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.
Implementation for Object-Oriented Languages

Bc. Rasul Seidagul
Acknowledgements

I would like to thank my supervisor Petr Maj for his consultation and advising throughout writing of this paper.

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.
Abstract

This work will discuss translating simple Object-Oriented programming language down to its simpler non-Object-Oriented language subset. In the chapter 2, the author briefly discusses existing Object-Oriented programming features. The chapter 3 describes the design of TinyC+ and what features from the chapter 2 were adopted or rejected. For methodology and as an example the chapter 4 is dedicated to the overview of how features were implemented in the transpiler. In addition, in the chapter 5, the author evaluates TinyC+ in terms of the structural overhead of an output program and provides the author’s comments. Finally, the author shares the conclusion of the final output.

**Keywords:** object-oriented programming, transpiler, tinyc

**Supervisor:** Ing. Petr Máj
## Contents

**Introduction**  
1 Object-Oriented Features
  1.1 Class  
  1.2 Inheritance  
  1.3 Interfaces  
  1.4 Virtual inheritance  
  1.5 Virtual methods  
  1.6 Abstract methods  
  1.7 Access modifiers  
  1.8 Typecasting  
  1.9 Messaging  

2 Tiny C++ Design
  2.1 TinyC++ Design  
  2.2 Grammar
    2.2.1 Class  
    2.2.2 Method  
    2.2.3 Polymorphism  

3 Implementation
  3.1 Class  
  3.2 Methods  
  3.3 Name collisions
    3.3.1 Virtual tables  
    3.3.2 Implicit initialization and construction  

4 Evaluation
  4.1 Classes  
  4.2 Methods  
  4.3 Runtime initialization  

5 Conclusion

Bibliography
Introduction

This work aims to demonstrate how to translate features of an Object-Oriented programming language down to its simpler subset. The main goal is to supplement educational material for the course NI-GEN, which teaches students how to write compilers to a C-like programming language called “TinyC” and translate its high-level features down to low-level assembly language like “Tiny86”. This paper among with the resulting language reference and the transpiler is an introductory material of Object-Oriented features translation, that will be a part of “tiny-verse” - an umbrella project for all educational materials for the NI-GEN course of the Faculty of Information Technologies, Czech Technical University in Prague.

This work briefly introduces existing object-oriented features used in other programming languages with respect to the problems they solve and the complexities they introduce.

Finally, the paper describes the implementation of OOP translations and evaluation of the TinyC+ transpiler that lowers the OOP features down to the minimal TinyC feature set. The translation would be viewed in simplified form as 1:1 mapping between OOP features and their representation in TinyC, omitting how the transpiler really implements it.
Chapter 1
Object-Oriented Features

This chapter will briefly present features that were considered by the author during the development of the transpiler to be added into TinyC+ programming language. For each feature will be discussed the purpose it serves. Features are described mainly based on how they appear in C++, however they were detached from language specifics in order to leave room for ideas brought by other languages as TinyC+ is not a strict copy of C++, but an alike project.

To have a better perspective of the following features, it is required to recall what they exist for. Object-Oriented Programming is a paradigm that involves identifying concepts and using objects to structure the way that these concepts are embodied in a software system (program). The fundamental concept of OOP is an object, which is an abstract logical unit of a program that addresses the program’s concern, represented by collection of data, procedures over that data, and identity (in relation to concern).

1.1 Class

A class is a fundamental language construct of OOP, which represents a template used to create objects in a program. The main idea of a class: it organizes a program into reasonable pieces of code where each is responsible for different program concerns. Class could be seen as collection of data declaration (”fields”) and ”methods”. Method represents function or procedure that implicitly operates on an instance of the method’s class. A method is accessible as a member of a class - the same way as a field. Conceptually a method is seen as an interface to an object, through which a program could interact with it. Classes alone do not bring much value, however it is a core feature all other OOP features are built on. Classes (or concepts like it) is a base to bring core OOP concepts such as polymorphism, inheritance, and encapsulation.
1. Object-Oriented Features

```csharp
// Example of a class in C# [5]
public class Person
{
    // Default constructor:
    public Person()
    {
        Name = "unknown";
    }
    // Explicit constructor:
    public Person(string name)
    {
        Name = name;
    }
    // Auto-implemented readonly property:
    public string Name { get; }  
    // Method that overrides the base class
    // (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
```

1.2 Inheritance

In object-oriented programming, when the features and functions of one class are obtained by another class for a reason to behave similarly - the process is called “Inheritance”. The class obtained from is called “Parent class” (“Base class”) and the class obtained to is called “Child class” (“Derived class”). Whenever, the child class obtains functionality and features from multiple parent classes, the process is called Multiple Inheritance [4]. Therefore, Single inheritance is when only one class is selected as a base. Inheritance provides code reusability, code organization, and a framework for code extension. Multiple inheritance contributes to polymorphism allowing for a single object to care properties of many base classes, however it comes with a variety of problems:

- The addressing problem - class A and B declares a member with the same name, but with different purpose, making it ambiguous what to return when a member is addressed via name inside a program.
- The combination problem - should a member of class A and B, having the same name, be combined into one member or kept separate.
- The problem of representation - how to inherit content of multiple classes in the predictable order, where the compiler could treat access to the
base class member the same for single and multiple inheritance cases.

```cpp
// Example of multiple inheritance in C++
#include <iostream>
using namespace std;

class Polygon {
    protected: int width, height;
    public:
        Polygon (int a, int b) : width(a), height(b) {}
    }

class Output {
    public: static void print (int i);
};
void Output::print (int i) {
    cout << i << std::endl;
}

class Rectangle: public Polygon, public Output {
    public:
        Rectangle (int a, int b) : Polygon(a,b) {}
        int area () { return width*height; }
    }

class Triangle: public Polygon, public Output {
    public:
        Triangle (int a, int b) : Polygon(a,b) {}
        int area () { return width*height/2; }
    }

    int main () {
        Rectangle rect (4,5);
        Triangle trgl (4,5);
        rect.print (rect.area());
        Triangle::print (trgl.area());
        return 0;
    }
```

### 1.3 Interfaces

Modern OOP languages provide a feature that ensures an object meets certain expectations, like a method with exact name and signature. Such features implementation differs from language to language. For example C# presents an 'interface', a special type contract that enforces the class/struct it was declared on and will implement interface: methods, properties, and delegates. In C++ it is achievable via multiple inheritance with self-restrictions (e.g. pure virtual function declaration only). In Rust it embodies in form of Traits, requiring (as C# interfaces) declaration of certain functions for the type it declared on. The contract may also provide a default implementation, leaving
an option for the contracted type to declare its implementation only when it
differs from the default one.

### 1.4 Virtual inheritance

Virtual inheritance is the technique used along with the multiple inheritance
feature to avoid duplication of functionality and features of parent classes
to be obtained by child class, when multiple parent classes have the same
base class. For instance, if class D following multiple inheritance is a child
class for classes B and C, which are children of class A, virtual inheritance
avoids duplication of features and functionality inherited by class B and C,
but rather picks only one of them.

### 1.5 Virtual methods

Virtual methods is a concept of allowing method definition to be overridden
by derived classes, while keeping the same name and type signature. This is
useful to extend or change base class methods’ behaviour to suit new problems
that the derived class addresses. By keeping the same type signature and
name, now other objects could interact through a single interface and yet get
different results depending on the callable object hidden type.

### 1.6 Abstract methods

Usually what is referred to as abstract methods is a method declaration
that does not have a definition and it is intended that a derived class gives
definition to it. Simply put, it is a promise of the current class that his
children will express it in their own way but preserving the method type
signature and name, forming an object API that greatly contributes to the
concept of polymorphism.

### 1.7 Access modifiers

Access modifiers represent encapsulation, which is considered one of the core
concepts of OOP. The feature contributes to complexity decrease, allowing to
keep consistent API for an object without exposing unnecessary information.
There are a variety of access modifiers for different purposes, in example
below a reader could see:

- public modifier - allowing outside code to access the marked member.
- final modifier - marking class as non-inheritable.
- private modifier - marking member as accessible within the class scope
  only.
1.8 Typecasting

Strongly typed languages are required to implement object type casting, allowing it to turn into a more generalized or specific type of object when needed. There are two types of cast in relation to inheritance:

- **downcasting** - when an instance of derived type is casted into base type.
- **upcasting** - when an instance of base type is casted into a specific derived type.

Both castings assume a given instance to be valid for both types, otherwise a failure should be provided. In languages such as C++ it is achieved via "dynamic_cast" and on failure it returns nullptr. In C# it is represented in more natural language via the "as" and "is" operators, the first tries to cast into target class type and the second casts into target class only when cast is possible. While the "is" operator may be used as a boolean operator, in case of casting as the "as" operator they both return null objects on failure.

1.9 Messaging

Alan Kay, one of the designers of Smalltalk - one of the very first OOP languages and unique in its own way, noted that the language started around the idea of messaging between objects. However, what messages is a board term accommodating an immense amount of ideas on how objects should interact between each other. For example, C# presents “events” - a special language construct representing a collection of methods, lambdas, and functions. C# "events" key features is the fact that it is invokable/callable only by object it belongs to, and other objects could only listen for its invocation. Usually messaging is not presented in any form in the language, implicitly dedicating method calls or function pointers to serve this purpose.
Chapter 2
Tiny C+ Design

2.1 TinyC+ Design

TinyC+ as a language serves only one purpose - to be a simple Object-Oriented programming language, whose translation into TinyC is simple, concise, and yet valuable in terms of education. Meaning that demonstration of its translation is much more valuable than its practical use. Therefore, the author decided to make TinyC+ as small as possible consisting only of fundamental features that together forms core of Object-Oriented programming such as:

- **Class** – as mentioned earlier, the class is the most common language construct in object-oriented programming languages, therefore, is simple for learning and usage of students. Additionally, the class feature represents the idea of data and logic encapsulation into a single unit.

- **Single-Inheritance** – inheritance is a core feature of object-oriented programming language, therefore, it’s important to demonstrate for students how it is translatable into simpler language constructs.

- **Virtual methods** – this feature is added to support polymorphism, which is essential part of object-oriented programming language.

The multiple inheritance was rejected in favor of single inheritance because it drags with it a lot of problems that needs to be addressed and its solution mostly relies on transpiler inside work rather than translated output produced by it. On the other hand, the presence of single inheritance translation is already enough to cover the topic as students would be able to support multiple inheritance once they learn how a simpler case was made. Therefore, the necessity of multiple inheritance was low, so the addition of it would complicate the development process in comparison to the educational value it brings. It is worth clarifying that the transpiler itself is not a goal of the project, rather it is a means of demonstration. The virtual inheritance without multiple inheritance does not make sense threfor was rejected along with it.

The access modifier was not added to the language design, because they only exist as meta information for the transpiler and does not translate to
anything in TinyC at all. It could be mentioned for students that some features do not need a representation in the final output acting only as contextual information, and that is it.

Interfaces share a lot of similarities with multiple inheritance, with exception that it does not provide code reuse rather it forces user to write more code. Therefore, they have been rejected for the same reason multiple inheritance was. The below example shows how similar the translation of interface’s method and class’s virtual method is in the result.

```cpp
// Example of possible TinyC+ code using interfaces

interface I {
    void execute();
};

class A : I {
    void execute() { ... }
    void print() virtual { ... }
};
class B : A {

    // ... the above translates to ...

    struct I {
    }
    struct A {
    // (A) vtable points to (A__execute, A__print)
        void __tinycpp__A__execute(I* this) { ... }
        void __tinycpp__A__print(A* this) { ... }
    struct B {
    // (B) vtable points to (A__execute, A__print)

    
Abstract methods feature is addition to the whole concept of virtual methods. However, as the access modifiers they only bring contextual information and do not change the resulting translation. Students are free to add it after implementing virtual tables as it is only a matter of time to add it into their transpiler. In the case of TinyC+, it is not necessary.

In case of type casting feature - TinyC already has a dedicated operator for casting and the TinyC+ transpiler as a consequence of virtual methods adoption will allow TinyC cast operator to handle upcasting and downcasting of instances of class type. Therefore the only practical value of special objects-dedicated cast operators will be a transpiler error(warning) when non-class types are used and a zero-value pointer as a result of cast failure, which would really translate into a function with if statement inside. Adding it to the language will make it less minimalistic while bringing insignificant value to the final translation.
Messaging feature was rejected because it requires a more complex underlying mechanic, such as memory management (for seamlessly subscribing/unsubscribing/notifying listeners), which is not a thing nor in TinyC, nor in TinyC+. TinyC does not even specify a way for dynamic memory allocation. Therefore, any feature that bases itself on the concept of dynamic collections is not just implementable. A concept of Smalltalk-like messaging is possible, however, as with interfaces its TinyC representation would structurally be very similar to virtual method calls. In fact, it will make translation worse by requiring all types to have a virtual table, which will make the result output more homogeneous with less distinct cases.

## 2.2 Grammar

The TinyC+ is a superset of TinyC grammar, meaning that every TinyC language reference is preserved ‘as-is’ with no modification or removal. The grammar will be described using Enhanced Backus-Naur Form [2]. For sake of simplicity, several non-terminal rules were omitted from E-BNF representation, and these are:

- **identifier** - alphanumeric string allowing use of underscores, but requires the first character to be an alphabet character.
- **integer** - integer number
- **double** - rational number in finite decimal representation
- **char** - an ASCII character surrounded by single quote
- **string** - sequence of ASCII characters surrounded by double quote

To distinguish grammar added by TinyC+ it has been colored blue.

```
PROGRAM := { FUNDECL | FUNPTRDECL | VAR_DECLS ';'
  | STRUCT_DECL | CLASS_DECL }
FUNDECL := TYPE_FUN_RET identifier '('
  [ FUN_ARG { ',' FUN_ARG } ] ')' [ BLOCK_STMT ]
FUN_ARG := TYPE identifier
TYPE_FUN_RET := TYPE | void
FUNPTRDECL := typedef TYPE_FUN_RET
  '(' '*' identifier ')'  
  '(' [ TYPE { ',' TYPE } ] ')'  ';'
VAR_DECLS := VARDECL { ',' VARDECL }
VARDECL := TYPE identifier [ '[' E9 ']' ] [ '=' EXPR ]
STRUCT_DECL := 'struct' identifier [ '{
  { TYPE identifier ';' } '}' ] ';
CLASS_DECL := 'class' identifier [ : identifier ]
  [ '{' { TYPE identifier ';' | METHODDECL } '}' ] ]
  ';
```
METHOD_DECL := TYPE identifier '('
    [ TYPE identifier { ',' TYPE identifier } ]
')' [ 'virtual' | 'override' ] [ BLOCK_STMT ]
TYPE := ('int' | 'double' | 'char' | identifier) {'*'}
    | = 'void' {'*' | '***'}
BLOCK_STMT := '{' { STATEMENT } '}
STATEMENT := BLOCK_STMT | IF_STMT | SWITCH_STMT
    | WHILE_STMT | DO_WHILE_STMT | FOR_STMT
    | BREAK_STMT | CONTINUE_STMT | RETURN_STMT
    | EXPR_STMT
IF_STMT := 'if' '(' EXPR ')' STATEMENT
    [ 'else' STATEMENT ]
SWITCH_STMT := 'switch' '(' EXPR ')' '{' { CASE_STMT }
    [ 'default' ':' CASE_BODY ]
    { CASE_STMT } '}'
CASE_STMT := 'case' [ integer | char ] ':' CASE_BODY
CASE_BODY := { STATEMENT }
WHILE_STMT := 'while' '(' EXPR ')' STATEMENT
DO_WHILE_STMT := 'do' STATEMENT 'while' '(' EXPR ')' ';
FOR_STMT := 'for' '(' [ EXPR_OR_VAR_DECL ] ';' 
    [ EXPR ] ';' [ EXPR ] ')' STATEMENT
BREAK_STMT := 'break' ';
CONTINUE_STMT := 'continue' ';
RETURN_STMT := 'return' [ EXPR ] ';
EXPR_STMT := EXPR_OR_VAR_DECL ';
F := integer | double | char | string | identifier
    | (' EXPR ') | E_CAST
E_CAST := 'cast' '<' TYPE '>' '(' EXPR ')' E_CALL := '(' [ EXPR ']' E_INDEX := [' ' EXPR ']
E_INDEX := [' ' EXPR ']
E_CALL_INDEX_MEMBER_POST := F { E_CALL | E_INDEX
    | E_MEMBER | E_POST }
E1 := E_UNARY_PRE { ('*' | '/' | '%' ) E_UNARY_PRE }
E2 := E1 { ('+' | '−') E1 }
E3 := E2 { ('<<' | '>>') E2 }
E4 := E3 { ('<' | '<=' | '>' | '>=' ) E3 }
E5 := E4 { ('==' | '!=' ) E4 }
E6 := E5 { ('&' E5 }
E7 := E6 { ('|' E7 }
E8 := E7 { ('&&' E7 }
E9 := E8 { ('||' E8 }
12
2.2. Grammar

```
EXPR := E9 [ ‘=’ EXPR ]
EXPRS := EXPR { ‘,’ EXPR }
EXPR_OR_VAR_DECL := VAR_DECLS | EXPRS
```

### 2.2.1 Class

A class declaration is a collection of fields and methods declaration. The order of declaration is important: a member (A) declared after declaration of member (B) is not visible to member (B). A class inherits all members of a class declared after colon (second identifier in CLASS_DECL, the "base" class). All members of the base class are visible to methods of derived class.

```
// Example of simple File class addressing I/O concern
class File {
  int descriptor;
  char[256] buffer;
  int cursor;
  void open(char * path) {
    descriptor = sys_open(path, ...other parameters);
    cursor = 0;
  }
  void close() {
    sys_close(descriptor, ...other parameters);
  }
  char read_char() {
    char c = sys_read(descriptor, buffer, cursor, ...
    return c;
  }
}
```

### 2.2.2 Method

A method declaration is almost identical to a function declaration, but adds two new keywords: 'virtual' and 'override'. The 'virtual' keyword marks method as virtual, allowing the method to be re-defined in derived classes with use of 'override' keyword.

```
class Shape {
  double compute_area() virtual { return 0.0; }
  void scale_by(double value) virtual { }
};
class Rectangle : Shape {
```
2. Tiny C+ Design

```cpp
double width; double height;
double compute_area() override {
    return this->width * this->height;
}
void scale_by(double value) override {
    this->width = this->width * value;
    this->height = this->height * value;
}
```

Methods are accessed as members of a class. When called it implicitly operates on a pointer to the class instance (callee) and adds it to the method’s body context. The pointer is accessible via "this" keyword. When a method is declared in derived class it implicitly adds another pointer to the body context, which is accessible via "base" keyword and has the same value as "this", but with a different type - a type of the base class of the method’s class. A user could expect a method call via base pointer to call the method’s last definition known to the base class.

```cpp
Shape * make_rectangle(
    double width,
    double height,
    double scale
) {
    Shape result;
    result.width = width;
    result.height = height;
    result.scale_by(scale);
    return result;
}
int main(int argc, char** argv) {
    if (argc < 3) return -1;
    double width = str_to_double(argv[1]);
    double height = str_to_double(argv[2]);
    double scale = str_to_double(argv[3]);
    Shape * s = make_rectangle(width, height, scale);
    return s->compute_area();
}
```

### 2.2.3 Polymorphism

A user is able to expect upcasting and downcasting to work on correct types (e.g. derived class pointer type to base class pointer type). However, the transpiler does not guarantee safety in case of incorrect types (e.g those that are not within the same inheritance tree) - validity of cast is defined by user only. This decision is motivated by the design of TinyC+ to extend the base language without modifying existing features such as cast. Consider an example:
class A {
    // content of A
};

class Union {
    int data[64];
    // methods to store values...
    int read_int() {
        // the reason for offset of 1
        // will be discussed in
        // Implementation chapter.
        return data[1];
    }
    // ...other reads
    A read_A() {
        return *cast<A*>(cast<int>(this) + 1);
    }
};

Even though class (A) and (Union) does not derive from the same class or have any other relations - the above code is meaningful and should not be rejected. It is fair to say that addition of other cast operators that specify on classes may solve the issue and secure a user from making mistakes. Like "dynamic_cast" in C++ that returns the nullptr on failure, or "is" and "as" constructs of C#. However, for sake of TinyC+ simplicity, such complications were omitted.
Chapter 3
Implementation

The translation happens via a transpiler. A transpiler is a program that is similar to a compiler: both translates one source code into another, however, the compiler produces executable code whereas the transpiler produces declarative-only code. The TinyC+ transpiler was developed by using tinyverse common shared libraries and was based on the source code of the TinyC frontend of tiny-verse. The existing AST tree, type context, and type-checker were modified to support the new OOP features. Due to the visitor-based API of TinyC AST the development of new stages and modification of existing ones was easier than writing everything from scratch. The original list of ASTs was replenished with ASTs of class and method declarations. In code examples, important parts will be colored blue.

3.1 Class

Translation of TinyC+ class involves the usage of multiple TinyC language constructs. The core representation of a class in TinyC language is a struct with the same name. The struct contains all fields and a pointer to the virtual table of the translating class. The pointer to the virtual table is declared first, before the fields. Classes that didn’t declare or inherit any virtual methods will have their pointer to the virtual table be equal to zero (null reference).

Due to single inheritance, the declaration of the fields in the target struct are straightforward: starting from a class on top of the inheritance tree (the first ancestor class) following down to the translating class with respect to their declaration order.
3. Implementation

```cpp
class Counter {
    int value;
    void tick() {
        this->value++;
    }
};

int test() {
    Counter c;
    c.value = 0;
    c.tick();
    c.tick();
    c.tick();
    return c.value;
}
// the above is translatable
// by the transpiler to:
struct Counter {
    int value;
};
void __tinycpp__Counter__tick(Counter * this) {
    this->value++;
}
int test() {
    Counter c;
    c.value = 0;
    __tinycpp__Counter__tick(&c);
    __tinycpp__Counter__tick(&c);
    __tinycpp__Counter__tick(&c);
    return c.value;
}
```

### 3.2 Methods

Their representation comes right after the struct representation of a class when translated to TinyC. A representation of a method is the TinyC function with almost 1:1 mapping of the declaration and body part. The only changes necessary are the name and insertion of a pointer to the class instance as the first parameter. The name of the pointer is obviously "this" allowing to leave all class member accesses untouched. However, the use of the "base" keyword does not have a direct representation in TinyC but translates to a cast of "this" pointer to the type of 'base' pointer.

For simplicity sake, below is dummy code of game written in TinyC+, which will be used in all following examples of Implementation Chapter as code to translate. The code successfully utilizes all features of TinyC+ in the
most minimal way, which is great for demonstration of long translations.

```cpp
// (*) Imagine code of some RPG game
// written in TinyC+
class Npc {
    int pos_x;
    int pos_y;
    int attack(int x, int y) virtual {
        ... // punch animation and effects
        return damage;
    }
};
class Soldier : Npc {
    int sword_id;
    int attack(int x, int y) override {
        // While soldier uses sword for most of the times. The sword still may be dropped during a battle. If that happens he will try to punch.
        ...
        if (this->sword_id == 0) {
            damage = base->attack(x, y);
        }
        ...
        // * return damage
        return damage;
    }
};
void fight() {
    Npc bandit;
    Soldier guard;
    guard.sword_id = 0;
    guard.attack(bandit.pos_x, bandit.pos_y);
    bandit.attack(guard.pos_x, guard.pos_y);
}
```

Below is the TinyC representation of the previous code. Virtual table declaration and initialization code was omitted for sake of readability.

```cpp
... 
struct Npc{
    int pos_x; int pos_y;
};
int __tinycpp__Npc__virtual__attack (Npc* this, int x, int y) {
    ... // punch animation and effects
    return damage;
}
... 
```
struct Soldier{
    int pos_x; int pos_y; int sword_id;
};
int __tinycpp__Soldier__virtual__attack (Soldier* this, int x, int y) {
    ...
    if (this->sword_id == 0){
        damage = __tinycpp__Npc__virtual__attack (cast<Npc*>(this), x, y);
    }
    ...
    return damage;
}
...
void fight() {
    Npc bandit = __tinycpp__make__Npc();
    Soldier guard = __tinycpp__make__Soldier();
guard.sword_id = 0;
guard.__tinycpp__vtable->attack(&guard, 
                          bandit.pos_x, bandit.pos_y);
    bandit.__tinycpp__vtable->attack(bandit, 
                         guard.pos_x, guard.pos_y);
}

As you might notice, the "base" keyword when used to call a method translates to a direct call of the most recent method declaration inside the inheritance hierarchy down to the base class of the current class. This feature helps to re-declare previous implementation inside an overridden method or to call base class exact method implementation without worrying for it to be overridden by a child class. Such control makes class inheritance more predictable. If a user wants to call a method virtually expecting it to be overridden in child classes then calling them via "this" achieves that.

### 3.3 Name collisions

TinyC+ was designed as a clear extension of TinyC without any restrictions for existing language constructs. This means that any TinyC source code must work without any modification in TinyC+. However, because of the presence of methods in TinyC+ a name collision with the TinyC function is almost certain. That is why, a design decision was made that requires a user to access class members via "this" and "base" keywords mitigating the need to prioritize members. This design feature is mostly motivated by the desire to allow users to freely choose names without worrying about the existence of identically named functions of TinyC. However, implementation of virtual methods required creation of structs, function pointer types, and global and local variables that all require a unique name. For this purpose
it was decided to use prefix technique as a simple solution to the problem putting only small limitations on a user. So any user-defined name inside a program must not start as '__tinycpp__' in addition to not allowing use of keywords as names. While it is possible to resolve naming collisions by modifying all names, the prefix solution in comparison is much easier and requires only small limitations to be enforced. The example of the use of such a solution could be seen in C standardization, where users are recommended to avoid using double underscores as they consider names that could be used by the compiler or be converted into a keyword or operate with future releases of C standard.

### 3.3.1 Virtual tables

For methods dispatching, the concept of virtual tables, similar to that of C++, was adopted. Simply put, a virtual table is a collection of function pointers, pointing to the relevant implementation of virtual methods declared and inherited by the target class. The order of pointers to virtual methods implementation is similar to field translation - the order is kept based on an inheritance tree (from top to bottom) and the order of declaration within a class is preserved as well. In TinyC+ class’s virtual table translates into TinyC representation in the form of ‘struct’ declaration with fields being function pointers. Each of those function pointers has the name of the virtual method it represents and after being assigned will point to the TinyC function representation of their methods.

```c
struct __tinycpp__Npc__vtable__ {
    __tinycpp__Npc__vtable__attack attack;
};

__tinycpp__Npc__vtable__
__tinycpp__Npc__vtable__instance__;

struct Npc {
    __tinycpp__Npc__vtable__* __tinycpp__vtable;
    ...
};

int __tinycpp__Npc__virtual__attack
(Npc* this, int x, int y) {
    ...
    return damage;
}

void __tinycpp__Npc__vtable__init__() {
    __tinycpp__Npc__vtable__instance__.attack =
    &__tinycpp__Npc__virtual__attack;
}
```
3. Implementation

```c
Npc __tinycpp__make__Npc() {
    Npc this;
    __tinycpp__Npc__vtable__init__();
    this.__tinycpp__vtable = &__tinycpp__Npc__vtable__instance__;
    return this;
}

void fight() {
    Npc bandit = ...;
    ...
    bandit.__tinycpp__vtable->attack(&bandit, ...);
}
```

Information about the virtual table of a class instance is stored as a pointer to the virtual table’s struct global instance, and the name of the pointer uses reserved name prefixes and sum ups to '____tinycpp____vtable'.

### 3.3.2 Implicit initialization and construction

Unfortunately, TinyC does not support the assignment of default values for struct fields. That is why, not only pointers to the virtual table need to be assigned for each class instance, but virtual tables’ fields need to be initialized at some point as well. To do that TinyC+ implicitly creates an initialization function for a virtual table and a special 'constructor' function for every class and some struct. Firstly, let’s look at the virtual table initialization function. Their declaration in TinyC is straightforward: it returns void, requires no parameters, has a name that consists of the reserved TinyC+ prefix followed by the name of the virtual table’s class, and ends with a special '__vtable__init' suffix. Finally, the function body is just a sequence of function pointer assignments for the TinyC global instance of virtual table struct.

```c
struct __tinycpp__Npc__vtable__ {
    __tinycpp__Npc__vtable__attack attack;
};

__tinycpp__Npc__vtable__
    __tinycpp__Npc__vtable__instance__;
...

int __tinycpp__Npc__virtual__attack (Npc* this, int x, int y) {
    ...
    return damage;
}

void __tinycpp__Npc__vtable__init__() {
```
Now, let’s examine the “constructor” functions declaration. For class type, the constructor:

- Returns a class instance
- The name starts with TinyC+ prefix followed by the word "make" and ends with the class name.
- Body starts with the assignment of pointer to the virtual table with the virtual table global instance, followed by assignment of struct or class type fields that have their own implicit constructor to calls of their constructor function.
```c
}  
void fight() {
    Npc bandit = __tinycpp__make__Npc();
    ....
    bandit.__tinycpp__vtable->attack(&bandit, ...);
}
```

Additionally, a reader could notice in the code above, when the variable of struct or class type is declared without immediate assignment, the assignment to the result of a call of its constructor function (if it has one) is implicitly added when translated to TinyC.
Chapter 4

Evaluation

This chapter will evaluate the TinyC representation of implemented TinyC+ features. However, because the performance cannot be really tested as there is no optimizing TinyC compiler and the target available provides artificial performance counts not based on time, the evaluation will be solely based on the amount of used TinyC language constructs in the representation and how they potentially could affect the program performance.

4.1 Classes

TinyC+ class representation in TinyC is a struct declaration and a sequence of function declarations. Therefore, the overhead of the class language construct alone is almost non-existent, with exception of overhead of the class’s virtual table declaration consisting of: number of function pointers of declared and inherited methods in addition to virtual table pointer per declared object.

4.2 Methods

The non-virtual methods are as fast as TinyC functions. There is no downside in their use. However, it is different for virtual methods. The virtual methods are as slow as the use of function pointers. The use of function pointers is slower than direct call because an optimized compiler can’t guarantee what function will be called, therefore, optimizations such as inlining or/and control flow analysis are not possible. Besides that, virtual method calls always require dereference of the virtual table pointer from the class instance further delaying the whole process and possibly introducing cache misses if the selected compiler will produce binary runnable on real hardware.

4.3 Runtime initialization

TinyC does not specify a program entry point in its language reference, which means that pretty much any function could be set as the entry. This is expected, after all - TinyC is a replica of C and as an educational language it is valuable to challenge students addressing this problem in their way.
However, because of that, TinyC+ transpiler does not really know when to create the runtime necessary for the initialization of virtual tables. That is why there are two ways transpiler address the issue.

In the default mode, each time a class instance is declared in function/method scope - the transpiler must initialize a global instance of the virtual table (even though it might have been initialized before). Meaning that the global instance of a virtual table will be initialized countless times throughout the life of a program at each class instance declaration. You, as a reader, may think that this horrible outcome might be solved by storing a flag as the first field of the virtual table representing its initialization status. However, it is based on the false assumption that a target TinyC compiler will initialize fields with some default value like zero, which is not guaranteed by TinyC language reference either. That is why, the transpiler can’t optimize initialization of virtual tables and will produce unnecessary slower programs than what is really able to produce.

In the second mode, when the program entry is given to the transpiler via optional argument "–entry". This way a transpiler gives a user the ability to get a more optimized version by specifying the entry point, therefore, allowing the transpiler to initialize all virtual tables once in the beginning of the program(start of the entry point). If the program is huge enough, there might be noticeable short slowdown at the start of the program execution, however it pays off in comparison to the default blind solution.
Chapter 5

Conclusion

The final version of TinyC+ represents simple but complete Object-Oriented programming language and as an extension was able to preserve all features of TinyC allowing for a student of NI-GEN to effortlessly rewrite TinyC program into TinyC+ program. In the result the complexity added by TnyC+ to the TinyC language reference is simple and affordable for students to learn the language almost instantly and start engaging in transpiler development. Most of the features considered by the author on the stage of language design development were rejected as an unnecessary complication of the problem as they did not produce unique representations, but a repetition of existing forms. However, the necessary forms of translation were demonstrated allowing students to implement MOST OF OOP feature according to this work.


