created objects (more on prototypes later below).

Dynamic typing is the main feature of JavaScript. Typical JavaScript code looks like a sequence of variable declarations and expressions. A variable is a container that holds either primitive data or references objects. A variable can be overridden with a value of a different type it holds at the moment. A JavaScript object is a dynamically sized collection of properties, where the property behaves just like a variable. A special case of an object is function. A function definition consists of arguments and the main code block. Thus, a variable that holds a function can be called.

1.4.1 Prototype-based inheritance

An ability to get properties via name lays the fundament for the distinctive feature of JavaScript - prototype-based inheritance. Every object has a special compulsory property called prototype. The prototype property serves a special purpose: when other code tries to read property (x), which the target object (y0) does not own (have in its internal property collection), then JavaScript looks for the property (x) in the prototype (y1) of the target object. If property was not found in the prototype (y1) it repeats process but this time delegates search to the prototype’s prototype (y2) and so on until prototype is null: (y1) -¡ (y2) -¡ ... -¡ (yN). If the property was found, it returns its value as if the originally accessed object (y0) owned it. In case the property was not found - a primitive value: undefined is returned, which directly means: “value for property X is not defined”. An object implicitly inherits the content of its prototype - delegation of the property access, which also means that when the prototype object is changed - all inheritor objects adopt the change too. Consider the following example, which uses “Object.create(...)” to create an object with a prototype.

Code listing 1.23 JavaScript. Example of prototype usage in resolution of property access.

```javascript
var person = {
```
```javascript
"age": 21,
  "country": "UK",
};

var student = Object.create(person);
student.name = "Jessy Brown";
student.country = "US";
student.grade = 2.1;

var teacher = Object.create(person);
teacher.name = "Walter Smith";
teacher.age = 45;

console.log(student.name);
// Access: student (found)
// Result: "Jessy Brown"

console.log(teacher.country);
// Access: teacher (not found) => person (found)
// Result: "UK"

console.log(student.university);
// Access: student (not found) => person (not found) => Object.prototype
  /* (not found) => null (stop)
// Result: undefined

Remarks. Null is a primitive data representing "nothing"; for example, JavaScript "Object"
has null prototype.

Nevertheless, it only works when the property is read - write access either updates property
value or adds property with the given value when the accessed object does not own a property
with the asked name. Therefore, to set prototype’s property value it must be accessed directly:
"object.prototype.property = value". Property assignment can be seen in the example below.

Code listing 1.24 JavaScript. Example of object creation with prototype.

```
1.4.2 Encapsulation, Abstraction and Polymorphism

The prototype-based inheritance satisfies the inheritance principle of OOP. However, other principles were not addressed until the ES6 standard was released. The standard introduced the class feature that alone embodies the abstraction principle. Before classes, only constructor functions played the role of forming reliable abstractions. As nothing can change the function’s content, it would produce an object with predictable properties when it was used as a constructor. The following example demonstrates the class declaration of a simple queue data structure with private: “array” field, public: “enqueue” and “dequeue” methods, “count” and “first” getters.

Code listing 1.25 JavaScript. Example of class declaration.

```javascript
class Queue() {
    #array;

    function enqueue(value) {
        array.push(value);
    }

    function dequeue() {
        var value = array[0];
        array.shift();
        return value;
    }

    get count() {
        return array.length;
    }

    get first() {
        return array[0];
    }
}
```

The class’s ability to define private fields addresses the encapsulation principle, which previously was made by capturing variables inside the context of the constructor function where only the resulted object’s functions, getter, and setters refer to an enclosed variable. The previous example of queue data structure can be rewritten using the old technique of variable encapsulation as shown below; where the “array” variable, even though not part of the queue object, is effectively sealed within the declaration of “Queue” constructor - nothing outside of properties defined by “Queue” function can access the array instance.
### Code listing 1.26 JavaScript. Example of variable capture during construction.

```javascript
function Queue() {
    var array = [];
    this.enqueue = function(value) {
        array.push(value);
    }
    this.dequeue = function() {
        var value = array[0];
        array.shift();
        return value;
    }
    Object.defineProperty(this, "count", {
        get: function() { return array.length; }
    });
    Object.defineProperty(this, "first", {
        get: function() { return array[0]; }
    });
}

var q = new Queue();
q.enqueue(9);
q.enqueue(4);
q.enqueue(5);
q.dequeue();
console.log("count: \u201c\u201d + q.count + ", first: \u201c\u201d + q.first); // prints: 4
```

The polymorphism in JavaScript finds itself the inability of the same-named method on different objects to have different behavior. Thanks to dynamic typing, multiple objects with a set of properties (in other words having different runtime “types”) that have the same method can be substituted as instances of one abstract type and provide different execution on the method call.

In summary, a distinctive OOP feature of JavaScript is prototype-based inheritance. The abstraction principle of OOP is achieved via constructor functions or classes, while the encapsulation principle found direct feature support via class private members. Only polymorphism acts more like a programming practice and does not find a feature in the language that addresses it.
The goal of the work is to create learning material for students as part of the tiny-verse project. Therefore, the resulting language is an OOP language with minimal features. Such language must address and directly support via feature all main OOP principles: abstraction, encapsulation, inheritance, and polymorphism.

The design is based on features explored in the first chapter. However, only some discussed features found a place in the final design of tinyC+: features that either extend (but do not provide the principle on its own), have a better alternative (in case of design and implementation complexity), or require the presence of additional non-OOP features, or for other specific reasons were not included. The rejected features are multiple inheritance, prototype-based inheritance, events, properties, destructors, and traits. While adopted with change features are classes (along with constructor and access modifiers), single inheritance, interfaces, and virtual methods. Each feature is discussed in the related OOP principle part.

In the “Code Generators” course, students study how to write a tinyC compiler that generates tiny86 assembly code. This is why tinyC+ takes the form of an OOP extension over tinyC with almost no modifications to facilitate the gained knowledge. The OO features are layered on top of the existing tinyC design.

The chapter content is divided into OOP principles to articulate what principle each feature address.
2.1 Rejected features

The *multiple inheritances* was rejected in favor of single inheritance. Multiple inheritance introduces ambiguity, which raises both the design and implementation complexity. The language user must manually resolve issues like member access ambiguity and duplicate inheritance mentioned in the first chapter. On the other hand, single inheritance is straightforward and introduces no ambiguity. The idiom of single inheritance is to extend existing code specializing it for certain use cases. In contrast, multiple inheritance is unbound, making it easy to mix loosely coupled code that lowers code quality and makes it less predictable.[9]

An important note could be that multiple inheritance is not that popular among new languages. For example, from the same TIOBE index, only C++ and Python use multiple inheritance - most other OOP languages restrict inheritance to a single type only. Along with decreasing popularity, the implementation of multiple inheritance is substantially complicated. The time it would require to implement does not fit this work’s time scope.

The *prototype-based inheritance* is quite a unique way to achieve a seamless inheritance. However, it was rejected because it suits languages with a dynamic type system, where new types could be created during runtime. The prototype-based inheritance feature presented in JavaScript requires dynamic pair-based collections (like map and dictionary), where an item key is an object’s member name. First, tinyC does not have a standard library that could provide dynamic memory allocation. Therefore, while some complicated solutions could be implemented, fully-fledged dynamic memory is impossible without changing tinyC specifications. Secondary, the implementation would apply more runtime cost compared to the statically-based inheritance approach, which is not decisive for the argument but makes the prototypes-based system less appealing.

In the case of *properties* discussed in the section about C# - the feature was rejected as it was not essential to achieving OOP. While, as was described before, the feature does facilitate such principles as abstraction and encapsulation. Nevertheless, to maintain the minimum set of features - properties were not incorporated into tinyC+. In fact, the next explored feature of C# - *events* was also not included in the final design. While the feature allows objects to initiate communication between each other, it does not directly address any of the mentioned OOP principles (even though it does support them when they are addressed).

In the case of *traits*, it is not rejected; instead, a simplified feature (x) was derived both from C# interfaces and Rust traits. Both features address the lack of type formation flexibility given by single inheritance - and the derived feature was incorporated into the design of tinyC+. However, the name “traits” is not mentioned further below as the derived tinyC+ feature resembles the
Rejected features

earliest version of C# interfaces.

Finally, the last feature that didn’t make it into the list is *destructors*. The primary use of destructors is to control the lifetime of an object’s owned resources (like closing files and releasing internal buffer) and as the finalization point of the object’s overall execution (closing network communication, notifying other objects about destruction). While destructors enhance the autonomy of objects and give more control over an object’s execution. However, it would require tracking all local scopes, the implementation of which is quite complex. It would greatly contrast with the rest of the work; the future work may focus on adding constructors as the next step in the maturity of tinyC+.
2.2 Abstraction

The abstraction principle is embodied in tinyC+ in the form of the class language construct. The class feature is a fundamental feature of tinyC+ - it represents object types and becomes a basis for all other features. As described in C++ language evaluation, the feature directly addresses the abstraction principle - a static scheme of data and associated operations for creating objects of the same type. Those, in tinyC+, an object is an instance of a class.

A class declaration may consist of: field, method, and constructor type members. A class field declaration is identical to those in struct and expresses the data available to an object. A method is similar to a plain tinyC function; however, it represents an operation associated with an object of the class type it was declared in. For articulating this fact, an implicit variable “this” is used to represent a class instance reference - the subject of method operations. The “this” variable is a keyword of tinyC+ language. Apart from the use of special variables, a method declaration may differ from a plain function declaration to express additional properties, more on that later.

The following tinyC+ example code demonstrates a simple class declaration of a vector type that can be found in programs related to 3D graphics.

Code listing 2.1 TinyC+. Example of class declaration.

```c
class Vector3D {
    double x;
    double y;
    double z;

    double sqrMagnitude() {
        double x2 = this->x * this->x;
        double y2 = this->y * this->y;
        double z2 = this->z * this->z;
        return x2 + y2 + z2;
    }
};
```

With the given information, the EBNF part of the class production rule looks as follows (with respect to revealed features).

Code listing 2.2 TinyC+. Class declaration in EBNF.

```plaintext
CLASS_DECL := 'class' identifier [ '{' { FIELD_DECL | METHOD_DECL } '}' ] ';'
```
Remark. “BLOCK_STMT”, “TYPE_FUN_RETURN” “TYPE”, and “identifier” are reused from tinyC EBNF representation.

As seen from the method production rule - it does not support the forward declaration. Because there is no need to do it - all class methods are accessible to each other whether they have been declared after or before their use. The feature of methods independent declaration only works inside their class scope.

With those two class member types, the class reaches the requirement of an object specification: a scheme that describes the object’s data and logic and satisfies the abstraction principle.
2.3 Encapsulation

Before discussing constructors and their purpose in a class, it is important to introduce the feature responsible for the encapsulation in tinyC+ - access modifiers. An access modifier is a special annotation in a class member declaration that specifies what parts of code can access it. There are free modifiers added to the language: public - member is accessible from everywhere, private - member is accessible only inside its class scope, and protected - member is accessible only inside the scope of its class (x) and other classes that inherits (x).

Many OOP languages with classes introduce their own set of access modifiers and usually make one of them implicit. However, in tinyC+, each class member must sound its access modifier. The decision was primarily motivated by the language implementation, making parsing easy, thus making declaration uniform.

Now, let us examine the following version of the previous tinyC+ class example that restricts access to its data and only exposes the method for outside use.

**Code listing 2.3** TinyC+. Example of class declaration with access modifiers.

```c
class Vector3D {
    private double x;
    private double y;
    private double z;

    public double sqrMagnitude() {
        double x2 = this->x * this->x;
        double y2 = this->y * this->y;
        double z2 = this->z * this->z;
        return x2 + y2 + z2;
    }
};
```

The access modifiers add one more nonterminal symbol into the field and method productions rules. The respectful EBNF part updates to the following representation.

**Code listing 2.4** TinyC+. Class declaration in EBNF.

```c
FIELD_DECL := ACCESS TYPE identifier ';'
METHOD_DECL := ACCESS FUN_HEAD BLOCK_STMT
ACCESS := 'public' | 'private' | 'protected'
```

The reader can notice that something is wrong with the example above: while no one can influence the internal values of the “Vector3,” it also means that none could set them. Unless
some setter methods are declared, which would allow outside code to initialize those private values. However, if they were forgotten to be called or called in the wrong order - the object would exist in an invalid state. This is why tinyC+ design incorporates constructors into the class declaration. The class declaration allows the specification of a constructor - a function-like member that helps set private and protected fields, thus establishing a valid state in a controlled environment.

Calling constructor allows creating an instance of its class. The declaration of a constructor is similar to an ordinary function; however, the name changes to a class name. Using the “this” keyword is compulsory for accessing class members. With constructors, the class example can be rewritten as follows:

Code listing 2.5 TinyC+. Example of class declaration with constructors.

```cpp
class Vector3D {
    private double x;
    private double y;
    private double z;

    public Vector3D(int x, int y, int z) {
        this->x = x;
        this->y = y;
        this->z = z;
    }

    public double sqrMagnitude() {
        double x2 = this->x * this->x;
        double y2 = this->y * this->y;
        double z2 = this->z * this->z;
        return x2 + y2 + z2;
    }
};

void foo() {
    Vector3D v = Vector3D(1, 2, 2);
    double sqrM = v.sqrMagnitude(); // 9
}
```
According to the addition of constructors, the class production rule is excited by the non-terminal “CONSTR_DECL” symbol representing a constructor member. The respectful EBNF part updates to the following representation.

<table>
<thead>
<tr>
<th>Code listing 2.6 TinyC+. Class declaration in EBNF.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DECL := 'class' identifier [ '{' { FIELDDECL</td>
</tr>
</tbody>
</table>
| \rightarrow CONSTRDECL ) '}' ] ';
| CONSTRDECL := ACCESS identifier FUN_ARGS BLOCK_STMT |

In summary, the three access modifiers: public, private, and protected, along with the ability to define constructors that initialize a created object, together form a minimal representation of encapsulation in tinyC+. From this, a user can separate the codebase into classes and control execution within their scope.
2.4 Inheritance

The inheritance principle in tinyC+ is provided by the single inheritance feature. As described previously, single inheritance allows one class to inherit the contents of another single class. The syntax is subtle: a base class name can be specified after the colon after the derived class name; the colon is missing if the class does not inherit anything. The previous class example can be rewritten with the addition of a less specialized type: “Vector2D”, as follows.

Code listing 2.7 TinyC+. Example of single inheritance.

```cpp
class Vector2D {
    public double x;
    public double y;

    protected double sqr(double v) { // returns (v) in the power of 2
        return v * v;
    }
};
class Vector3D : Vector2D {
    private double z;

    public Vector3D(int x, int y, int z) {
        this->x = x;
        this->y = y;
        this->z = z;
    }
    public double sqrMagnitude() {
        double x2 = this->sqr(this->x);
        double y2 = this->sqr(this->y);
        double z2 = this->sqr(this->z);
        return x2 + y2 + z2;
    }
};
```

With the given information, the EBNF part of the class production rule extends by a combination of terminal symbols and looks as follows.

Code listing 2.8 TinyC+. Class declaration in EBNF.

```plaintext
CLASS_DECL := 'class' identifier [ '::' identifier ] [ '{' { FIELD_DECL | METHOD_DECL } '}' ] ';'
```
In the example above, the reader could also notice how changing access modifiers of “x” and “y” to protected allows them to be used by the derived class “Vector3D”. Inheritance allows members of the base class to be substituted as members of the class that derives it.

Apart from inheriting fields and methods, a derived class inherits constructors and the responsibility to call them. There is a reason why derived class needs to do it: initialization of the base state of an object along with potentially setting unaccessible private members. The syntax of constructor reuse resembles inheritance syntax. All constructors are equal in responsibility to set an object with the correct state; therefore, a derived class constructor needs to reuse any (but one) constructor of the base class.

Concerning the previous example, the code can be modified as follows.

**Code listing 2.9** TinyC+. Example of the base constructor reuse.

```c
class Vector2D {
    protected double x;
    protected double y;

    public Vector2D(double x, double y) {
        this->x = x;
        this->y = y;
    }
    ...
};
class Vector3D : Vector2D {
    ...
    public Vector3D(int x, int y, int z) : Vector2(x, y) {
        this->z = z;
    }
    ...
};
```

According to the addition of constructor reuse, its production rule in the respectful EBNF part updates to the following representation.

**Code listing 2.10** TinyC+. Class declaration in EBNF.

```csharp
CLASS_DECL := 'class' identifier [ ':' identifier ] '{' { FIELD_DECL | METHOD_DECL | CONSTR_DECL } '}' ';' | CONSTR_DECL := ACCESS identifier FUN_ARGS [ ':' identifier '(' [ EXPR { ',' EXPR } ] ')' ] BLOCK_STMT
```
2.5 Polymorphism

The polymorphism principle finds itself in tinyC+ as a dynamic dispatch of virtual methods. Declaring the method as virtual means that class allows inheritor classes to redefine this method in their declaration. Because of that, virtual methods may have more than one definition, making it impossible for the compiler to know what version to call. Therefore virtual method call is dynamically dispatched - the method version to call is chosen at runtime based on a class used to instantiate the object, regardless of what type the object has right now.

By default, tinyC+ does not allow member redefinition (base and derived class having members of the same name) or method overloading (having different methods under one name and differ only in type signature). The reason for this decision has practical reasons. Firstly, unique names would help students easily match tinyC+ code with its representation in tinyC. Secondary, having members of the same name complicates the review and validation of the output code. On the other hand, it has its benefits in language concept sense-making class member naming consistent: one name - one entity.

One may think that the redefinition of methods contradicts the inability to use base member names. Still, it perfectly fits into the “one name - one entity” policy as an overridden (redefined) method should appear as the same method the accessed class has for outside code.

A virtual method declaration consists of a special annotation between the end of arguments parenthesis block and opening curly brace (scope opening). The available annotations are: “abstract”, “virtual”, and “override”. Those words are language keywords and cannot be used as identifiers. The purpose of the “override” keyword is to specify that method is a redefined base class virtual method. The transpiler will raise an error if a user tries to override a method that was not declared in the base classes or does not have the “virtual” or “abstract” annotation in any of them. The reversed condition is also true: “virtual” or “abstract” annotations cannot be used to redefine virtual methods - they are always placed at the first declaration of a virtual method in the related class inheritance tree.

The previously shown “Vector” method to calculate squared magnitude could be rewritten as follows. Which now allows both “Vector2D” and “Vector3D” have the same sqrMagnitude defined to their needs. If object of type “Vector3D” is substituted as object of type “Vector2D” and “sqrMagnitude” method is call for it then the “Vector3D.sqrMagnitude” definition is used.

Code listing 2.11 TinyC+. Example of virtual method declarations.

```cpp
class Vector2D {
    ...
```
The syntax decision was motivated by existing ambiguity in the parsing of field and method declaration: both have the same pattern (access modifier + type + name) and diverge only after name declaration, where the method expects arguments block, but field declaration terminates. This is why tinyC+ expects method virtualization annotation happens after arguments, as it makes field and method parsing less ambiguous.

The updated EBNF part of the class method production rule looks as follows.

```
METHOD_DECL := ACCESS FUN_HEAD (((‘virtual’ | ‘override’) ↦ BLOCK_STMT ) | ‘abstract’ ‘;’ )
```

Virtual methods annotated using the “abstract” keyword are called abstract methods. Their distinct difference compared to other annotations is the restriction to specify body. The idea of the abstract method is that it only specifies a method, which the inheritor class must define on its own. The benefits of abstract methods were discussed in the previous section about C++ language. As abstract classes do not have a definition, classes that contain them cannot be considered complete. Therefore, those classes cannot be instantiated and act only as of the abstract object type. An abstract method

When overriding, the new method definition may use the definition given by the base class. For that, tinyC+ adds the “base” keyword. When used, the keyword translates to *statically dispatched call* of the base class method - no virtualization is performed. The keyword can be
used to call any base method at any place inside the method scope. In other scenarios, the “base” keyword behaves like “this” keyword as it points to the same class instance.

The implementation of “Vector3D.sqrMagnitude” can be changed to benefit from calling the base class method to remove code repetition. The new version looks as follows.

- **Code listing 2.13** TinyC+. Example of calling base method in its override.

```cpp
... class Vector3D : Vector2D {
    ...
    public double sqrMagnitude() override {
        base->sqrMagnitude(); // calls Vector2D.sqrMagnitude (a statically dispatched call)
        double z2 = this->sqr(this->z);
        return x2 + y2 + z2;
    }
    ... // other member functions...
};
```

A side effect of inheritance is the formation of type sets - the effect has been discussed previously in sections about C++ and C#. The ability of one object to look like an object of another type but behave differently - is an important part of the polymorphism principle. However, the single inheritance alone is very limited in this regard compared to multiple inheritance. The *interface* feature was incorporated into tinyC+ to address the lack of flexibility in object types formation.

An interface is an abstract type that cannot instantiate objects on its own but rather is applied to another type. When applied, it “forces” the receiving type to implement all its members. In the case of tinyC+, the implementer types are classes. An implementer must provide a definition for methods declared by the applied interface. An interface, as the name suggests, provides an interaction framework abstracted from implementation details. In tinyC+, an interface serves as a type an object can reduce its type to - be substituted as an interface instance.

- **Remark.** Together class and interface represent object types, other types like structs, function pointers, and primitive datatypes - do not.

In tinyC+, an interface declaration resembles a struct declaration, but instead of holding fields, it only holds method declarations without definitions. All interface members must be public as the interface is purely an interaction layer. This is why the interface method does not specify an access modifier in its declaration. When implemented, the interface method must keep its access level - the user cannot redefine interface methods in a class using a private or
protected modifier. The following example shows a simple interface declaration and subsequent class implementation.


```cpp
interface ISummableDouble {
    double sum();
};

class Vector2D :: ISummableDouble {
    protected double x;
    protected double y;

    public double sum() {
        return this->x + this->y;
    }
};
```

Interface implementation by class looks similar to how inheritance happens, with the difference that the name of interfaces must come after the second colon. The second colon marks the name sequence start of the applied interfaces. Such syntax was chosen as it makes parsing easy and predictable: whatever comes after the first colon is base class (and it can be only one), and whatever comes after the second colon is interface names separated by a comma.

With the given information, the EBNF part of class production rule extends by addition of one more combination of optional terminal symbols and looks as follows.

Code listing 2.15 TinyC+. Class declaration with interfaces specification in EBNF.

```plaintext
CLASS_DECL := 'class' identifier [ ':' [ identifier [ '::' 
                       ↦ identifier { ',', identifier } ] [ '(' { FIELDDECL | METHODDECL | CONSTR_DECL } ')'] ] ];
```

Unlike Rust and modern C#, interfaces in tinyC+ do not have a default implementation. The default interface/traits implementation organizes code and naturally allows for writing common sharable code that only depends on object abstract type(super traits in Rust). However, without it, the language does not lose the ability to form types and provide common behavior via single inheritance. The primary reason behind interfaces in tinyC+ is to allow a user to connect horizontally separate vertical class hierarchies formed via single inheritance - it places bridges between class islands.

tinyC has restrictions in type casting where all cast must be explicit, which raises a particular problem. Currently, the one way to cast values of a different type in tinyC is by using the special
“cast<X>(y)” operator, where “X” is the cast type and “y” is a thing to cast. The tinyC cast type might be either a primitive data or pointer of any type - cast to struct value type is not allowed. The operator exists purely for the explicitness of type casting and should convert to nothing (as it is just a type requirement). It casts one pointer to another regardless of whether it actually points to a value of the given cast type.

However, the object casting needs to perform a runtime operation to define whether the object can be cast to the given type and return a valid pointer. The runtime operation also assumes a certain structure of the value the object pointer points to (discussed later in the implementation chapter). This is why the tinyC cast operator cannot be used to cast tinyC+ objects, as it may produce an invalid pointer in the best case and violate memory by substituting unrelated memory as the domain of the object’s content.

Considering this issue, tinyC+ introduces a special cast operator named trivially “classcast”. The class cast operator accepts any object (interface or class instance) and casts it to a given object type (interface or class).

Continuing the previous vector examples, the object type casting looks as follows.

### Code listing 2.16 TinyC+. Example of casting class instance to interface instance.

```c
double bar(ISummableDouble * s) {
    return s->sum();
}

void foo() {
    Vector3D v1 = Vector3D(1, 2, 3);
    int sum1 = v1.sum(); // 6
    ISummableDouble * s = classcast<ISummableDouble*>(&v1);
    int sum2 = bar(s); // 6
}
```

The production rule of the class cast is identical to the tinyC cast and is substituted as a nonterminal rule in the same place; the EBNF looks as follows.

### Code listing 2.17 TinyC+. Class cast operator in EBNF.

```c
E_CLASSCAST := classcast '<' TYPE '>' '(' EXPR ')' '
```

If an object does not implement the interface of or inherit/derived from the given cast type, then the class cast operator returns a special null pointer of the given cast type. The result is always either a valid casted object or a null pointer. A user can check the operator result against
the “null” keyword (the same null pointer). The “null” keyword refers to an implicitly declared global variable of type “object” with an address equal to zero.

▶ Remark. A little note from implementation: there always would be something at address zero because the null variable is declared at the top of the output tinyC code, so it effectively points to itself. This was made to ensure the existence of an “invalid” address not used by anything and to directly specify where null is used in the generated tinyC code so that students would not need to guess it.

As the reader can notice, the null has a special type called “object”. The “object” is a class type. In fact, for the class cast operator to properly work, the “object” is implicitly defined as a base class for all classes that do not specify their base. So that, when an object is cast to the “object” type, it would return the same object but cast - not null. Otherwise, the feature would not work.

Furthermore, the class cast operator guarantees to return a properly work and return acceptable value even if a variable of object type, which previously was assigned to null, is supplied. It is worth mentioning that the incorporation of the null pointer is considered a bad design. The problem is that pointer may not be checked before use, while being a null pointer, invalid dereferencing and its subsequent value use. However, a null pointer is the easiest minimal solution for the problem of safe object casting that does not require incorporating features like Rust enums, which would have a significant impact on the language design and implementation complexity.

To summarize, in tinyC+, the polymorphism principle of OOP is represented via dynamic dispatch of a class of virtual methods and interface methods. The cast between types is driven by a special class cast operator, which tries to cast the object to the asked type if possible. However, if not - the null is returned. By checking the result against null, a user can always identify whether the class cast succeeds or not.
The implementation of the tinyC+ language is a transpiler program, which translates tinyC+ code text (input) into tinyC code text (output). A transpiler is like a compiler, but instead of executable code, it produces declarative code of another language. The output code is ready for compilation by any compiler that fully implements tinyC language reference. The translation process would be viewed in a simplified form as a 1:1 mapping between input and output, omitting the internal work of the transpiler.

Remark. In the following examples of tinyC+ code representation in output tinyC code, the reader can often notice “...” inside code. However, they show code not important for the current discussion and represent either some feature representation in tinyC, which is not relevant to the given case, generated code that was previously shown, or unimportant content (like ordinary operation in the function block).

3.1 Name mangling

Before describing how the selected features were implemented, an important note must occur. For achieving implicit behavior, tinyC+ introduces a lot of supportive tinyC structures, variables, and functions. That additional code provides the functionality of those features. Throughout the following sections, each feature will introduce bit by bit of supportive code generated for it. However, they all share the same naming technique: the first character starts with an underscore followed by a unique string - a specialized prefix. Each prefix is responsible for splitting namespaces that do not intersect ever. One restriction is imposed on the user - its identifier is not allowed to start with an underscore to achieve this. This is a well-known practice for dividing user and language-specific code and can be seen in languages like C, Python, and Javascript.
3.2 Class and its simple members

The class translation from tinyC+ to tinyC is complex and consists of multiple parts because it serves as a fundament for all other features.

The first thing to examine is how class fields and methods are translated. The fields are translated to the declaration of struct, which has the same name as the class - this is the main data package that holds all fields. Such struct declaration looks as follows.

### Code listing 3.1 TinyC+ code to translate. Class declaration consisting of fields.

```c
class A {
    public int f1;
    private double f2;
    protected char f3;
};
```

Class field translation is straightforward. An access modifier is not translated as it only exists as a code writing restriction. The dots represent the currently unseen field declaration. The above class translates to a struct as follows.

### Code listing 3.2 Generated tinyC code. Class data representation.

```c
struct A {
    ...
    int f1;
    double f2;
    char f3;
};
```

A non-virtual method of a class is translated to a tinyC function, where the first argument is of type "void*" and called “this”. This is the main reason why the tinyC+ design forces users to explicitly use “this” variable, as nothing needs to be changed when it comes to translation. It is a little quirk that made the development of the transpiler easier.

### Code listing 3.3 TinyC+ to translate. Class method declaration and call.

```c
class A {
    ...
    public int m1(int arg1) {
        return arg1 + this->f1 + getf2();
    }
```
The non-virtual method calls are static dispatch to their tinyC function equivalence. One of the qualities of life improvement of tinyC+ was the independent declaration of methods and constructors - users could call them in each other’s bodies without worrying about the order of declaration or providing forward declarations. The implementation behind it is trivial and consists of placing forward declaration for each method and constructor tinyC function representation before any of their full declaration takes place. The method translation of declaration and call looks as follows.

**Code listing 3.4** Generated tinyC code. Class method declaration and call.

```c
struct A {
    int f1;
    int f2;
};
int _CF_A_m1(void* this, int arg1);
int _CF_A_getf2(void* this);
int _CF_A_m1(A* this, int arg1) {
    return arg1 + this->f1 + _CF_A_getf2(this);
}
int _CF_A_getf2(A* this) {
    return this->f2;
}
void m1(A a) {
    int r = _CF_A_m1(&a, 1);
}
```

When translated to tinyC, the method’s name is prefixed with a special string in case a user has defined functions with the same name (as was in this case) to ensure no naming collision.

**Code listing 3.5** Generated tinyC code. Method name prefixing.

```c
generated method name: '_CF_ + class name + '_ + method name
```
The translation of a class constructor is similar to method translation with a difference in naming generation pattern and specified return type. Consider the following example.

**Code listing 3.6** TinyC+ to translate. Class constructor declaration and call.

```cpp
class A {
    ...
    public void A(int v1, double v2) {
        this->f1 = v1;
        this->f2 = v2;
    }
};
void foo() { A a = A(1, 2); }
```

The name generation looks as follows.

**Code listing 3.7** Generated tinyC code. Method name prefixing.

```cpp
generated constructor name: '_Cmake_' + unique number to resolve multiple declarations + '_' + class name
```

As the constructor is for creating objects, it cannot take an instance from outside, and usage of “this” variable to set object values means that it must be declared somehow. For that, the transpiler places hidden unaccessible by user “this” variable declaration of the target class value type. On the next line, the “this” pointer is declared and immediately assigned to point to the “this” address. The reason for the double declaration is to preserve the pointer access type for all member access of “this” variable. In the end, the “_this” is returned. With this, the final translation is as close to the original code as possible, making it easy to compare and review.

**Code listing 3.8** Generated tinyC code. Class constructor declaration and call.

```cpp
A _Cmake_1_A(int v1, double v2) {
    A _this;
    A * this = &_this;
    ...
    this->f1 = v1;
    this->f2 = v2;
    return _this;
}
void foo() { A a = _Cmake_1_A(1, 2); }
```
3.3 Single inheritance and constructor reuse

In the case of the single inheritance, the transpilation process is simpler as fields are copied to the inheritor class’s struct, and base class methods are substituted when access throw derived class instance. Consider the following example.

Code listing 3.9 TinyC+ code to translate. Inheritance of base class fields and methods.

```cpp
class A {
    private int f1;
    public void foo() { ... }
};
class B : A {
    public int f2;
};
void foo(B * b) { b->foo(); }
```

The fields are copied with respect to class precedence in inheritance hierarchy: base class fields are declared before the inherit class fields. This allows the inheritor class instance to be substituted as the base class instance without violating the address position of fields in the struct. For base method invocation, the necessary step is to cast the object to an acceptable pointer type - the type of the base class that has declared that method. The above code translates to the following.

Code listing 3.10 Generated tinyC code. Inheritance of base class fields and methods.

```cpp
struct A {
    int f1;
};
... void _CF_A_foo(A * a) { ... }
... struct B {
    int f1;
    int f2;
};
... void foo(B * b) { _CF_A_foo(cast<A*>(b)); }
```

Previously, declarations were one to one match. However, it was the simplest example. Depending on their case, the feature discussed further produces multiple declarations and supportive
The first such case is the translation of a derived class constructor. For other classes to reuse their base class constructors, one more tinyC function representation needs to be generated for initialization purpose. Consider the following example of single inheritance.

**Code listing 3.11** TinyC+ code to translate. Example of the base class constructor reuse.

```c
class A {
    public A(int v1) { ... }
};
class B : A {
    public B(int v1) : A(v1) { ... }
};
```

The translation process creates two tinyC functions each representing the constructor use-case: object creation (make) or object setup (init). When the base class's constructor is reused, the derived class calls the init function of the base class right after class instance declaration. The name of the init function, as seen below, simply replaces the word “make” with “init”.

**Code listing 3.12** Generated tinyC code. Example of the base class constructor reuse.

```c
A _Cmake_1_A(int v1) { ... }
void _Cinit_1_A(A * this, int v1) { ... }
B _Cmake_1_B(int v1) {
    B _this;
    B * this = &_this;
    ...
    _Cinit_1_A(cast<A*>(this), v1);
    ...
    return _this;
}
void _Cinit_1_B(B * this, int v1) {
    _Cinit_1_A(cast<A*>(this), v1);
    ...
}```
### 3.4 Virtual methods

One virtual method may have multiple implementations across the given inheritance chain, which means that the transpiler cannot just call an exact tinyC function representation. Thankfully, tinyC has the ability to declare function pointers. Unlike regular pointers, a function pointer can be called the same way tinyC functions do. Before calling it, the function pointer must have its value assigned. The assignable values are tinyC function address or another function pointer. A function pointer type declaration consists of a function signature (return and arguments types) and its name.

Therefore, virtual methods can be represented as function pointers inside the class struct and called from within it. However, while straightforward, such a technique might be quite inefficient when a class instance is copied around. Additionally, because objects are created via classes, the implementation of their virtual methods never changes, which means placing them in each class instance unnecessarily wastes memory.

A smarter solution can be applied - the use of virtual tables. A virtual table is a struct that contains function pointer fields, each representing a virtual method of the dedicated class. One virtual table is specified for one class. This way, the class struct needs to have only one pointer pointing to the vtable instead of numerous function pointers. tinyC+ uses virtual tables to deal with method virtualization. Consider the following example for a demonstration of the virtual table’s declaration.

**Code listing 3.13** TinyC+ code to translate. Virtual method declaration.

```c
class A {
    public int field1;
    public void method1() virtual {
        ...
    }
};
```

The translation looks as follows: the first line declares the function pointer type of the virtual method “method1”, and the second line declares the virtual table structure responsible for holding the function pointers. The function pointers have the name of the method they point to; in this example below, the only pointer is for the “method1” method. The global instance of the virtual table is declared right after the struct declaration. Followed by a class struct declaration where, now, the first field is always the pointer to its virtual table: “_vt”; class fields (including inherited ones) come after it. After class declaration comes to its implicit default
constructor, which assigns “_vt” to point to the global class virtual table instance. Finally, the method declaration is placed.

**Code listing 3.14** Generated tinyC code. Example of the virtual method declaration.

```c
typedef void (*_CFtype_A_method1)(A*); // function pointer type
struct _VTtype_A { // virtual table of the class (A)
    ... _CFtype_A_method1 method1;
};
_VTtype_A _VTinst_A; // virtual table instance of the class (A)
struct A { // class A
    _VTtype_A * _vt; // class’s pointer to the virtual table
    int field1;
};
A _Cmake_0_A() {
    A this;
    this._vt = &_VTinst_A;
    // virtual table assingment during class construction
    return this;
}
void _CF_A_method1(A* this);
void _CF_A_method1(A* this) { ... } // virtual method’s function
```

The above example only consists of declarations. However, tinyC has no idea about the relations between that structures. Therefore, an action that connects all of it (in other words setups classes) needs to occur. In tinyC+, that function is generated for each class and called at the top of the entry function. A tinyC+ program does not compile if the entry function is missing because of this reason. By default entry function is any function with the name: “main”. This is not a design requirement but rather specifics of the written transpiler. The transpiler allows to define entry function name by calling it with “–entry=x” argument, where (x) is the name of the new entry.

The previous example’s class setup function declaration and call look as follows.

**Code listing 3.15** Generated tinyC code. Example of the class setup function.

```c
void _Csetup_A() { // setup of global variables related to class (A)
    ... _VTinst_A.method1 = &_CF_A_method1; // assingning function address to
    the pointer
}
...
A tinyC+ program can be written without setup classes in the entry function. The alternative solution could be to check for class setup inside class constructors, and if the class is not initialized - initialize it. However, that would require an unnecessary check each time a class instance is created, while it only needs to be run once. Secondary, it enlarges output code, making it harder to read. This is why tinyC+ transpiler uses the entry function as a single place for all class setups.

With the virtual table ready, let’s extend the previous example by adding a new class (B) that inherits (A) and overrides its virtual method. The (B)’s override of (A)’s virtual method uses the “base” keyword to reuse (A)’s method execution - the virtual table is not involved in this case: (A)’s method call is statically dispatched inside (B)’s method when accessed using the “base” keyword.

Code listing 3.16 Generated tinyC code. Example of the virtual method use.

class A {
    ...
    public void method1() virtual { ... }
};
class B : A {
    public void method1() override {
        if (...condition x...) {
            base->method1();
        } else {
            ...
        }
    }
};

Everything that was generated for the (A) class and shown in the previous examples applies to any other class. The function pointer types (for its methods), virtual table struct and global instance, class struct, constructor, and method function declarations are generated for each class.
The tinyC+ code of the example above translates to the following tinyC representation omitting supportive code.

**Code listing 3.17** Generated tinyC code. Difference between usage of “base” and “this” keywords.

```c
typedef void (*_CFtype_A_method1)(A*);
typedef void (*_CFtype_B_method1)(B*);
struct _VTtype_A {
    ...
    _CFtype_A_method1 method1;
};
struct _VTtype_B {
    ...
    _CFtype_B_method1 method1;
};
void _CF_A_method1(A * this) { ... }
void _CF_B_method1(B * this) {
    if (...condition x...) {
        _CF_A_method1(cast<A*>(this)); // statically dispatched
    } else{ ... }
}
void foo(A * a, B * b) {
    a->_vt->method1(a); // dynamically dispatched
    b->_vt->method1(b); // dynamically dispatched
}
```
3.5 Interfaces

Interfaces have adaptable translation: when the class implements interface methods as non-virtual methods and the method is called on some instance of that class: the call is statically dispatched as there is only one implementation for the given method (inside the class).

However, when an interface instance is obtained from the class instance, and the same method needs to be called - static dispatch cannot be used. As with virtual methods, the method call on the interface instance must be dynamically dispatched because it can have several implementations across different classes. Therefore, interfaces need a similar structure to a virtual table to hold function pointers to one of its implementations. For that, the tinyC+ transpiler generated a special “impl” struct for each interface type. Consider the following example of a simple “ICollection” interface declaration.

Code listing 3.18 TinyC+ to transpile. Example of the interface implementation declaration.

```c
interface ICollection {
    int get_count();
};
class Stack :: ICollection {
    public int get_count {...}
};
class List :: ICollection {
    public int get_count {...}
};
```

When interface declaration is translated, the above code generates the following structures: function pointer types (for methods declaration) and structures like virtual table, which can hold pointers to functions of interface method type. The method names are preserved.

Code listing 3.19 Generated tinyC code. Example of the interface implementation struct.

```c
typedef int (*_IFtype_ICollection_get_count)(void*);
struct _Iimpl_ICollection {
    _IFtype_ICollection_get_count get_count;
};
```

However, the pointer to the interface implementation cannot be stored inside the class struct (like how the virtual table is referenced) because the pointer position inside the class struct cannot have the same position relative to class struct fields for all classes implementing it. It
was not a problem with the virtual table pointer as it is always the first field of any class struct, but the class can implement an arbitrary amount of interfaces - transpiler cannot have any assumption on their pointers position inside the class struct. A solution is to collect all interface implementation pointers into a function, which could return the correct pointer based on some unique identifier. So the interface implementation representation by “Stack” and “List” with their “get interface implementation” function looks as follows.

Code listing 3.20 Generated tinyC code. Example of the interface implementation by class.

```c
_Impl_ICollection _Cimpl_Stack_ICollection;
_Impl_ICollection _Cimpl_List_ICollection;
...
struct _VTtype_Stack { _Cgetif_ _gi; ... };    
struct _VTtype_List { _Cgetif_ _gi; ... };    
// Both 'List' and 'Stack' virtual tables reference their get
  interface impl function
...
void* _Cgeti_Stack(int id) { // Stack's get interface implementation
    switch(id){
      case 0: return cast<void*>(&_Cimpl_Stack_ICollection);
      default: return null;
    }
}
void* _Cgeti_List(int id) { // List's get interface implementation
    switch(id){
      case 0: return cast<void*>(&_Cimpl_List_ICollection);
      default: return null;
    }
}
...
void _Csetup_Stack() {
  _VTinst_Stack._gi = _Cgeti_Stack;
  // assigning 'get interface impl' function for class virtual
    table
  _Cimpl_Stack_ICollection.get_count = cast<
    _IFtype_ICollection.get_count>(&_CF_Stack_get_count);
  // initialization of class interface implementation instance
}
void _Csetup_List() {
  _VTinst_List._gi = _Cgeti_List;
  // assigning 'get interface impl' function for class virtual
    table
```

tinyC+ Language Implementation
The number in the case statement represents the static id of "IimplICollection". When a tinyC+ program is transpired, a unique id is assigned to each interface type. The “get interface implementation” function is stored on the top of class virtual table struct at the same position for all classes - this way, it can be accessed without the worry of reading something else instead of the interesting function pointer. Because the “get interface implementation” function has the same signature across all classes, and in order to operate on a class instance of an unknown class type (interface type), a generalized virtual table structure can be defined at the top of the program as a cast target for all class instances. With this information given, the “get_count” method call can look as follows.

**Code listing 3.21** Generated tinyC code. Example of calling an interface method.

```c
typedef void* (*_Cgetif_)(int);
struct _VTany_ { // generalized virtual table structure
  _Cgetif_ _gi;
};

int foo(void * classInst) // inst: class instance of unknown class type
{ // interface type
  _VTany_ * vt = cast<_VTany>(inst); // as table is the first field
  _ImplICollection * impl = cast<_ImplICollection>(vt->_gi(0)); // gets implementation of 'ICollection'
  if (impl != cast<_ImplICollection>(null)) { // if class implements '
    return impl->get_count();
  } else {
    return 0;
  }
}
...
```

However, this is a bit inefficient solution: each time the interface method is called, the access of interface implementation requires a function call which imposes runtime cost: for example, the switch case can be implemented as a binary search tree and, therefore, take $O(\log_2 n)$ where $n$
is the number of implemented interfaces. Ideally, the access to interface implementation should take $O(1)$. Interface implementation needs to be cached somewhere to do that. As there is no place inside the class struct where to store interface implementation and at the same time make it uniquely accessible from all classes, the transpiler changes the interface instance itself to a tuple of class instance and interface implementation pointers - an interface view. For that, a generalized interface structure named “_Iview_” is declared at the program’s top. A special “cast” function is declared for each interface to create the view struct. With this information given, the final look at how the interface method is called looks as follows (only parts changed from the previous example are demonstrated).

**Code listing 3.22** Generated tinyC code. Example of calling an interface method (implementation caching version).

```
struct _Iview_ {
    void* this;
    void* impl;
};
...
_Iview_ _Icast_ICollection(void* inst) {
    _Iview_ view;
    if (inst == null) {
        view.impl = null;
        view.this = null;
    } else {
        _VTany_* vtable = cast<_VTany_*>(inst);
        _Iimpl_ICollection* impl = cast<_Iimpl_ICollection*>(vtable->_gi(0));
        if(impl == null){
            view.impl = null;
            view.this = null;
        } else{
            view.impl = impl;
            view.this = inst;
        }
    }
    return view;
}
...
int foo(void* classInst) // inst: class instance of unknown class type (interface type)
    _Iview_ c = _Icast_ICollection(classInst); // creation of
```
An interface method call on its instance translates to an interface accessing its implementation and calling the associated function pointer on it, passing all arguments after passing the class instance into it as the first argument.

Everywhere where the interface pointer is declared, it is translated to interface view variable/-field declaration. When the interface is passed as an argument, the copy of the view is passed. When the interface instance is cast back to the class instance or used in binary operation (like in the case of checking against null) - the “this” field, which holds the reference to the class instance, is returned. Integration of an interface view instead of an interface pointer is seamless and goes unnoticed during the program execution.
### 3.6 Object type casting

The final part of implementation is the transpilation of the class cast operator. Consider the following tinyC+ example, where two objects of different types (classes) are converted to interface type then back again, and finally, \( b \) is cast to the type of \( a \).

#### Code listing 3.23 TinyC+ code to transpile. Example of object type casting.

```c
interface I { void foo(); }

class A : I { public void foo() virtual { } }

class B : A : I { public void foo() override { } }

void main() {
    A a = A();
    B b = B();
    // cast from class to interface
    I * i1 = classcast<I*>(&a);
    // cast from interface to class
    A * a1 = classcast<A*>(i1);
    // cast from class to class
    A * a2 = classcast<A*>(&b); // upcasting
    B * b1 = classcast<B*>(a2); // downcasting
}
```

Before showing the transpilation of class to the class operator, the reader can see in the transpilation of “class to interface” located below, which is just a call to the global “Icast,” function of the interface (I) that returns interface view - as casting class instance to interface type is about creating interface view on it.

#### Code listing 3.24 Generated tinyC code. Class to interface casting.

```c
void main() {
    A a = _Cmake_0_A(); // creation of object (a)
    B b = _Cmake_0_B(); // creation of object (b)
    // cast from class to interface
    _Iview_ i1 = _Icast_I(cast<void*>(&a));
}
```

In the upcasting case of the class cast operator, the transpilation does not impose any runtime cost as the operation cannot fail. Therefore, just casting pointers is enough. The transpilation of upcasting looks as follows.

void main() {
    ...
    A * a2 = cast<A*>(kb);
}

However, in the case of downcasting and casting from interface to class type, a special function
is necessary. That function would check whether the given class instance (in the interface view,
the class instance is stored at “this”) is null and whether the class used to create it inherits from
or is the target class (class pointer type between <...> symbols).

In order to check inheritance, a special “_Ccheck_” function is generated for each class. The
function consists of a switch statement that checks integer argument “id” against ids of classes.
The class id is unique to it and assigned during transpilation - the same technique used to generate
interface ids. The function’s switch-case consists of cases representing inheritance relation with
a class (represented by its id). The generated inheritance check functions of class (A) and (B)
looks as follows.

- Code listing 3.26 Generated tinyC code. Class inheritance check function example.

```c
// --- class A --- id:1
void* _Ccheck_A(void* inst, int id) {
    switch(id){
        case 1: return inst;
        case 0: return inst;
        default: return null;
    }
}
// --- class B --- id:2
void* _Ccheck_B(void* inst, int id) {
    switch(id){
        case 2: return inst;
        case 1: return inst;
        case 0: return inst;
        default: return null;
    }
}
```

- Remark. The zero in the switch case of classes inheritance check functions is the id of the
“object” class. As it was mentioned in the design part of “classcast”, every class implicitly
inherits from the hidden “object” class.

Those check functions are accessible from the virtual table of each class (the same way
“interface get implementation” was done). Following respectful parts of virtual table generations are changed: virtual table struct of each class and global generalized virtual table struct, class setup functions.

**Code listing 3.27** Generated tinyC code. Upcasting case.

```c
typedef void* (*_Ccheckf_)(void*, int);
typedef void* (*_Cgetif_)(int);
struct _VTany_ {
    _Ccheckf_ _cc; // class inheritance check function pointer
    _Cgetif_ _gi;  // interface impl. getter function pointer
};
...
struct _VTtype_A {
    _Ccheckf_ _cc;
    _Cgetif_ _gi;
    ...
    // method function pointers
};
...
void _Csetup_A() {
    _VTinst_A._cc = &_Ccheck_A;
    ...
}
...
struct _VTtype_B {
    _Ccheckf_ _cc;
    _Cgetif_ _gi;
    ...
    // method function pointers
};
...
void _Csetup_B() {
    _VTinst_B._cc = &_Ccheck_B;
    ...
}
```

Now, with class inheritance check functions declared, the global class cast function (placed at the top of the function, right after the “_VTany_” declaration) looks as follows.

**Code listing 3.28** Generated tinyC code. Global class cast function.

```c
void* _Ccast_(void* inst, int id){
    if (inst == null) {
        return null
    } else{
```
With this information given, downcast and interface to class casting cases look as follows.

**Code listing 3.29** Generated tinyC code. Downcast and interface to class casting cases example.

```c
void main() {
    A * a2 = ...;
    _IviewI i1 = ...;
    // cast from interface to class
    A* a1 = cast<A*>(_Ccast_(i1.this, 1)); // (A) class id: 1
    // downcasting
    B* b1 = cast<B*>(_Ccast_(cast< void*>(a2), 2)); // (B) class id: 2
}
```

To conclude, the most challenging part of implementation is the transpilation of features related to the polymorphism principle of OOP: interfaces, virtual methods, and object casting. All other features related to the encapsulation, abstraction, and inheritance principles were the simplest to transpile and impose almost no runtime cost on a generated code.
Conclusion

In conclusion, the work has explored some mainstream OOP languages and, based on the findings, selected a minimal viable subset of OO features. The design chapter justified the rejection and selection of each feature where the main decisive attribute was how it addresses OOP principles. The work succeeds as educational material providing students of the “Code Generators” course with minimal OOP language based on the language they are already familiar with. The resulting tinyC+ language was able to preserve all features of tinyC, allowing students of the course to effortlessly rewrite a tinyC program into a tinyC+ program with a minimum amount of change. The language is implemented as a transpiler program that supports the transpilation of all described language features.

The future work might include the evolution of tinyC+ from a programming language with a minimalistic object-oriented feature set to a more mature version usable outside the tiny-verse educational context. The future language version can incorporate previously rejected features such as events, properties(getter & setter), and something like Rust super traits with the default interface implementation. The language implementation can take the form of a full-featured compiler that can generate linkable executable code for hardware architectures like x86 or RISC-V.
Bibliography


