# 2. Object Oriented Programming und Templates

AlDaBi Praktikum

**Fundamental Programming Styles** 

### **PROGRAMMING PARADIGMS**

### Imperative Paradigmen

- Programm besteht aus Anweisungen, die den Status des Programms ändern
- Programmierer beschreibt, wie diese Anweisungen auszuführen sind, ähnlich einem Kochrezept
- Wiederverwendbare Anweisungsblöcke werden meist in Funktionen ausgelagert
- Ergebnisse von Funktionen sind oftmals vom Programmstatus abhängig
  - Problem: Programmstatus ist global sichtbar

# Beispiel Naives Pattern Matching

```
#include <iostream>
#include <string>
#include <vector>
// Iteration über das Pattern (inner loop)
int computeLocalScore(std::string const & text, std::string const & pattern,
                      int const & textPos) {...}
// Iteration über den Text (outer loop)
std::vector<int> computeScore(std::string const & text, std::string const &
                                pattern) { . . . }
int main() {
    // Initialisierung
    std::string text = "This is an awesome tutorial to get to know some C++!";
    std::string pattern = "some";
     // Computation of the similarities
    std::vector<int> score = computeScore(text, pattern);
    // Ergebnisausgabe
    for (unsigned i = 0; i < score.size(); ++i)</pre>
         std::cout << score[i];</pre>
    std::cout << std::endl;</pre>
    return 0:
```

### OOP Paradigmen

#### Ausführung von Anweisungen mit Hilfe von interagierenden Objekten

- Objekte sind reale oder abstrakte Einheiten mit einer best. Rolle
- Objekte haben klar definierte Verhaltensweisen
- Objekte enthalten private Informationen
- Objekte interagieren miteinander über ihre Funktionalitäten

#### Programmierer designen ein Set von Objekten für das aktuelle Problem

Menschen sehen die Welt als eine Zusammenwirkung von Objekten

#### Wichtig für Softwarekonzeption, Wartung und Erweiterbarkeit

Objekt ist kleinste Moduleinheit, erweiterbar und wiederverwendbar

#### Moderne Programmiersprachen unterstützen OOP

Java und C++ wurden direkt f
ür OOP konzipiert

### OOP Prinzipien

#### **Abstraktion**

- Funktionalität gegeben über Interface (Memberfunktionen) wie bei abstrakten Datentypen
- Komplexität der Implementierung beschränkt sich auf das Objekt, d.h.
   für den Benutzer nicht notwendig und ersichtlich

#### **Encapsulation**

- Interner Status des Objektes nicht nach außen ersichtlich
- Objekt (Member) kann nur über seine Methoden verändert und eingesehen werden
- Methoden sichern ab, dass Member nur gültige Zustände erreichen

#### Vererbung

Erlaubt Wiederverwendung und Erweiterung von Objekten

#### Polymorphismus

 Erlaubt Spezialisierung von Subtypen von Objekten (z.B. nach Vererbung)

### Beispiel Parkplatz für Autos

#### Anweisungen

- Programmiere Anwendung zum Parkplatzmanagement
- Berechne zu jeder Zeit den aktuellen Gewinn durch die parkenden Autos
- Parkgebühr abhängig von der Automarke

#### Objekte

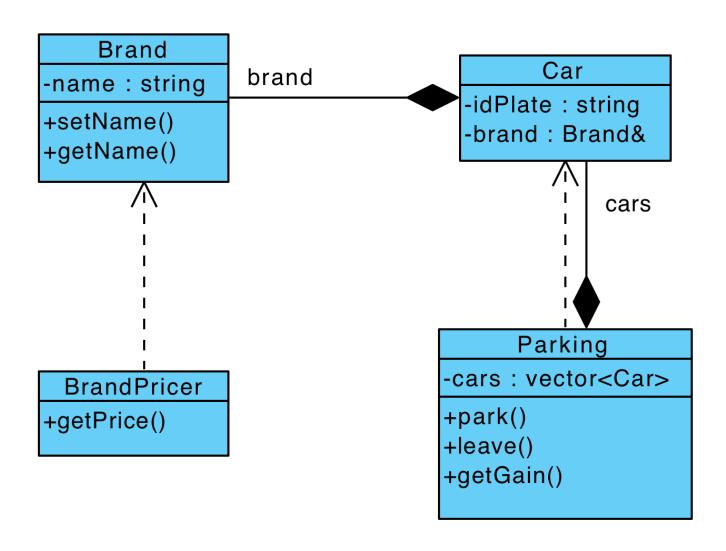
Parking: enthält Autos

– Car: Auto

– Brand: Automarke

BrandPricer: Bestimmt Gebühr anhand der Automarke

# Beispiel Parkplatz für Autos (II)



# Objektbeziehungen

#### **Assoziation**

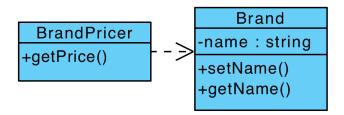
- Generische Beziehung zwischen zwei Objekten
- Objekte benutzen andere
   Objekte oder stellen sie zur
   Verfügung

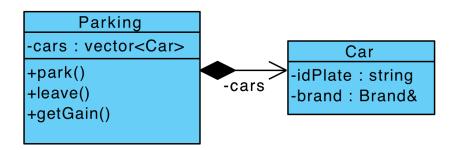
#### Aggregation

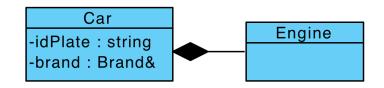
- "Hat ein(e)" Beziehung
- Tritt z.B. in Containern (z.B. Vektoren) auf

#### Komposition

- "Teil eine(s)" Beziehung
- Objekt existiert nur als Teil eines anderen Objektes







A Very Brief Overview

### OOP IN C++

### Klassen und Objekte

Klasse definiert *Implementierung* von Objekten

```
class Car { };
struct Car { };
```

#### Objekt ist Instanz einer Klasse

a, b, c sind alles Instanzen derselben Klasse Car

```
void main() {
  Car a;
  Car b;
  Car c;
}
```

### Member

- Eigenschaften eines Objektes sind die Member (Typen, Variablen) der Klasse
- Sichtbarkeit der Member kann eingeschränkt sein
  - Keyword private beschränkt Sichtbarkeit außerhalb der Klasse
  - Keyword protected erlaubt Sichtbarkeit für Subklassen

```
class Car {
private:
    string idPlate;
protected:
    unsigned seats;
public:
    string brandName;
};
```

```
void main() {
   Car c;
   // OK
   c.BrandName = "BMW";
   // Compile Error
   c.seats = 5;
   // Compile Error
   cout << c.idPlate;
}</pre>
```

### Methoden

- Methoden sind Funktionen mit einem impliziten Argument this
- this ist ein Pointer auf das spezielle, instanziierte Objekt

```
class Car {
private:
  string idPlate;
public:
  string getIdPlate() {
    return this->idPlate;
  bool setIdPlate(string &idPlate) {
    if (!idPlate.empty())
      return false:
    this->idPlate = idPlate;
    return true:
};
```

```
int main() {
   Car c;
   // Returns false
   c.setIdPlate("");
   // Returns true
   c.setIdPlate("B ER 5");
   // Prints B ER 5
   cout << c.getIdPlate();
}</pre>
```

### Methoden extern definieren

- Methoden müssen in der Klasse deklariert werden
- Der Übersichtlichkeit wegen werden sie aber oft außerhalb definiert

```
class Car {
private:
  string idPlate;
public:
   bool setIdPlate(string &);  // Deklaration
};
// Definition
bool Car::setIdPlate(string &idPlate) {
  if (!idPlate.empty())
    return false;
  this->idPlate = idPlate;
  return true;
}
```

# Method Overloading

#### Methoden (und Funktionen) können überladen werden

- Zwei Funktionen k\u00f6nnen denselben Namen, aber andere Signaturen haben
- Der Compiler wählt den mit der passendsten Signatur aus
- Overloading bezieht sich dabei nicht auf den Rückgabewert!!

```
void main() {
    // Prints 1
    Class::m(5);
    // Prints 2
    Class::m((char)5);
    // Prints 3
    Class::m(3.1, 2);
}
```

Methoden nur überladen, wenn sie *inhaltlich* (mehr oder weniger) dasselbe machen!

### Vererbung

#### Vererbung besteht aus drei Konzepten

- Strukturelle Vererbung der Methoden und Member
- Subtyping
- Method overloading

```
struct Car {
  void refill() {}
  void drive() {}
};
struct ElectricCar : Car {}
```

```
void main() {
  ElectricCar e;
  e.drive();
}
```

#### Die abgeleitete Klasse ElectricCar

- Erbt Methoden refill und drive von Car
- Ist ein Subtyp der Klasse Car
- Hat die Möglichkeit die Klasse Car zu erweitern und spezialisieren

# Subtyping

- Problem: Können wir ein ElectricCar Auto auf einem Parkplatz für Car Autos parken?
  - Yes we can! ☺

```
struct Parking {
  vector<Car> cars;

Parking(unsigned places) {
    cars.resize(places);
}

  void park(unsigned place, Car &car) {
    cars[place] = car;
}

Car & leave(unsigned place) {
  return cars[place];
}
};
```

```
void main() {
    Parking p(2);

    Car c;
    p.park(0, c);

    ElectricCar e;
    p.park(1, e);

    Car & b = p.leave(1);
    b.drive();
}
```

### Quiz

Woher weiß die Memberfunktion Date::month(), dass sie beim ersten Aufruf d1 .m und beim zweiten d2 .m zurückgeben muss?

```
Class Date {
    // ...
    int month() {return m;}
    // ...
    private:
    int y, m, d; // Jahr, Monat, Tag
};

void f(Date d1, Date d2) {
    cout << d1.month() << d2.month() << endl;
};</pre>
```

### Quiz

Woher weiß die Memberfunktion Date::month(), dass sie beim ersten Aufruf d1 .m und beim zweiten d2 .m zurückgeben muss?

```
Class Date {
  // ...
                                       int month(Date *this) {
  int month() {return m;}
                                         return *this.m;
 // ...
 private:
  int y, m, d; // Jahr, Monat, Tag
};
void f(Date d1, Date d2) {
  cout << d1.month() << d2.month() << endl;</pre>
};
          month (&d1)
                              month (&d2)
```

### **TEMPLATES**

David Weese © 2010/11

### Templates: Motivation

**Aufgabe**: Schreibe eine Funktion max(a, b), die das Maximum zweier Zahlen a und b ausgibt:

```
int max (int a, int b)
{
   if (a > b) return a;
   return b;
}
```

**Problem**: Funktion wird für alle Typen benötigt

```
float x = max(1.4, y);
```

# Lösung: Templates

Templates sind Schablonen, nach denen der Compiler Code herstellt

```
template <typename T>
T max (T a, T b)
{
   if (a > b) return a;
   return b;
}
...
float x = 1, y = 2;
float z = max (x, y);
```

### Template-Argumente

Beim Kompilieren müssen die Template-Argumente bestimmt werden. Es gibt zwei Möglichkeiten:

1. Explizit:

```
template <typename T1, typename T2>
void f (T1 a, T2 b) { ... }

f<int,float>(0, 3.5);
```

1. Implizit:

```
int x = 0;
float y = 3.5;
f(x, y);
```

# Template-Argumente, gemischt

Mischung von expliziter und impliziter Bestimmung von Template-Argumenten:

```
template <typename T1, typename T2, typename T3>
void f (T3 x, T2 y, T1 z)
{ ... }

f<int>( (char) 0, 3.5F, 7 );
```

Argument: char x
 Argument: float y

3. Argument: int z

### Implizit: Problem 1

Folgender Code verursacht ein Problem:

```
template <typename T>
T max (T a, T b)
{
    //...
}
double x = 1.0;
double y = max(x, 0);
```

```
error: template parameter 'T' is ambiguous
```

#### Grund:

- x (double) und 0 (int) haben unterschiedliche Typen
- In Funktionssignatur müssen beide Argumente denselben Typ haben

### Implizit: Problem 2

Bei folgendem Beispiel funktioniert eine implizite Bestimmung des Template-Arguments gar nicht:

```
template <typename T>
T zero ()
{
   return 0;
}
int x = zero();
```

error: could not deduce template argument

#### Grund:

Implizite Bestimmung benutzt Argumente, nicht den Rückgabewert

### Parameter-Deklarationen

Statt eines Types kann auch eine Konstante als Template-Parameter spezifiziert werden:

```
template <int I>
  void print ()
{
    std::cout << I;
}
print<5>();
```

Ausgabe: 5

### Template-Klassen

Wie Template-Funktionen lassen sich auch Template-Klassen definieren:

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
};
```

#### Beispiel:

vector, map, list (STL Template Klassen)

### Defaults für Template-Parameter

Für Template-Klassen können Default-Argumente definiert werden:

```
template <typename T = int>
struct Pair
{
    T element1;
    T element2;
};
Pair < > p;
```

Nachfolgende Parameter müssen dann auch Defaults haben.

### Template Beispiele

Frage: Wozu kann man die Argumente von Template-Klassen verwenden?

Beispiel: Typen für Member

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
    T get_max();
    void set_both(T elm1, T elm2);
};
```

### typename

Hinweis für den Compiler: "hier kommt ein Type!"

#### typename steht vor ...

- 1. Template-Argumenten, die Typen sind (keine Konstanten)
- 2. Typen, die von Template-Argumenten abhängen

### "Patterns" von Template Typen

Typen von Klassentemplates können als eine Art "Pattern" angegeben werden:

```
template <typename T1, typename T2>
struct MyClass;

template <typename T>
void function1 (T & obj);

template <typename T1, typename T2>
void function2 (MyClass<T1, T2> & obj);

template <typename T>
void function3 (MyClass<T, int> & obj);
```

# Template Spezialisierung

Templates können für bestimmte Template-Argumente spezialisiert werden.

```
template <typename T>
struct MyClass
{ int foo; };

template < >
struct MyClass <int>
{ int x; };

MyClass<int> obj;
obj.x = 17;

obj.foo = 18;  // ERROR! Keine Vererbung
```

#### Grund:

- foo ist in MyClass<int> nicht bekannt.

### Quiz: Was wird ausgegeben?

```
template <typename T>
struct C
    static void f() { std::cout << "allgemein"; }</pre>
};
template < >
struct C <char*>
    static void f() { std::cout << "speziell"; }</pre>
};
template <typename T>
void call f(T & t)
    C<T>::f();
}
typedef char * c ptr;
c ptr x;
call f(x);
```

### Quiz: Was wird ausgegeben?

```
template <typename T>
struct C
    static void f() { std::cout << "allgemein"; }</pre>
};
template < >
struct C <char*>
    static void f() { std::cout << "speziell"; }</pre>
};
template <typename T>
void call f(T & t)
    C<T>::f();
typedef char * c ptr;
c ptr x;
call f(x);
```

### Quiz: Und jetzt?

```
template <typename T>
struct C
    static void f() { std::cout << "allgemein"; }</pre>
};
template < >
struct C <char*>
    static void f() { std::cout << "speziell"; }</pre>
};
template <typename T>
void call f(T & t)
    C<T>::f();
}
call f("hallo");
```

## Quiz: Und jetzt?

```
template <typename T>
struct C
    static void f() { std::cout << "allgemein"; }</pre>
};
template < >
struct C <char*>
    static void f() { std::cout << "speziell"; }</pre>
};
template <typename T>
void call f(T & t)
                                 const char*
    C<T>::f();
call f("hallo");
```

### **TEMPLATE SUBCLASSING**

David Weese © 2010/11

### Das Delegation-Problem

Problem: Es sei z.B. folgendes Programm gegeben:

```
struct Car {
  void refill() { cout << "Gas Please"; }</pre>
  void drive() {
    if (tank.empty())
      this->refill();
};
struct ElectricCar: Car {
  // Override function from Car
 void refill() { cout << "Energy Please"; }</pre>
};
```

### Das Delegation-Problem

Problem: Was passiert hier?

```
ElectricCar e;
e.drive();
```

```
struct Car {
  void refill() { cout << "Gas Please"; }</pre>
 void drive() {
    if (tank.empty())
      this->refill();
};
struct ElectricCar: Car {
  // Override function from Car
 void refill() { cout << "Energy Please"; }</pre>
};
```

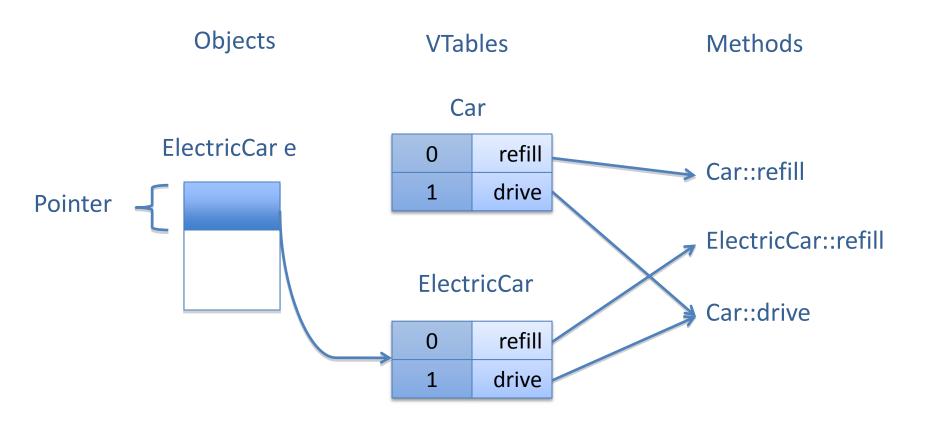
## Das Delegation-Problem

Ausgabe: Gas Pleases

```
ElectricCar e;
                     struct Car {
 e.drive();
                       void refill() { cout << "Gas Please"; }</pre>
                       void drive() {
                         if (tank.empty())
                            this->refill();
                     };
Lösung:
                     struct ElectricCar: Car {
  virtual functions
                       // Override function from Car
                       void refill() { cout << "Energy Please"; }</pre>
 struct Car {
   virtual void refill()
     { . . . }
 };
```

### Wie funktioniert virtual?

Prinzip: Objekt hält Zeiger auf eine Tabelle mit Zeigern auf die richtigen Funktionen



### Eigenschaften virtueller Funktionen

#### dynamic binding

Die tatsächlich aufgerufene Funktion wird erst zur Laufzeit bestimmt

#### Nachteile des dynamic binding

- zusätzlicher Speicherbedarf pro Objekt
- langsamer, indirekter Sprung
- kein Inlining

#### Beobachtung

- oft braucht man gar kein dynamic binding
- Typen stehen schon zur Compile-Zeit fest

## Template Subclassing

Eine alternative Lösung des Delegation-Problems mit static binding

Bisher (objektorientiert):

```
struct Car { /*Car*/ };
struct ElectricCar : Car { /*ElectricCar*/ };
```

# Template Subclassing

Schritt 1: "Template Spezialisierung statt Ableitung"

```
template <typename T>
struct Car { /*Car*/ };

struct ElectricCar;

template <>
struct Car <ElectricCar> { /*ElectricCar*/ };
```

# Template Subclassing (II)

Schritt 2: "Globale Funktionen statt Member Funktionen"

```
template <typename T>
void refill (Car<T> & obj)
    std::cout << "Gas Please"; ...</pre>
void refill (Car<ElectricCar> & obj)
    std::cout << "Energy Please"; ...</pre>
template <typename T>
void drive (Car<T> & obj)
    refill(obj); ...
```

# Template Subclassing (III)

Template Subclassing löst das Delegation Problem:

```
Car<ElectricCar> car;
drive(car);
// Output: "Energy Please"
```

## Anwendungsbeispiel

#### Gesucht: Funktion, die das größte Element eines Feldes bestimmt

- Gegeben ist einen Feld von Zeigern auf eigentliche Elemente
- Vergleichsfunktion soll frei wählbar sein

#### Lösung 1: Objektorientiert

- Basisklasse Comparable mit virtueller Vergleichsfunktion less
- Definiere Elementtyp als Kindklasse und überlade less

#### Lösung 2: Templates

- Definiere globale less-Funktion und templatisiere maxArg
- Spezialisiere less für Elementtyp

## Objektorientiert

```
struct Comparable
    virtual bool less(Comparable &right);
};
struct Pair: public Comparable
    int a, b;
    bool less(Comparable &right) {
        return a < static cast<Pair&>(right).a;
};
Comparable * maxArg(Comparable *arr[], int size)
    Comparable *max = NULL;
    for (int i = 0; i < size; ++i)</pre>
        if (max == NULL || max->less(*arr[i]))
            max = arr[i];
    return max;
```

### **Templates**

```
template <typename T>
inline bool less(T &left, T &right)
    return left < right;  // allgemein</pre>
template < >
inline bool less(Pair &left, Pair &right)
    return left.a < right.a; // speziell</pre>
template <typename T>
T * maxArg(T *arr[], int size)
    T *max = NULL;
    for (int i = 0; i < size; ++i)</pre>
        if (max == NULL || less(*max, *arr[i]))
            max = arr[i];
    return max;
```

### Vergleich

#### Objektorientiert mit virtueller Funktion

- Es existiert genau eine maxArg-Funktion
- Indirektion beim Lesen der Funktionsadresse von less
- Sprung und Rücksprung, Stackframe auf- und abbauen

#### Templates und inline

- Für jeden benutzten Elementtyp wird eine eigene maxArg-Funktion erzeugt
- less-Funktion wird direkt in maxArg eingebaut (inline)

#### Laufzeitmessung mit 1 Mrd. Elementen

- 3350 ms mit OOP
- 2150 ms mit Templates

### **BEMERKUNGEN ZUR P-AUFGABE**

David Weese © 2010/11 Sascha Meiers WS 2013/14

### Hinweise

- Punkte erscheinen in Datei punkte.txt im Repository
- Musterlösung zum Horspool unter <a href="https://svn.imp.fu-berlin.de/aldabi/WS13/material/">https://svn.imp.fu-berlin.de/aldabi/WS13/material/</a>
- Mehr zum Horspool im nächsten Tutorium
- Beachten Sie die Abgabehinweise!
  - Datei sollte aufgabe1.cpp heissen, nicht anders
  - Die n\u00e4chste soll also aufgabe2.cpp heissen (!)
  - Aufruf muss über die Kommandozeile erfolgen, nicht über cin

## Tipps zu Aufgabe 2

Funktor = Objekt mit einer Methode namens operator()
 In diesem Fall soll der Funktor 2 Suffixes vergleichen:

 Eine Funktion bool compare(int i, int j) genügt nicht, denn diese Funktion kennt den Text nicht!

## Tipps zu Aufgabe 2

 Die Sortierfunktion std::sort kann nach einer beliebigen Funktion bzw. einem beliebigen Funktor sortieren

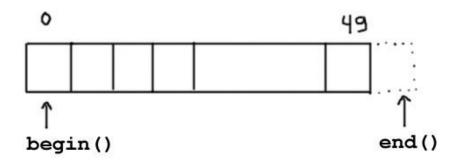
Iteratoren sind ein viel genutztes Konzept in C++

### Iteratoren

Iteratoren = verallgemeinerter Pointer

```
vector<char> vec(50);
vector<char>::iterator anfang = vec.begin();
vector<char>::iterator ende = vec.end();

cout << ende - anfang; //Ausgabe: 50</pre>
```



### Warum Iteratoren benutzen?

• summiere alle Elemente von Feld *f* 

```
int sum = 0;
for (int i = 0; i < n; ++i)
  sum += f[i];</pre>
```

wird vom Compiler genauso übersetzt wie:

```
int sum = 0;
for (int i = 0; i < n; ++i)
  sum += *(f + i);</pre>
```

effizienter ist (ohne Optimierung):

```
int sum = 0;
int *f_end = f + n;
for (int *p = f; p < f_end; ++p)
   sum += *p;</pre>
```

### Warum Iteratoren benutzen? (II)

- Warum sorgen, wenn der Compiler optimiert?
  - die Optimierung funktioniert nur bis zu einer gewissen Komplexität
- Was passiert, wenn f eine verkettete Liste ist?
  - f[i] hat dann eine Laufzeit von O(i)
  - die ganze Schleife hat dann also O(n²) Laufzeit

```
std::list<int> f(n);
for (int i = 0; i < n; ++i)
    sum += f[i];</pre>
```

- Iteratoren sind optimal f
  ür sequentielle Zugriffe
  - Schleife mit Iteratoren hat O(n) Laufzeit

```
std::list<int> f(n);
std::list<int>::iterator i = f.begin();
std::list<int>::iterator i_end = f.end();
for (; i != i_end; ++i)
    sum += *i;
```

### Iteratoren (II)

Mit Iteratoren kann man durch Container traversieren.

```
template<typename TIn, typename TOut>
inline TOut copy(TIn first, TIn last, TOut dest)
while ( first != last )
   *dest = *first;
   ++dest; ++first;
return (dest);
```