Основы разработки на C++. shared_ptr и RAII

Оглавление

Умные указатели. Часть 2

1.1 Умный указатель shared_ptr

Давайте, собственно, посмотрим на то, как `shared_ptr` работает. У нас есть некоторый код, который мы использовали для демонстрации возможностей `unique_ptr`. Давайте его запустим.

```
#include <iostream>
#include <memory>

using namespace std;

struct Actor {
    Actor() { cout << "I was born! :)" << endl; }

    ~Actor() { cout << "I died :(" << endl; }

    void DoWork() { cout << "I did some job" << endl; }
};

void run(Actor* ptr) {
    if (ptr) {
        ptr->DoWork();
    } else {
        cout << "An actor was expected :(" << endl;
    }
}

int main() {
    auto ptr = make_unique<Actor>();
    run(ptr.get());
    auto ptr2 = move(ptr);
    run(ptr2.get());
    run(ptr.get());
    return 0;
}
```

Так, мы видим, что он создает `Actor`, дальше его вызывает несколько раз и после этого `Actor` удаляется. Хорошо! В принципе все как и раньше.

Если здесь мы создавали именно `unique_ptr`, то давайте начнем с того, что вместо `unique_ptr` будем создавать `shared_ptr`. И если `unique_ptr` мы создавали с помощью функции `make_unique`, то `shared_ptr`, как вы можете догадаться, мы будем создавать с помощью функции `make_shared`.

```
int main() {
    auto ptr = make_shared<Actor>();
    ...
}
```

Соберем программу таким образом. Она у нас действительно собралась. Запустим, и она отработала точно таким же образом. Что приводит нас к первому важному выводу: `shared_ptr` умеет делать всё то же самое, что и `unique_ptr` и кое-что ещё. Соответственно, давайте посмотрим, что ещё он может делать.

Для начала вспомним немного пример — как то, что у нас выводится соответствует тому, что у нас написано в коде. Значит, сначала у нас создается `Actor`, дальше дважды выполняется работа, а дальше у нас «An actor was expected», то есть передается нулевой `Actor`.

Соответственно `unique_ptr` у нас копировать было нельзя. Главное отличие `shared_ptr` в том, что `shared_ptr` копировать можно, потому что `shared_ptr` разделяемый. Несколько объектов `shared_ptr` могут ссылаться на один и тот же объект, и это совершенно нормально.

Поэтому, чтобы превратить перемещение в копирование мы займемся тем, что удалим функцию `move`. Теперь `ptr2` является копией `ptr`. Давайте соберем такой код.

```
...
auto ptr2 = ptr;
...
```

Видим, что он у нас успешно собрался. Запустим, и как мы видим теперь: последний вызов функции `run` у нас точно так же вызывает у `Actor` некоторую работу, то есть он продолжает ссылаться на тот же самый `Actor`. Мы видим, что две функции `run` вызваны для `ptr2` и `ptr` они обе отрабатывают, обе вызывают вывод в консоль, поэтому `ptr` и `ptr2` они оба одновременно указывают на этот объект. Это в общем то то, что нам и нужно было получить. Этих знаний о `shared_ptr` нам с вами на самом деле уже достаточно, для того, чтобы реализовать наши новые требования.

1.2 shared_ptr в дереве выражения

Теперь, когда мы с вами познакомились с возможностями `shared_ptr`, а конкретно с главной его возможностью, что его в отличие от `unique_ptr` можно копировать, мы готовы изменить решение задачи для дерева выражений таким образом, чтобы удовлетворить наше новое требование. В авторском решение с помощью `unique_ptr` все умные указатели переделаем с `unique_ptr` на `shared_ptr`. И дальше у нас есть функции, которые создают новые объекты. Они у нас вызывают `make_unique`, а нам нужно использовать `make_shared`. И как мы помним, `shared_ptr` умеет делать всё то же самое, что и `unique_ptr`. Поэтому такую программу мы уже можем собрать и запустить — она у нас будет выводить все то же самое, но при этом она работает уже на `shared_ptr`. И теперь вопрос: как нужно изменить функцию `main`, чтобы повторное обращение функции `Print` для `e1` у нас выводило не строчку о том, что там нулевой указатель, а чтобы оно точно так же снова выводило содержимое дерева `e1`?

```
int main() {
    ExpressionPtr e1 = Product(Value(2), Sum(Value(3), Value(4)));
    Print(e1.get());

    ExpressionPtr e2 = Sum(move(e1), Value(5));
    Print(e2.get());

    Print(e1.get());
}
```

Почему у нас вообще там выводилось сообщение о том, что у нас нулевое выражение? Потому что мы переместили из указателя `e1`. В случае с `unique_ptr` выхода у нас не было, нам нужно было перемещать. Но у нас же теперь `shared_ptr`, соответственно нам нужно вместо перемещения использовать копирование.

И ответ очень простой: нам нужно удалить функцию `move`. То есть теперь вместо перемещения у нас будет использоваться копирование.

Отлично, давайте посмотрим, как работает такое исправление. Компилируем, запускаем нашу программу и видим, что всё отлично работает. То есть мы снова обратились по указателю `e1` и напечатали исходное дерево. Хорошо.

Давайте теперь проверим, что у нас действительно при удалении указателя `e2` наше поддерево, на которое указывает `e1`, останется без изменений. То есть, что `shared_ptr` в данном случае лучше, чем сырой указатель.

Так. Давайте заключим создание `e2` и его распечатывания в блок, и как мы знаем, у нас `e2` будет уничтожен по выходу из блока. Проверим, что у нас `e1` будет продолжать работать.

```
    ...
    {
        ExpressionPtr e2 = Sum(move(e1), Value(5));
        Print(e2.get());
    }
    ...
```

Соберем такую программу и запустим её. И видим, что у нас вывод корректный, то есть действительно у нас то поддерево, на которое указывает `e1`, не пострадало, несмотря на то, что `e2` удалилось. То есть все работает нормально.

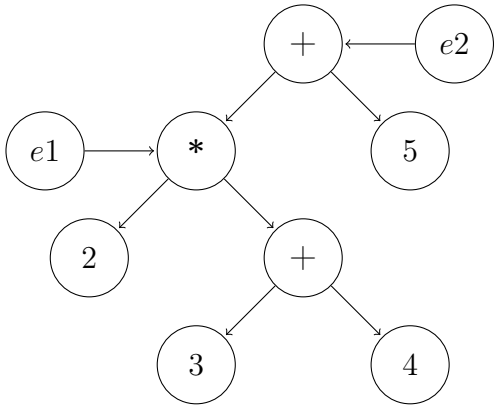
Но вот уберем этот блок. А теперь смотрите, на самом деле мы можем делать даже более интересные вещи. Например, давайте заведем такой `ExpressionPtr e3`, который будет равен сумме `e1` и `e2`, и напечатаем уже вот такой `e3`.

```
    ...
    ExpressionPtr e3 = Sum(e1, e2);
    Print(e3.get());
```

И теперь вопрос к вам: как вы думаете, что напечатает вот эта последняя строчка `Print(e3)`?

Давайте соберем программу и посмотрим, что же она на самом деле напечатает. Так, мы запускаем и видим, что она работает на самом деле абсолютно логично: `e1` мы знаем, `e2` мы тоже знаем. Значит, `e1 + e2` — это просто их сумма. А то, что на самом деле там дерево `e1` входит в

это выражение дважды, это сути дела не меняет, всё и так отлично работает, как раз потому что `shared_ptr` отлично умеет указывать на один и тот же узел из нескольких мест. Хорошо, давайте теперь рассмотрим чуть подробнее, как же у нас эти узлы расположены в памяти и друг на друга ссылаются.



Теперь давайте посмотрим на то, как у нас происходит удаление `e2`.

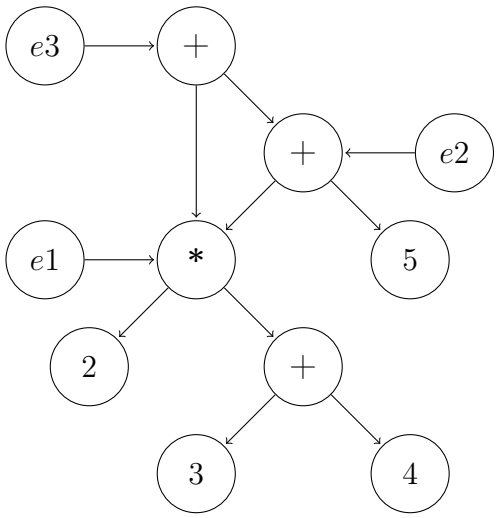
1. Итак, начинается удаление e2. Это `shared_ptr`, он указывает на узел `SumExpr`, начинает удалять `SumExpr`.

2. `SumExpr` при своем удалении начинает удалять свои поля, начинает с поля `right`. Дальше он начинает удалять поле `left`.

3. Поле `left` ссылается на узел умножения. И вот тут самое интересное: `shared_ptr` видит, что на этот узел умножения кроме него ссылается еще другой `shared_ptr` — `e1`, и поэтому он не будет удалять этот узел. Он просто удалится сам, а узел трогать не будет.

На этом удаление `SumExpr` заканчивается, и удаление `e2` тоже заканчивается. А поддерево `e1` у нас остается без изменений.

Теперь давайте посмотрим на вот этот наш пример, когда мы сделали сумму выражений `e1` и `e2`. Если на `unique_ptr` у нас было дерево, то на `shared_ptr` мы смогли сделать направленный ациклический граф. При этом, опять же, если в случае с `unique_ptr` у нас получался рекурсивный обход, абсолютно бесплатный — компилятор сам его делал, здесь точно так же: то, что это граф, и то, что в нем нет циклов, нам гарантирует просто семантика работы с `shared_ptr`. Вот таким, по сути, тривиальным изменением нашей реализации: просто заменив `unique_ptr` на `shared_ptr`, мы смогли реализовать новое требование.



1.3 Внутреннее устройство умных указателей

Теперь давайте посмотрим на то, как умные указатели устроены внутри и как у них получается делать то, что они умеют делать.

Начнем с создания `unique_ptr` на примере вызова функции `make_unique`.

1. Мы вызываем функцию `make_unique`, она создает новый динамический объект T и получает сырой указатель, который на него ссылается.

2. Затем она создает объект собственно `unique_ptr` и помещает в него этот указатель. Сам указатель ей больше не нужен.

3. Этот `unique_ptr` она отдает наружу.

Вот, в общем-то, и все. Так незамысловато это устроено. `unique_ptr` действительно очень простой. Указатель на объект — это единственное, что он хранит внутри себя.

Посмотрим, как происходит перемещение `unique_ptr`.

1. У нас создается новый `unique_ptr`, в который мы перемещаем.

2. Мы берем этот указатель копируем его в новый `unique_ptr` и зануляем его в исходном `unique_ptr`.

Вот так у нас произошло перемещение. Получилось, что в исходном `unique_ptr` у нас ничего не осталось, а новый `unique_ptr` указывает на тот же самый объект.

Хорошо, теперь посмотрим, как происходит разрушение `unique_ptr`.

1. Начнем с `unique_ptr`, который никуда не указывает. Он собственно берет и умирает, потому что у него указатель никуда не указывает, поэтому никаких дополнительных действий не происходит.

2. Теперь, как у нас умирает `unique_ptr`, который куда-то указывает? В своем деструкторе он смотрит: так вот указатель, который у меня лежит, он куда-то указывает? В данном случае да, он указывает на динамический объект. Ну отлично. Значит, он вызывает для него delete и удаляет этот динамический объект, после чего удаляется собственно сам `unique_ptr`.

Вот так, с `unique_ptr` все достаточно просто.

Теперь давайте посмотрим, как у нас устроен `shared_ptr`. Начнем с создания `shared_ptr`, например, при вызове функции `make_shared`.

1. Начало точно такое же: у нас создается динамический объект T, оператор new возвращает нам сырой указатель на этот динамический объект, затем создается объект `shared_ptr`, и этот указатель помещается в `shared_ptr`.

А вот после этого происходит интересное. Смотрите, мы помним, что `shared_ptr` умеет некоторым образом отвечать на вопрос: а существуют ли еще другие `shared_ptr`, которые указывают на тот же самый объект, то есть у него есть некоторая дополнительная информация. Давайте подумаем, где эта дополнительная информация может находиться.

У нас есть сам объект. То есть можно попытаться положить эту информацию в сам объект. Но у нас объект произвольный, по идее его вообще не должно волновать, что он был создан с помощью функции `make_shared`. Поэтому когда мы говорим о стандартных умных указателях, там в сам объект мы ничего добавить не можем, потому что объект абсолютно произвольный.

У нас есть объект `shared_ptr`, который создался. Казалось бы, в принципе, вроде бы да, мы можем положить туда произвольную информацию. Однако мы помним, что `shared_ptr` можно будет скопировать, и скопировать несколько раз. Эти `shared_ptr`, куда мы скопировали, будут жить, а исходный `shared_ptr` у нас уже умрет. Получается, что если мы положим какую-то информацию в этот `shared_ptr`, то тогда она в какой-то момент нам станет уже недоступна. То есть туда ее складывать тоже нельзя.

В сам объект складывать нельзя, в `shared_ptr` складывать нельзя. Что нам остается? Создать некоторое новое место. И действительно, `shared_ptr` **создает в куче новый небольшой утилитарный объект под названием** `ControlBlock`. В этом контрольном блоке он сохраняет счетчик ссылок, то есть текущее количество `shared_ptr`, которые указывают на этот динамический объект. А у себя, в объекте `shared_ptr`, он сохраняет только указатель на этот контрольный блок. Временный указатель нам уже не нужен. Получается, что **в** `shared_ptr` **у нас есть два указателя: и на сам объект, и на контрольный блок**. Это очень важно.

Давайте теперь посмотрим, как происходит копирование `shared_ptr`.

1. Создается новый объект `shared_ptr`. В него копируются указатели на тот же самый объект.

2. В него копируются указатели на тот же самый контрольный блок.

3. Дальше, поскольку у нас создался новый `shared_ptr`, который ссылается на тот же самый объект, нам нужно увеличить счетчик ссылок, что и происходит.

4. Мы увеличим счетчик ссылок, теперь он равен 2. Теперь у нас два `shared_ptr` указывают на один и тот же объект, и что характерно, если мы пойдем в любой из этих `shared_ptr`, мы сможем попасть в один и тот же контрольный блок, где будет записана вот эта информация.

Хорошо, давайте посмотрим, как происходит перемещение `shared_ptr`.

1. Создается новый `shared_ptr`, в него копируется указатель из того `shared_ptr`, из которого мы перемещаем. А в исходном `shared_ptr` этот указатель зануляется, как и в случае с `unique_ptr`.

2. И то же самое происходит с указателем на контрольный блок. Он копируется и зануляется в исходном `shared_ptr`.

3. При этом обратите внимание, что счетчик ссылок у нас не меняется, поскольку действительно у нас один `shared_ptr` стал указывать на этот объект, а другой перестал указывать на этот объект. Поэтому перемещение `shared_ptr` — это более эффективная операция, чем его копирование. И в принципе, как всегда, **перемещение — это в некотором смысле оптимизация операции копирования**.

Давайте теперь посмотрим на разрушение `shared_ptr`. Начнем с простой ситуации, когда разрушается `shared_ptr`, который никуда не указывает.

1. Как и в случае с `unique_ptr`, он просто берет и уничтожается. Это не влечет за собой никаких последствий.

Следующим примером рассмотрим удаление `shared_ptr`, когда существует какой-то другой `shared_ptr`, указывающий на тот же самый объект.

1. Начинает разрушаться `shared_ptr`, и в своем деструкторе он идет в контрольный блок, и на этот раз он уменьшает счетчик ссылок. Их было две, он уменьшает до одной.

2. Смотрит: а есть там еще какие-то ссылки? Да, счетчик не упал до нуля. Если счетчик не упал до нуля, значит, кто-то еще указывает на тот же самый объект, и значит, нам ничего делать больше не нужно.

3. После этого мы просто берем и уничтожаем текущий объект `shared_ptr`.

И самый интересный пример — это у нас удаляется последний `shared_ptr`, указывающий на данный объект.

1. Он начинает удаляться, он идет в контрольный блок и уменьшает счетчик ссылок.

2. Видит, что счетчик ссылок на этот раз упал до нуля. Значит, он последний, все, больше никого нет. За нами Москва. Нужно за собой прибраться. Первым делом он удаляет контрольный блок. В конце концов, контрольный блок он сам же и завел, для того чтобы отследить количество ссылок.

3. После этого он, очевидно, удаляет объект, поскольку именно этим занимаются умные указатели — они удаляют объекты.

4. И вот теперь, когда он за собой прибрался, он может быть удален сам.

На самом деле всё, конечно же, не так просто. На самом деле `unique_ptr` работает несколько сложнее, `shared_ptr` работает намного сложнее, чем было описано. Но это некоторая модель, которой на самом деле вам более чем достаточно для понимания большинства случаев работы с умными указателями, и которая вам позволит решать большинство практических задач.

1.4 Владение

На текущий момент мы с вами уже довольно подробно познакомились с управлением памятью в C++. Настало время несколько структурировать наши знания. Давайте посмотрим, какие вообще в C++ бывают виды объектов с точки зрения времени их жизни.

1. У нас бывают **автоматические** объекты, временем жизни автоматических объектов управляет компилятор. К автоматическим объектам относятся локальные переменные, глобальные переменные и члены классов.

2. Также у нас бывают **динамические** объекты. Временем жизни динамических объектов управляет программист. Мы уже знаем, что динамические объекты создаются с помощью ключевого слова `new`, которое используется в недрах функций `make_unique` и `make_shared`. А удаляются они с помощью ключевого слова `delete`, которое опять же используется в деструкторах умных указателей.

3. Также в C++ существует понятие **владения**. Считается, что некоторая сущность владеет объектом, если она отвечает за его удаление.

Давайте посмотрим, кто владеет разными видами объектов.
Для **автоматических** объектов:

- если говорить о локальных переменных, то локальными переменами владеет окружающий их блок кода. Действительно, когда поток управления программы выходит из блока кода, мы знаем, что уничтожаются все локальные переменные, которые были объявлены в этом блоке.

- Глобальные переменные: можно считать, что ими владеет сама программа. Мы знаем, что когда программа завершается, уже после завершения функции `main`, происходит удаление всех глобальных переменных.

- А членами класса владеет сам объект класса. Действительно, мы никогда не задумываемся о том, когда нам нужно удалять члены класса. Мы знаем, что в любой момент, как бы это ни произошло, когда будет уничтожен сам объект класса, он позаботится о том, чтобы в своем деструкторе удалить свои члены.

И с автоматическим объектами: эти правила, которые мы здесь перечислили, они очень четкие, и им следует компилятор. Они всегда работают ровно так.

С **динамическими** объектами ситуация несколько интереснее. Как вы думаете, кто владеет динамическими объектами? Как вы можете догадаться по названию данного блока, динамическими объектами владеют умные указатели. При этом

- `unique_ptr` обеспечивает уникальное, эксклюзивное владение объектами. На один объект может указывать только один `unique_ptr`, и один `unique_ptr` может указывать только на один объект.

- `shared_ptr` — это разделяемое, или совместное, владение. На один и тот же объект может указывать несколько `shared_ptr`.

- А вот **сырой указатель не владеет объектом**. Считается, что если у нас есть сырой указатель, мы его получили каким-то образом, то мы никаким образом не отвечаем за время жизни этого объекта, на который он указывает.

И вот тут ситуация достаточно интересна с динамическими объектами в том плане, что то, что мы сейчас перечислили, это соглашение. Это не совсем правила. И этим соглашениям должен следовать программист. При этом программист, вообще говоря, может их нарушить.

Давайте посмотрим на эти соглашения в действии на конкретном примере кода.

```
unique_ptr<Animal> CreateAnimal(const string& name) {
    if (name == "Tiger")
        return make_unique<Tiger>();
    if (name == "Wolf")
        return make_unique<Wolf>();
    if (name == "Fox")
        return make_unique<Fox>();
    return nullptr;
}
```

Нам даже не нужно заглядывать внутрь этой функции, чтобы понять, что эта функция создает объект и возвращает нам его во владение. Потому что она возвращает нам `unique_ptr`. Теперь мы у себя сохраняем `uniqie_ptr`, мы как-то им пользуемся, и владение находится на нашей стороне. Эта функция следует данным соглашениям.

Следующий пример

```
class Shape {
    shared_ptr<Texture> texture_;

public:
    Shape(shared_ptr<Texture> texture) : texture_(move(texture)) {}
};
```

Из сигнатуры конструктора мы можем понять, что новый объект фигуры, который создан с помощью этого конструктора, он будет находиться в совместном владении этой текстурой. Этот пример также следует нашим соглашениям.

Следующий пример с использованием сырого указателя.

```
void Print(const Expression* e) {
    if (!e) {
        cout << "Null expression provided" << endl;
        return;
    }
    cout << e->ToString() << "=" << e->Evaluate() << endl;
}
```

Функция, разумеется, не удаляет объект и ничего не делает с ним, что относилось бы к времени жизни этого объекта. То есть указатель, который она получила, — не владеющий.

Таким образом, существуют соглашения по владению динамическими объектами. Этим соглашениям следует программист. И эти соглашения могут быть нарушены. Они могут быть нарушены по ошибке, например, из-за простого незнания, или они могут быть нарушены по необходимости. Необходимость может возникнуть в том случае, если нам нужно ручное, низкоуровневое управление динамической памятью. Например, мы пишем свой контейнер.

В случае соблюдения этих соглашений, вы можете гарантировать, и это очень важно, и это то, зачем люди придерживаются данных соглашений, что в вашей программе не может быть никаких утечек памяти и не может быть двойного удаления объекта. То есть если вы следуете этим соглашениям, у вас такие ситуации просто теоретически невозможны. Нельзя написать такой код.

Однако если вы эти соглашения нарушаете, то все может сработать корректно, но при определенных ситуациях вы можете создать программы, в которых будут утечки памяти или двойное удаление. Давайте с вами посмотрим примеры, какие именно могут возникать проблемы.

Вот первый пример такой программы. Вопрос: как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
}
```

Это очень простая программа, здесь мы создаем динамический объект и потом его не удаляем. Очевидно, здесь будет утечка. Причем в данной программе мы нарушили наше соглашение — мы получили владеющий сырой указатель, но нам нужно было на нём вызвать `delete`, что мы забыли сделать.

Посмотрим следующий пример. Вот такая программа, чуть-чуть посложнее. Как вы думаете, как она отработает?

```
struct A {/*...*/};

void UseA(int x) {
    A* ptr = new A;
    if (x < 0) {
        return;
    }
    delete ptr;
}

int main() {
    UseA(-1);
}
```

В этой программе мы уже вызываем `delete`, после того как мы вызвали `new`, но дело в том, что до вызова `delete` у нас присутствует некоторое условие, причем в результате выполнения этого условия мы можем выйти из функции, и `delete` не будет вызван. Здесь функция вызывается как

раз с таким аргументом, что у нас условие будет выполнено и до `delete` мы не дойдем. То есть в данном случае у нас опять же будет утечка.

Этот пример может выглядеть немного надуманным, но на практике вот именно такая проблема и встречается чаще всего. Как это происходит? Функция обычно начинается с того, что она достаточно небольшая. Где-то в начале этой функции мы объект создаем, в конце мы его удаляем, сама функция в пять строчек — очевидно, что всё будет хорошо. Никакой утечки не будет. Потом ваш проект эволюционирует, эта функция увеличивается. В ней в какой-то момент становится уже сотни строчек, на ней начинают работать люди, которые даже не знают тех, кто изначально ее написал, и дальше в какой-то момент кто-то вставляет посередине вот такое условие — с досрочным выходом из функции. И забывает посмотреть, что у нас где-то там находится `delete`, его тоже нужно вызвать. И у нас получается вот такая ситуация — возникает утечка. Здесь мы нарушили то же самое соглашение, что и в предыдущем примере: мы создали владеющий сырой указатель, потому что здесь мы как бы сами его удаляем, то есть мы оставляем на нем ответственность за то, чтобы объект был удален.

Посмотрим следующий пример. Здесь у нас уже появляются умные указатели. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
    unique_ptr<A> up1(ptr);
    unique_ptr<A> up2(ptr);
}
```

Несмотря на то, что мы здесь используем умные указатели, все равно программа отработает неправильно. Недостаточно просто использовать умные указатели, нужно использовать их правильно. Что у нас здесь происходит? Мы создаем новый динамический объект, сохраняем его в сыром указателе. А потом передаем этот сырой указатель в конструктор двух `unique_ptr`. И у нас получается, что два `unique_ptr` указывают на один и тот же объект. Но `unique_ptr` — это очень простой класс, он в своем деструкторе смотрит: объект есть какой-то, на который я указываю? Есть. Значит, я его удалю. И оба `unique_ptr` в данном случае попытаются удалить объект. Это приведет к некорректной работе программы. Здесь мы нарушили соглашение об эксклюзивном владении `unique_ptr`. Мы знаем, что одним объектом должен владеть только один `unique_ptr`. Здесь мы его нарушили и получили проблему.

Посмотрим следующий пример. Здесь все то же самое, но вместо `unique_ptr` используются `shared_ptr`. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

int main() {
    A* ptr = new A;
    shared_ptr<A> sp1(ptr);
    shared_ptr<A> sp2(ptr);
}
```

Несмотря на то, что, казалось бы, `shared_ptr` как раз и нужен для того, чтобы организовать разделяемое владение и несколько `shared_ptr` действительно могут ссылаться на один и тот же объект, здесь ситуация гораздо более интересная. Мы создали новый объект, поместили его в сырой указатель. И опять же, мы передаем сырой указатель в конструктор двух `shared_ptr`. Когда мы создаем первый `shared_ptr`, он видит: сырой указатель, отлично, создам для него контрольный блок, все хорошо. Когда мы создаем второй `shared_ptr`, у него нет никакой возможности узнать, что есть уже какой-то другой `shared_ptr`, который указывает на тот же самый объект, поэтому он спокойно создаст еще один контрольный блок и будет указывать на тот же самый объект. И у нас получится два никак не связанных `shared_ptr`, каждый со своим контрольным блоком, которые указывают на один и тот же объект. И каждый из них, когда будет удаляться, будет считать, что на этот объект больше никто не указывает, и попытается его удалить. У нас, опять же, здесь произойдет, как и в предыдущем примере, двойное удаление. В этой программе мы более тонко нарушили наше соглашение, на самом деле `ptr` в данном случае является у нас владеющим. Мы об этом чуть подробнее поговорим попозже.

Давайте рассмотрим следующий пример. Как вы думаете, как отработает данная программа?

```
struct A {/*...*/};

A* makeA() {
    return new A;
}

int main() {
    unique_ptr<A> up(makeA());
    shared_ptr<A> sp(makeA());
}
```

Эта программа похожа на предыдущую, но она уже отработает корректно, потому что здесь у нас создаются два умных указателя, оба создаются из сырых указателей, но каждый раз у нас создается новый динамический объект. То есть здесь у нас всё будет хорошо: `unique_ptr` будет указывать на свой объект, `shared_ptr` будет указывать на свой объект. Каждый из них его удалит. То есть вроде бы проблем нет.

Но давайте подумаем, что будет, если кто-то вызовет функцию `MakeA` и забудет передать этот указатель в конструктор умного указателя. Вот в этом случае у нас может произойти утечка, может быть точно такая же ситуация, которую мы рассмотрели в первых нескольких примерах. Поэтому несмотря на то, что эта программа работает корректно, она нарушает соглашение. Она создает владеющий сырой указатель, который возвращает функция `MakeA`. Поэтому её следует переписать с выполнением наших соглашений, например, вот таким образом:

```
struct A {/*...*/};

unique_ptr<A> makeA() {
    return make_unique<A>();
}

int main() {
```

```
    unique_ptr<A> up(makeA());
    shared_ptr<A> sp(makeA());
}
```

Это абсолютно корректная ситуация. И вот такая программа работает точно так же, корректно, но уже следует нашим соглашениям. Именно так и следует писать.

И давайте посмотрим ещё один, самый интересный пример. Как вы думаете, как отработает такая программа?

```
struct A {/*...*/};

int main() {
    auto up1 = make_unique<A>();
    unique_ptr<A> up2(up1.get());
}
```

В этой программе, несмотря на то, что мы даже не писали в явном виде `new` и `delete`, мы с вами всё равно умудрились нарушить соглашение и создать такую ситуацию, что произойдет двойное удаление.

Смотрите, что получается. Мы создаем новый динамический объект с помощью функции `make_unique`. Всё замечательно. Получаем `unique_ptr up1`. После этого мы достаем из него сырой указатель и подаем его в конструктор второго `unique_ptr`. И у нас опять получается ситуация, когда два `unique_ptr` указывают на один и тот же объект. В итоге это, конечно, приведет к двойному удалению. Идея здесь в том, что указатель, который передается в конструктор умного указателя, трактуется как владеющий. То есть в данном случае сырой указатель, который возвращает `get()`, был трактован как владеющий.

Давайте подведем небольшой итог. Мы рассмотрели соглашение по владению динамическими объектами. И после этого привели множество примеров, как эти соглашения можно нарушить. Причем некоторые из них были не совсем очевидными, как, например, последний пример. Давайте теперь сформулируем несколько практических положений, то есть что на практике означают эти соглашения, что это значит на уровне написания кода. На самом деле это означает

- Не нужно использовать `new` и `delete`, как мы это уже знаем.
- Поскольку нельзя использовать `new` для создания объектов, нужно использовать функции `make_unique` и `make_shared`.
- Нельзя использовать конструкторы умных указателей, которые принимают сырые указатели. То есть нельзя создавать умные указатели напрямую из сырых. А это ровно то, что мы сделали в последнем примере, потому что вот такой конструктор — он трактует сырой указатель, который ему передают, как владеющий, ведь он же сейчас создаст умный указатель, который будет владеть этим объектом. А кто им владел до того? Значит, это сырой указатель.

new и конструкторы скрываются от нас в недрах функций `make_unique` и `make_shared`. То есть `new` и конструкторы — это некоторый низкоуровневый инструмент управления, которым пользоваться не нужно. Вместо этого нужно пользоваться высокоуровневым — функциями `make_unique`

и `make_shared`. Это ровно то, что они делают: создают объект и конструируют умные указатели из указателя на динамический объект. А вот вызов `delete` скрывается от нас в деструкторах умных указателей. То есть, опять же, `delete` мы напрямую не вызываем. Мы полагаемся на то, что этим займутся за нас умные указатели.

Таким образом, мы с вами выяснили, что есть два вида объектов, с точки зрения времени их жизни: автоматические и динамические. Мы знаем, что **владение автоматических объектов берет на себя компилятор и существуют строгие правила, которым он следует**. А вот **владение динамическими объектами описывается соглашениями, которым следует программист**. Но эти соглашения могут быть нарушены. И их нарушение может привести к проблемам, как мы видели. Если же им следовать, то тогда мы гарантируем, что в нашей программе будут отсутствовать утечки памяти и отсутствовать двойные удаления.

1.5 Присваивание умных указателей

До сих пор мы с вами инициализировали умные указатели в момент их создания, и после этого их не меняли. Однако умные указатели можно присваивать, и периодически это бывает полезно. В римере, который мы использовали для демонстрации возможностей `shared_ptr`, мы создаем один `Actor`. Для того чтобы показать, как умные указатели присваиваются, нам понадобится парочка. Поэтому давайте дадим актору имя и будем передавать его в конструкторе, и сделаем так, чтобы у нас, когда `Actor` что-то говорит — он выводил информацию о том, кто это делает, то есть будем вводить имя.

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

using namespace std;

struct Actor {
    Actor(string name) : name_(move(name)){
        cout << name << "I was born! :)" << endl;
    }

    ~Actor() {
        cout << name << "I died :(" << endl;
    }

    void DoWork() {
        cout << name << "I did some job" << endl;
    }

    string name_;
};
```

Теперь давайте изменим нашу функцию `main`: продемонстрируем возможности присваивания на примере `unique_ptr`

```
int main() {
    auto ptr1 = make_unique<Actor>("Alice");
    auto ptr2 = make_unique<Actor>("Boris");
    run(ptr1.get());
    run(ptr1.get());
    return 0;
}
```

Мы видим достаточно ожидаемый вывод. У нас сначала создалась Алиса, потом создался Борис, они оба выполнили некоторую работу и после этого оба дружно умерли в обратном порядке. Все вполне ожидаемо. Хорошо.

Давайте теперь выполним присваивание, то есть напишем

```
    ...
    ptr1 = move(ptr2);
    run(ptr1.get());
    run(ptr1.get());
    return 0;
```

И теперь, прежде чем мы запустим программу, вопрос к вам: как вы думаете, что напечатают вот эти три строчки? Для того чтобы нам с вами было лучше видно, что же эти строчки напечатают, давайте сделаем вот такую отбивку. Чтобы в выходе их явно было видно.

```
    ...
    cout << "----" << endl;
    ptr1 = move(ptr2);
    run(ptr1.get());
    run(ptr1.get());
    cout << "----" << endl;
    return 0;
```

Соберем программу в таком виде и запустим. Итак, что же мы видим? Что первым делом после присваивания у нас удаляется Алиса. Как же так произошло?

Смотрите. У нас `ptr1` указывал на Алису, а потом `ptr1` присваивается тому, на что указывал `ptr2`. То есть получается, что на Алису в этот момент уже больше никто не указывает. Раз на нее больше никто не указывает, но она при этом управлялась умным указателем, она соответственно должна быть удалена, иначе бы она утекла. Соответственно, в этот момент она удаляется. Смотрим дальше на вывод. У нас какую-то работу выполняет Борис, и после этого вызывается `run` с нулевым актором. Ну действительно, `ptr1` у нас теперь указывает туда, куда указывал ptr2, то есть на Бориса, а из `ptr2` у нас было выполнено перемещение. То есть в нем у нас ничего не осталось. И поэтому при этом втором вызове `run` у нас выводится сообщение, что актора там нет. Все вполне логично.

Давайте теперь попробуем сформулировать точное правило: в какой момент у нас удаляются объекты, которые управляются умными указателями. До этого момента мы как бы говорили,

что умные указатели удаляют объект в своем деструкторе. И это правда, но не вся. Дело в том, что умные указатели могут не удалить объект в своем деструкторе, например, у нас есть один `shared_ptr`, он указывает на объект, он удаляется, но при этом другой `shared_ptr` продолжает указывать на тот же самый объект. Как мы прекрасно знаем, в этом случае объект не будет удален.

Кроме того, умные указатели могут удалить объект не в деструкторе, как мы это с вами только что видели, умный указатель удалил объект в момент присваивания, то есть после присваивания у нас Алиса была удалена. Наиболее точное правило будет звучать подобным образом: **динамический объект удаляется, когда им перестают владеть умные указатели**. При этом перестать владеть объектом умные указатели могут в несколько моментов.

- Во-первых, при разрушении, с этого мы собственно и начали.
- Во-вторых, при перемещении из умного указателя, это мы тоже подробно рассмотрели.
- И при присваивании умного указателя чему-то. Это мы с вами только что видели на примере присваивания `unique_ptr`.

Давайте теперь подробнее поговорим именно про присваивание, потому что в C++ их существует несколько разных видов

- Для начала **перемещающее присваивание** — ровно то, что мы сейчас с вами делали в программе. У нас `ptr1` указывает на Алису, `ptr2` указывает на Бориса. После того как мы выполняем перемещающее присваивание, у нас `ptr1` начинает указывать туда, куда указывал `ptr2`, то есть на Бориса. При этом `ptr2` у нас не указывает больше никуда, потому что мы из него переместили, а на Алису в этот момент никто не указывает. То есть в соответствии с нашей формулировкой умные указатели прекратили владение Алисой. Следовательно, в этот момент она должна быть удалена, раз на нее больше никто не указывает. Вот она и удаляется. Собственно, то, что мы с вами и видели. Это перемещающее присваивание, и оно будет одинаково работать и для `unique_ptr`, и для `shared_ptr`.
- Теперь посмотрим на **копирующее присваивание**. Копирующее присваивание, очевидно, применимо только к `shared_ptr`, потому что `unique_ptr` копировать нельзя. Пусть у нас вот такая ситуация: опять же, у нас Алиса и Борис, и три `shared_ptr`. Первый указывает на Алису, а остальные два указывают на Бориса. Мы присваиваем `ptr2 = ptr1`. После этого присваивания у нас `ptr2` начинает указывать вместо Бориса на Алису, потому что ровно туда указывал `ptr1`. Но при этом у нас на все объекты продолжают указывать какие-то умные указатели, то есть больше ничего в этот момент не происходит.

  Однако дальше давайте выполним еще одно присваивание. Теперь `ptr3` присваивается `ptr1`, и `ptr3` тоже начинает указывать на Алису вместо Бориса. Теперь Борисом больше не владеет ни один умный указатель, следовательно, в этот момент он должен быть удален, что и происходит.
- Теперь давайте посмотрим на еще один интересный вид присваивания — это **присваивание** `nullptr`. Оно, опять же, применимо и к `unique_ptr`, и к `shared_ptr`. Когда мы присваиваем умному указателю `nullptr`, он просто становится простым и прекращает владение объектом,

на который он указывал. И если на объект больше никто не указывает, следовательно, объект удаляется. **Подобное присваивание очень полезно, если нам нужно освободить владение объектом до того, как умный указатель прекратил свое существование**. То есть здесь `ptr1` у нас продолжает существовать, но больше уже никуда не указывает.

1.6 shared_ptr и многопоточность

На текущий момент мы с вами рассматривали только программы, которые работают в однопоточном режиме. Однако умные указатели и в частности `shared_ptr` можно очень часто встретить в многопоточных программах. Поэтому давайте сейчас напишем небольшой пример многопоточной программы, где `shared_ptr` используется для разделения некоторого ресурса.

```
#include <future>
#include <iostream>
#include <memory>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

class Data {
public:
    Data(string data) : data_(data) {
        cout << "Data constructed\n";
    }
    ~Data() {
        cout << "Data destructed\n";
    }

    const string& Get() const {
        return data_;
    }
    string& Get() {
        return data_;
    }
private:
    string data_;
};

void ShareResource(shared_ptr<Data> ptr) {
    cout << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << endl;
}

vector<future<void>> spawn() {
    vector<future<void>> tasks;
```

```
    auto data = make_shared<Data>("meow");

    for (int i = 0; i < 10; ++i) {
        tasks.push_back(async([=]() {
            ShareResource(data);
        }));
    }

    return tasks;
}

int main() {
    cout << "Spawning tasks...\n";
    auto tasks = spawn();
    cout << "Done spawning.\n";
}
```

Причём `data` в нашу лямбда функцию мы захватываем по значению. Потому что, как мы уже ожидаем, у нас эти задачи будут выполняться после того, как данная функция завершится, потому что мы возвращаем вектор из `future`. То есть захватывать по ссылке мы не можем. Если бы мы захватили по ссылке, мы могли бы обратиться к этому `data`, когда у нас он уже был разрушен, ведь это локальная переменная функции.

Давайте теперь запустим. Так, что у нас происходит? Чего-то он нам тут выводит, на самом деле не очень понятно чего выводит, как-то оно все вперемешку, и на самом деле он выводит нам это все вперемешку, потому что у нас несколько потоков одновременно начинают писать в `cout`. Одновременно писать в `cout` можно. То есть доступ к `cout` в принципе синхронизирован. Там не будет data race. Проблема в том, что у нас каждый вот этот оператор вывода `<<` может перекрываться в разных потоках. Получается у нас такая чересполосица. Соответственно, для того, чтобы этой чересполосицы избежать, нам нужно использовать ровно один оператор вывода.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << endl;
    cout << ss.str();
}
...
```

Соберём и запустим. Так, вот теперь мы видим, что у нас программа делает что-то осмысленное, значит, давайте читать. Мы начали создавать задачи. Затем зашли в функцию `spawn`. У нас вывод, что создались данные, а дальше у нас уже начали работать наши задачи, которые мы создали с помощью `async`. Вот мы видим, что у нас был расшарен наш ресурс из тредов с разными `id`. Дальше мы закончили исполнять, но при этом задачи продолжили выполняться, потому что это как раз ровно то, на что мы рассчитывали, что задачи будут продолжаться уже продолжать выполняться уже после того как отработала наша функция.

Итак, давайте теперь сделаем такую интересную вещь. Давайте выведем сколько у нас в дан-

ный момент существует на наши данные ссылок из различных `shared_ptr`. Для того, чтобы это сделать мы воспользуемся методом `use_count()`, который есть у `shared_ptr`. Он на практике не то, чтобы очень полезен. Потому что как раз в данном случае у нас многопоточное выполнение, и как только мы спросили некоторые `use_count()`, сразу же после этого этот `use_count()` у нас может измениться, поскольку у нас несколько потоков с ним взаимодействуют. Но при этом это будет достаточно полезно, чтобы примерно прикинуть некоторый масштаб количества использований.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << ", counter =
        " << ptr.use_count() << endl;
    cout << ss.str();
}
...
```

Запустим теперь. Мы видим, что у нас происходит с `counter`. 11, 10, 11, 8. Мы видим, что в процессе выполнения у нас счетчик действительно меняется, то есть `shared_ptr` у нас как-то копируются, умирают, то есть что-то происходит. Так, можно еще запустить. Картинка немного меняется, но общая суть остается примерно одинаковой: из разных тредов мы меняем этот счетчик. Какой отсюда можно сделать важный вывод? Несмотря на то, что мы не предпринимали никаких дополнительных усилий к тому чтобы обеспечить синхронизированный доступ к этому счетчику, программа у все равно работает корректно. Дело в том, что `shared_ptr` сам синхронизирует доступ к своему счетчику,и абсолютно нормально: из разных потоков этот счетчик менять. Поэтому мы как раз безопасно это можем сделать. А вот если мы попытаемся проделать аналогичную операцию с нашим `data`, вот тут у нас могут быть проблемы. Потому что доступ к `data` у нас не особо синхронизирован.

Так что давайте в нашем `ShareResource` попробуем взять и `data` поменять. Конкретно давайте в нашей строчке чего-нибудь допишем. И теперь давайте еще включим оптимизацию (Release), потому что, если оптимизации нет, то у нас это может не сработать.

```
...
void ShareResource(shared_ptr<Data> ptr) {
    stringstream ss;
    ss << "Shared resource" << ptr->Get() << " in " << this_thread::get_id() << ", counter =
        " << ptr.use_count() << endl;
    cout << ss.str();
    ptr->Get().push_back('x');
}
...
```

Так. Соберем программу. Мы видим, что наши данные начинают меняться и в принципе пока вроде бы оно выглядит даже нормально. Однако! Давайте увеличим количество итераций — поставим, скажем, тысячу, и давайте уберем вывод в выходной поток, потому что вывод в выходной поток на самом деле нам обеспечивает синхронизацию собственно через этот вывод, потому что сам доступ к потоку синхронизирован, и запустим теперь. Вроде сработало. А вот теперь уже не сработало. Мы видим, что наша программа упала, а упала она ровно потому, что мы из нескольких потоков

модифицируем одни и те же данные, а это как мы уже знаем состояние гонки, так делать нельзя. Поэтому наша программа упала.

Соответственно, здесь для того, чтобы избежать этой проблемы нам нужно либо синхронизировать доступ к этим данным как мы это, собственно, делали раньше, либо в явном виде запретить модификацию этого объекта из нескольких потоков, и самый удобный способ это сделать — это на самом деле сказать, что объект, который нам приходит, он константный (`const Data`).

Так, давайте вернем небольшое количество итераций и соберем программу. Запускаем. Программа у нас работает как раньше, при этом мы на уровне интерфейса `ShareResource` мы как бы ограничили себя — мы не можем модифицировать данные через функцию `ShareResource`, — то есть обезопасили себя от состояния гонки. Этот приём достаточно часто применяется на практике.

Таким образом, мы узнали, что `shared_ptr` обеспечивает потокобезопасный доступ к счетчику ссылок. То есть вы можете безопасно копировать `shared_ptr` из нескольких потоков и удалять `shared_ptr` в этих потоках. Все это приводит к модификации счетчика, но вам не нужно беспокоиться о его синхронизации.

А вот о чем вам нужно беспокоиться, так это о синхронизации доступа к самому объекту, потому что здесь `shared_ptr` никак не влияет и никак не модифицирует правила игры. Если вы хотите менять объект из нескольких потоков, тогда обеспечьте некоторую синхронизацию для этого объекта. Однако же простой способ сделать всё-таки безопасный доступ — это в явном виде запретить модификацию объекта из нескольких потоков. То есть передавать `shared_ptr` на константный объект, тогда вы не сможете его модифицировать. Нет модификации — нет проблем.

И сейчас должно быть понятно, что перемещение `shared_ptr` — это некоторая оптимизация. Как вы помните, она позволяет избежать изменений счетчика, а изменение счетчика к тому же еще и синхронизировано, а синхронизированные изменения счетчика — это несколько более дорогая операция, чем простое изменение счетчика. И поэтому за счёт перемещения `shared_ptr` мы опять же можем избежать вот такой чуть более дорогой операции.

1.7 Умный указатель weak_ptr

На текущий момент мы с вами познакомились с основными и наиболее полезными возможностями умных указателей, которые приходится часто использовать на практике. Теперь мы поговорим о некоторых дополнительных возможностях, которые на практике используются не так часто, но если они нужны, то лучше знать о том, как их делать, иначе без них будет достаточно тяжело. Конкретно, мы поговорим о двух вещах: о ещё одном умном указателе `weak_ptr` и о пользовательском `deleter`, который можно использовать в `shared_ptr`.

Начнём с `weak_ptr`. Пусть у нас есть следующий пример

```
class Cache {
public:
    shared_ptr<Widget> GetWidget(const string& name) {
        if (!map_[name]) {
            map_[name] = make_shared<Widget>(name);
        }
```

```
        return map_[name];
    }

private:
    map<string, shared_ptr<Widget>> map_;
};
```

Это отлично работает, но есть одна небольшая проблема. Даже если мы создали какой-то `Widget` и в программе его уже больше не используем, этот `Widget` всё равно будет существовать, потому что на него сохраняется `shared_ptr` внутри этого объекта кеша, то есть `Widget` мы не используем, но он всё равно у нас висит и отжирает некоторые ресурсы. Нам бы хотелось сделать так, чтобы если в остальной программе мы перестали пользоваться этим `Widget`, то этот `Widget` удалялся бы. Как это можно сделать?

Первая мысль, которая приходит в голову, почему бы нам не использовать `use_count()`? Мы уже видели, что он вернет нам текущее количество ссылок из `shared_ptr` на данный объект. Если мы видим, что этот `use_count() == 1` — это значит, что только внутри нашего кеша хранится `shared_ptr` на этот объект, а больше в программе нигде не используется, и соответственно, мы можем его удалить. Казалось бы да, но есть некоторая проблема. Мы же не знаем точный момент, когда в остальной программе у нас уничтожится последний `shared_ptr`, который указывает на этот `Widget`, то есть непонятно когда этот `use_count()` вызывать.

Мы можем подумать, ладно, давайте заведем какой-нибудь фоновый поток в котором будем по таймеру смотреть на `use_count()`. Хорошо, так в принципе можно сделать, но тут возникает другая проблема: как мы уже знаем `use_count()` в многопоточной среде использовать очень ненадёжно, потому что значение, которое возвращает `use_count()` устаревает ровно в тот момент, когда мы его получили, ибо ровно в этот же самый момент из другого потока кто-то может у нас запросить этот же самый `Widget`, и, соответственно, счетчик ссылок у нас увеличится на одиночку. Получается, что если нам нужен `use_count()`, то нам нужен фоновый поток, но если фоновый поток — мы не можем использовать `use_count()`.

Есть другое решение, более подходящее в данном случае — это как раз использование `weak_ptr`: `weak_ptr` — **это невладеющий умный указатель**. Казалось бы, он вроде бы умный, но почему он тогда не владеющий? Чем он лучше обычного сырого указателя? Дело в том, что из `weak_ptr` можно корректно создать `shared_ptr`, который уже будет владеющим. Как мы знаем, из сырого указателя создавать `shared_ptr` — это гиблое дело. Это вообще считается низкоуровневым управлением динамической памятью, потому что каждый раз, когда мы создаем `shared_ptr` из сырого указателя у нас для этого объекта заводится свой контрольный блок, и созданные таким образом `shared_ptr` оказываются не связанными друг с другом, и они скорее всего приведут к двойному удалению объекта. А вот из `weak_ptr` можно абсолютно корректно создавать `shared_ptr`, и все эти `shared_ptr` будут иметь один и тот же контрольный блок. Давайте посмотрим, как у нас изменится код с использованием `weak_ptr`.

Метод `lock()` у `weak_ptr` как раз создает `shared_ptr`, для того объекта, на который указывает этот `weak_ptr`, если такой объект есть. Соответственно, если такой объект есть, то мы получаем у него него `shared_ptr` и возвращаем. А вот если этого объекта нет, то что происходит?

Объекта может не быть по двум причинам: либо этот объект ещё не был создан вообще, в принципе мы только создали кэш и в кэше `Widget` с этим именем нет, либо — и это самое главное,

этот объект когда-то был создан, но с тех пор им перестали пользоваться, то есть мы когда-то отдали на него `shared_ptr`, но потом этот `shared_ptr` и все его копии были уничтожены, и объектом больше никто не пользуется — счетчик ссылок упал до нуля, тогда этот объект будет удален и `weak_ptr` сможет понять, что объект был удален.

Соответственно, в обоих этих случаях метод `lock()` вернет нам пустой `shared_ptr` — это будет обозначать, что объекта у нас нет по той или иной причине.

В этом случае мы зайдем в условие, и, поскольку у нас объекта нет, но соответственно, его нужно создать, поэтому мы точно таким же образом вызываем функцию `make_shared` для `Widget` с данным именем, она возвращает `shared_ptr`, этот `shared_ptr` мы неявно конвертируем в `weak_ptr` и складываем его в `map`. После этого мы этот `shared_ptr` возвращаем.

```
class Cache {
public:
    shared_ptr<Widget> GetWidget(const string& name) {
        auto ret = map_[name].lock();
        if (!ret) {
            map_[name] = make_shared<Widget>(name);
        }
        return map_[name];
    }

private:
    map<string, weak_ptr<Widget>> map_;
};
```

Вот таким несложным исправлением мы смогли добиться ровно того поведения этой программы, которое нам нужно, за счёт использования ещё одного специфического умного показателя `weak_ptr`.

1.8 Пользовательский deleter

Пусть у нас есть следующая задача.

```
Widget* GetNonOwningPtr();
shared_ptr<Widget> GetOwningPtr();

/*?..*/ GetWidget(bool owning) {
    if (owning) {
        return GetOwningPtr();
    } else {
        return GetNonOwningPtr();
    }
}
```

Казалось бы, а какой тип будет возвращать эта наша написанная функция? Давайте подумаем.

- Допустим, она возвращает сырой указатель. Тогда вызвать функцию, которая возвращает сырой указатель, никаких проблем нет, мы просто вернем тот же самый сырой указатель.

Но что делать, если нам нужно вызвать функцию, которая возвращает `shared_ptr`? Мы можем, конечно, из этого `shared_ptr` достать сырой указатель с помощью `get()`, мы так уже делали. И это действительно сработает. Но мы вернем сырой указатель, который не будет участвовать во владении этим объектом. А нам бы хотелось, чтобы этот указатель смог поучаствовать во владении. Получается, что сырой указатель нам не подходит.

- Допустим, будем возвращать тоже `shared_ptr`. Тогда у нас не будет никаких проблем вызвать функцию, которая сама возвращает `shared_ptr` — мы просто вернем тот же самый `shared_ptr`. Но что делать с функцией, которая возвращает сырой указатель? Ведь мы знаем, что нельзя просто так взять и засунуть сырой указатель в `shared_ptr`. Тем более, если мы вернем `shared_ptr`, мы знаем, что `shared_ptr` — это владеющий умный указатель, и когда он будет удален или когда его копии будут удалены, он попытается удалить этот объект. А нам не нужно, чтобы он удалялся, ведь нам возвращают невладеющий указатель. Что же делать?

На самом деле мы можем сделать хитрый ход конем и всё-таки использовать `shared_ptr`. Смотрите, за счёт чего.

```
Widget* GetNonOwningPtr();
shared_ptr<Widget> GetOwningPtr();

shared_ptr<Widget> GetWidget(bool owning) {
    if (owning) {
        return GetOwningPtr();
    } else {
        Widget* ptr = GetNonOwningPtr();
        auto dummyDeleter = [](Widget*) {};
        return shared_prt<Widget>(ptr, dummyDeleter);
    }
}
```

Что делать, если нам нужно вызвать функцию, которая возвращает обычный указатель? Смотрите, мы его вызываем, получаем обычный указатель, и дальше мы этот указатель засунем в новый `shared_ptr`, который мы создали. Как же так, скажете вы, он же попытается его удалить. И мы знаем, что действительно, умные указатели удаляют объект, на который они ссылаются, когда они заканчивают владение этим объектом. Но это же C++. То есть на самом деле всё немного не так, всё немного все хитрее. **Умные указатели не удаляют объект, когда заканчивают владение объектом, на самом деле они для этого объекта вызывают `deleter`, который по умолчанию этот объект удаляет**; `deleter` — это просто некоторая функция. И эта функция по умолчанию реализована таким образом, что она просто вызывает `delete` для переданного указателя.

А здесь мы с вами завели свой `deleter`, в которм мы не делаем ничего. То есть получается, мы создаем такой `shared_ptr`, который указывает на некоторый объект и который, когда он заканчивает владение этим объектом, вызывает для него `deleter`, который не делает ничего. То есть по сути мы создали `shared_ptr`, который этим объектом не владеет. Получается, что в первой ветке мы возвращаем владеющий `shared_ptr`, который нам вернула функция, а во второй ветке мы возвращаем невладеющий `shared_ptr`, то есть мы возвращаем некоторый `shared_ptr`, который

условно владеет объектом. Обратите внимание, что в данном случае мы нарушили соглашение по владению динамическими объектами. Ну потому что как бы предполагается, что `shared_ptr` безусловно владеет объектами, на которые он ссылается. Однако здесь у нас была определенная причина. Мы решили, что для реализации такого поведения программы мы нарушим эти соглашения. И это может быть оправдано в зависимости от задачи, которую мы решаем.

Таким образом, использование `deleter` на самом деле позволяет нам создать условно владеющий `shared_ptr`, хотя это приведет к нарушению соглашения по владению. И на самом деле `deleter` доступен не только для `shared_ptr`, но для `unique_ptr` тоже можно указать свой `deleter`. Правда, работать это будет немножко по-другому.

Небольшое напутствие: всегда старайтесь следовать соглашению по владению динамическими объектами. На практике их нужно будет нарушать разве что в тех случаях, когда вам придется взаимодействовать с компонентами, которые уже по тем или иным причинам нарушают эти соглашения. Например, это может быть компонента, написанная с использованием старого стандарта C++, еще до C++11, когда не было умных указателей. Или это может быть компонента, написанная на чистом C, где, очевидно, нет никаких умных указателей. В этом случае рекомендуется эти компоненты в явном виде изолировать от всего остального кода и во всем остальном коде придерживаться этих соглашений.

Идиома RAII

2.1 Знакомство с редактором vim и консольным компилятором

Давайте познакомимся с той средой, в которой будем писать программы, а также контролировать их и запускать. Работать будем в операционной системе Linux, используя консольный текстовый редактор vim. Давайте с его помощью напишем простейшую программу, которая печатает «Hello, world!» на экране.

Пишем знакомый нам текст. К vim можно подключить различные плагины, которые позволят нам использовать и автодополнения кода и поиск по коду, но не будем сейчас это делать.

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!\n";
}
```

Итак, выполняем команду `g++`, то есть вызываем компилятор, указываем ему опцию `-std=c++17`, хотя в программе нет ничего, что требовало бы C++17, указываем имя файла и указываем с помощью опции `-o` имя того выходного исполняемого файла, в который компилятор запишет скопмилированный результат. Пусть это будет файл `hello.out`. Выполняем команду, и если вы не увидели на экране никаких сообщений — это значит, что программа скомпириловалась успешно. Теперь запускаем файл `hello.out` и видим на экране «Hello, world!».

```
# g++ --std=c++17 hello.cpp -o hello.out
# ./hello.out
```

2.2 Жизненный цикл объекта

Вспомним, что такое **жизненный цикл объекта**. Блок кода, грубо говоря, — это фрагмент программы, ограниченный операторными или фигурными скобками. Например, тело функции — это блок кода. Тело условного оператора или оператор цикла — это тоже блок кода. Блоки кода могут быть вложены друг в друга, как матрёшки. Если внутри какого-то блока кода объявляется переменная, то обычно такая переменная считается автоматической. Если тип этой переменной —

это какой-то класс, то такую переменную мы называем объектом. Автоматические объекты создаются на стеке и живут до конца блока. Это значит, что как только мы из блока выходим, для таких созданных к этому моменту переменных компилятор вызывает деструктор у соответствующего класса, а также генерирует и исполняет код, который очищает память, которую эта переменная на стеке занимала. Заметьте, что порядок удаления таких переменных обратен порядку их создания. В этом примере в искусственном блоке кода написаны две переменных:

```
{
    // ...
    string s;
    // ...
    vector<int> v = {1, 2, 3};
    //...
}
```

И, как бы мы из этого блока не вышли, гарантированно будут вызваны для них деструкторы. Эти деструкторы будут очищать ту память, в которой вектор и строка хранили свои данные.

В противовес автоматическим переменным, бывают переменные, которые можно создать в динамической памяти. Время жизни таких переменных управляется программистом. Создаются они с помощью конструкции `new`, удаляются с помощью `delete`, но дополнительная гибкость с управлением времени и жизни разменивается здесь на потенциальный набор проблем, который может быть связан с утечкой памяти. Ведь программисту приходится не забывать всякий раз освобождать переменную, когда она уже не нужна. Вот еще один искусственный пример.

```
string* p1;

{
    string* p2 = new string;
    p1 = p2
    // ...
}

delete p1;
```

Заметим, что умирает лишь сам указатель `p2`, но никак не та память, на которую он указывал. Мы нарочно сохраним этот указатель в переменной `p1`, тоже типа указатель, который переживет этот блок. И вот в конце вызовем `delete` для этой переменной `p1`. Именно в этот момент мы вручную удалим тот объект, который мы породили в динамической памяти.

Вспомним также про исключения. **Если в программе произошло исключение, то текущий блок покидается аварийно**. Это означает, что для переменных, которые к этому моменту автоматически были созданы на стеке, автоматически же будут генерироваться и вызываться деструкторы. Это явление называется **раскруткой стека**. Важно: **можно генерировать исключения в конструкторах. Более того, это единственный способ сообщить внешнему миру о том, что объект не может быть создан в конструкторе**. Однако генерировать исключения, которые покидают пределы деструкторов, крайне опасно. Считается, что все деструкторы должны отработать без сбоя. Представьте себе, что будет если в деструкторе, который

работает в момент раскрутки стека, произойдет еще одно исключение. Вложенные поля сложных объектов ведут себя похожим на стек образом. Перед телом конструктора они инициализируются, после выполнения тела деструктора они уничтожаются. Поэтому если внутри конструктора произошло какое-то исключение, то все проинициализированные, к этому моменту вложенные, поля — несмотря на то что сам объект, который конструируется, не будет считаться созданным, — будут корректно уничтожены с помощью деструкторов. Давайте рассмотрим вот такой пример.

```
class C {
    vector<int> vals;
public:
    C(const vector<int>& given) : vals(given) {
        if (any_of(begin(vals), end(vals), [](int v) {return v < 0;})) {
            throw runtime_error("negative values!");
        }
    }

    ~C() {
    }
};
```

Давайте напишем код, который все это иллюстрирует.

```
#include <iosream>

using namespace std;

class C {
public:
    C() {
        cout << "C()\n";
    }
    ~C() {
        cout << "~C()\n";
    }
};

int main() {
    {
        C c;
    }
    {
        C c;
    }
}
```

Скомпилируем эту программу. Запустим. И мы увидим, что сначала создан объект типа `C`, потом уничтожен. И опять создан объект типа `C` и снова уничтожен, то есть при выходе из блока автоматически вызывается деструктор.

Пусть теперь у нас есть какая-то вспомогательная функция, которую мы будем вызывать из функции `main`.

```
#include <iostream>
#include <stdexcept>
...
void foo() {
    C c1;
    throw runtime_error("");
    C c2;
}

int main() {
    try {
        foo();
    } catch (...) {
        cout << "exception\n";
    }
}
```

Скомпилируем такую программу, запустим её. Смотрите: несмотря на то, что в функции `foo` должны были создаваться два объекта, на самом деле, конечно же, был создан только один, который `c1` и который создаётся до генерации исключения.

Обратите внимание, что мы покинули функцию `foo` аварийно и для этого объекта деструктор был также вызван автоматически.

Таким образом, для автоматических объектов, то есть объектов, которые создаются на стеке, гарантированно будут вызываться деструкторы, каким бы способом мы не покинули блок. То же самое относится и к полям вложенных объектов классов.

2.3 Идея RAII

Вообще, **RAII расшифровывается как Resource Aquisition Is Initialization**. Считается, что это не очень удачное название. Перевести на русский его можно как «выделение ресурса (или получение ресурса, или захват ресурса) есть инициализация», то есть такая операция должна являться инициализацией в правильно написанном коде, инициализация какой-то переменной. Но давайте сначала разберемся, что такое ресурс.

Ресурс нам предоставляется нам напрокат какой-то третьей стороной, например, операционной системой. Этот ресурс надо не забыть вернуть, когда он нам больше не нужен, потому что ресурс ограничен. Типичный пример ресурсов — это файл, мьютекс или память. Так вот, эта идиома предлагает с каждым фактом запроса, захвата ресурса связывать какую-нибудь автоматическую переменную. Такая переменная с помощью своего деструктора будет автоматически возвращать этот ресурс, когда он окажется не нужен. Можно сказать, что идиома RAII — это такой рай для программиста, который позволяет легко следить за ресурсами. Проведем аналогию с обычной жизнью.

Пусть блок кода — это комната в каком-то помещении, в которой ходит программист. Выход из блока — это значит выход из комнаты через какую-то дверь. Исключение — это срочная эвакуация через запасной выход. Что в этой терминологии является ресурсом? Можно считать, что ресурсом является электричество, а освобождение ресурса — это требование выключить свет, когда мы выходим из комнаты. Нам нужно не забыть выключить свет, даже если объявлена эвакуация.

Что предлагает подход RAII? Подход RAII говорит, что выключать свет каждый раз вручную утомительно. Очень легко забыть это сделать, особенно в чрезвычайной ситуации, поэтому давайте поставим автоматический датчик, который будет в комнате следить за тем, что никого нет. Включать свет мы по-прежнему будем вручную, но если датчик сказал, что комната пуста и её все покинули, свет будет автоматически выключатся. Наверное, вы знаете, что есть языки с так называемой «сборкой мусора». Они предлагают совершенно другой подход. В этих языках по комнатам периодически ходит вахтер, который вручную выключает свет в тех комнатах, где никого нет. C++ не такой язык. Здесь всякая работа с ресурсом, следуя этой идиоме, должна быть обернута в какой-то блок кода, в начале которого специальные переменные инициализируются, а в ее деструкторах этот ресурс автоматически освобождается.

2.4 RAII-обёртка над файлом

Попробуем применить идиому RAII на практике к конкретному ресурсу. Давайте в качестве ресурса выберем банальный файл. Для начала попробуем поработать с файлом так, как мы это бы делали с помощью библиотеки языка C, где никакой идиомы RAII нет. Вот пример программы.

```
#include <cstdio>
#include <iostream>

using namespace std;

int main() {
    FILE* f = fopen("output.txt", "w");

    if (f != nullptr) {
        fputs("Hello, world!\n", f);
        fputs("This file is written with fputs\n", f);
        fclose(f);
    } else {
        printf("Cannot open file\n");
    }
}
```

У нас в программе может быть много мест, из которых мы можем выйти. В каждом из них, если мы работали с файлом, нам надо не забыть вернуть этот ресурс операционной системе. Закрывался файл с помощью функции `fclose`. Давайте запустим компилятор, программа успешно скомпилирована, запускаем её и видим, что у нас теперь появился файл `output.txt`, в который мы действительно что-то записали.

Теперь давайте попробуем переписать эту программу в духе идиомы RAII. Для этого объявим

специальный класс, который будет оборачивать в себе этот ресурс.

```
class File {
private:
    FILE* f;

public:
    File(const string& filename) {
        f = fopen(filename.c_str(), "w");
        if (f == nullptr) {
            throw runtime_error("cannot open " + filename);
        }
    }

    void Write(const string& line) {
        fputs(line.c_str(), f);
    }

    ~File() {
        fclose(f);
    }
};

int main() {
    try {
        File f("output.txt");
        f.Write("Hello, world!\n");
        f.Write("This is RAII file\n");
    } catch (...) {
        cout << "cannot open file\n";
    }
}
```

Посмотрите, мне нигде не пришлось писать явно функцию `fclose`, которая этот файл закрывает, кроме как в деструкторе класса. Она написана один раз, потому что моя переменная автоматическая. Каким бы способом она не вышла бы из этого блока, вот если бы было бы написано посередине `return`, или если бы здесь произошло еще какое-то исключение, созданная к этому моменту переменная будет автоматически освобождена.

Давайте скомпилируем нашу программу и убедимся, что она работает. Успешно скомпилировалась и запустилась, и мы видим, что она действительно записала в файл `output.txt` эти строки.

2.5 Копирование и перемещение RAII-обёрток

Итак, мы с вами написали класс `File`, который в духе идиомы RAII, являлся оберткой над низкоуровневой файловой переменной. В его конструкторе мы пытались выделить этот ресурс,

запросить его у операционной системы, то есть открыть файл. В его деструкторе мы этот файл закрывали. Давайте попробуем написать небольшой кусок кода, который покажет, что не все так просто и на самом деле наш файл нуждается в некоторых улучшениях.

```
int main() {
    try {
        File f("output.txt");
        File f2 = f;
        ...
    }
    ...
}
```

Давайте попробуем скомпилировать такую безобидную программу и посмотрим как она себя поведет. Скомпилировалась, но при запуске она как-то очень странно себя повела. Мы видим на экране какой-то непонятный дамп. Что же произошло? Написано «double free» — двойное освобождение.

Действительно, давайте разберемся: мы с вами создали копию нашей файловой переменной. Для переменной `f2` был неявно вызван конструктор копирования. Мы не написали в нашем классе никакого конструктора копирования, поэтому компилятор предоставил его нам по умолчанию. Что же делает этот конструктор копирования по умолчанию? А он просто копирует все поля, которые были в классе. В нашем классе единственное поле — это указатель. Он и был скопирован. Теперь получается, что два объекта `f` и `f2` хранят внутри себя указатель на одну и ту же область памяти, два одинаковых указателя. Что произойдет, когда эти переменные начнут выходить из блока? Как вы помните, первой умирает та переменная, которая была создана последней, то есть `f2`, и в момент работы деструктора для `f2` ничего плохого не происходит, закрывается наш файл. А вот в тот момент, когда умирает переменная `f`, мы пытаемся этот файл закрыть дважды. В этот момент и происходит ошибка. **Возвращение ресурса дважды — это большая ошибка, которую нельзя допускать**. Что же делать?

Давайте разберемся, в чем вообще причина этой проблемы? На самом деле причина в том, что мы не определили семантику копирования. Можно было бы делать по разному, например, можно было бы предположить, что в случае такого копирования объектов у нас должен создаваться какой-то новый файл, рядышком, с каким-то дополнительным другим именем, в который бы копировалось содержимое предыдущего файла и с которыми мы бы теперь работали независимо.

А можно сделать проще. Можно сказать, что объекты типа `File` мы просто запрещаем копировать. На самом деле, если мы что то подобное делаем с конструктором копирования, скорее всего, наверняка нам надо сделать что-то похожее и с оператором присваивания. Это можно написать так.

```
...
class File {
private:
    FILE* f;

    File(const File&) = delete;
    void operator = (const File&) = delete;
```

```
public:
    ...
};
```

Таким образом мы предостерегли себя от неосторожного использования этого класса.

2.6 RAII вокруг нас

Давайте обратим внимание, что **идиома RAII**, которую мы изучаем, на самом деле **повсеместно встречается в стандартной библиотеке языка C++**. Типичный пример — это, стандартные классы `string` и `vector`. Эти классы хранят свои элементы в динамической памяти. В конструкторе это динамическая память выделяется. В каких-то функциях, например, если мы в вектор добавляем новые элементы и требуется реаллокация, мы эту динамическую память перераспределяем. В деструкторе этих классов эта память непременно освобождается. Пользователю этих классов нет необходимости думать об этом. Всё это скрыто под капотом внутри. Именно в этом и удобство этой идиомы. Мы просто пользуемся этими классами, создаем такие переменные на стеке и совершенно не задумываемся о том, что когда эти переменные выходят из соответствующего блока, где-то выполняется код, который эту память освобождает.

Другой типичный пример идиомы RAII — это умный указатель `unique_ptr`. Здесь, в отличие от `vector`, динамическая память не выделяется в конструкторе. Наоборот, мы захватываем владение уже существующим указателем, который указывает на какую-то память, и который нам предоставил пользователь. А вот деструктор точно так же эту динамическую память освобождает.

Ещё в качестве примера можно привести файловый поток `fstream`, который чем-то напоминает нашу обертку над файлом, которую мы написали в прошлый раз. Есть небольшое отличие: у него другой интерфейс для ввода-вывода, он по другому обрабатывает ошибки в случае, если файл открылся неуспешно, но суть точно такая же. В своём деструкторе он пытается закрыть файл.

Ещё классический пример идиомы RAII в стандартной библиотеке, это работа с `mutex`: `mutex` — это тоже ресурс. Давайте попробуем посмотреть на голый `mutex` непосредственно. У него в интерфейсе есть две функции: заблокировать `mutex` и разблокировать: `lock()` и `unlock()`. Если бы мы пользовались им непосредственно, то вначале каждой функции нам бы приходилось брать блокировку, при выходе из этой функции её снимать, возвращать `mutex` обратно. Давайте рассмотрим пример, который вам уже знаком. Вы уже писали такой код, где в несколько потоков вызывается функция `Spend` которая пытается уменьшить баланс на какую-то величину.

```
struct Account {
    int balance = 0;

    bool Spend(int value) {
        if (value <= balance) {
            balance -= value;
            return true;
        }
```

```
        return false;
    }
};
```

Я напомню, что если этот код скомпилировать непосредственно, то может произойти такое, что у нас в итоге баланс окажется отрицательным, поскольку несколько потоков, которые не договорившись друг с другом, одновременно начнут уменьшать переменную `balance`. Для того, чтобы этого не было, надо внутри этой функцию взять блокировку. Попробуем это сначала сделать с помощью голого `mutex`.

```
#include <mutex>

...

struct Account {
    int balance = 0;

    bool Spend(int value) {
        m.lock();
        if (value <= balance) {
            balance -= value;
            m.unlock();
            return true;
        }
        m.unlock();
        return false;
    }
};
```

Программа скомпилируется и будет работать, баланс будет всё время нулевым, но обратите внимание, помнить о том, что при каждом выходе с функций надо не забыть написать `unlock` — это довольно утомительно, это чревато ошибками. Если наша функция окажется сложнее, из нее может быть очень много точек выхода, причем как явных, как здесь, так и не явных. Дело в том, что какие-то функции внутри могут сгенерировать исключение, которые мы не обработаем и тогда мы функцию `Spend` будем аварийно покидать, `mutex` в этом случае окажется в состоянии блокировки. Блокировку мы забудем снять. Что же делать?

Давайте воспользуемся таким вспомогательным классом `lock_guard`, также вам знакомым, который позволяет написать всё то же самое, но только в одну строчку.

```
#include <mutex>

...

struct Account {
    int balance = 0;

    bool Spend(int value) {
        lock_guard<mutex> guard(m);
```

```
        //m.lock();
        if (value <= balance) {
            balance -= value;
            //m.unlock();
            return true;
        }
        //m.unlock();
        return false;
    }
};
```

Давайте убедимся, что этот код тоже работает. Да, баланс все время получается нулевым.

На самом деле, классы, которые вы писали на занятиях, тоже в каком-то смысле исповедуют идиому RAII. `ObjectPool`, который хранил объекты, и даже `LogDuration`, который вы использовали для того, чтобы заметить время работы какого-то блока кода, можно считать примерами RAII. В `LogDuration`, конечно же, никакой ресурс не захватывался в самом начале, но его деструктор вызывался всякий раз, когда переменная выходила из блока, и он выполнял нетривиальные действия по замеру времени.

Давайте рассмотрим, почему в языке C++ нет блока `try/finally` как в некоторых других языках, например в Java. Блок `finally` нужен в тех языках для того, чтобы гарантированно освободить ресурсы, как бы ни завершился основной код, с исключением или без. Обычно пишется обертка `try`, в который помещается какой-то код, потенциально опасный, в котором производится работа с каким-то ресурсом, например, открывается файл. После этого в блоке `catch` перехватываются быть может какие-то исключения, и в самом конце в блоке `finally` вы пытаетесь закрыть файл, что бы ни произошло.

Можно провести такую аналогию из жизни, продолжая те аналогии, которые мы уже приводили. Если блок кода — это какая-то комната с множеством выходов, то в таких языках все выходы ведут в какой-то общий коридор, единый коридор, в котором есть один выключатель, который выключает свет. Этот выключатель непременно надо выключить руками. Почему этого нет **в языке C++**? Потому что полностью **блок `finally` заменен на идиому RAII**. Код, который мог бы быть написан при каждой обработке ошибки в таком блоке `finally`, пишется на самом деле ровно один раз и ровно в одном месте: в деструкторе соответствующего класса, который оборачивает этот ресурс.

Разбор задачи

3.1 Разбор задачи с использованием идиомы RAII

Давайте посмотрим на вспомогательные классы

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class FlightProvider {
public:
    using BookingId = int;

    struct BookingData {
        string city_from;
        string city_to;
        string date;
    };

    BookingId Book(const BookingData& data) {
        ++counter;
        cerr << "Flight booking: " << counter << "\n";
        return counter;
    }

    void Cancel(const BookingId& id) {
        --counter;
        cerr << "Cancel flight: " << id << "\n";
    }

private:
    int counter = 0;
};
```

По аналогии с этим классом `FlightProvider`, есть очень похожий класс `HotelProvider` для бронирования гостиниц.

```
class HotelProvider {
public:
    using BookingId = int;

    struct BookingData {
        string city;
        string date_from;
        string date_to;
    };

    BookingId Book(const BookingData& data) {
        ++counter;
        cerr << "Hotel booking: " << counter << "\n";
        return counter;
    }

    void Cancel(const BookingId& id) {
        --counter;
        cerr << "Cancel hotel: " << id << "\n";
    }

private:
    int counter = 0;
};
```

Давайте также предположим, что функция `Book` и в том, и в другом классе, вообще говоря, может генерировать исключения, например, если мест в гостинице нет или ни на какой рейс не удалось забронировать билет. Давайте сымитируем эту ситуацию

```
...
class FlightProvider {
    ...
    BookingId Book(const BookingData& data) {
        ++counter;
        if (counter > 1)
            throw runtime_error("Overbooking");
        cerr << "Flight booking: " << counter << "\n";
        return counter;
    }
    ...
```

Давайте также заметим, что отмена бронирования никогда не приводит к сбоям, а отменить бронирование какой-либо поездки или проживание в гостинице, которое было в прошлом, то есть уже совершено, это тоже нормальная, законная ситуация, которая не приводит к какой-либо ошибке.

Теперь давайте попробуем воспользоваться этими классами для того, чтобы написать систему управления командировками сотрудников. Мы хотим посылать сотрудников в командировки, и нам надо знать, на каких рейсах они летят в другие города и в каких гостиницах живут. Поэто-

му, пользуясь этими классами, мы напишем класс `TripManager`, у которого тоже будет похожий интерфейс.

```
struct Trip {
    vector<HotelProvider::BookingId> hotels;
    vector<FlightProvider::BookingId> flights;
};

class TripManager {
public:
    using BookingId = int;

    struct BookingData {
        string city_from;
        string city_to;
        string date_from;
        string date_to;
    };

    Trip Book(const BookingData& data) {
        Trip trip;
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        {
            HotelProvider::BookingData data;
            trip.hotels.push_back(hotel_provider.Book(data));
        }
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        return trip;
    }

    void Cancel(Trip& trip) {
        for (auto& id : trip.hotels) {
            hotel_provider.Cancel(id);
        }
        trip.hotels.clear();
        for (auto& id : trip.flights) {
            flight_provider.Cancel(id);
        }
        trip.flights.clear();
    }
```

```
private:
    HotelProvider hotel_provider;
    FlightProvider flight_provider;
};
```

Ну, что ж, давайте посмотрим, как теперь заработает такая функция `main`.

```
int main() {
    TripManager tm;
    auto trip = tm.Book({});
    tm.Cancel(trip);
}
```

Запускаем программу. Ой, и внезапно видим на экране, что перелёт забронирован, гостиница забронирована, а вот попытка забронировать обратный перелёт обернулась исключением, которое мы не перехватили. Ну, действительно, мы же там специально предусмотрели, что в случае, когда перелётов больше одного, то происходит исключение, мы это смоделировали. Да, мы вспомнили, что функция `Book`, вообще говоря, может выкидывать исключение, если бронирование не удалось, поэтому этот кусочек кода давайте обернём в `try/catch`, чтобы исключения цивилизованно ловить, чтобы наша программа не завершалась аварийно с такой ошибкой.

```
int main() {
    try {
        TripManager tm;
        auto trip = tm.Book({});
        tm.Cancel(trip);
    } catch (...) {
        cout << "Exception\n";
    }
}
```

Ещё раз запускаем, смотрим. И что же мы видим? Посмотрите, мы смогли забронировать перелёт, забронировать гостиницу, поймали даже это исключение. Но почему мы не видим сообщение об отмене этих перелётов? Ведь если всю командировку не получилось забронировать целиком, то те её составные части, которые мы уже забронировали, надо отменить. Предположим, что бронирование устроено так, что в случае, если мы его не отменяем, автоматически списываются деньги с какой-то карты. Это не здорово. Давайте посмотрим, как грамотно отменять такие результаты в случае каких-то внутренних сбоев, как сделать нашу командировку транзакционной.

Бронирование — это такой ресурс, который нам предоставляют соответствующие провайдеры `HotelProvider` и `FlightProvider`. Получается, что этот ресурс утекает. Для того чтобы справиться с этой проблемой, нам надо поправить функцию `Book`. Давайте обернем опасный блок кода, а именно вызовы функции `Book` у этих провайдеров, в `try/catch`.

```
class TripManager {
    ...
    Trip Book(const BookingData& data) {
        Trip trip;
```

```
      try {
        {
          FlightProvider::BookingData data;
          trip.flights.push_back(flight_provider.Book(data));
        }
        {
          HotelProvider::BookingData data;
          trip.hotels.push_back(hotel_provider.Book(data));
        }
        {
          FlightProvider::BookingData data;
          trip.flights.push_back(flight_provider.Book(data));
        }
      } catch (...) {
          Cancel(trip);
          throw;
      }
      return trip;
  }
  ...
```

Давайте посмотрим, как теперь изменится программа. Посмотрите, теперь действительно в случае исключения мы отменяем два предыдущих бронирования. Это ожидаемое поведение, которого мы и хотели достичь.

Однако поглядите на эту функцию `Book`. Кажется, что она устроена слишком сложно. Она теперь превращается в такие макароны кода, которые обернуты вот этими блоками `try/catch` на случай, как бы чего не вышло. На самом деле, мы можем элементарно забыть написать этот `try/catch` в какой-нибудь аналогичной функции, которая, например, добавляет новые города к командировке. Более того, интерфейс нашего класса, нашей структуры `Trip` вообще открыт, и кто угодно может добавить новую командировку, если у него есть доступ к соответствующему провайдеру. В случае, если произойдет такая же ошибка, он может позабыть просто отменить предыдущее бронирование. Такой код чреват утечками наших ресурсов.

Давайте посмотрим, что предлагает нам идиома RAII взамен. **Идиома RAII гласит: каждый ресурс следует обернуть в объект, который за него отвечает**. RAII, напомним, расшифровывается как Resource Acquisition Is Initialization. И там, казалось бы, речь идет про инициализацию, про выделение ресурса. Но **гораздо важнее в этой идиоме помнить про деструктор**, ведь именно в деструкторе класса будет написан код, который этот ресурс возвращает. Давайте, следуя этой идиоме, перепишем эту структуру `Trip`.

```
class Trip {
private:
    HotelProvider& hotel_provider;
    FlightProvider& flight_provider;

public:
    vector<HotelProvider::BookingId> hotels;
```

```
    vector<FlightProvider::BookingId> flights;

    Trip(HotelProvider& hp, FlightProvider& fp)
        : hotel_provider(hp), flight_provider(fp) {}

    Trip(const Trip&) = delete;
    Trip(Trip&&) = default;

    Trip& operator=(const Trip&) = delete;
    Trip& operator=(Trip&&) = default;

    void Cancel() {
        for (auto& id : hotels) {
            hotel_provider.Cancel(id);
        }
        hotels.clear();
        for (auto& id : flights) {
            flight_provider.Cancel(id);
        }
        flights.clear();
    }

    ~Trip() {
        Cancel();
    }
};
```

еперь давайте попробуем переписать наш `TripManager`, чтобы он смог работать с этой новой версией класса `Trip`. Нам уже не нужен блок `try/catch`, мы можем смело его убрать. И код становится таким же, как был в нашей первой, наивной версии. Действительно, если в какой-то момент одно из бронирований окажется неудачным, и произойдет исключение, то раскрутка стека гарантирует нам, что все созданные к этому моменту автоматические объекты будут уничтожены, для них будет вызван деструктор. И поэтому для объекта `trip` будет вызван деструктор, который вызовет `Cancel`, такая поездка будет отменена.

```
class TripManager {
    ...
    Trip Book(const BookingData& data) {
        Trip trip(hotel_provider, flight_provider);
        {
            FlightProvider::BookingData data;
            trip.flights.push_back(flight_provider.Book(data));
        }
        {
            HotelProvider::BookingData data;
            trip.hotels.push_back(hotel_provider.Book(data));
        }
```

```
    {
        FlightProvider::BookingData data;
        trip.flights.push_back(flight_provider.Book(data));
    }
    return trip;
}

void Cancel(Trip& trip) {
    trip.Cancel();
}
...
```

Попробуем скомпилировать нашу программу. Наша программа запустилась, и она ведет себя так же ожидаемо, как и в версии с обёрткой `try/catch` внутри функции `Book`.

Обратите внимание, что нам теперь нигде не пришлось писать этот `try/catch`. В коде класса `Trip` мы его не написали. Давайте посмотрим, что нам это дало. Мы заметили, что у нас есть определенный ресурс — это командировка. Он, в свою очередь, состоит из каких-то составных элементарных ресурсов, которые нам предоставляются провайдерами: `HotelProvider` и `FlightProvider`. Если проводить аналогию, скажем, с памятью или с файлами, то в роли таких провайдеров системных ресурсов обычно выступает операционная система, которая не представлена каким-то объектом явно, она находится за кулисами. Но вот здесь у нас особый тип ресурса, и поэтому эти провайдеры должны быть нам известны явно. Можно было бы развивать эту идею дальше и переделать интерфейс наших провайдеров, чтобы они в функции бронирования возвращали бы не идентификатор бронирования, а тоже какой-нибудь объект, который умеет сам себя отменять в случае, если вызван его деструктор или что-то пошло не так. Но мы ограничились тем, что написали такую обертку `Trip` для составного бронирования.

Таким образом, оборачивайте ваши ресурсы в классы, следуя идиоме RAII, пишите деструкторы этих классов. **Именно в деструкторах должно возвращаться владение этими ресурсами тем, кто их предоставил**.