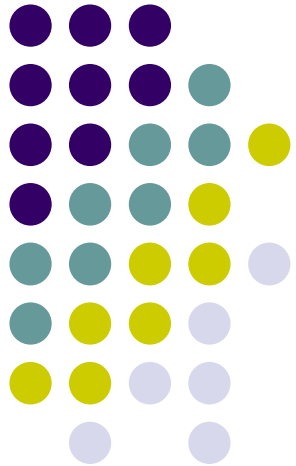
C++

April 25, 2022

Mike Spertus

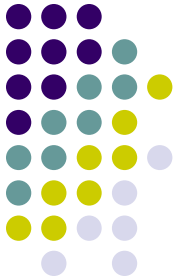
spertus@uchicago.edu



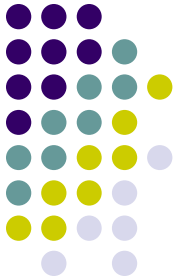
MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

A problem with our tuple implementation

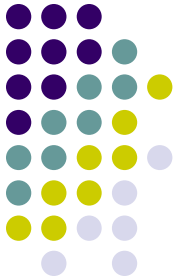


- h/t Bihao
- https://godbolt.org/z/TW9fq5j4T



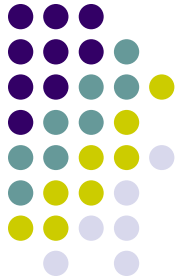
DESIGN PATTERNS

Design Patterns



- The book that introduced Design Patterns was the Gang of Four's Design Patterns: Elements of Reusable Object Oriented Software
  - ACM SIGPLAN Programming Language Achievement Award
  - One of only three software engineering books on Wikipedia's list of Important Publications in Computer Science
- According to Wikipedia, a Design Pattern is
  - In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
  - A design pattern is not a finished design that can be transformed directly into source or machine code
    - Wanna bet? It is a description or template for how to solve a problem that can be used in many different situations.
    - Hmmm…, "Template"
  - Patterns are formalized best practices that the programmer must implement in the application

Andrei Alexandrescu
Modern C++ Design

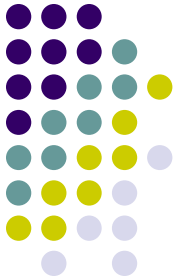


- Major theme: Implementing Design Patterns as Templates
- The Gang of Four's John Vlissides writes in the forward
  - This book documents a convergence of programming techniques—generic programming, template metaprogramming, object-oriented programming, and design patterns—that are well understood in isolation but whose synergies are only beginning to be appreciated. These synergies have opened up whole new vistas for C++, not just for programming but for software design itself, with profound implications for software analysis and architecture as well.
  - Wouldn't it be great if we could realize the theoretical benefits of code generation—quicker, easier development, reduced redundancy, fewer bugs—without the drawbacks? That's what Andrei's approach promises. Generic components implement good designs in easy-to-use, mixable-and-matchable templates. They do pretty much what code generators do: produce boilerplate code for compiler consumption. The difference is that they do it within C++, not apart from it. The result is seamless integration with application code.

Example: AbstractFactory and ConcreteFactory

- These are two of the most popular Design Patterns
- In my experience, most large scale programs have something like the following

AbstractFactory pattern



```
class WidgetFactory
{
public:
  virtual
    unique_ptr<Window> CreateWindow() = 0;
  virtual
    unique_ptr<Button> CreateButton() = 0;
  virtual
    unique_ptr<ScrollBar> CreateScrollBar() = 0;
};
```

ConcreteFactory pattern

- By using a factory for your particular UI, you don't need to worry about creating incompatible objects, like accidentally creating a Windows widget in an Android app
- ```
  class MSWindowsWidgetFactory
    : public WidgetFactory {
    unique_ptr<Window> CreateWindow() override
    { return make_unique<MSWindow>(); }
    /* ... */
  };
  class AndroidWidgetFactory
    : public WidgetFactory { ... };
  ```
  ```

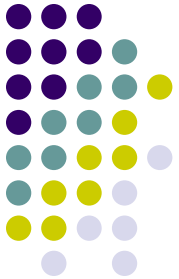
Putting them together

- Easy-to-use and hard to misuse
-
```
unique_ptr<WidgetFactory> widgets
  = make_unique<MSWindowsWidgetFactory>();
// All code below here can only use
// generic widget functionality so
// if we switch to an
// AndroidWidgetFactory later, it will
// still work
/* ... */
auto myButton = widgets.CreateButton();
```

So what's the problem

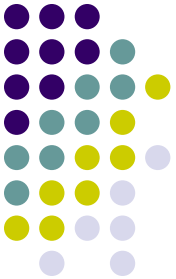
- Factories are easy to use but hard to write
- There are actually dozens of different kinds of widget
- Manually implementing all of the different factorys is
  - **tedious**: programmers will avoid using factories even when beneficial
  - **error-prone and brittle**: vulnerable to cut-and-paste errors, getting out of sync when new widget types appear, etc.
- Can we use a template to generate for us?

Another problem: Calling factory methods uniformly?



```
// Legal but hard to use and maintain
class RedWidgetFactory : public WidgetFactory {
public:
  RedWidgetFactory(shared_ptr<WidgetFactory> f) : wrapped(f) {}
  virtual unique_ptr<Window> CreateWindow() {
    auto pW = wrapped->CreateWindow();
    pW->SetColor(RED);
    return pW;
  }
  virtual unique_ptr<Button> CreateButton() {
    auto pB = wrapped->CreateButton();
    pB->SetColor(RED);
    return pB;
  }
  virtual unique_ptr<ScrollBar> CreateScrollBar() {
    auto pW = wrapped->CreateScrollBar();
    pW->SetColor(RED);
    return pW;
  }
private:
  shared_ptr<WidgetFactory> wrapped;
};
Window *redWindow = RedWidgetFactory(wf).CreateWindow();
```

Couldn't we use a template to say `MakeRedWidget<Button>(f)`?



```
// The following almost works
template<class T>
unique_ptr<T>
MakeRedWidget(WidgetFactory &f)
{
  auto pW = f.CreateT(); // Illegal!
  pW->SetColor(Red);
  return pW;
};
```

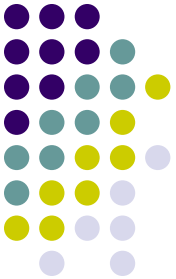
Suppose we change the factory to have template methods?

- The following interface is nice, but not legal

```
class WidgetFactory {
 public:
   // Oops! Virtual template
   // methods not legal!
   template<class T>
   virtual unique_ptr<T> Create() = 0;
 };
```

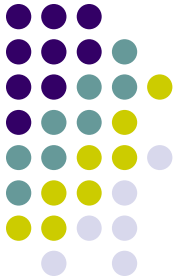
- This is why we learned how to simulate template virtuals last week
- Let's do a quick review

Now it will work



```
// The following almost works
template<class T>
unique_ptr<T>
MakeRedWidget(WidgetFactory &f)
{
   auto pW = f.Create<T>();
   pW->SetColor(Red);
   return pW;
};
```

Our goal: Automate with factory templates



- Easy to create

```
using WidgetFactory
  = AbstractFactory<Window, Button, Scrollbar>;

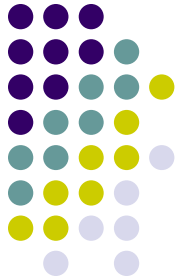
using QtFactory
  = ConcreteFactory
    <WidgetFactory, QtWindow, QtButton, QtScrollbar>;

using GTKFactory
  = ConcreteFactory
    <WidgetFactory, GTKWindow, GTKButton, GTKScrollbar>;
```

- Easy to use

```
unique_ptr<WidgetFactory> wf
  = make_unique<QtFactory>(); // Easy to use
auto b = wf->Create<Button>();
```

Simulating template virtuals



- One of the big problems is that factories use virtual methods so the abstract factory methods can be overridden, but method templates can’t be virtual
- Key idea (Alexandrescu): Have the method template call an ordinary virtual method selected by overload resolution

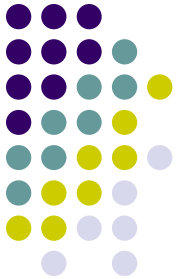
First swing

- A first attempt might be something like:

```
template<class T>
T *create() {
  return createHelper(T());
}
virtual
  unique_ptr<Window> createHelper(Window&&)=0;
virtual
  unique_ptr<Button> createHelper(Button&&)=0;
virtual
  unique_ptr<Scrollbar> createHelper(Scrollbar&&) = 0;
};
```

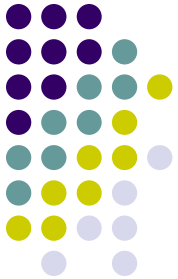
- The idea is that overload resolution selects the right `createHelper()`
- This actually almost works, except `T` will usually be abstract and can't be created. Even if it can, it might not have a default constructor

TT



- What we need is a “tag type” that can always be created
  - Not abstract
  - Default constructible
  - Cheap to construct
- `template<class T>`
  `struct TT {`
  `};`
- Fits the bill. If I construct with `TT<Window>()`, it’s legal because it is not abstract and is default constructible, and it’s cheap because it’s just an empty class

A better attempt



- ```
  template<class T>
  T *create() {
    return doCreate(TT<T>());
  }

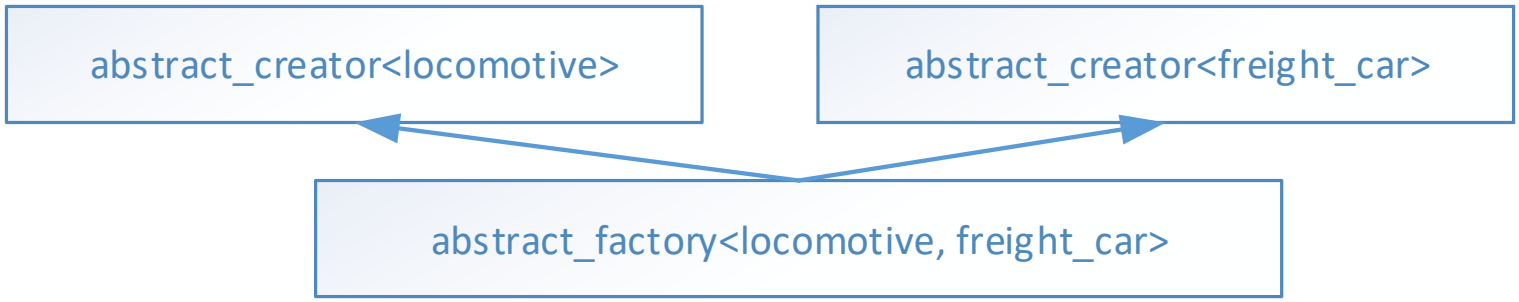
  virtual unique_ptr<Window>
    doCreate(TT<Window> &&)=0;
  virtual unique_ptr<Scrollbar>
    doCreate(TT<Scrollbar> &&)=0;
  virtual unique_ptr<Button>
    doCreate(TT<Button> &&)=0;
  };
  ```
  This works because `TT<T>` is guaranteed to be default constructible
- Now all we need is to do is automate the generation of the `doCreate` methods, so we don't need to manually enter them all like above
  ```

The abstract creator

- Build up from a single `doCreate`
- Since all of the `doCreate` methods have the same name and method hiding takes place, we need to be a little careful
- ```
  template<typename T>
  struct abstract_creator{
    virtual unique_ptr<T>
      doCreate(TT<T>&&) = 0;
  };
  ```
- Our real abstract factory will get all the methods it needs by inheriting from `abstract_creator<Button>`, …
  ```

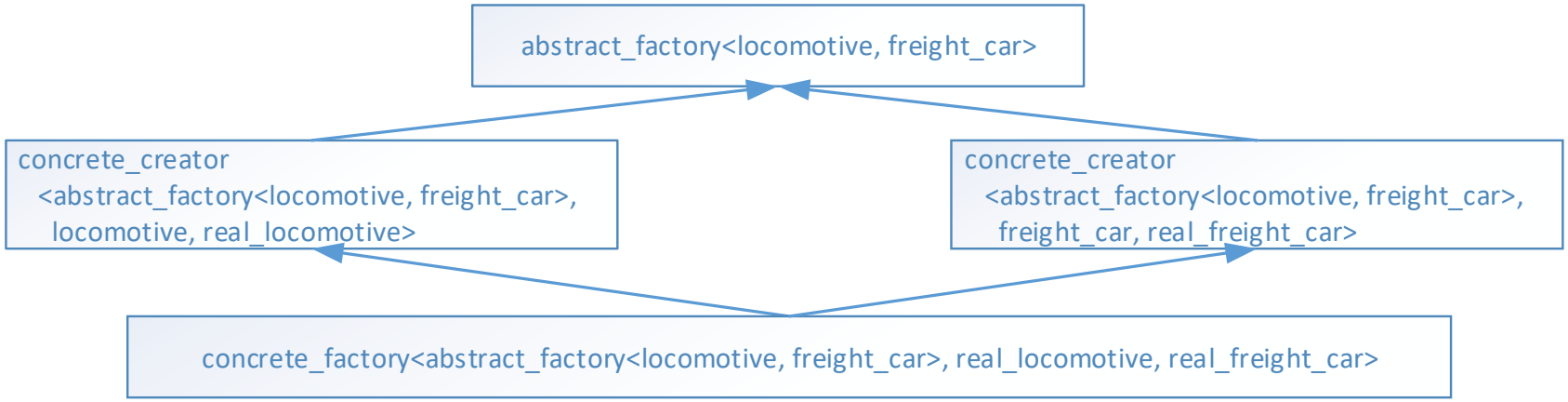
Abstract factory (factory.h)

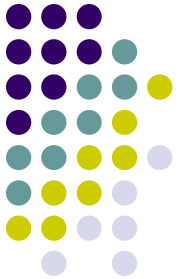
- By using a variadic expansion of the base class, we can inherit from all of the abstract creators at once



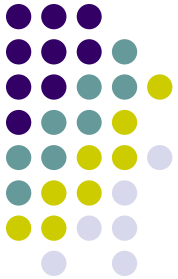
Concrete factory (factory.h)

- By using a variadic expansion of the base class, we can inherit from all of the abstract creators at once





LET’S SEE THE CODE



C++ MEMORY MODEL (HANS BOEHM SLIDES)

HW 14-1

- This exercise is to practice using factories
- Use the classes in factory.h to create
  - an abstract factory `train_factory` for building train cars like `Locomotive`, `FreightCar` and `Caboose`
    - The car classes don't need to have any particularly train-like functionality, so don't worry too much about how many methods you define (if you give them any methods at all).
  - concrete factories `model_train_factory` and `real_train_factory`
- Use these factories to create cars for a model train
  - You can model your solution off of factory.cpp in Canvas

Advanced Factories

- The following problems can selected from for practice designing advanced template classes using the techniques we have discussed
- You are not expected to be comfortable writing code like this, so definitely not required
- …but I guarantee you will learn a lot trying them if you try (and I will give partial credit)
- **Note:** Do 1 for full credit, more for extra credit

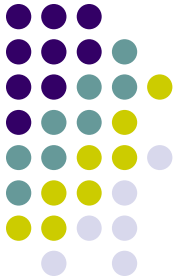
HW 14-2: Extra credit

- Often factory methods need to be called with specific arguments.
- Create a `flexible_abstract_factory` that that can take signatures
  - If you just give it a class instead of a signature, then it calls the default constructor like our original factory
- `flexible_abstract_factory`

```
<Locomotive(double horsepower),
 FreightCar(long capacity),
 Caboose>
```

- Redo 14-1 with this class template

HW 14-2: Extra credit



- Instead of having a `ModelLocomotive` class, etc., define them as template specializations `Model<Locomotive>`, etc.
- Now change the `concrete_factory` class so you can create a model train factory like:

```
using parameterized_model_train_factory
  parameterized_factory<train_factory, Model>;
```

- Redo 14-1 with this class

HW 14-3: Distributed Counters

- I have posted our atomics-based DistributedCounter4 from last quarter is on Canvas
- Can it be improved with different memory orderings
- Why or why not?