4

Брандмауэр и идиома скрытой реализации

Управление зависимостями между частями кода является далеко не самой маловажной частью разработки. Я считаю [Sutter98], что в этом плане сильные стороны C++ — поддержка двух мощных методов абстракции: объектно-ориентированного и обобщенного программирования. Оба эти метода представляют собой фундаментальный инструментарий, помогающий программисту в управлении зависимостями и, соответственно, сложностью проектов.

Задача 4.1. Минимизация зависимостей времени компиляции. Часть 1 *Сложность: 4*

Когда мы говорим о зависимостях, то обычно подразумеваем зависимости времени выполнения, типа взаимодействия классов. В данной задаче мы обратимся к анализу зависимостей времени компиляции. В качестве первого шага попытаемся найти и ликвидировать излишние заголовочные файлы.

Многие программисты часто используют гораздо больше заголовочных файлов, чем это необходимо на самом деле. К сожалению, такой подход существенно увеличивает время построения проекта, в особенности когда часто используемые заголовочные файлы включают слишком много других заголовочных файлов.

Какие из директив `#include` могут быть удалены из следующего заголовочного файла без каких-либо отрицательных последствий? Кроме удаления и изменения директив `#include`, вы не вправе вносить в текст никакие другие изменения. Обратите также внимание на важность комментариев.

```
//  x.h: исходный заголовочный файл
//
#include <iostream>
#include <ostream>
#include <list>

// Ни один из классов A, B, C, D и E не является шаблоном
// Только классы A и C имеют виртуальные функции
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
#include "e.h"  // class E

class X : public A, private B
{
```

```
public:
    X(const C&);
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E )
    virtual std::ostream& print( std::ostream& ) const;
private:
    std::list<C> clist_;
    D            d_;
};

inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Решение

Из первых двух стандартных заголовочных файлов, упомянутых в `x.h`, один может быть немедленно удален за ненадобностью, а второй — заменен меньшим заголовочным файлом.

1. Удалите `#include <iostream>`.

Многие программисты включают `#include <iostream>` исключительно по привычке, как только видят любое упоминание потоков в тексте программы. Класс `X` использует потоки, это так, но ему вовсе не требуется все упомянутое в файле `iostream`. Максимум, что ему необходимо, — это включение одного лишь `ostream`, да и этого слишком много.

Рекомендация

Никогда не включайте в программу излишние заголовочные файлы.

2. Замените `ostream` на `iosfwd`.

Для работы нам требуется знать только параметры и возвращаемые типы, так что вместо полного определения `ostream` достаточно только предварительного объявления этого класса.

Раньше мы могли бы просто заменить "`#include <ostream>`" строкой "`class ostream;`", поскольку `ostream` представлял собой класс и не располагался в пространстве имен `std`. Теперь же такая замена некорректна по двум причинам.

1. `ostream` теперь находится в пространстве имен `std`, а программисту запрещается объявлять что-либо, находящееся в этом пространстве имен.
2. `ostream` в настоящее время представляет собой синоним шаблона, а именно шаблона `basic_ostream<char>`. Дать предварительное объявление шаблона `basic_ostream` нельзя, поскольку реализации библиотеки могут, например, добавлять свои собственные параметры шаблона (помимо требуемых стандартом), о которых ваш код, естественно, не осведомлен. Кстати, это одна из основных причин, по которой программисту запрещается давать собственные объявления чего-либо, находящегося в пространстве имен `std`.

Однако не все потеряно. Стандартная библиотека предупредительно обеспечивает нас заголовочным файлом `iosfwd`, который содержит предварительные объявления

всех шаблонов потоков (включая basic_iostream) и стандартных определений typedef этих шаблонов (включая ostream). Так что все, что нам надо, — это заменить директиву #include <ostream> директивой #include <iosfwd>.

Рекомендация

Там, где достаточно предварительного определения потока, предпочтительно использовать директиву #include <iosfwd>.

Кстати, если, узнав о существовании заголовочного файла iosfwd, вы сделаете вывод, что должны существовать аналогичные файлы для других шаблонов, типа stringfwd или listfwd, то вы ошибетесь. Заголовочный файл iosfwd был создан с целью обратной совместимости, чтобы избежать неработоспособности кода, использующего старую, не шаблонную подсистему потоков.

Итак, все просто. Мы...

"Минутку! — слышу я голос кого-то из читателей. — Не так быстро! Наш operator<< реально использует объект ostream, так что нам будет недостаточно упоминания типов параметров или возвращаемого типа — нам требуется полное определение ostream, так?"

Нет, не так. Хотя вопрос вполне разумный. Рассмотрим еще раз интересующую нас функцию.

```
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Здесь ostream& упоминается как тип параметра и как возвращаемый тип (большинство программистов знают, что для этого определение класса не требуется) и передается в качестве параметра другой функции (многие программисты *не знают*, что и в этом случае определение класса не требуется). До тех пор пока мы работаем со ссылкой ostream&, полное определение ostream необходимым не является. Конечно, нам потребуется полное определение, если мы попытаемся вызвать любую функцию-член, но здесь ничего подобного не происходит.

1. Замените "#include "e.h"" предварительным определением "class E;".

Класс E упоминается только в качестве типа параметра и возвращаемого типа, так что его определение нам не требуется.

Рекомендация

Никогда не используйте директиву #include там, где достаточно предварительного объявления.

Задача 4.2. Минимизация зависимостей времени компиляции. Часть 2 *Сложность: 6*

Теперь, когда удалены все излишние заголовки, пришло время для второго захода: каким образом можно ограничить зависимости внутреннего представления класса?

Далее приведен заголовочный файл из задачи 4.1 после первого преобразования. Какие еще директивы #include можно удалить, если внести некоторые изменения, и какие именно изменения требуются?

Мы можем вносить только такие изменения в класс X, при которых базовые классы X и его открытый интерфейс остаются без изменений; любой код, который использует X в настоящий момент, не должен требовать изменений, ограничиваясь простой перекомпиляцией.

```
//  x.h: без лишних заголовочных файлов
//
#include <iosfwd>
#include <list>

// Ни один из классов A, B, C, D и E не является шаблоном
// Только классы A и C имеют виртуальные функции
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D

class E;

class X : public A, private B
{
public:
    X(const C&);
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E )
    virtual std::ostream& print( std::ostream& ) const;
private:
    std::list<C> clist_;
    D            d_;
};

inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Решение

Давайте рассмотрим, чего мы не можем сделать.

- Мы не можем удалить a.h и b.h, поскольку X является наследником как A, так и B. Для того чтобы компилятор мог получить размер объекта X, информацию о виртуальных функциях и прочую существенную информацию, требуется полное определение базовых классов.
- Мы не можем удалить list, c.h и d.h, поскольку list<C> и D представляют собой типы членов данных X. Хотя C не является ни базовым классом, ни членом, он используется для настройки члена list, а большинство современных компиляторов требуют полного определения C в случае настройки list<C>. (Стандарт не требует полного определения в этой ситуации, так что в будущем можно ожидать снятия этого ограничения.)

А теперь поговорим о красоте скрытой реализации.

C++ позволяет нам легко инкапсулировать закрытые части класса от неавторизованного *доступа*. К сожалению, поскольку подход с использованием заголовочных файлов унаследован от C, для инскапсуляции *зависимостей* закрытых частей класса требуется бо́льшая работа. “Но, — скажете вы, — ведь суть инкапсуляции и состоит в

том, что клиентский код не должен ничего знать о деталях реализации закрытых частей класса, не так ли?" Да, так, и клиентский код в C++ не должен ничего знать о закрытых частях класса (да и если это не друг класса, то он просто не получит доступ к ним), но поскольку эти закрытые части видны в заголовочном файле, код клиента вынужденно зависит от всех упомянутых там типов.

Каким же образом можно изолировать клиента от деталей реализации? Один из хороших способов — использование специального вида идиомы handle/body [Coplien92] (которую я называю идиомой Pimpl[1] из-за удобства произношения "указатель `pimpl_`") в качестве брандмауэра компиляции [Lakos96, Meyers98, Meyers99, Murray93].

Скрытая реализация представляет собой не что иное, как непрозрачный указатель (указатель на предварительно объявленный, но не определенный вспомогательный класс), использующийся для сокрытия защищенных членов класса. То есть вместо

```
// Файл x.h
class X
{
    // Открытые и защищенные члены
private:
    // Закрытые члены. При внесении изменений
    // в эту часть весь клиентский код должен
    // быть перекомпилирован.
};
```

мы пишем

```
// Файл x.h
class X
{
    // Открытые и защищенные члены
private:
    struct XImpl;
    XImpl* pimpl_; // Указатель на предварительно
                   // объявленный класс
};

// Файл x.cpp
struct X::XImpl
{
    // Закрытые члены; внесение изменений
    // не требует перекомпиляции кода клиентов
};
```

Для каждого объекта `X` динамически выделяется объект `XImpl`. Мы по сути обрезаем большой кусок объекта и оставляем на этом месте только маленький кусочек — указатель на реализацию.

Основные преимущества идиомы скрытой реализации обусловлены тем, что она позволяет разорвать зависимости времени компиляции.

- Типы, упоминающиеся только в реализации класса, больше не должны быть определены в коде клиента, что позволяет устранить лишние директивы `#include` и повысить скорость компиляции.
- Реализация класса может быть легко изменена — вы можете свободно добавлять или удалять закрытые члены без перекомпиляции клиентского кода.

Платой за эти преимущества является снижение производительности.

---

[1] Pimpl означает "указатель на реализацию" (**p**ointer to **impl**ementation), но использование в качестве перевода термина "скрытая реализация" в данном случае отражает суть идиомы — сокрытие деталей реализации класса. — *Прим. перев.*

- Каждое создание и уничтожение объекта сопровождается выделением и освобождением памяти.
- Каждое обращение к скрытому члену требует как минимум одного уровня косвенности. (Уровней косвенности может оказаться и несколько, например, если обращение к скрытому члену использует указатель для вызова функции в видимом классе.)

Мы вернемся к этим и другим вопросам использования скрытой реализации немного позже, а пока продолжим решение нашей задачи. В нашем случае три заголовочных файла нужны только потому, что соответствующие классы встречаются при описании закрытых членов X. Если реструктуризировать X с использованием идиомы скрытой реализации, мы тут же сможем внести в код ряд дальнейших упрощений.

Итак, используем идиому Pimpl для сокрытия деталей реализации X.

```
#include <list>

#include "c.h"  // class C
#include "d.h"  // class D
```

Один из этих заголовочных файлов (c.h) может быть заменен предварительным объявлением (поскольку C упоминается в качестве типа параметра и возвращаемого типа), в то время как остальные два могут быть просто удалены.

Рекомендация

Для широко используемых классов предпочтительно использовать идиому сокрытия реализации (брандмауэр компиляции, Pimpl) для сокрытия деталей реализации. Для хранения закрытых членов (как переменных состояния, так и функций-членов) используйте непрозрачный указатель (указатель на объявленный, но не определенный класс), объявленный как "struct XxxxImpl pimpl_;". Например: "class Map { private: struct MapImpl* pimpl_; };".*

После внесения дополнительных изменений заголовочный файл выглядит следующим образом.

```
// x.h: Использование Pimpl
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
#include "b.h" // class B (виртуальных функций нет)
class C;
class E;
class X : public A, private B
{
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

Детали реализации находятся в файле реализации X, который не виден клиентскому коду, который, соответственно, никак не зависит от этого файла.

```
// Файл реализации x.cpp
//
struct X::XImpl
{
    std::list<C> clist_;
    D            d_;
};
```

Итак, у нас осталось только три заголовочных файла, что само по себе является существенным усовершенствованием первоначального кода. А нельзя ли еще больше изменить класс X и добиться дальнейшего усовершенствования полученного кода? Ответ на это вы найдете в задаче 4.3.

Задача 4.3. Минимизация зависимостей времени компиляции. Часть 3 *Сложность: 7*

Теперь, когда удалены все излишние заголовочные файлы и устранены все излишние зависимости во внутреннем представлении класса, остались ли какие-либо возможности для дальнейшего разъединения классов? Ответ возвращает нас к базовым принципам проектирования классов.

Верные принципу не останавливаться на достигнутом, ответьте на вопрос: каким образом можно продолжить уменьшение количества зависимостей? Какой заголовочный файл может быть удален следующим и какие изменения в X для этого следует сделать?

На этот раз вы можете вносить любые изменения в X, лишь бы открытый интерфейс при этом оставался неизменным и для работы клиентского кода было достаточно простой перекомпиляции. Вновь обратите особое внимание на комментарии в приведенном тексте.

```
// x.h: После использования Pimpl для
//      сокрытия деталей реализации
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
#include "b.h" // class B (виртуальных функций нет)
class C;
class E;
class X : public A, private B
{
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

```
// Файл реализации x.cpp
//
struct X::XImpl
{
    std::list<C> clist_;
    D            d_;
};
```

Решение

Согласно моему опыту, множество программистов, работающих на C++, все еще полагают, что без наследования нет объектно-ориентированного программирования и используют наследование везде, где только можно. Этому вопросу посвящена задача 3.5, главный вывод которой заключается в том, что наследование (включающее отношение ЯВЛЯЕТСЯ, но не ограничивающееся им) — существенно более сильное отношение классов, чем СОДЕРЖИТ или ИСПОЛЬЗУЕТ. Таким образом, при работе с зависимостями следует отдавать предпочтение включению.

В нашем случае `X` открыто наследует `A` и закрыто — `B`. Вспомним, что открытое наследование всегда должно моделировать отношение ЯВЛЯЕТСЯ и удовлетворять принципу подстановки Лисков.[2] В данном случае `X` ЯВЛЯЕТСЯ `A` и в этом нет ничего некорректного, так что оставим открытое наследование от `A` в покое.

Но не обратили ли вы внимания на одну интересную деталь, связанную с `B`?

Дело в том, что класс `B` является закрытым базовым классом для `X`, но не имеет виртуальных функций. Но ведь обычно для того, чтобы использовать закрытое наследование, а не включение, имеется только одна причина — получить доступ к защищенным членам, что в большинстве случаев означает "заместить виртуальную функцию".[3] Как мы знаем, виртуальных функций в классе B нет, так что, вероятно, нет и причины использовать наследование (разве что классу `X` требуется доступ к каким-либо защищенным функциям или данным в `B`, но мы предполагаем, что в нашем случае такой необходимости нет). Соответственно, от наследования можно отказаться в пользу включения, а значит, и удалить из текста ненужную теперь директиву "`#include "b.h"`".

Поскольку объект-член `B` должен быть закрытым (в конце концов, он — всего лишь деталь реализации), его следует скрыть за указателем `pimpl_`.

Рекомендация

Никогда не используйте наследование там, где достаточно включения.

Итак, предельно упрощенный в смысле заголовочных файлов код выглядит следующим образом.

```
// x.h: После удаления ненужного наследования
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
class B;
class C;
class E;
class X : public A
{
```

[2] Материал на эту тему можно найти по адресу `http://www.gotw.ca/publications/xc++/om.htm` и в [Martin95].

[3] Конечно, бывают и другие возможные причины для выбора наследования, но эти ситуации крайне редки. Подробное изложение данного вопроса можно найти в [Sutter98a, Sutter99].

```
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Сюда включен объект B
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

После трех последовательных усовершенствований файл `x.h` использует имена других классов, но из восьми включаемых заголовочных файлов осталось только два. Неплохой результат!

Задача 4.4. Брандмауэры компиляции — *Сложность: 6*

Использование скрытой реализации (идиомы Pimpl) существенно снижает взаимозависимости кода и время построения программ. Но что именно следует скрывать за указателем `pimpl_` и как наиболее безопасно использовать его?

В C++ при любых изменениях в определении класса (даже при изменениях в закрытых членах) требуется перекомпиляция всех пользователей этого класса. Для снижения этой зависимости обычной методикой является использование непрозрачного указателя для сокрытия некоторых деталей реализации.

```
class X
{
public:
    /* ... Открытые члены ... */
protected:
    /* ... Защищенные члены? ... */
private:
    /* ... Закрытые члены? ... */
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
```

Ответьте на следующие вопросы.

1. Что следует внести в `XImpl`? Имеется четыре распространенные стратегии.
   - Разместить в `XImpl` все закрытые данные (но не функции).
   - Разместить в `XImpl` все закрытые члены.
   - Разместить в `XImpl` все закрытые и защищенные члены.
   - Сделать `XImpl` классом, полностью повторяющим класс `X`, которым он должен был бы быть, и оставить в классе `X` только открытый интерфейс, просто перенаправляющий вызовы функций классу `XImpl`.

   Каковы преимущества и недостатки каждой из стратегий? Как выбрать среди них наиболее подходящую?
2. Требуется ли наличие в `XImpl` обратного указателя на объект `X`?

Решение

Класс `X` использует вариант идиомы “handle/body” [Coplien92], которая применяется в первую очередь для счетчиков ссылок разделяемой реализации, но может использоваться и для более общих целей сокрытия реализации. Для удобства далее я буду называть `X` “видимым классом”, а `XImpl` — Pimpl-классом.

Большим преимуществом этой идиомы является разрыв зависимостей времени компиляции. Во-первых, ускоряется компиляция и сборка программы, поскольку использование сокрытия реализации, как мы видели в задачах 4.2 и 4.3, позволяет устранить излишние директивы `#include`. В своей практике мне приходилось работать над проектами, в которых применение сокрытия реализации позволяло сократить время компиляции вдвое. Во-вторых, при этом возможно свободное изменение части класса, скрытой за указателем `pimpl_`, так, например, здесь можно свободно добавлять или удалять члены без необходимости перекомпилировать клиентский код.

1. Что следует внести в `XImpl`?

Вариант 1 (оценка: 6/10): разместить в `XImpl` все закрытые данные (но не функции). Это хорошее начало, поскольку теперь нам достаточно предварительного объявления классов, использующихся только в качестве членов данных (мы можем обойтись без директивы `#include` и полного определения класса, что автоматически приведет к зависимости от этого класса клиентского кода).

Вариант 2 (оценка: 10/10): разместить в `XImpl` все закрытые невиртуальные члены. В настоящее время я (почти) всегда поступаю именно таким образом. В конце концов закрытые члены в C++ — это те члены, о которых клиентский код не должен ничего знать вообще. Однако следует сделать ряд предостережений.

- Вы не можете скрывать виртуальные функции-члены, даже если это закрытые функции. Если виртуальная функция замещает виртуальную функцию, унаследованную от базового класса, то она обязана находиться в классе-наследнике. Если же виртуальная функция не является унаследованной, то она все равно должна оставаться в видимом классе, чтобы производные классы могли ее заместить.

  Обычно виртуальные функции являются закрытыми, за исключением ситуаций, когда функция производного класса вызывает функцию базового класса — тогда виртуальная функция должна быть защищенной.
- Если функция в Pimpl-классе в свою очередь использует функции видимого класса, то ей может потребоваться “обратный указатель” на видимый объект, что приводит к дополнительному уровню косвенности (по соглашению такой обратный указатель обычно носит имя `self_`).
- Зачастую наилучшим компромиссом является использование варианта 2, с дополнительным размещением в `XImpl` тех не закрытых функций, которые должны вызываться закрытыми.

Рекомендация

Для широко используемых классов предпочтительно использовать идиомы сокрытия реализации (брандмауэра компиляции, Pimpl) для сокрытия деталей реализации. Для хранения закрытых членов (как переменных состояния, так и функций-членов) используйте непрозрачный указатель (указатель на объявленный, но не определенный класс), объявленный как “`struct XxxxImpl* pimpl_;`”.

Вариант 3 (оценка: 0/10): разместить в `XImpl` все закрытые и защищенные члены. Распространение идиомы Pimpl на защищенные члены является ошибкой. Защищенные члены никогда не должны переноситься в Pimpl-класс. Они существуют для того, чтобы быть видимыми и используемыми производными классами, так что ничего хорошего от того, что они будут скрыты от производных классов, не получится.

Вариант 4 (оценка: 10/10 в некоторых ситуациях): сделать `XImpl` классом, полностью повторяющим класс `X`, которым он должен был бы быть, и оставить в классе `X` только открытый интерфейс, просто перенаправляющий вызовы функций классу `XImpl`. Это неплохое решение в некоторых случаях, обладающее тем преимуществом, что при этом не требуется обратный указатель на видимый объект (все сервисы доступны внутри Pimpl-класса). Основной недостаток данного метода в том, что обычно он делает видимый класс бесполезным с точки зрения наследования.

2. Требуется ли наличие в `XImpl` обратного указателя на объект `X`?

Требуется ли наличие в Pimpl-классе обратного указателя на видимый объект? Ответ — иногда, к сожалению, да. В конце концов, мы делаем не что иное, как (несколько искусственное) разделение каждого объекта на две части с целью скрыть одну из них.

Рассмотрим вызов функции видимого класса. Обычно при этом для завершения запроса требуется обращение к каким-то данным или функциям скрытой части. Это просто и понятно. Однако, на первый взгляд, не столь очевидно, что функции скрытой части могут вызывать функции, находящиеся в видимом классе, обычно потому, что это открытые или виртуальные функции. Один способ минимизации количества таких вызовов состоит в разумном использовании описанного ранее варианта 4, т.е. в реализации варианта 2 и добавлении в Pimpl-класс тех не закрытых функций, которые используются закрытыми.

Задача 4.5. Идиома "Fast Pimpl" *Сложность:6*

Иногда хочется схитрить во имя "уменьшения зависимостей" или "эффективности", но не всегда это приводит к хорошим результатам. В данной задаче рассматривается превосходная идиома для безопасного достижения обеих целей одновременно.

Стандартные вызовы `malloc` и `new` относительно дорогостоящи.[4] В приведенном ниже коде изначально в классе `Y` имелся член данных типа `X`.

```
// Попытка №1
//
// Файл y.h
#include "x.h"
class Y
{
    /* ... */
    X x_;
};

// Файл y.cpp
Y::Y() {}
```

Данное объявление класса `Y` требует, чтобы класс `X` был видимым (из `x.h`). Чтобы избежать этого, программист пытается переделать код следующим образом.

```
// Попытка №2
//
// Файл y.h
class X;
class Y
```

[4] По сравнению с другими типичными операциями, типа вызова функции.

```
{
    /* ... */
    X* px_;
};

// Файл y.cpp
Y::Y() : px_(new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Таким образом удается скрыть X, но из-за частого использования Y и накладных расходов на динамическое распределение памяти это решение приводит к снижению производительности программы.

Тогда наш программист находит решение, которое не требует включения x.h в y.h (и даже не требует предварительного объявления класса X) и позволяет избежать неэффективности из-за динамического распределения.

```
// Попытка №3
//
// Файл y.h
class X;
class Y
{
    /* ... */
    static const size_t sizeofx = /* Некоторое значение */;
    char x_[sizeofx];
};

// Файл y.cpp
#include "x.h"
Y::Y()
{
    assert( sizeofx >= sizeof(X) );
    new(&x_[0]) X;
}

Y::~Y()
{
    (reinterpret_cast<X*>(&x_[0]))->~X();
}
```

1. Каковы накладные расходы памяти при использовании идиомы Pimpl?
2. Каково влияние идиомы Pimpl на производительность?
3. Рассмотрите код из третьей попытки. Не могли бы вы предложить лучший способ избежать накладных расходов?

Примечание: просмотрите еще раз задачу 4.4.

Решение

1. Каковы накладные расходы памяти при использовании идиомы Pimpl?

В случае использования идиомы Pimpl нам требуется память как минимум для одного дополнительного указателя (а при необходимости размещения обратного указателя в XImpl — двух) в каждом объекте X. Обычно это приводит к добавлению как минимум 4 (или 8) байт, но может потребовать 14 байт и более — в зависимости от требований выравнивания. Например, попробуйте скомпилировать следующую программу, используя ваш любимый компилятор.

```
struct X { char c; struct XImpl; XImpl* pimpl_; };
struct X::XImpl { char c; };
int main()
```

```
{
    cout << sizeof(X::XImpl) << endl
         << sizeof(X) << endl;
}
```

Многие популярные компиляторы, использующие 32-битовые указатели, создадут программу, вывод которой будет следующим.

```
1
8
```

В этих компиляторах накладные расходы, вызванные дополнительным указателем, составляют не 4, а 7 байт. Почему? Потому что платформа, на которой работает компилятор, требует, чтобы указатель размещался выровненным по 4-байтовой границе, иначе при обращении к нему резко уменьшится производительность. Зная это, компилятор выделяет для выравнивания 3 байта неиспользуемого пространства внутри каждого объекта `X`, так что стоимость добавления указателя оказывается равной 7 байтам. Если требуется и обратный указатель, то общие накладные расходы достигнут 14 байт на 32-битовой машине и 30 байт на 64-битовой.

Каким образом можно избежать этих накладных расходов? Говоря коротко — избежать их невозможно, но можно их минимизировать.

Имеется один совершенно безрассудный путь для устранения накладных расходов, который вы не должны никогда использовать (и даже говорить кому-либо, что слышали о нем от меня), и корректный, но непереносимый путь их минимизации. Обсуждение пути устранения накладных расходов, как слишком некорректного, вынесено во врезку "Безответственная оптимизация и ее вред".

Тогда, и только тогда, когда указанное небольшое различие в выделяемой памяти действительно играет важную роль в вашей программе, вы можете воспользоваться корректным, но непереносимым путем, использовав специфичные для данного компилятора директивы `#pragma`. Во многих компиляторах допускается возможность отмены выравнивания по умолчанию для данного класса (как это делается в конкретном компиляторе, вы узнаете из соответствующей документации). Если ваша целевая платформа рекомендует, но не заставляет использовать выравнивание, а компилятор поддерживает возможность отмены выравнивания по умолчанию, вы сможете избежать перерасхода памяти за счет (возможно, очень небольшого) уменьшения производительности, поскольку использование невыровненных указателей несколько менее эффективно. Но перед тем как приступать к такого рода действиям, всегда помните главное правило — *сначала сделай решение правильным, а уж потом быстрым*. Никогда не приступайте к оптимизации — ни в плане памяти, ни в плане скорости работы, — пока ваш профайлер и прочий инструментарий не подскажут вам, что вы должны этим заняться.

2. Каково влияние использования идиомы Pimpl на производительность?

Использование идиомы Pimpl приводит к снижению производительности по двум основным причинам. Первая из них состоит в том, что при каждом создании и уничтожении объекта `X` происходит выделение и освобождение памяти для объекта `XImpl`, что является относительно дорогой операцией.[5] Вторая причина заключается в том, что каждое обращение к члену в Pimpl-классе требует как минимум одного уровня косвенности; если же скрытый член использует обратный указатель для вызова функции из видимого класса, уровень косвенности увеличивается.

Каким образом можно обойти эти накладные расходы? Путем использования идиомы Fast Pimpl, о котором рассказывается далее (имеется еще один способ, которым лучше никогда не пользоваться, — см. врезку "Безответственная оптимизация и ее вред").

[5] По сравнению с другими типичными операциями C++, типа вызова функции. Замечу, что я говорю о стоимости использования распределителя памяти общего назначения, обычно вызываемого посредством встроенного `::operator new()` или `malloc()`.

3. Рассмотрите код из третьей попытки.

Вкратце можно сказать одно: так поступать не надо. C++ не поддерживает непрозрачные типы непосредственно, и приведенный код — непереносимая попытка (я бы даже сказал — "хак") обойти это ограничение. Почти наверняка программист хотел получить нечто иное, а именно — идиому Fast Pimpl.

Вторая часть третьего вопроса звучит так: **не могли бы вы предложить лучший способ избежать накладных расходов?**

Главное снижение производительности связано с использованием динамического выделения памяти для Pimpl-объекта. Вообще говоря, корректный путь обеспечить высокую производительность выделения памяти для определенного класса — это обеспечить его специфичным для данного класса оператором `new()` и использовать распределитель памяти фиксированного размера, поскольку такой распределитель может быть сделан существенно более производительным, чем распределитель общего назначения.

```
// Файл x.h
class X
{
    /* ... */
    struct XImpl;
    XImpl* pimpl_;
};

// Файл x.cpp
#include "x.h"
struct X::XImpl
{
    /* ... */
    static void* operator new( size_t )   { /* ... */ }
    static void  operator delete( void* ) { /* ... */ }
};
X::X() : pimpl_( new XImpl ) {}
X::~X() { delete pimpl_; pimpl_ = 0; }
```

"Ага! — готовы сказать вы. — Вот мы и нашли священную чашу Грааля — идиому Fast Pimpl!" Да, но подождите минутку и подумайте, как работает этот код и во что обойдется его использование.

Уверен, что в вашем любимом учебнике по программированию или языку C++ вы найдете детальное описание того, как создать эффективные функции выделения и освобождения памяти фиксированного размера, так что здесь я не буду останавливаться на этом вопросе, а поговорю об используемости данного метода. Один способ состоит в размещении функций распределения и освобождения в обобщенном шаблоне распределения памяти фиксированного размера примерно таким образом.

```
template<size_t S>
class FixedAllocator
{
public:
    void* Allocate( /* Запрашиваемый объем памяти
                       всегда равен S */ );
    void  Deallocate( void* );
private:
    /* Реализация с использованием статических членов */
};
```

Поскольку закрытые детали скорее всего используют статические члены, могут возникнуть проблемы при вызове `Deallocate` из деструкторов статических объектов. Вероятно, более безопасный способ — использование синглтона, который управляет отдельным списком для каждого запрашиваемого размера (либо, в качестве компромисса, поддерживает списки для наборов размеров, например, один список для блоков размером от 0 до 8 байт, второй — от 9 до 16 и т.д.).

```
class FixedAllocator
{
public:
    static FixedAllocator& Instance();
    void* Allocate( size_t );
    void  Deallocate( void* );
private:
    /* Реализация с использованием синглтона,
       обычно с более простыми, чем в приведенном
       ранее шаблонном варианте, статическими
       членами */
};
```

Давайте добавим вспомогательный базовый класс для инкапсуляции вызовов (при этом производные классы наследуют базовые операторы).

```
struct FastArenaObject
{
    static void* operator new( size_t s )
    {
        return FixedAllocator::Instance()->Allocate(s);
    }
    static void operator delete( void* p )
    {
        FixedAllocator::Instance()->Deallocate(p);
    }
};
```

Теперь мы можем легко написать любое количество объектов Fast Pimpl.

```
// Требуется, чтобы это был объект Fast Pimpl?
// Это очень просто - достаточно наследования...
struct X::XImpl : FastArenaObject
{
    /* Закрытые члены */
}
```

Применяя эту технологию к исходной задаче, мы получим вариант попытки □2 из условия задачи.

```
// Файл y.h

class X;
class Y
{
    /* ... */
    X* px_;
};

// Файл y.cpp

#include "x.h" // X наследует FastArenaObject
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Однако будьте осторожны и не используйте эту идиому бездумно где только можно. Да, она дает увеличение скорости, но не следует забывать о том, какой ценой это делается. Работа с отдельными списками памяти для объектов различного размера обычно приводит к неэффективному использованию памяти в силу большей, чем обычно, ее фрагментации.

И последнее напоминание: как и в случае любой другой оптимизации, использовать идиому Pimpl вообще и Fast Pimpl в частности следует только тогда, когда профилирование программы и опыт программиста доказывают реальную необходимость повышения производительности в вашей ситуации.

Рекомендация

Избегайте использования встроенных функций и настройки кода, пока необходимость этого не будет доказана путем профилирования программы.

Безответственная оптимизация и ее вред

Приведенное далее решение, казалось бы, позволяет избежать излишних накладных расходов при использовании идиомы Pimpl как с точки зрения используемой памяти, так и с точки зрения производительности. Иногда такой метод даже рекомендуют использовать, хотя совершенно зря.

Это совершенно бездумный, небезопасный и вредный метод. Вообще не следовало бы упоминать о нем, но мне приходилось видеть людей, которые поступали следующим образом.

```
// Файл x.h
class X
{
    /* ... */
    static const size_t sizeofximpl
                    = /* Некоторое значение */
    char pimpl_[sizeofximpl];
}:

// Файл реализации x.cpp
#include "x.h"
X::X()
{
    assert( sizeofximpl >= sizeof( XImpl ) );
    new( &pimpl_[0]) XImpl;
}

X::~X()
{
    (reinterpret_cast<XImpl*>(&pimpl_[0]))->~XImpl();
}
```

НЕ ДЕЛАЙТЕ ЭТОГО! Да, используя этот метод, вы сохраняете память — в количестве, достаточном для размещения целого указателя.[6] Да, вы избегаете снижения производительности — в коде нет ни одного вызова `new` или `malloc`. Может даже случиться так, что этот метод заработает при использовании текущей версии вашего конкретного компилятора.

Но этот метод абсолютно непереносим и может испортить всю вашу программу, даже если сначала покажется, что он отлично работает. Тому есть несколько причин.

1. *Выравнивание.* Любая память, выделяемая посредством вызова `new` или `malloc`, гарантированно оказывается корректно выровненной для объекта любого типа, но на буферы, не выделенные динамически, эта гарантия не распространяется.

```
char* buf1 = (char*)malloc( sizeof(Y) );
char* buf2 = new char[ sizeof(Y) ];
char buf3[ sizeof(Y) ];
new (buf1) Y;       // OK, buf1 выделен динамически
new (buf2) Y;       // OK, buf2 выделен динамически
new (&buf3[0]) Y;  // Ошибка, buf3 может быть не выровнен
(reinterpret_cast<Y*>(buf1))->~Y();      // OK
```

[6] При использовании этого метода Pimpl-класс оказывается полностью скрыт, но очевидно, что клиентский код должен быть перекомпилирован при изменении `sizeofximpl`.

```
(reinterpret_cast<Y*>(buf2))->~Y();       // OK
(reinterpret_cast<Y*>(&buf3[0]))->~Y();   // Ошибка
```

Чтобы не оставалось неясностей: я не рекомендую использовать не только третий, но и первые два метода тоже. Я просто указываю корректность первых двух вариантов с точки зрения выравнивания, но никоим образом не призываю использовать их для получения Pimpl без динамического выделения и не говорю о корректности с этой точки зрения.[7]

2. *Хрупкость.* Автор `X` должен быть чрезвычайно осторожен при работе с обычными функциями класса. Например, `X` не должен использовать оператор присваивания по умолчанию и должен либо запретить присваивание вовсе, либо снабдить класс собственным оператором присваивания. (Написать безопасный `X::operator=()` не так сложно, так что я оставлю это занятие читателям в качестве упражнения. Не забудьте только о безопасности исключений в этом операторе и в `X::~X`.[8] Когда вы выполните это упражнение, я думаю, вы согласитесь, что мороки с ним гораздо больше, чем кажется на первый взгляд.)
3. *Стоимость сопровождения.* При увеличении `sizeof(XImpl)` и превышении значения `sizeofximpl` программист должен не забыть увеличить последнее, что усложняет сопровождение программы. Конечно, можно заранее выбрать заведомо большое значение для `sizeofximpl`, но это приведет к неэффективному использованию памяти (см. п. 4).
4. *Неэффективность.* При `sizeofximpl > sizeof(XImpl)` наблюдается бесполезный перерасход памяти. Его минимизация приводит к увеличению сложности сопровождения кода (см. п. 3).
5. *Хакерство.* Из приведенного кода очевидно, что программист пытается сделать "нечто необычное". Из моего опыта "необычное" почти всегда означает "хак". Так что когда вы видите нечто подобное — размещение объекта в массиве символов или реализацию присваивания путем явного вызова деструктора с последующим размещением объекта (см. задачу 9.4), — вам следует решительно сказать "Нет!".

Основной вывод — C++ непосредственно не поддерживает непрозрачные указатели, и попытки обойти это ограничение к добру не приводят.

---

[7] Ну хорошо, я признаюсь: есть способ (хотя и не очень переносимый, но вполне безопасный) обеспечить размещение Pimpl-класса подобным образом. Этот способ включает создание структуры `max_align`, которая гарантирует максимальную степень выравнивания, и определение Pimpl-члена как `union { max_align dummy; char pimpl_[sizeofximpl]; };` — это гарантирует достаточное выравнивание. Детальную информацию вы можете найти, осуществив поиск "`max_align`" в Web или на DejaNews. Но я еще раз повторяю, что крайне не рекомендую идти по этому скользкому пути, так как применение `max_align` решает только один вопрос и оставляет нерешенными остальные четыре. И не говорите, что вас не предупреждали!

[8] Для чего обратитесь к задачам 2.1–2.10.