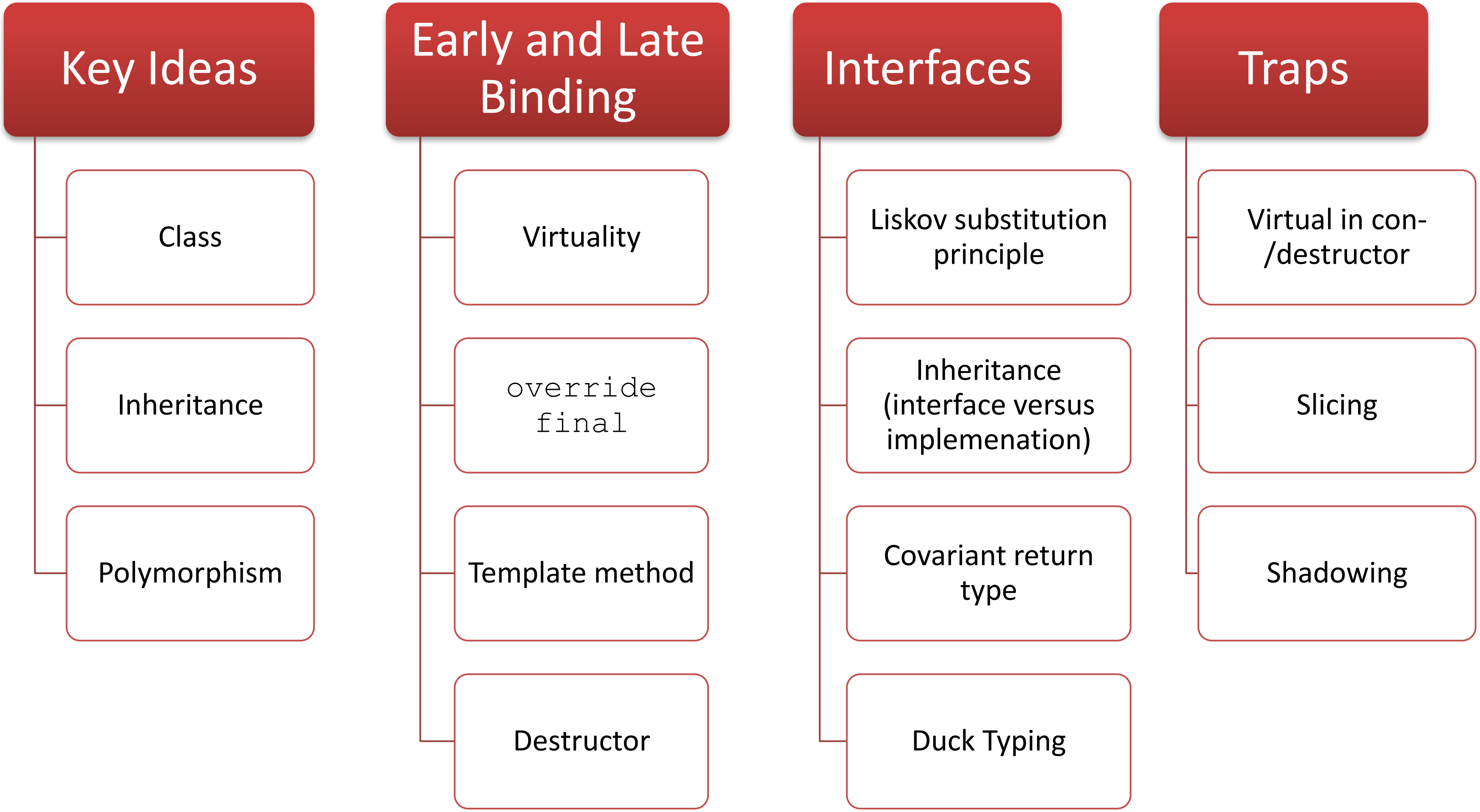
Back to Basics: Object-Oriented Programming

RAINER GRIMM

Cppcon
The C++ Conference

2021 | October 24-29

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override`
`final`
- Template method
- Destructor

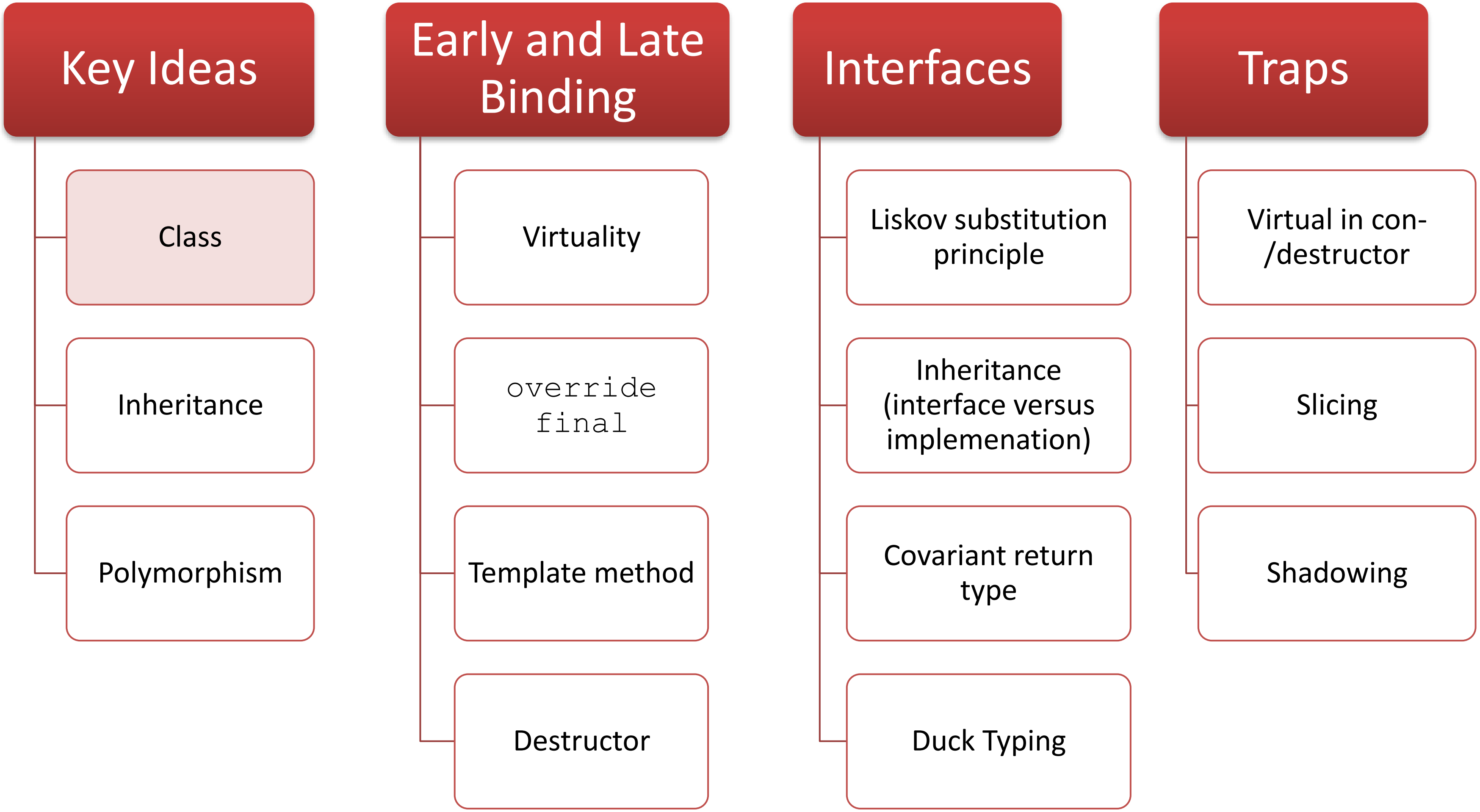
Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Object-Oriented Programming



- Key Ideas
  - Class
  - Inheritance
  - Polymorphism
- Early and Late Binding
  - Virtuality
  - `override` `final`
  - Template method
  - Destructor
- Interfaces
  - Liskov substitution principle
  - Inheritance (interface versus implemenation)
  - Covariant return type
  - Duck Typing
- Traps
  - Virtual in con-/destructor
  - Slicing
  - Shadowing

Class

C++ supports for class types:

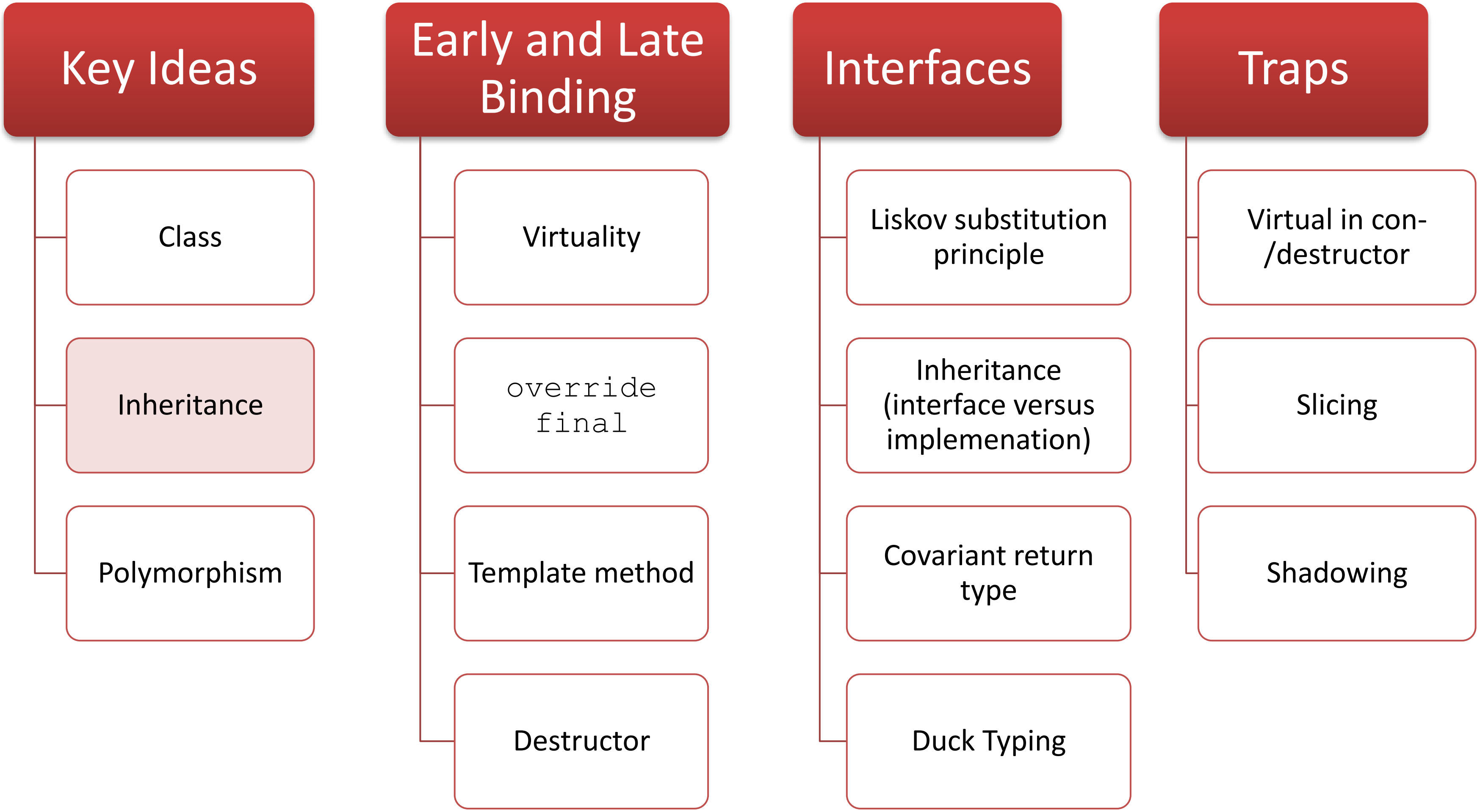
  - `class`
  - `struct`
  - `union` (I ignore them)

- Class types encapsulate its members and member functions from the outside world.

➡ Information hiding

Separation from interface and implementation

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Inheritance

The inheriting class

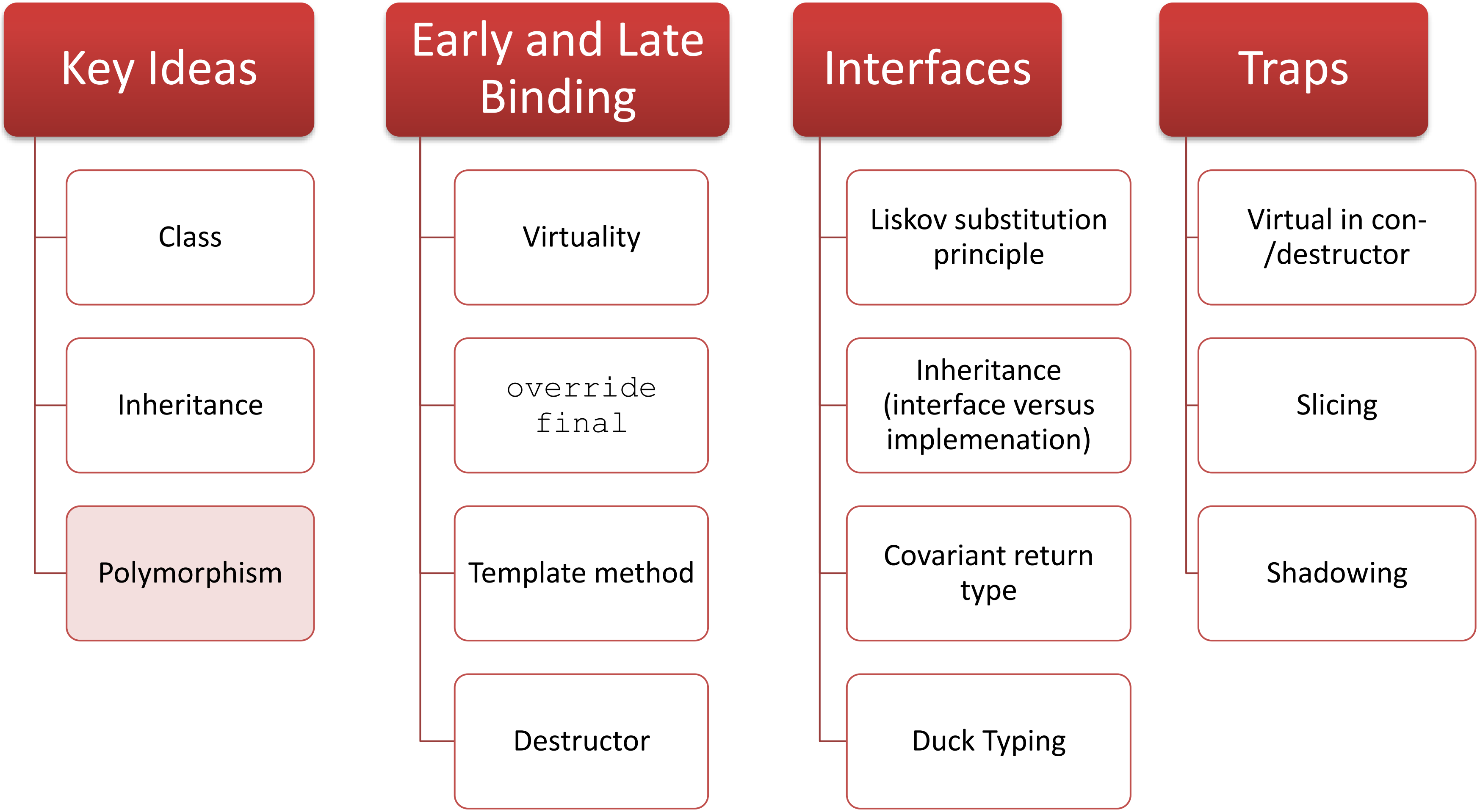
- gets all members and member functions from the inherited class.
- uses the members and the member functions of the inherited class and adds new ones.

- The access specifier of the inherited class and the access specifier of the inheritance must be considered.

Don’t inherit for code reuse. Inherit, when you want to express a logical structure.

inheritanceAccessRights.cpp

Object-Oriented Programming



- Key Ideas
  - Class
  - Inheritance
  - Polymorphism
- Early and Late Binding
  - Virtuality
  - `override` `final`
  - Template method
  - Destructor
- Interfaces
  - Liskov substitution principle
  - Inheritance (interface versus implemenation)
  - Covariant return type
  - Duck Typing
- Traps
  - Virtual in con-/destructor
  - Slicing
  - Shadowing

Polymorphism

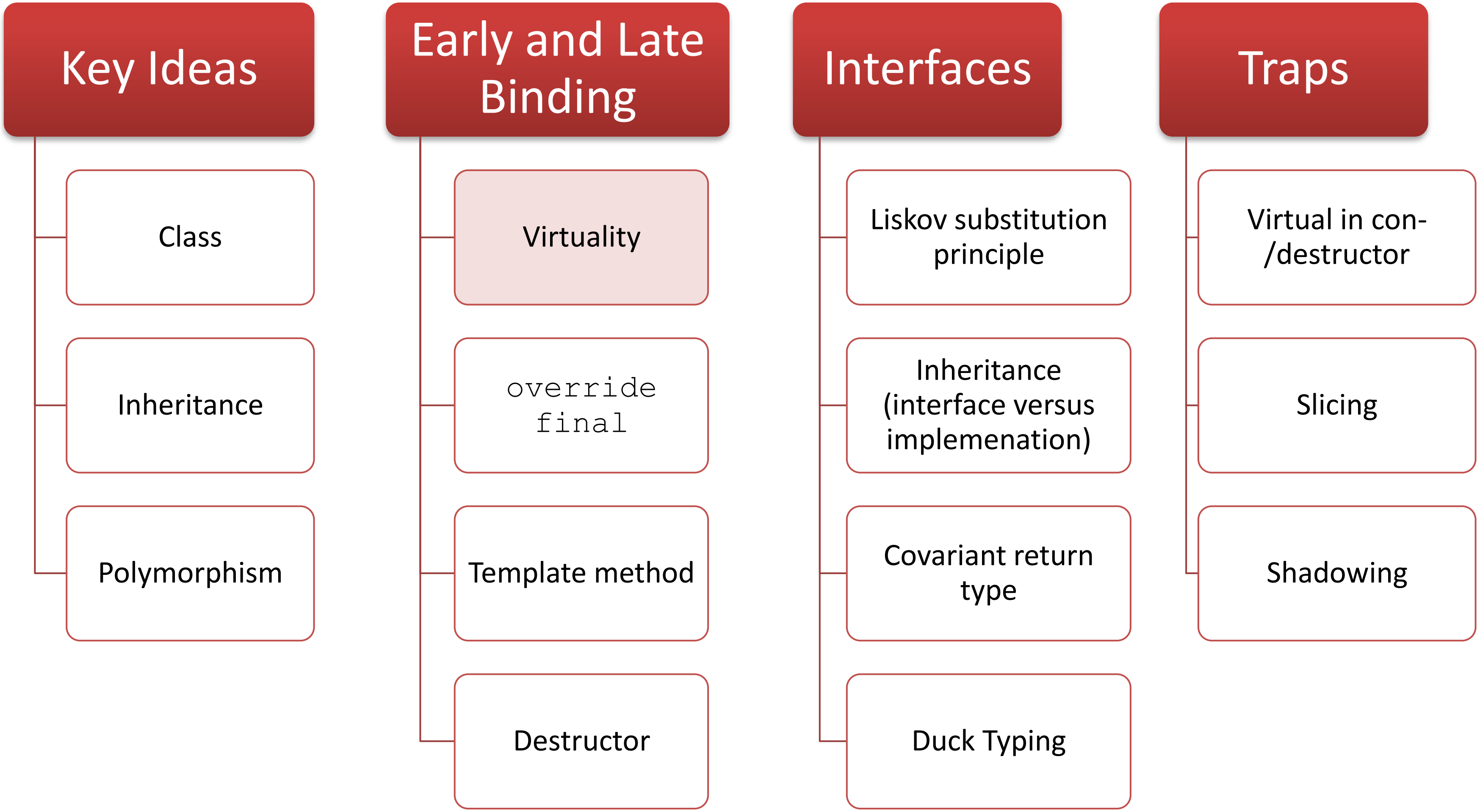
Polymorphism (poly morphs) is the characteristic of an object to behave differently at run time.

Polymorphism

- Inheritance is the base of polymorphism
- Enables the separation of interfaces and implementation.
- Involves a small overhead (pointer indirection).

The separation of the interface and its implementation is one of the crucial ideas of modern software design.

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Virtuality

Virtuality requires a

- virtual member function, and
- a pointer or reference.

```
struct Account {
  virtual void deposit(double) {...}
};

struct BankAccount: Account {
  void deposit(double) override {...}
};
```

```
BankAccount bankAccount;

Account* aPtr = &bankAccount;
aPtr->deposit(50.5);

Account& aRef = bankAccount;
aRef.deposit(50.5);
```

Distinguish between the static type and the dynamic type of an object.

Virtuality

Rules to keep in mind

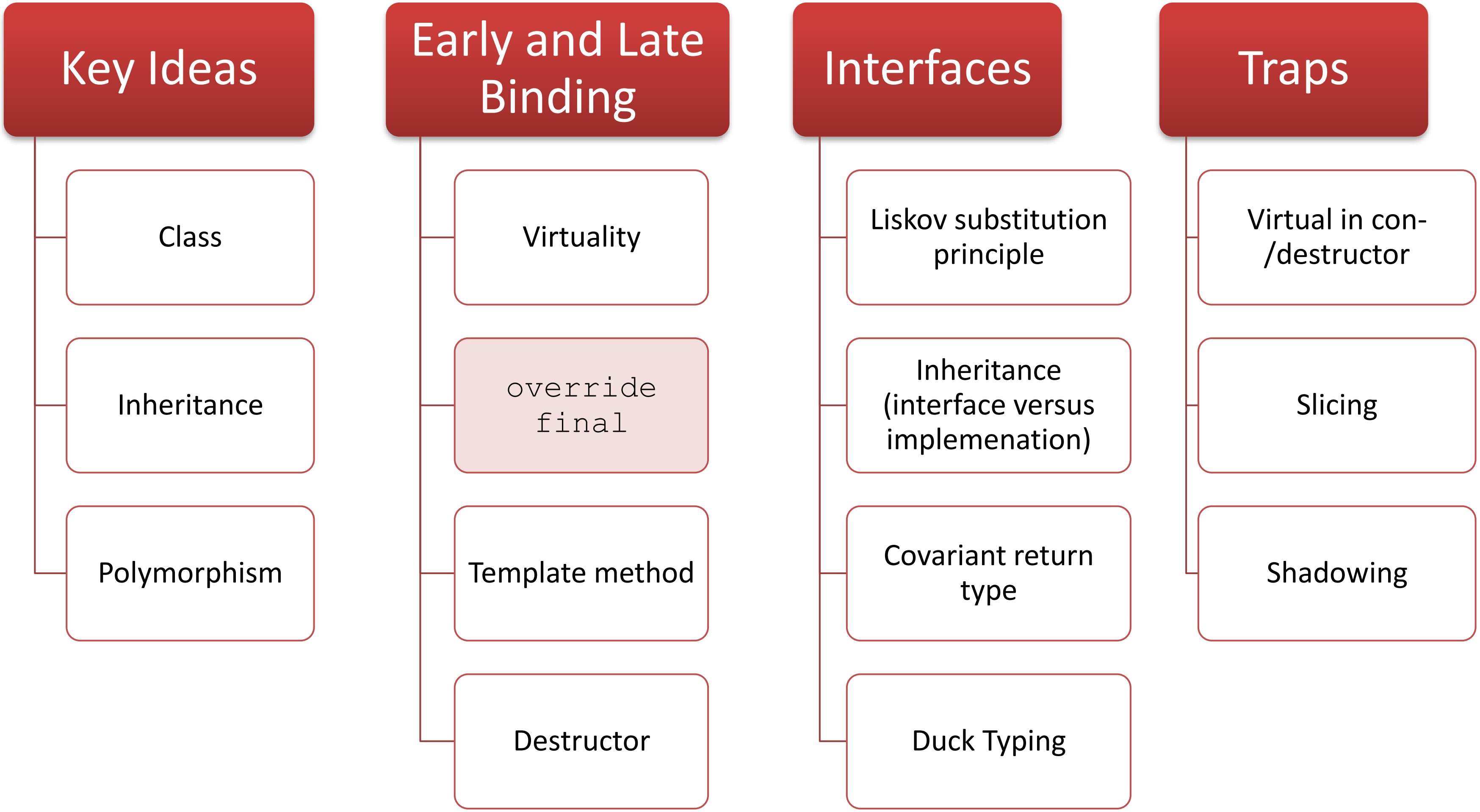
- Constructor cannot be virtual.
- A virtual member function stays virtual in the class hierarchy.
- The overriding member function must be identical to the overridden virtual function including the parameters, the return type, and the `const` qualifiers.

- Pure virtual member functions suppress the instantiation of a class and can have default implementations.

```
struct Window {
    virtual void show() = 0;
};
void Window::show() { // implementation }
```

➡ Window is an abstract base class.

Object-Oriented Programming



Key Ideas
- Class
- Inheritance
- Polymorphism

Early and Late Binding
- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces
- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps
- Virtual in con-/destructor
- Slicing
- Shadowing

`override` and `final`

An `override` declared function expresses that this function overrides a virtual function of a base class.

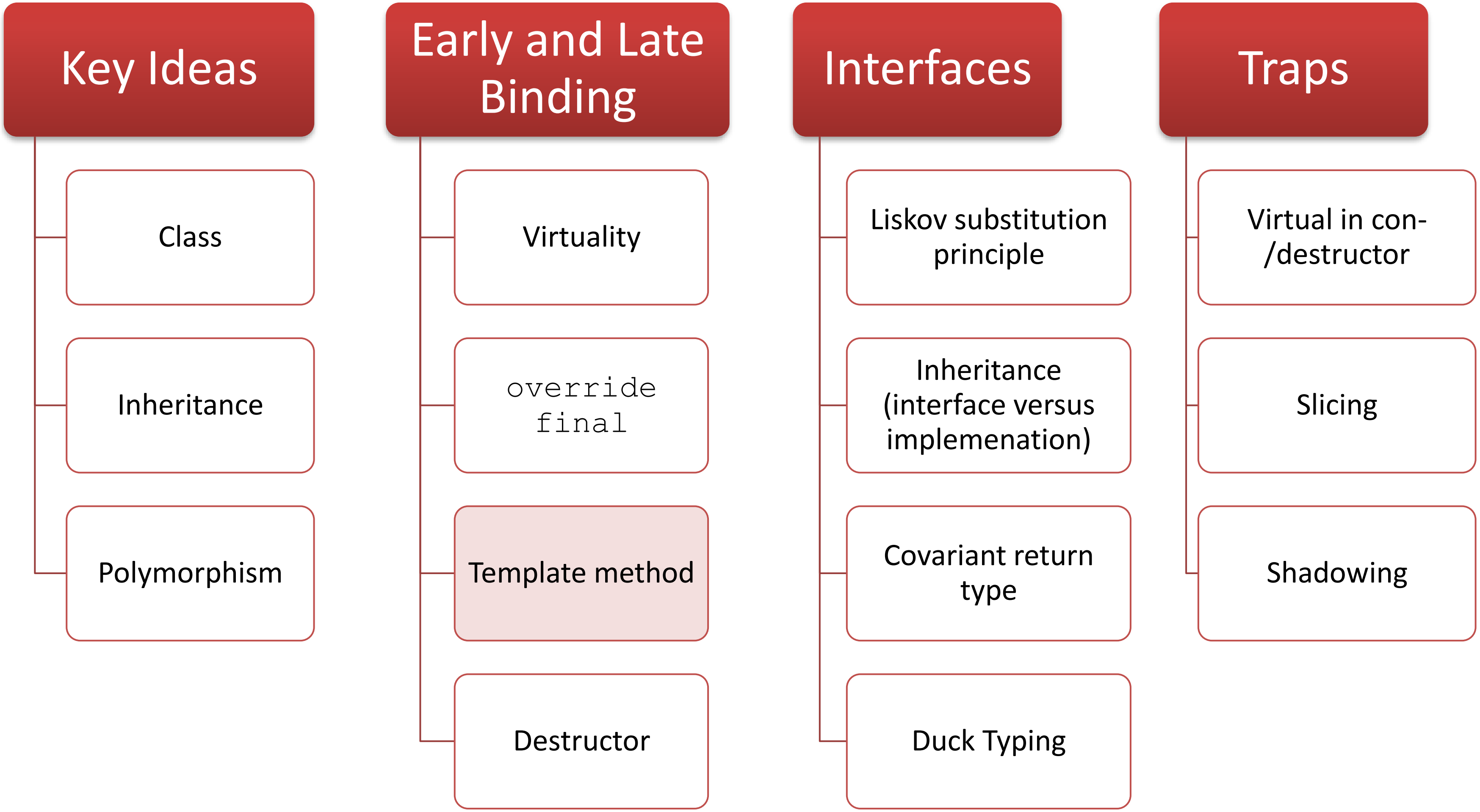
A `final` declared function expresses that this function overrides a virtual member and cannot be overridden.

- Member functions declared as `final` are an optimization opportunity for the compiler.
- Both variants are equivalent:

```
void func() final;
virtual void func() final override;
```

The compiler checks that the programmer follows the contract.

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Template Method

Type

- Behavioral pattern

Purpose

- An algorithm consists of a typical sequence of steps.
- Subclasses can adapt the steps, but not the sequence

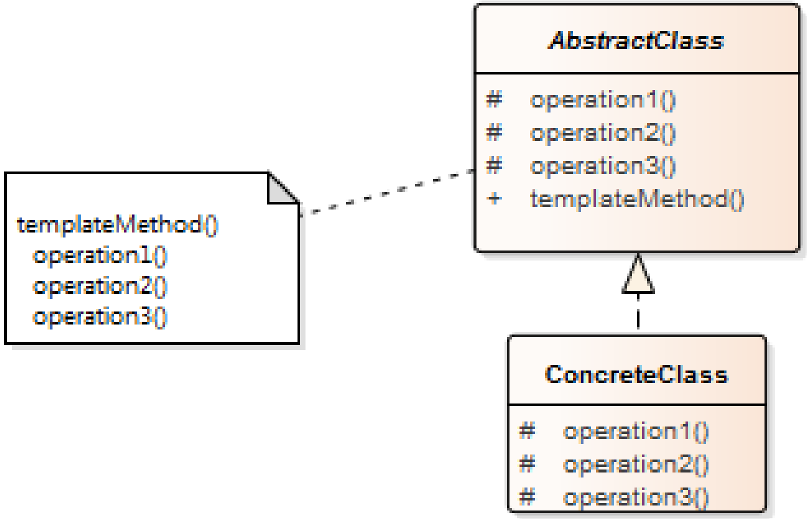
Use

- An algorithm consists of the same sequence of steps.
- The steps may vary between the variations of the algorithms.

Alternative

- Strategy Pattern

Template Method



`AbstractClass`

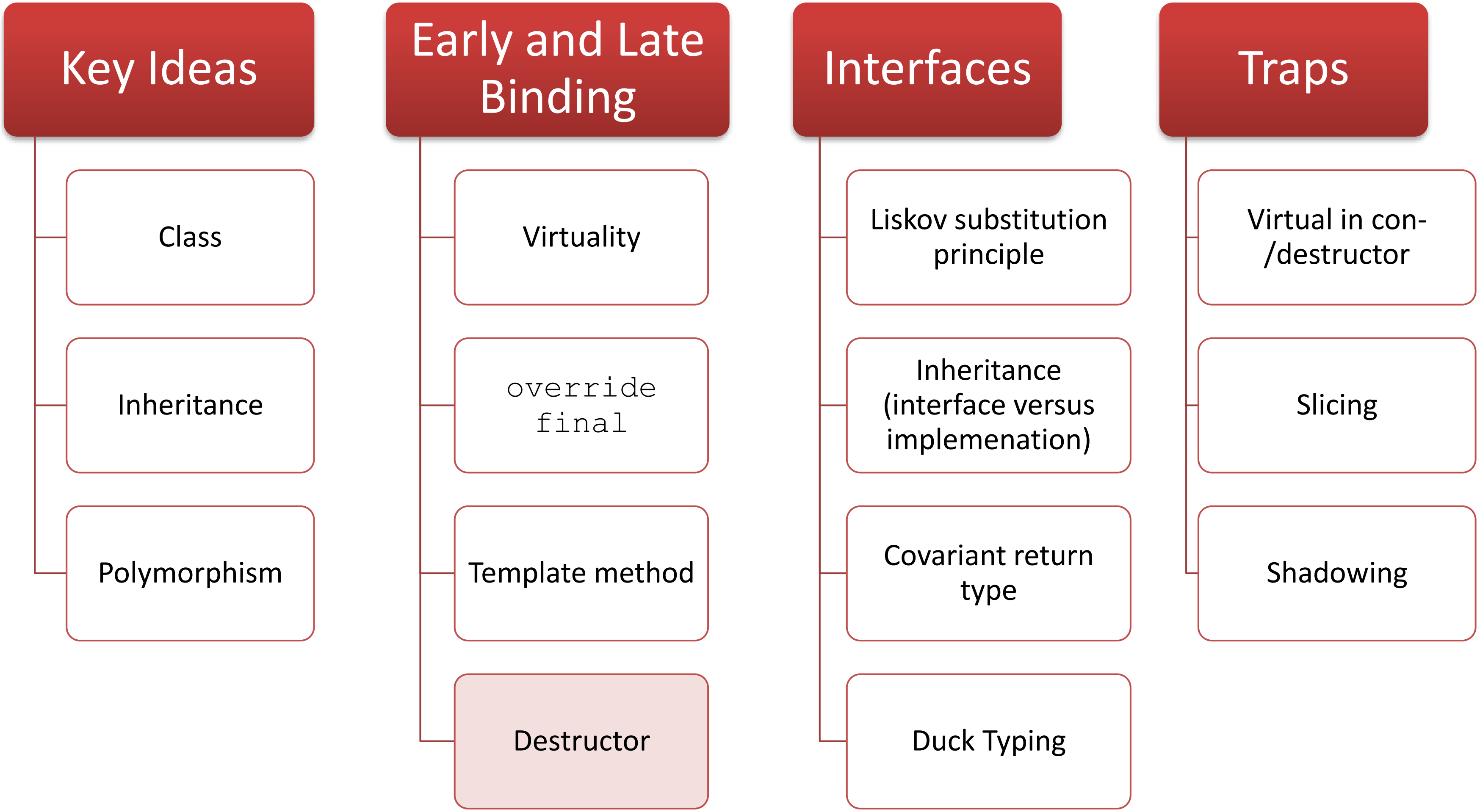
- Defines the structure of the algorithm.
- Defines the steps of the algorithm that can be adapted by subclasses.

`ConcreteClass`

- Overrides the specific steps of the algorithm.

`templateMethod.cpp`

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

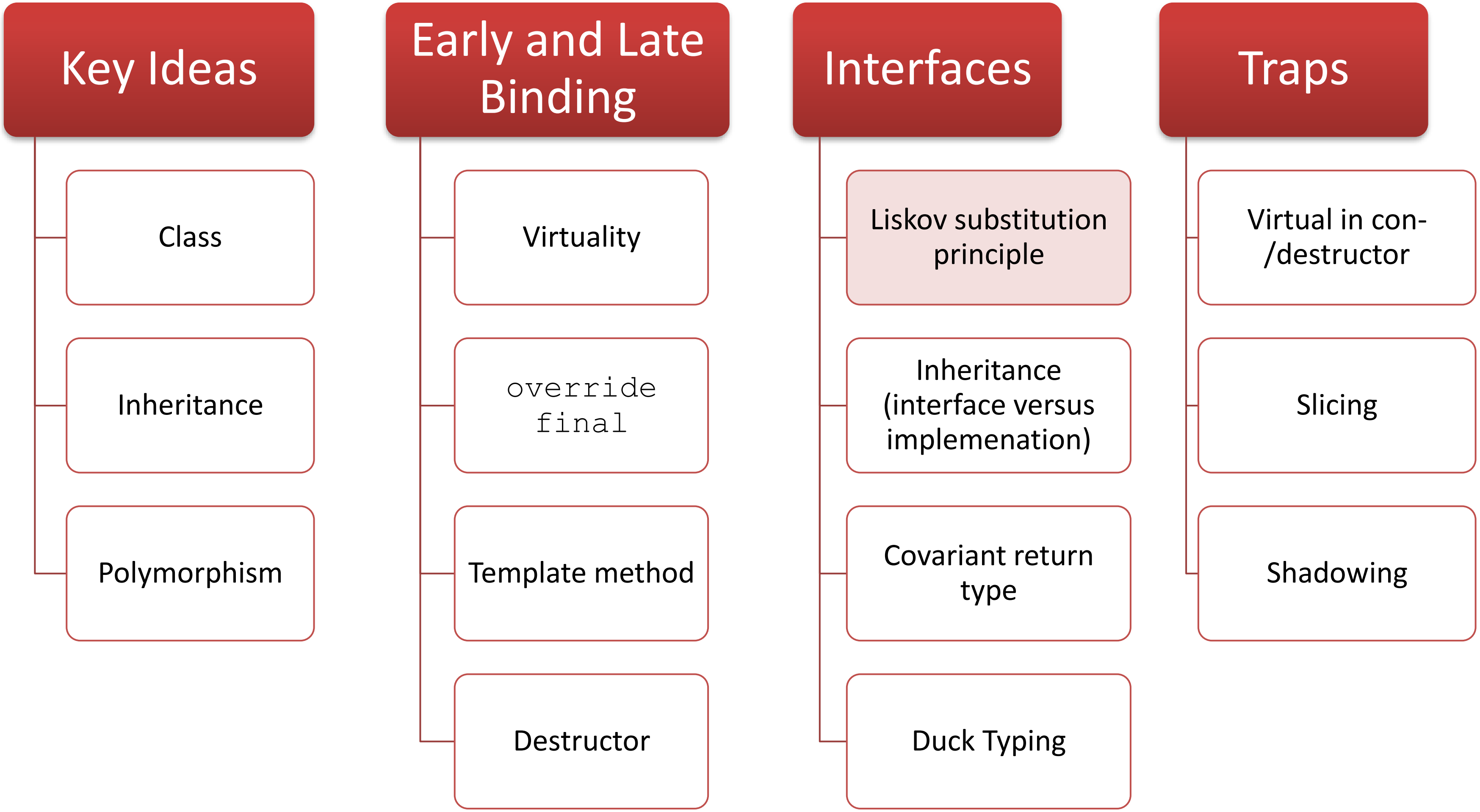
Destructors

Define a destructor if a class needs an explicit action at object destruction.

- A base class destructor should either be `public` and virtual, or `protected` and non-virtual.
  - `public` and virtual:
    - Base class pointers or references **can** destroy instances of derived classes.
  - `protected` and non-virtual:
    - Base class pointers or references **cannot** destroy instances of derived classes.

Destructors should not fail; make them `noexcept`

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

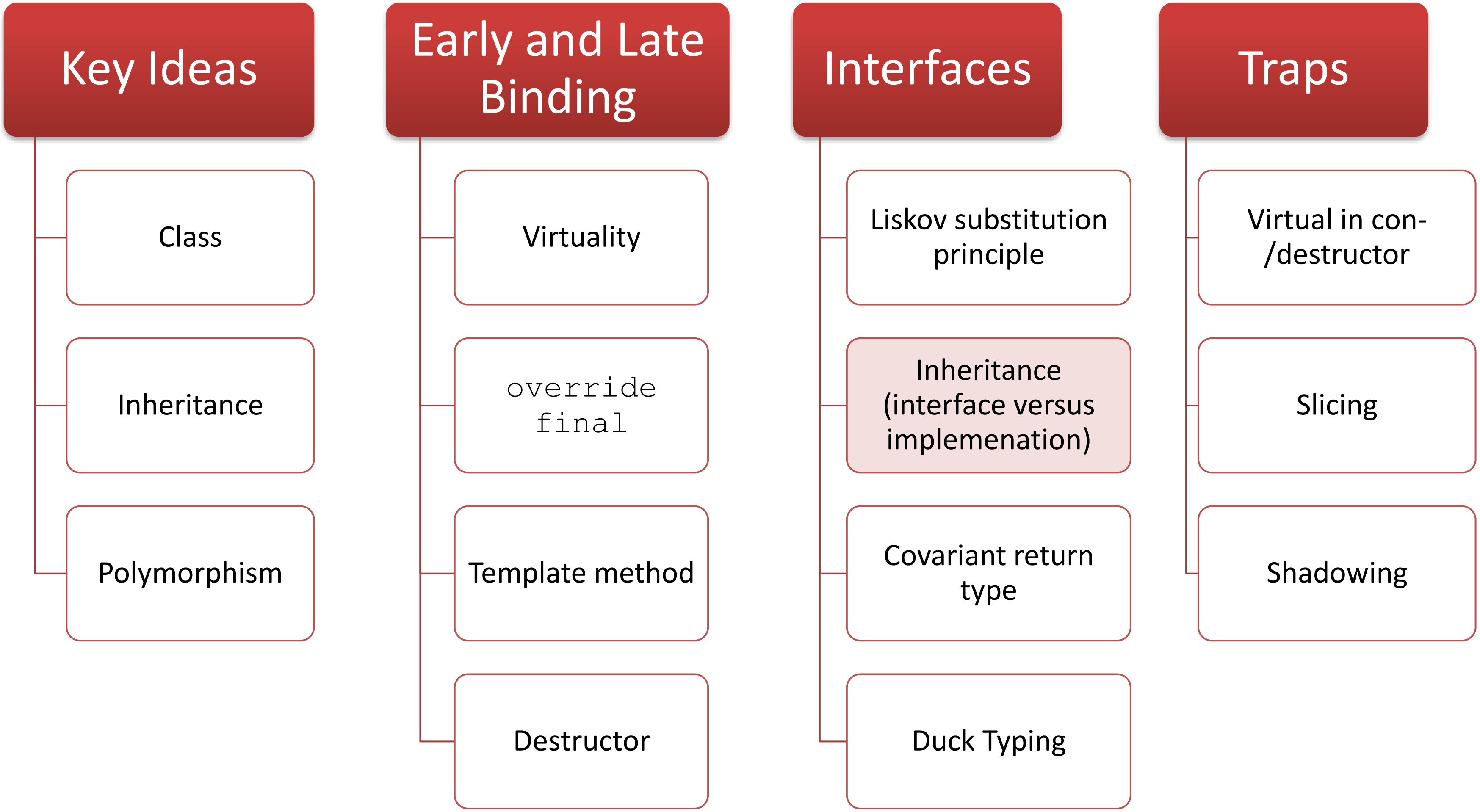
- Virtual in con-/destructor
- Slicing
- Shadowing

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program (L in SO**L**ID).

- Application of separation of interface and implementation in a class hierarchy
- Define the functionality of the interface and use an implementation.

Object-Oriented Programming



Key Ideas
- Class
- Inheritance
- Polymorphism

Early and Late Binding
- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces
- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps
- Virtual in con-/destructor
- Slicing
- Shadowing

Inheritance (Interface/Implementation)

A class hierarchy represents a set of hierarchically organized concepts. Base classes typically act as interfaces.

- **Interface inheritance** uses `public` inheritance. It separates users from implementations to allow derived classes to be added and changed without affecting the users of base classes.

- **Implementation inheritance** often uses `private` inheritance. Typically, the derived class provides its functionality by adapting functionality from base classes.

Implementation Inheritance (Adapter)

Type

- Structural pattern

Purpose

- Translate one interface into another interface

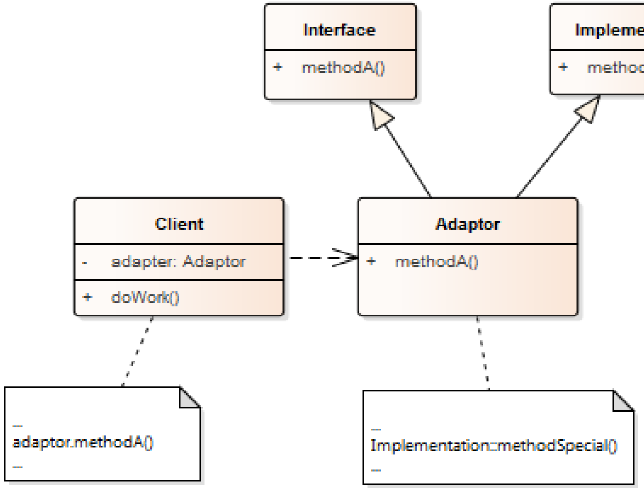
Use

- A class has the incorrect interface.
- Definition of an interface for many similar classes

Alternative

- Composition (The objects holds its adapted object.)

Implementation Inheritance (Adapter)



`Client`

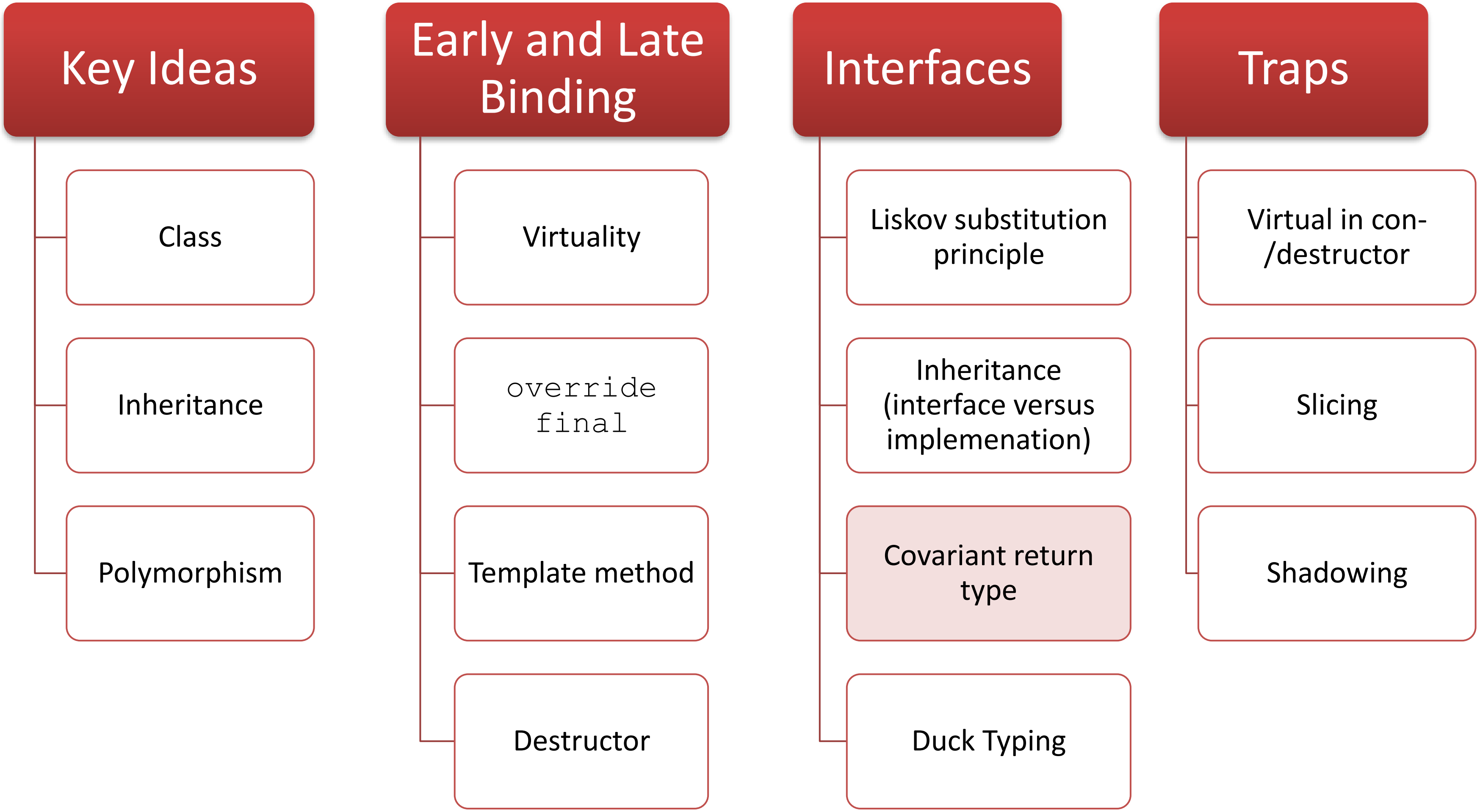
- Uses the `methodA()` of the `Adaptor`

`Adaptor`

- Derives `public` from `Interface` and `private` from `Implementation.`
- Supports the functionality of `methodA()` using multiple inheritance.

[`adapter.cpp`]()

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override`
`final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

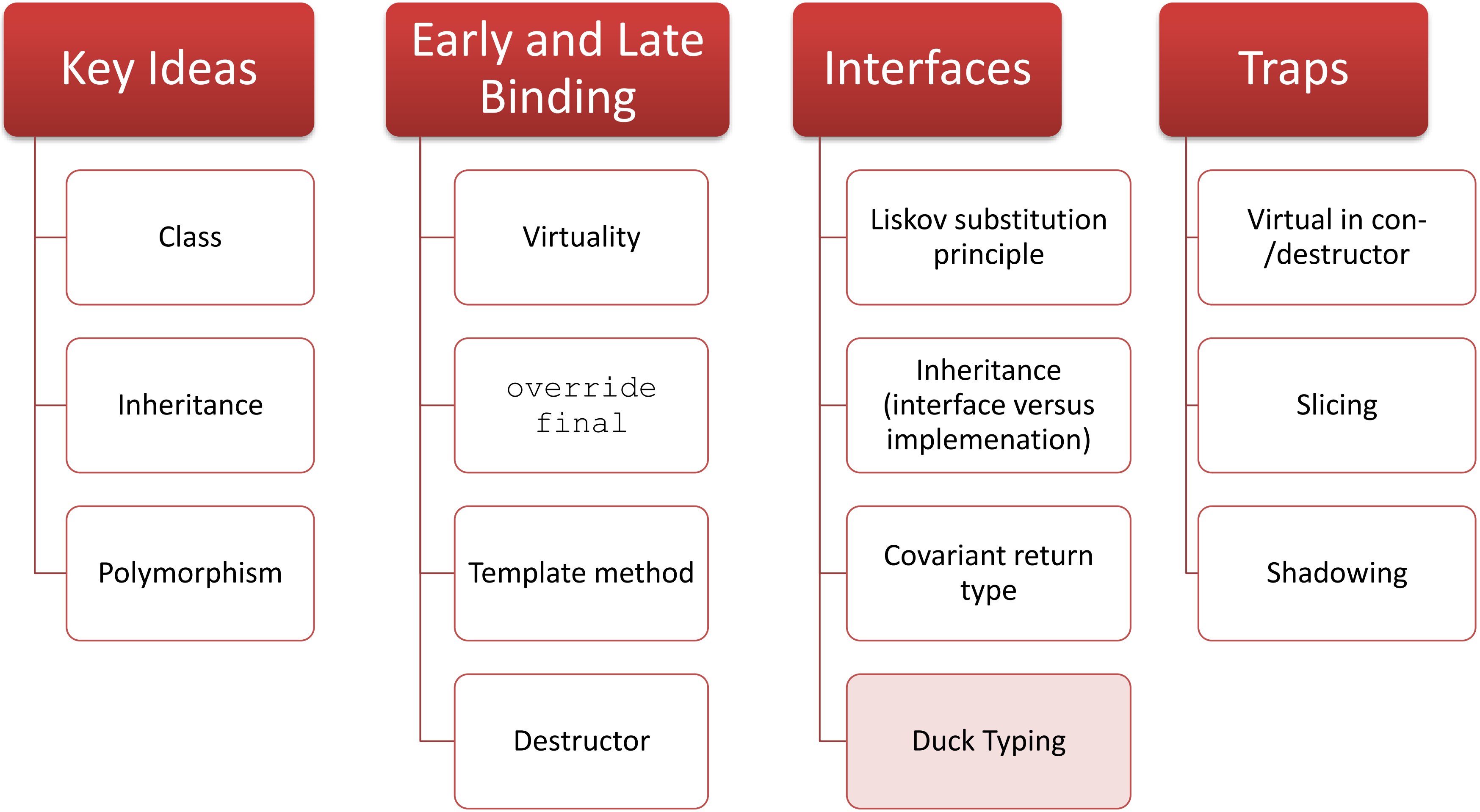
Covariant Return Type

Enables it for an overriding member function to return a subtype of the return type of the overridden member function.

```
class Base {
 public:
    virtual Base* clone() const {
        return new Base(*this);
    }
};

class Derived : public Base {
 public:
    Derived* clone() const override {
        return new Derived(*this);
    }
};
```

Object-Oriented Programming



- Key Ideas
  - Class
  - Inheritance
  - Polymorphism
- Early and Late Binding
  - Virtuality
  - `override` `final`
  - Template method
  - Destructor
- Interfaces
  - Liskov substitution principle
  - Inheritance (interface versus implemenation)
  - Covariant return type
  - Duck Typing
- Traps
  - Virtual in con-/destructor
  - Slicing
  - Shadowing

Duck Typing

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” (James Whitcomb Riley**)**

- Use:
  - Templates
  - Interpreter languages (Python)

Don’t ask for permisson, ask for forgiveness.

Duck Typing

Let it crash and deal with the error.

- Failed template instantiation of SFINAE
- Exception handling

```
try:
    swim(duck)
except TypeError:
    print(“This was not a duck!!!”)
```

Distinguish between:

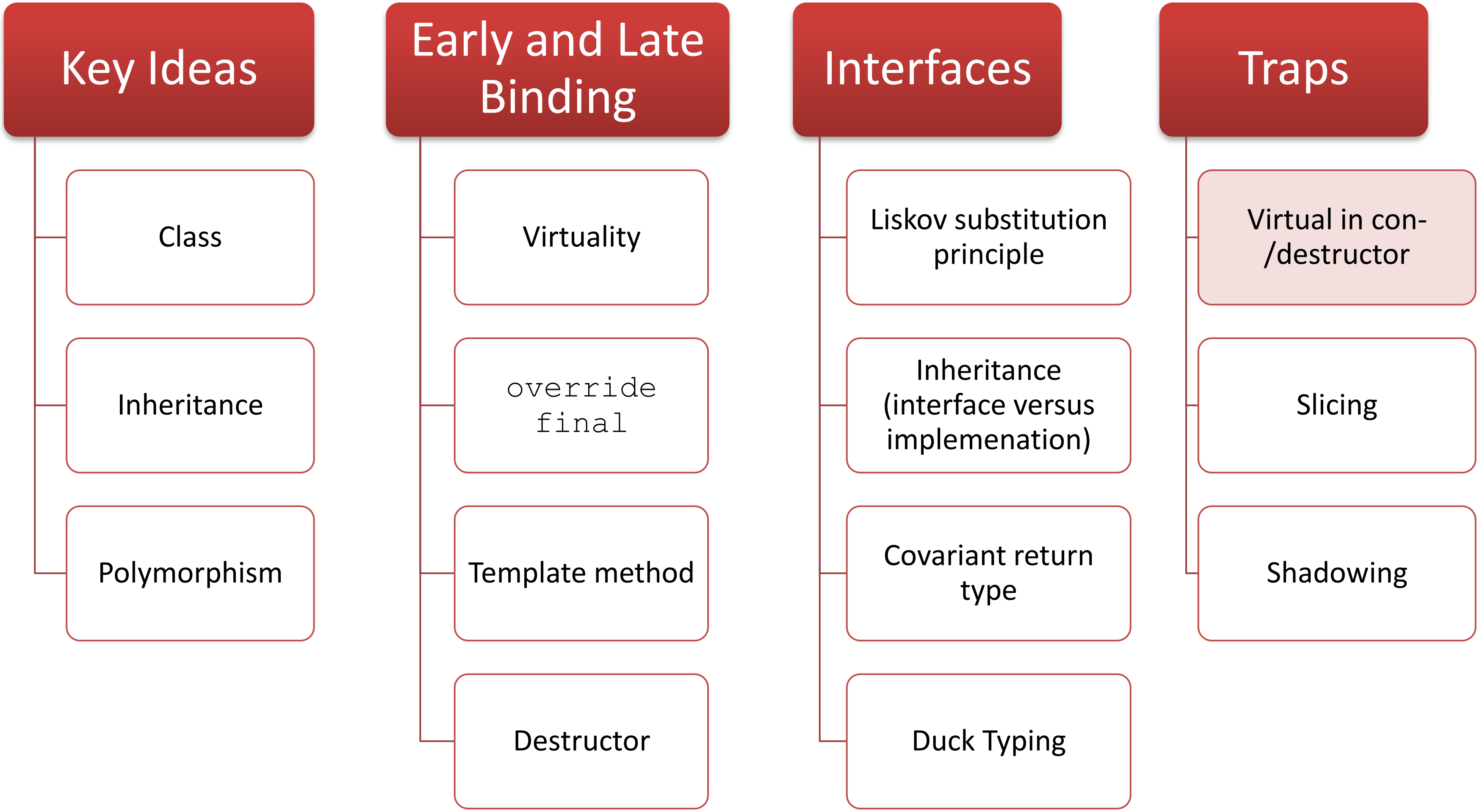
- Interface design: contract driven design

```
void swim(const Duck* duck)
```

- Duck typing: behavioral driven design

```
template <typename Duck>
void swim(Duck duck);
```

Object-Oriented Programming



Key Ideas
- Class
- Inheritance
- Polymorphism

Early and Late Binding
- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces
- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps
- Virtual in con-/destructor
- Slicing
- Shadowing

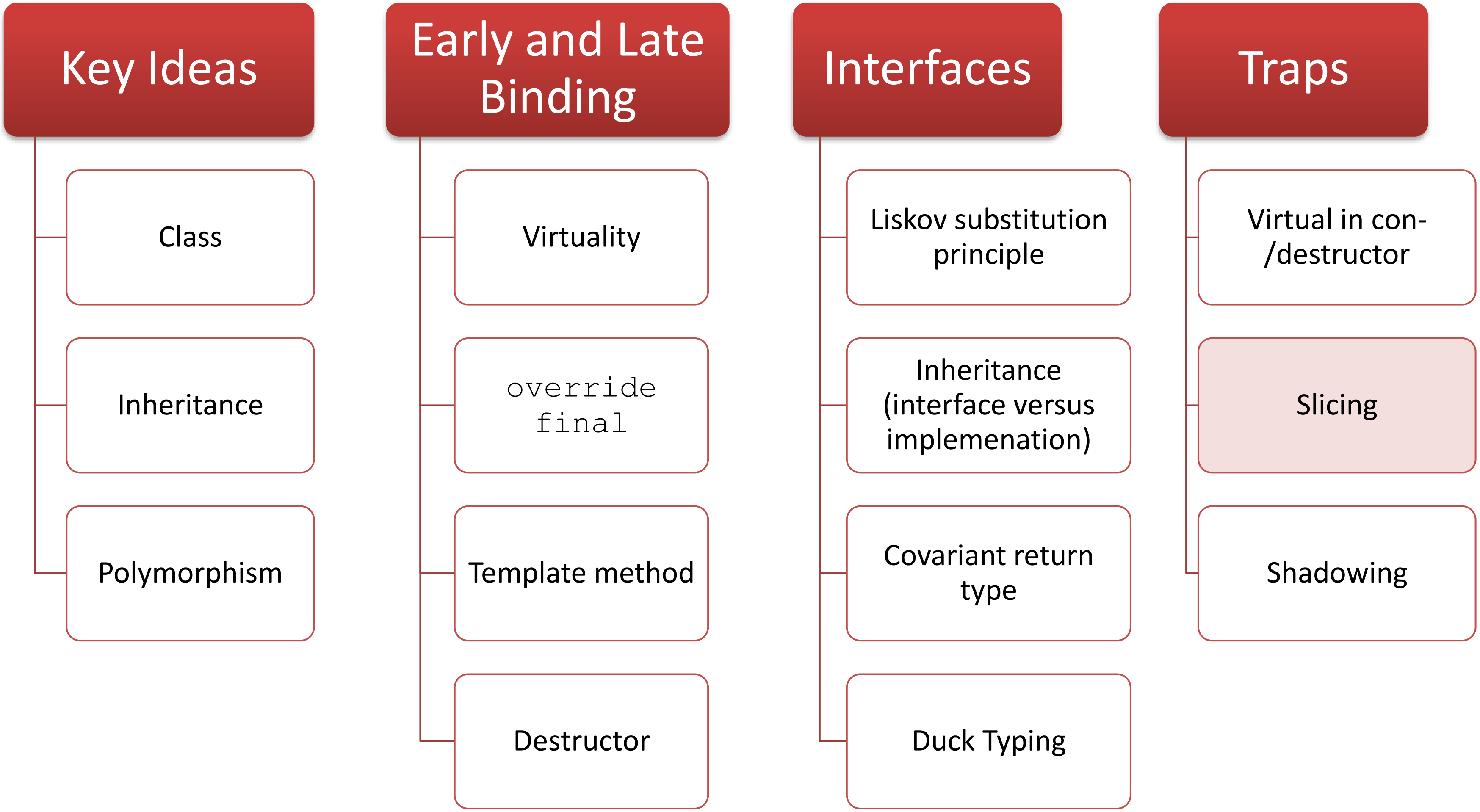
Virtual in Constructor/Destructor

Don’t call virtual functions in constructors and destructors.

- Pure virtual: → undefined behavior
- Virtual: → virtual call mechanism is disabled

virtualCall.cpp

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Slicing

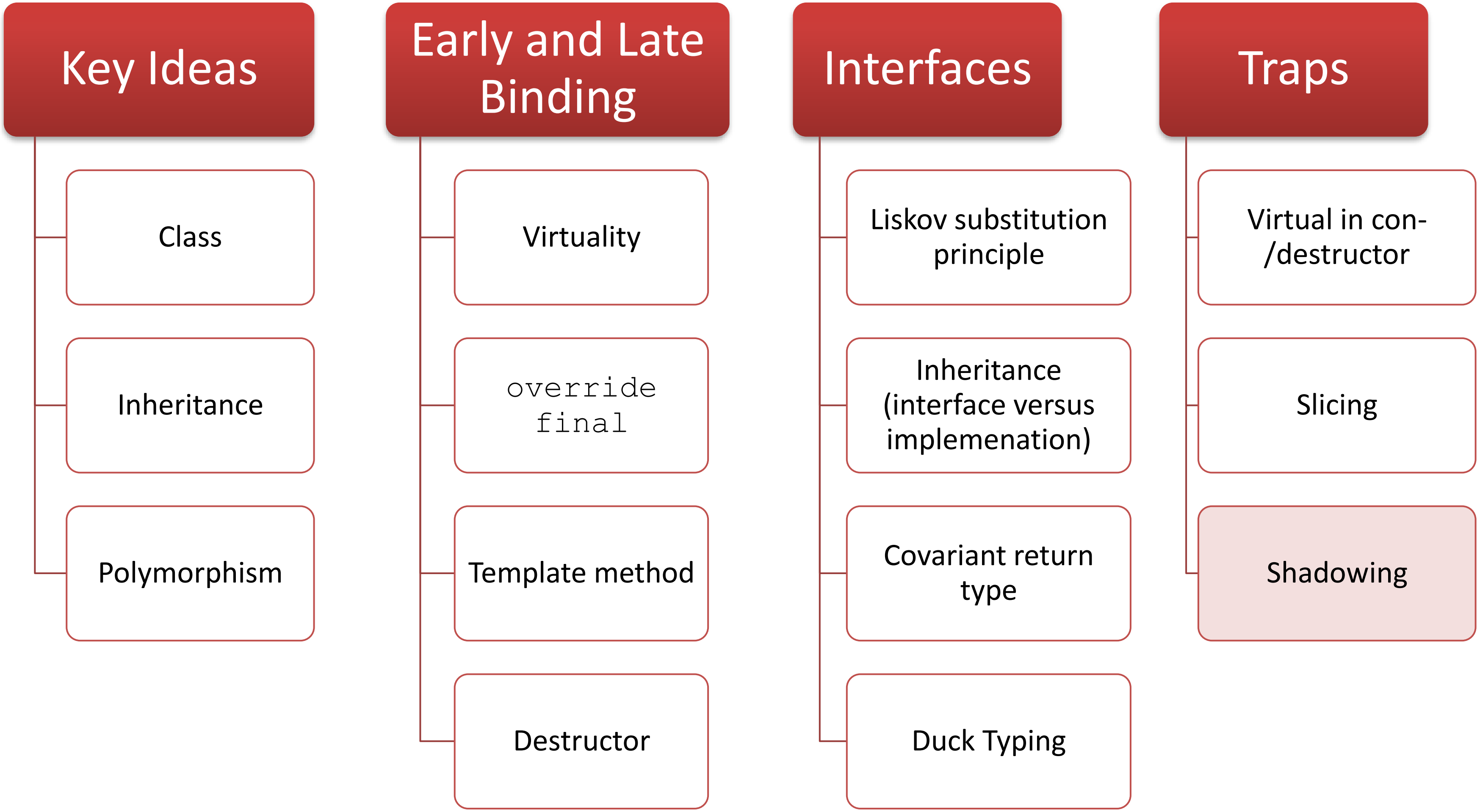
When a derived class is copied to a base class, the derived class becomes a base class.

- For making deep copies of polymorphic classes prefer a virtual member function `clone` instead of a copy constructor or copy assignment operator.

Factory method (virtual constructor)

`slice.cpp`
`clone.cpp`

Object-Oriented Programming



Key Ideas

- Class
- Inheritance
- Polymorphism

Early and Late Binding

- Virtuality
- `override` `final`
- Template method
- Destructor

Interfaces

- Liskov substitution principle
- Inheritance (interface versus implemenation)
- Covariant return type
- Duck Typing

Traps

- Virtual in con-/destructor
- Slicing
- Shadowing

Shadowing

A member function of a derived class shadows the member functions of its base class with the same name.

```
struct Base {
    void func(double d) { std::cout << "f(double) \n"; }
};

struct Derived: public Base {
     void func(int i) { std::cout << "f(int) \n"; }
};

Derived der;
der.func(2020.5);  // f.double()
```

➡ `Derived::func` shadows `Base::func`

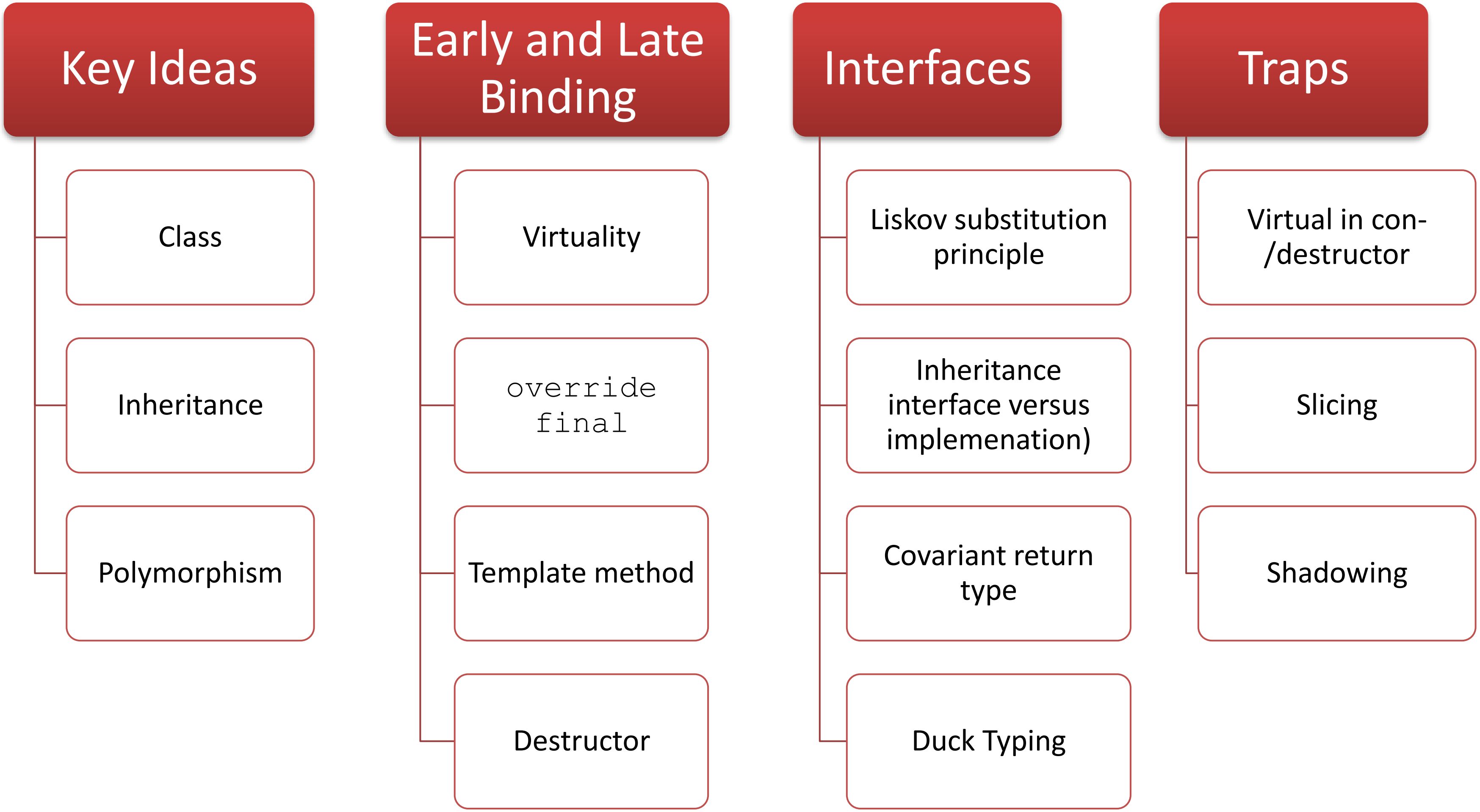
Shadowing

Create an overload set for a derived class and its base classes with `using`.

```
struct Derived: public Base {
    void func(int i) { std::cout << "f(int) \n"; }
    using Base::func; // exposes func(double)
};
```

shadowing.cpp

Object-Oriented Programming



- Key Ideas
  - Class
  - Inheritance
  - Polymorphism
- Early and Late Binding
  - Virtuality
  - `override` `final`
  - Template method
  - Destructor
- Interfaces
  - Liskov substitution principle
  - Inheritance interface versus implemenation)
  - Covariant return type
  - Duck Typing
- Traps
  - Virtual in con-/destructor
  - Slicing
  - Shadowing

#include
nt main(){
std::cout <<
std::vector<
std::iota(myVe
std::cout <<
for (
std::co
std::function< bool(int)> myBindPr
nd(std::logical
myVec.erase(std::remove_if(myVec.
.end(), myBind
std::cout << "myVec: ";
for (auto i: myVec) std::cout <<
std::cout << "\n\n";
std::vector<int> myVec2(20);
std::iota(myVec2.begin(), myVec2.e
std::cout << "myVec2: ";
for (auto i: myVec2) std::cout

www.ModernesCpp.com

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.net