DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

[CS 225] Advanced C/C++

Lecture 5: “Virtualization” of constructors

Sławomir “Swavek” Włodkowski
(Summer 2020)

Agenda

- Virtual constructors
- Design patterns
  - Factory method design pattern
  - Prototype design pattern
- Covariant return types

Virtual constructors

Objective:

- Polymorphism is about using an identifier or symbol to represent different operations depending on the context.
- We want to construct objects of different types based on types or counts of parameters; that is: polymorphically.

Virtual constructors

Problem:

- C++ is a statically-typed language and each constructor call requires a name of a specific class to instantiate. Constructors do not support ***static polymorphism*** across multiple classes (only overloading within a single class).

- C++ does not allow for `virtual` constructors, because it takes a constructor to initialize *vptr* of an object first. Constructors do not support ***dynamic polymorphism***.

Virtual constructors

Observation:

- We are not the first ones wanting to solve this problem...

Design patterns

A design pattern is a general, language-independent, reusable solution to a common problem in software design. It is a universally recognized best practice template for organizing class and object interactions.

Design patterns

- The concept originating from architecture [1966]
- Popularized in OOP by a book by “Gang of Four”:
  1. E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994.

“The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” [1]

Design patterns

Basic elements of a design pattern:

- A *pattern name* identifying the pattern.
- A *problem* describing a context in which it can be applied.
- A *solution* describing a design, relationships, responsibilities and collaborations of classes and object.
- The *consequences* describing results and trade-offs.

Design patterns

Types of design patterns:

- **Creational**
  For creating objects (i.e. *singleton*)
- **Behavioral**
  For facilitating objects' communication and coupling (i.e. *iterator*)
- **Structural**
  For shaping objects' functionality using inheritance and composition (i.e. *proxy*)

Design patterns

Our problem of **virtualization of constructors** can be addressed by some **creational** design patterns…

Factory method design pattern

Pattern names

Factory method
Virtual constructor

Problem

Instantiating specific derived classes through an abstract class, which we may not even be able to instantiate (i.e. an interface class, an abstract class with some pure virtual member functions, etc.).

Factory method design pattern

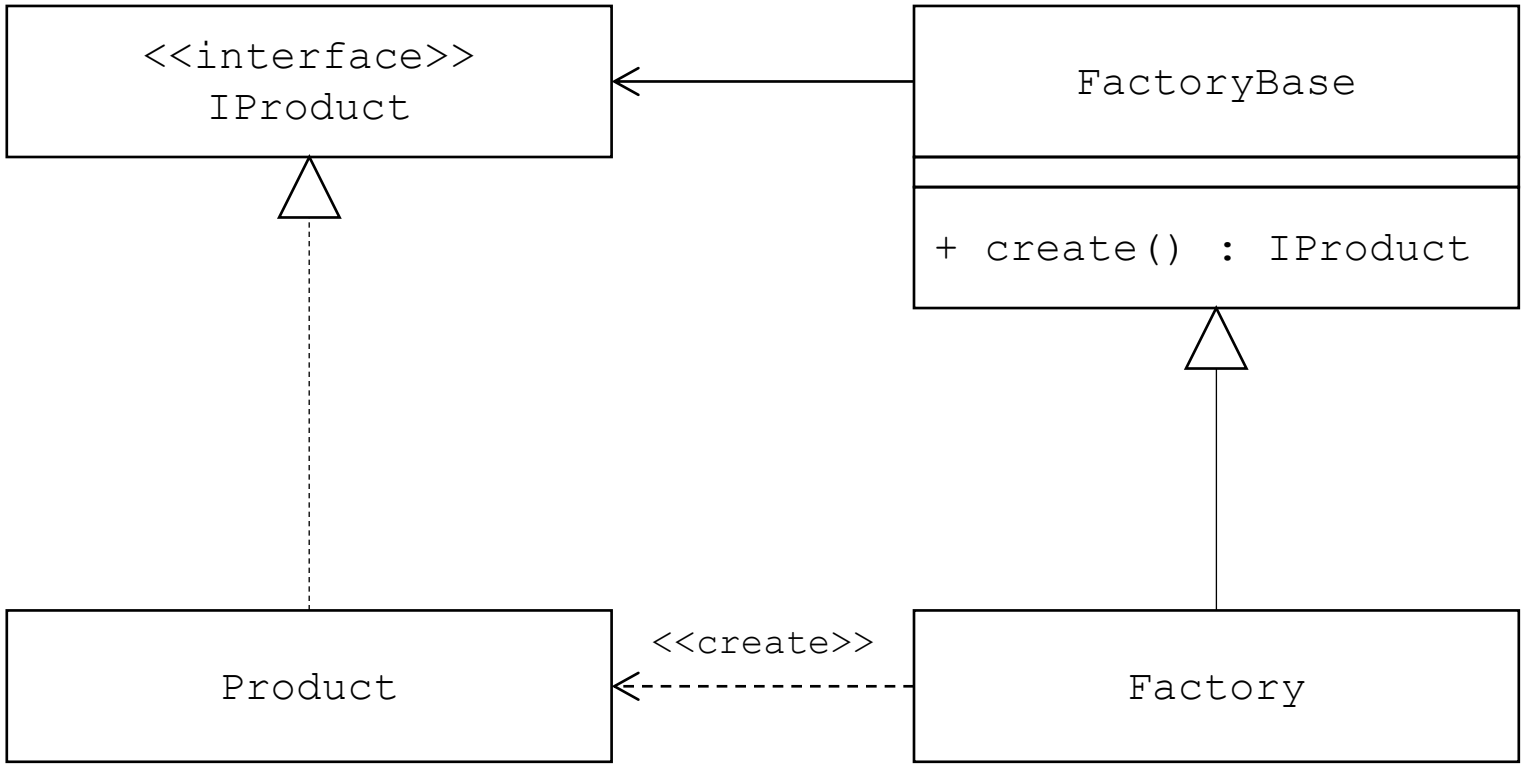
Solution

Delegate instantiation functionality to a separate class ("*creator*", "*factory*") that exposes a member function that is either:

1. Virtual and can be overridden to instantiate derived objects using the same parameters.
2. Non-virtual and can create derived objects using different parameters.

Factory method design pattern

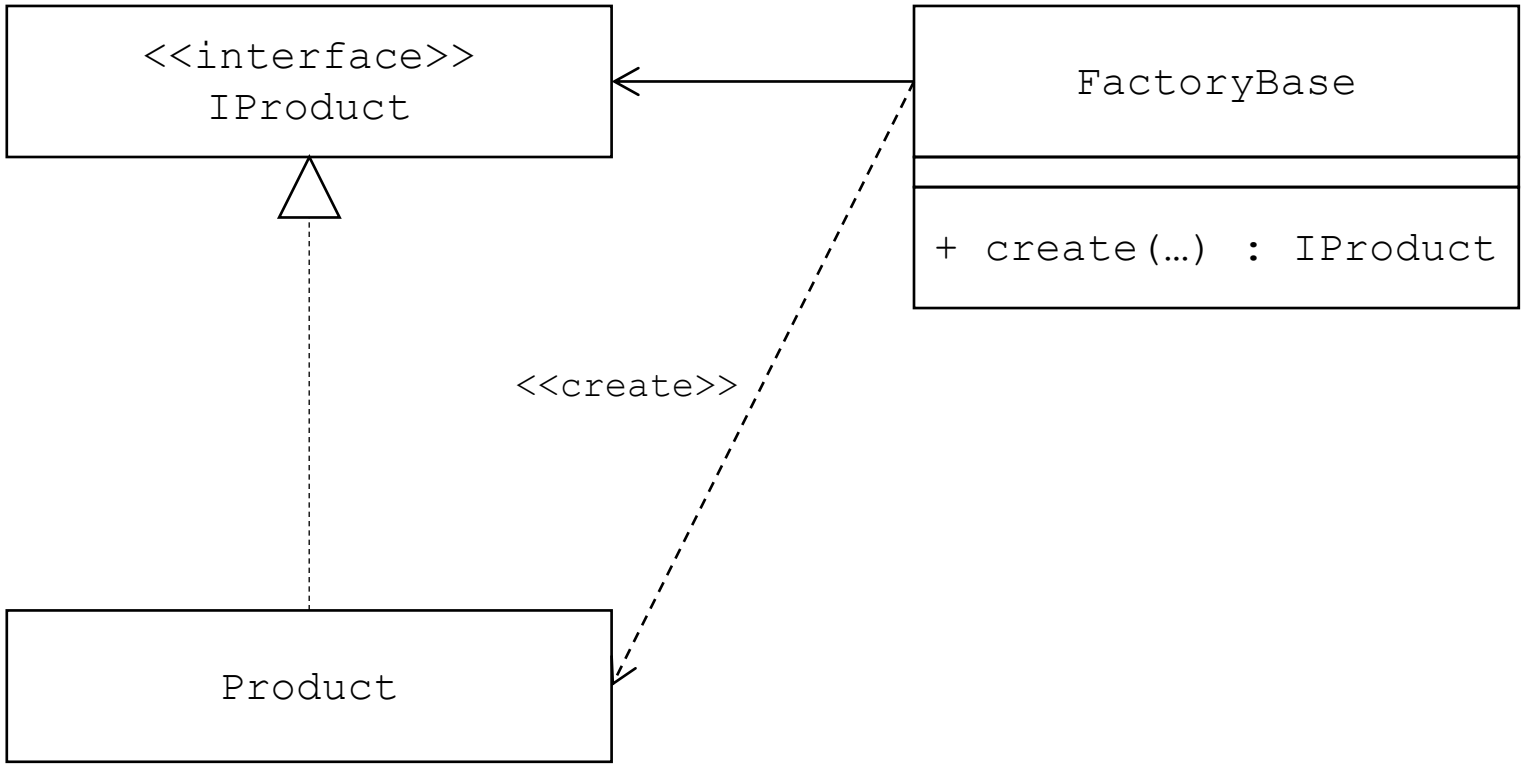
1



UML diagram for a factory method design pattern with a factory class that exposes a virtual member function that can be overridden to instantiate derived objects using the same parameters.

Factory method design pattern

2



UML diagram for a factory design pattern with a factory class that exposes a non-virtual member function that can create derived objects using different parameters.

Factory method design pattern

Consequences

- Requires

  (1) deriving concrete factory classes from a base factory.

  (2) or implementing one function that knows about all derived classes we instantiate.
- Provides a single place for hooking into creation of any subclasses.
- Connects parallel class hierarchies (when a derived class is associated with other derived classes).

Prototype design pattern

Pattern name

Prototype

Problem

Specify the kinds of object to create by using a *prototypical instance* and create objects by copying this prototype.

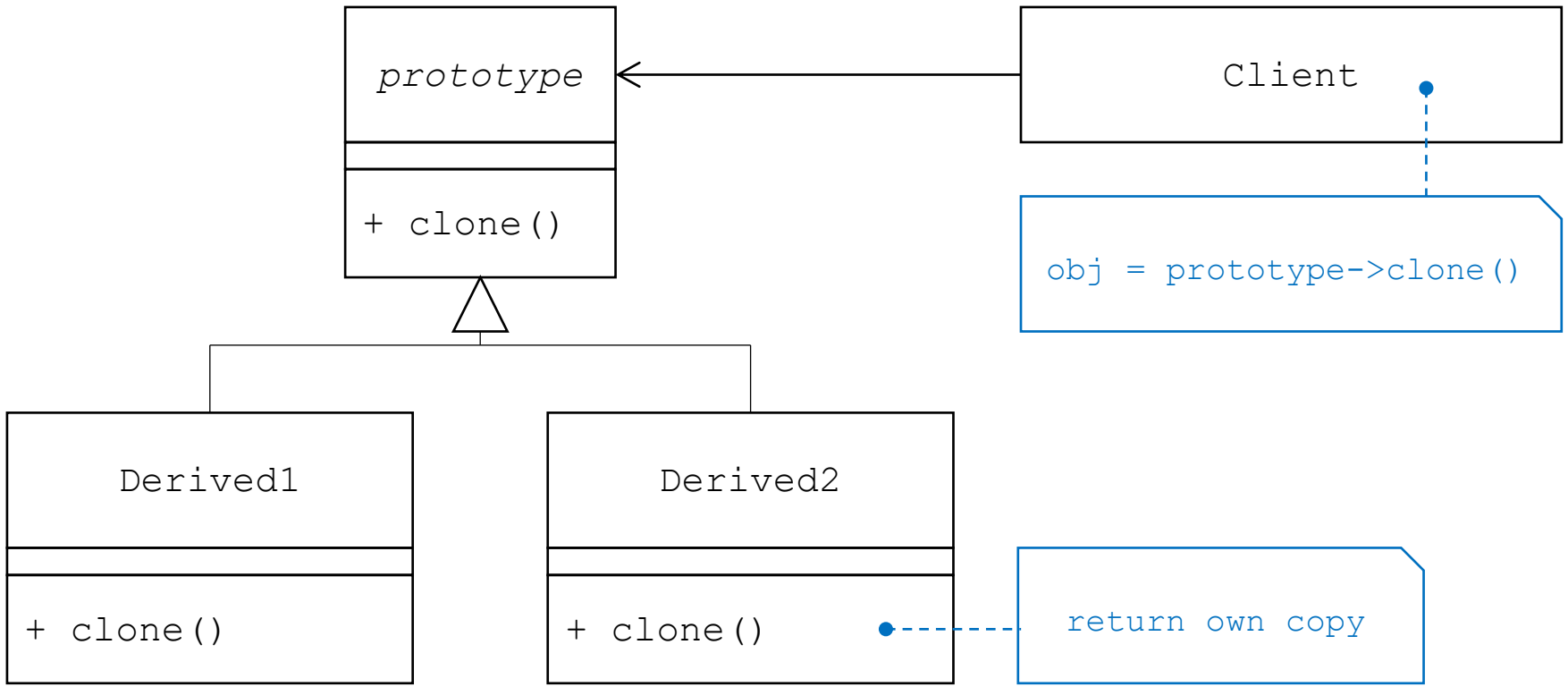
Prototype design pattern

Solution

Construct objects by polymorphically copying existing instances of “prototypes”. Each object must support on its interface a dedicated member function (“*clone*”, “*copy*”) that performs a deep copy. Ask objects to clone themselves as they know their own type.

We cannot use copy c-tors directly because we do not know a real type of a copied object (only its base class pointer) and the result must be of the same real type.

Prototype design pattern



Prototype design pattern

Consequences

- Allows for adding/removing prototypes in the run-time.
- Allows for dynamic configuration of application.
- Allows for specifying new object by varying their states, without adding more classes.
- Allows for reuse of complex, user-defined objects.
- Reduces inheritance, thus flattens the class hierarchy.
- Makes it difficult to avoid duplicated prototypes.

Covariant return types

The prototype design pattern requires implementing a `virtual clone()` member function. As the function is introduced in a base class, it should return a base class pointer.

- This is convenient when we implement code to work on any prototype via its base type pointer.
- When we work on a derived object directly, we know its exact type, so it would be better to return a derived class pointer.

Covariant return types

C++ supports a feature called **covariant return types**.

An override function in a derived class can return:

- A pointer to a base type as defined in a base class, or
- A pointer to a derived type of that base type.
  - When such a function is called via a base type pointer, the result is cast automatically to a base type.
  - When such a function is called via a derived type pointer, the result remains a derived type pointer.

Covariant return types

```
struct Base
{
  virtual Base* clone() const
  {
    // some copy logic, for example:
    return new Base{*this};
  };
  virtual ~Base() = default;
};
```

```
struct Derived : Base
{
  virtual Derived* clone() const override
  {
    // some copy logic, for example:
    return new Derived{*this};
  };
};
```

```
Derived d;

Derived* obj1 = d.clone();
delete obj1;

Base* pb = &d;
Base* obj2 = pb->clone();
delete obj2;
```