技术 (Techniques, Idioms, Patterns)

条款25：将 constructor 和 non-member function 虚化

将 constructor 虚化

所谓 virtual constructor 是某种函数，根据其所获得的输入，产生不同类型的对象。

```
class NLComponent {
public:
};

class TextBlock : public NLComponent {
public:
};

class Graphic : public NLComponent {
public:
};

class NewsLetter {
public:
  NewsLetter(istream &str);
private:
  list<NLComponet *> components;

  static NLComponent * readComponent(istream &str);
}

NewsLetter::NewsLetter(istream &str) {
  while (str) {
    components.push_back(readComponent(str));
  }
}
```

上面这段代码中的 readComponent 就是一个 virtual constructor。它通过获取的输入来向外生成 TextBlock 或者 Graphic。

一种特别的 virtual constructor 被成为 virtual copy constructor，它将调用这复制一个新的 副本并返回其指针。

```
class NLComponent {
public:
  virtual NLComponet * clone() const = 0;
};

class TextBlock : public NLComponent {
public:
```

```
  virtual TextBlock * clone() const {
    return new TextBlock(*this);
  }
};

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

可以看到，virtual copy constructor 实际上也是调用真正的 copy constructor 来实现。前者只 是利用 “虚函数之返回类型” 规则中的一个宽松点来实现。

当 NLComponent 拥有一个 virtual copy constructor 后，我们可以轻松的为 NewsLetter 实现 一个正常的 copy constructor。

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter &rhs);
private:
  list<NLComponent *> components;
};

NewsLetter::NewsLetter(const NewsLetter &rhs) {
  list<MLComponet *>::const_iterator it = rhs.components.begin();
  for (; it != rhs.components.end(); ++it) {
    components.push_back((*it)->clone());
  }
}
```

将 Non-Member Functions 的行为虚化

和 constructor 一样，non-member functions 无法真正意义上的虚化。但是有些场景下，我们仍然 希望能能够面向不同的对象，同一个函数实现不一样的功能。

这里以打印为例，operator<< 无法成为一个 member function 来使用，因此需要寻找其他方法。正确 的行为是，单独声明一个虚函数来执行打印操作，同时在 non-member functions 中调用该虚函数。

```
class NLComponent {
public:
  virtual ostream &print(ostream &os) const = 0;
};

class TextBlock : public NLComponent {
public:
  virtual ostream &print(ostream &os) const;
};
```

```
class Graphic : public NLComponent {
public:
  virtual ostream &print(ostream &os) const;
}

inline ostream &operator<<(ostream &os, const NLComponent &c) {
  return c.print(os);
}
```

条款26：限制某个 class 所能产生的对象数量

有时，我们需要对使用的对象个数进行限制，本条款用来讨论这些问题。

1. 允许 0 或 1 个对象

从 0 开始讨论是为了弄清楚一个最关键的问题——如何阻止对象被产生出来。

首先，阻止一个对象的创建最简单的方法就是 将 constructor 设置为 private。这样一来就不再有可能 调用 constructor 来创建对象。

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated &rhs);
}
```

有时，我们希望只存在一个对象，而并非不创建对象。这样就需要选择性的解除上面的约束限制。

```
class PrintJob;

class Printer {
public:
  void submitJob(const PrintJob &job);
  void reset();
  void performSelfTest();

friend Printer &thePrinter();
private:
  Printer();
  Printer(const Printer &rhs);
}

Printer &thePrinter() {
  static Printer p;
  return p;
}
```

以上代码能够生成唯一的一个 Printer 对象的原因具体有三点：

1. 构造函数设置为 private，阻止其他人调用生成 Printer 对象。
2. thePrinter 被声明为 class 的一个 friend，使得其可以调用 private constructor。
3. 使用 static 进行限制，限制全局只有一个对象。

上面的 thePrinter 函数是作为一个全局函数出现的。如果你认为不妥，可以使用 静态成员方法 或者 namespace 来解决这个问题。

在上面的代码中，还有两个细节值得讨论：

1. Printer 对象是一个 函数中的 static 对象 而非 class 中的 static 对象。 这两者的区别在于，class static object 当类型所在的文件被加载时就会被创建。而 function static objcet 则是在第一次调用时才被创建。

2. 如此短小的函数为什么不使用 inline 这需要理解一下 inline 的含义。除了概念上的编译时进行替代以外，对于 non-member functions，它 还意味着 这个函数有内部连接。

函数如果带有内部链接，可能会在程序中被复制。这也就意味着 static 对象可能会拥有多份该 static 对 象的副本。因此谨记：

> [!warning] 千万不要产生内含 local static 对象的 inline non-member functions。

除了这种方案外，是否还有更加简单易懂的方案。有，当存在对象超过限额时抛出异常。

```
class Printer {
public:
  class TooManyObjects {};

  Printer();
  ~Printer();
private:
  static size_t numObjects;
}

size_t Printer::numObjects = 0; // class statics 除了在 class 内声明，还需要在 class 外定义

Printer::Printer() {
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  ++numObjects;
}

Printer::~Printer() {
  --numObjects;
}
```

这种方式更容易一般化，只需更改限额数量。

2. 不同的对象构造状态

但是上面这种方法仍然有问题。当存在继承关系，或者被嵌入在其他对象中时可能会产生与预期不同的结果。

```
class ColorPrinter : public Printer {}

Printer p;
ColorPrinter cp;
```

因为，当子类调用 constructor 时，会调用父类的无参构造函数。因此上面这种本意是 printer 和 color printer 各一个的场景却会抛出 exception。因为申请了两个 printer。

同样的事情也会发生在内含 Printer 对象的其他对象中。

```
Class CPFMachine {
private:
  Printer p;
  FaxMachine f;
  CopyMachine c;
};

CPFMachine m1;
CPFMachine m2;
```

上述代码中，当生成 m2 时会抛出 exception，原因同上。

使用 private constructors 的 class 来实现，可以导致不可派生。如此一来可以实现包括两个特点的 对对象：1. 有限个对象。2. 不可被继承。

```
class FSA {
public:
  static FSA * makeFSA();
  static FSA * makeFSA(const FSA &rhs);

private:
  FSA();
  FSA(const FSA &rhs);
}

FSA * FSA::makeFSA() {return new FSA();}
FSA * FSA::makeFSA(const FSA &rhs) {return new FSA(rhs);}
```

上述的代码完全满足两个特性，首先可以被新建多个对象，同时使用前面的技术可以限制产生对象的数量。其次将 constructor 写为 private 函数，防止继承。

但是需要注意的是，其中使用 new 进行资源分配。所以必须记得调用 delete 来阻止资源的泄漏。或者， 可以选择使用 auto_ptr 来封装对象，在离开 scoop 时自动删除这些对象。

```
auto_ptr<FSA> pfsa1(FSA::makeFSA());
auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));
```

3. 允许对象生生灭灭

在 1 和 2 中已经了解到了如何设计 “只允许单一对象” 的 class，“追踪某特定 class 的对象个数”， 以及计数会导致的 constructor 调用异常，使用 private constructor 来防止这种事情的发生。

但是，当限制了个数为 1 的对象时，又很难实现在不同时间内维持 1 个对象的需求。

```
创建 Pointer p1;
使用 p1;
销毁 p1;

创建 Pointer p2;
使用 p2;
销毁 p2;
```

这种场景也是可预期的有限个数，但是与 [### 1.] 中所提到情形却不相同。实现这种效果的方法是：将 “对象计数” 和 “伪构造函数” 相结合。

```
class Printer {
public:
  class TooManyObjects{};

  static Printer * makePrinter();
  ~Printer();
  void submitJob(const PrintJob &job);
  void reset();
  void performSelfTest();

private:
  static size_t numObjects;
  Printer();
  Printer(const Printer &rhs);
}

size_t Printer::numObjects = 0; // class statics 除了在 class 内声明，还需要在 class 外定义

Printer::Printer() {
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  ++numObjects;
}
```

```
Printer * Printer::makePrinter() {
  try {
    return new Printer();
  }
  catch (TooManyObjects &e) {
    return nullptr;
  }
}

Printer::~Printer() {
  --numObjects;
}
```

上面代码所提供的 makePrinter 处理了 exception，因为有时我们不希望这个或称带来额外的困扰。但 是这也就意味着返回 nullptr，使用者需要对 makePrinter 进行验证。

4. 一个用来计算对象个数的 base class

对上述代码进行进一步的优化，其中的计数部分可以被单独提取出来写为一个 base class，作为对象计数 用。然后将诸如 Printer 的类来继承它。

```
template <class BeingCounted>
class Counted {
public:
  class TooManyObjects {};
  static int objectCount();

protected:
  Counted();
  Counted(const Counted &rhs);
  ~Counted(){ --numObjects; }

private:
  static int numObjectes;
  static const size_t maxObjects;
  void init();
};

template <class BeingCounted>
Counted<BeingCounted>::Counted() {
  init();
}

template <class BeingCounted>
Counted<BeingCounted>::Counted(const Counted<BeingCounted>&) {
  init();
}

template <class BeingCounted>
void Counted<BeingCounted>::init() {
  if (numObjects >= maxObjects) throw TooManyObjects();
```

```
  ++numObjects;
}

class Printer : private Counted<Printer> {
public:
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer &rhs);
  ~Printer();
  void submitJob(const PrintJob &job);
  void reset();
  void performSelfTest();

  using Counted<Printer>::objectCount;
  using Counted<Printer>::TooManyObjects;

private:
  Printer();
  Printer(const Printer &rhs);
}
```

Printer 使用 Counted template 追踪目前存在的 Printer 对象。这种实现使用 private 继承。 如果使用 public 继承，那么就需要为 Counted classes 提供一个 virtual destructor，否则当 有人通过 Counted * 指针来删除 Printer 对象时会产生意外的效果。

因为使用了 private 继承，因此 Counted classes 中的 public/protected 内容都变为了 private 内容。所以使用 using declaration 来恢复 public 访问层级。

使用上面的方案，可以轻松的将计数操作忽略，而方便 Printer 类似类型的设计。

其中最后需要被讨论的一个点是：关于 maxObjects 的值的定义。这个定义应该由 Printer 的作者进行 定义，他需要在实现文件中添加 `const size_t Counted<Printer>::maxObjects = 10;` 这样的 定义行。

条款27：要求 (或禁止) 对象产生于 heap 之中