实现

条款26: 尽可能延后变量定义式的出现时间

- 尽可能延后变量定义式的出现。这样做可以增加程序的清晰度并改善程序效率。

只要你定义一个变量，而其类型带有一个构造函数或者析构函数，那么当程序控制流到达这个位置时，就会 产生构造成本。离开作用域时你又需要承受析构成本。甚至你从未使用过这个变量。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  string encrypted;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }

  return encrypted;
}
```

上述代码就可能产生一个完全没有被使用的变量 encrypted。通过修改顺序，将 encrypted 的定义式 向后移动，可以避免这种事情的发生。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }
  string encrypted;

  return encrypted;
}
```

但是上述代码效率仍旧不高，因为 encrypted 并没有初始化，这意味着它会先调用 default constructor， 然后再调用 operator=。那前面 default constructor 的工作就前功尽弃了。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }
  string encrypted(password);
  encrypt(encrypted);
  return encrypted;
}
```

这种写法显然更好。这其中包含着延后变量定义式意味着直到能够给变量初值实参为止，再进行定义。但是，如果遇到循环应该怎么办呢？

```
Widget w;
for (int i = 0; i < n; ++i) {
  w = ...
}
```

```
for (int i = 0; i < n; ++i) {
  Widget w(...);
}
```

这两种方式哪个好。前者的成本更低，因为只需要 n 次赋值操作，但是不易理解和维护，因为 w 所在的 作用域更广，影响更大。而后者成本更高，但是更易维护。因此在选择时，除非 1. 知道赋值成本比构造+ 析构更低 2. 代码是效率高度敏感的部分，否则你应该选择下面的做法。

条款27: 尽量少做转型动作

在 c++ 中转型会破坏类型系统，而不像 c 一样无可厚非。c++ 提供了 4 种新式的转型方式：

- const_cast(expression): 去除常量性
- dynamic_cast(expression): 安全向下转型，即用来决定某对象是否归属继承体系中的某个类型。(无法通过旧式转型实现)
- reinterpret_cast(expression): 执行低级转型，实际动作取决于编译器，即不可移植。pointer to int
- static_cast(expression): 强迫隐式转型。

推荐使用新式转型，因为很容易在代码中分辨，其次转型动作的目标更窄，更容易被编译器诊断出错误。 转型操作会被编译器编译出对应动作，而非直接更改类型。例如指针，当使用 base class pointer 指 向 derived class 时，指针地址可能并不相同。这种情况下会有偏移量在运行期被施行于 Derived* 身上。也因此不要做出 “对象在 C++ 中如何布局” 的假设。

错误转型的另外一个场景是，可能写出在其他语言正确在 c++ 中却错误的代码，例如：

```
class Window {
public:
  virtual void onResize() {}
};

class SpecialWindow : public Window {
public:
  virtual void onResize() {
    static_cast<Window>(*this).onResize();
    ...
  }
};
```

上述代码本意是，在子类中先调用父类的 onResize 方法，然后再在当前子类中进行更改。但是这完全不 正确，因为 static_cast 返回的是一个副本，而不是当前对象父类的引用。这意味这上述更改均更改在 了一个 Window 副本中，而当前对象中的 Window 对象却丝毫未动，正确的写法应该如下：

```
class Window {
public:
  virtual void onResize() {}
};

class SpecialWindow : public Window {
public:
  virtual void onResize() {
    Window::onResize();
    ...
  }
};
```

当调用转型时，你很有可能走上了不归路。dynamic_cast更是如此。dynamic_cast 在许多版本中的实 现方案效率都不是很高，因此在注重效率的代码中应该避免使用 dynamic_cast。

通常你使用 dynamic_cast 是因为你认定了该 base class pointer 指向的对象是一个 derived class 对象。那想要避免使用 dynamic_cast 有两个方法。

1. 使用容器存储对应的 derived class 指针

```
class Window {};
class SpecialWindow : public Window {
public:
  void blink();
};

typedef std::vector<std::shared_ptr<SpecialWindow>> VPSW;
VPSW winPtrs;

for (VPSW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
}
```

这种做法不允许在一个容器内存储不同的 Window 派生类。

2. 通过 base class 接口来处理 “所有可能的各种 Window 派生类”，也就是使用 virtual。

```
class Window {
public:
  virtual void blink() {} // 缺省实现代码并不好
};
```

```
class SpecialWindow : public Window {
public:
  virtual void blink() {
    ...
  }
};

typedef std::vector<std::shared_ptr<Window>> VPW;
VPW winPtrs;

for (VPW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
}
```

一定要避免的代码是，一连串的 dynamic_casts。

```
for (VPW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
  if (SpecialWindow1* psw1 = dynamic_cast<SpecialWindow1*>(iter->get())) {
    ...
  } else if (SpecialWindow2* psw2 = dynamic_cast<SpecialWindow2*>(iter->get())) {
    ...
  } else if (SpecialWindow3* psw3 = dynamic_cast<SpecialWindow3*>(iter->get())) {
    ...
  }
}
```

这样的代码效率极低。总结来看，我们应该尽可能隔离转型动作，通常将它隐藏在某个函数内，函数的接口 会保护调用者不受内部任何动作的影响。

条款28: 避免返回 handles 指向对象内部成分

```
class Point {
public:
  Point(int x, int y);
  void setX(int newVal);
  void setY(int newVal);
};

struct RectData {
  Point ulhc; // upper left-hand corner
  Point lrhc; // lower right-hand corner
};

class Rectangle {
public:
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
```

```
private:
  std::shared_ptr<RectData> pData;
};
```

这样设计虽然能够通过编译，但是却是错误的，因为它违背了 reference constness 的原则。显然当你 使用一个 const Rectangle 对象调用 upperLeft() 时，你会获得一个指向内部数据 ulhc 的引用。 而这个引用却没有限定，因此可以被修改。而这一行为与 const 恰好冲突。

这就是 handles 的特点，它包括 reference、pointer 和 iterator。如果返回一个 “代表对象内部 数据” 的 handle，随之而来的就是 “降低对象封装性” 的风险。

使用 const 进行限定，可以实现由限度的松弛封装性，即可读但不可更改。

```
const Point& upperLeft() const {}
const Point& lowerRight() const {}
```

但是即使这样，还是会产生 dangling handles 的风险。这种风险往往来自于函数返回值。

```
class GUIObject {};
const Rectangle boundingBox(const GUIObject& obj);

GUIObject* pgo;
const Point* pUpperLeft = &(boundingBox(*pgo).upperLeft());
```

上述函数就会发生 dangling handles 的风险。因为实际上 pUpperLeft 指向的是一个临时变量的 ulhc， 即 temp.upperLeft()。而当这条语句运行结束之后，temp 会被销毁，那么此时 pUpperLeft 就变成 了空悬、虚吊。

条款29: 为“异常安全”而努力是值得的

```
class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  Image* bgImage;
  int imageChange;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  lock(&mutex);
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
  unlock(&mutex);
}
```

上述代码不是 异常安全 的。异常安全需要满足两个条件，当异常被抛出时：

- 不泄露任何资源: 一旦 new Image 抛出异常，unlock 就不再执行，也就是说 mutex 永远被锁。
- 不允许数据败坏: 如果 new Image 抛出异常，bgImage 就指向了一个已经删除的对象，imageChanges 也被累加，而没有新的图像产生。

资源泄露的解决方案在 条款13 中已经指出可以使用对象来管理资源，条款14 中则给出了一个 Lock 类 的实现方案。因此可以改写为:

```
void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock m(&mutex);
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

对于数据败坏的解决方案我们需要进行抉择，在此之前先了解一下选项的术语:

- 基本承诺: 如果异常抛出，程序内的任何事物仍然保持在有效状态下。然而此时程序的现实状态不可预料，可能是前一状态或是缺省状态。客户需要额外的函数来判断。
- 强烈保证: 如果异常抛出，程序状态不改变。成功就完全成功，否则就回复到调用之前的状态。这相较于前者调用起来更加简单，因为只有两个状态，前一状态和成功后状态。
- 不抛掷保证: 承诺不抛出异常，因为他们总是能够正确运行。比如内置类型就会提供 nothrow 保证。但是这并不是说绝不会抛出异常，而是如果产生异常那将是严重错误，而不可解决。

如果想要 异常安全性，那么就需要在上面三者中选择一个。选择的原则是，可能的话提供 nothrow 保证， 但是大多数条件下提供 基本保证 或者 强烈保证。

```
class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  std::shared_ptr<Image> bgImage;
  int imageChange;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock m(&mutex);
  bgImage.reset(new Image(imgSrc));

  ++imageChanges;
}
```

这样改写代码，可以保证 基本承诺。首先是使用 std::shared_ptr 来管理 bgImage。这有两点好处 1. 强化资源管理 2. 使用内置的 reset 函数来实现替换操作。其次将 ++imageChanges 放到最后，保证 只有正确运行才计数。

但是上面只保证了 基本承诺，因为 imgSrc 的状态并不确定，它的读取记号可能已经被移走。但这种更改 很容易实现，因此不再进一步考虑。

下面需要了解另一个实现 强烈保证的手段—— copy and swap。即 swap 之前先进行 copy 操作。

```
struct PMImpl {
  std::shared_ptr<Image> bgImage;
  int imageChange;
};

class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  std::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  using std::swap;

  Lock m(&mutex);
  std::shared_ptr<PMImpl> pNew(new PMImpl(*pImpl));
  pNew->bgImage.reset(new Image(imgSrc));
  ++pNew->imageChanges;

  swap(pImpl, pNew);
}
```

使用 copy-and-swap 能够实现强烈保证，但是并不是总能成功。

```
void someFunc() {
  f1();
  f2();
}
```

例如上面这种形式的代码，即使 f1 成功调用，只要 f2 失败，那么就破坏了强烈保证的原则。这问题出 在连带影响上。如果函数只操作局部性状态，那就相对容易提供强烈保证，否则就不行。例如数据库的修改 动作往往就无法提供强烈保证。

除此之外，copy-and-swap 的效率问题也很严重。所以，当强烈保证可以实现时，你的确应该提供它，但是“强烈保证”并非在任何时刻都显得实际。而不切实际时，就应该提供“基本保证”。

条款30: 透彻了解 inlining 的里里外外

如果 inline 函数的本体很小，编译器针对“函数本体”所产生的码可能比针对“函数调用”所产生的码更小。 将函数 inline 确实可以导致较小的目标码和较高的指令告诉缓存装置命中率。

inline 的定义可以分为隐式和显式:

```
// 隐式
class Person {
public:
  int age() const {return theAge;}
private:
  int theAge;
};
```

```
// 显式
template<typename T>
inline const T& std::max(const T& a, const T& b) {
  return a < b ? b : a;
}
```

编译器通常会拒绝将太过复杂的函数 inlining，而所有对 virtual 函数的调用也都会导致 inlining 落空。总的来看，一个表面上看似 inline 的函数是否真的是 inline 主要取决于你的建置环境，主要 取决于编译器。有时候编译器可能还会为 inlining 生成一个函数本体，比如当你需要获取函数指针时编 译器就不得不生成一个 outlined 函数本体。

关于构造函数和析构函数，使用 inline 可能是一个糟糕的想法。例如 default constructor，看上 去似乎是一段空代码，但是实际上编译器会给你填充大量的内容，防止其产生异常。而这恰恰违背了 短小 的要求。

除此之外，将函数声明为 inline 所需要面对更多的是，随着代码的维护可能函数会进行扩充，这样就不 适用 inline 标记了。

所以选择是否适用 inline 的逻辑策略是，一开始先不要将任何函数声明为 inline，或至少将 inlining 施行范围局限在那些 “一定成为 inline” 或者 “十分平淡无奇” 的函数身上。

条款31: 将文件间的编译依存关系降至最低

假设你对 c++ 程序的某个 class 实现做出了轻微修改，当你进行编译时可能遇到大量代码重新编译的情 况。这个原因在于 c++ 没有把 "将接口从实现中分离" 这件事做好。

```
class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};
```

上述代码在进行编译时，需要引入其他定义文件。这样一来就形成了一种编译依存关系。这种依存关系之下，如果头文件或者头文件依赖的头文件发生改变时，Person class 以及任何使用 Person class 的文件 都必须重新编译。那么如何将接口分离呢？使用 “声明依存性” 替换 “定义依存性”。

```
#include <string>
#include <memory>

class PersonImpl; // Person 实现类的前置声明
class Date; // 接口用到的 class 前置声明
class Address;

class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::shared_ptr<PersonImpl> pImpl; // pimpl idiom
};
```

上述代码的好处是，无论 PersonImpl 如何更改，或者其依赖 例如 Data，Address 如何更改，只需 重新编译到 PersonImpl 即可。因为 Person 中使用一个指针来指向 PersonImpl 对象，因此不需要 额外进行重新编译。这也就意味着，所有依赖 Person 的文件也不用重新更改了。这样一来，就完全分离 了他们之间的联系，也是真正的“接口与实现分离”。原则是：让头文件尽可能自我满足，万一做不到，则 让其他文件内的声明式相依。

其他的设计策略也都基于此:

- 如果使用 reference 或 pointers 可以完成任务，就不要使用 objects。你可以只靠一个类型声明式就定义出指向该类型的 references 和 pointers，但如果定义某类型的 objects， 就需要用到该类型的定义式。
- 如果能够，尽量用 class 声明式替换 定义。 当你声明一个函数而它用到某个 class 时，你并不需要该 class 的定义。
- 为声明式 和 定义式 提供不同的头文件。 为了遵守上述原则，需要两个头文件，一个用于声明式，一个用于定义式。引用时只引用声明式头文件。

```
/* date.h */
class Date {
  ...
};
```

```
/* datefwd.h */
class Date;
```

```
/* main.cpp */
#include "datefwd.h"
Date today();
void clearAppointments(Date d);
```

这种方式效仿了 c++ 标准程序库头文件的 。在其中包含了 iostream 各组件的声明式，其对 应定义则分布在若干不同的头文件内，包括，，，。它另一个特点是，如果建置环境允许 template 定义放在非头文件中，那么就可以通过声明式头文件来提 供 template。

另外一种方式时使用 c++ 提供的关键字 export。

回到 pimpl idiom，像 Person 这样使用 pimpl idiom 的 class 被称为 handle classes。他们 所有的函数都将任务转交给相应的实现类。

```
#include "Person.h"
#include "PersonImpl.h"

Person::Person(const std::string& name, const Date& birthday, const Address& addr)
:
  pImpl(new PersonImpl(name, birthday, addr)) {}
std::string Person::name() const {
  return pImpl->name();
}
```

实现 handle classes 的另一个方法时令 Person 称为 interface class。

```
class Person {
public:
  virtual ~Person();
  virtual std::string name() const;
  virtual std::string birthDate() const;
  virtual std::string address() const;

  static std::shared_ptr<Person> create(const std::string& name,
    const Date& date, const Address& address);
};

class RealPerson : public Person {
public:
  RealPerson(const std::string& name, const Date& birthday, const Address& addr);
  virtual ~RealPerson() {}
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::string theName;
  Date theBirthDate;
```

```
  Address theAddress;
};

std::shared_ptr<Person> Person::create(const std::string& name,
    const Date& date, const Address& address) {
  return std::shared_ptr<Person>(new RealPerson(name, date, addres));
}

std::shared_ptr<Person> pp(Person::create(name, date, address));
std::cout << pp->name()
          << pp->birthDate()
          << pp->address();
```

上面这段代码，将 Person 改为了 interface class，并提供了一个静态类方法用于实现 factory 函数。factory 函数的职责在于，根据条件生成一个实现类对象，这里就是 RealPerson。在使用过程 中，使用 Person pointer 进行操作，而实现则放在 RealPerson 中，这也同样实现了 pimpl idiom 类似的降低依赖的效果。

在成本角度，无论是 handle class 还是 interface class 都会带来额外的成本。handle class 通过 implementation pointer 进行对象访问，增加了一层间接访问。同时，每一个对象又会有额外的 开销。最后你要使用指针，那就可能发生动态内存分配的额外开销，以及 bad_alloc 的风险。而 interface class 则每次调用 virtual 都必须经过 vptr。同时 vptr 又增加了存储开销。

因此你需要在成本和耦合度之间做出抉择。