Синглтон (Singleton)

1. Шаблон Синглтон – как определяют его Банда Четырех

2. Ограничение экземпляров класса до одного – зачем, пример с файловой системой

3. Обеспечение глобальной области видимости – проблема, если мы это не обеспечим глобальность, как выглядит код с помощью старой реализации и современной

4. Зачем мы его используем – 3 плюса, в последнем расписать кодовую реализацию

5. Почему мы можем пожалеть, что стали его использовать – три проблемы глобальных переменных, решение двух проблем, даже если у нас всего одна (пример с логом), недостаток ленивой оптимизации (что может произойти из-за нее, как можно это исправить?)

6. Что можно сделать вместо этого – подумайте нужен ли вам класс вообще (пример с пулей и менеджером пулей), ограничение класса единственным экземпляром (когда хотим только один экземпляр, но не глобальный), удобство доступа к экземпляру: 3 способа

**1. Шаблон Синглтон**

Паттерны программирования определяют Синглтон следующим образом:

«*Обеспечивает существование единственного экземпляра класса и обеспечивает глобальный доступ к нему*».

**2. Ограничение экземпляров класса до одного**

Иногда класс может работать корректно только, если существует его единственный экземпляр. Зачастую, это необходимо, если класс взаимодействует с внешней системой, которая поддерживает собственное глобальное состояние.

Представим себе класс-обертку над низкоуровневым API файловой системы. Так как файловые операции занимают определенное время, наш класс выполняет операции асинхронно. Это значит, что несколько операций могут выполняться конкурентно и следовательно им нужно координировать свою работу. Если мы начнем с вызова для создания файла и затем попытаемся удалить тот же самый файл, наш класс-обертка должен знать об обеих вызовах, чтобы они друг другу не мешали.

Чтобы это сделать, наш класс-обертка должен иметь доступ ко всем предыдущим операциям. Если пользователь может свободно создавать любое количество экземпляров этого класса, каждый из этих экземпляров не будет значить ничего об операциях, выполняемых другими экземплярами. Вот здесь то и нужен Синглтон. Он позволяет обеспечить еще на этапе компиляции возможность существования только одного экземпляра класса.

**3. Обеспечение глобальной области видимости**

Наш класс-оболочку над файловой системой использует несколько других систем в игре: система отчетов, загрузка контента, сохранение состояния игры и т.д. Если эти системы не смогут создать собственный экземпляр обертки над файловой системой, то как они смогут ее использовать?

Синглтон решает и эту проблему. Он не только создает единственный экземпляр класса, но и делает его глобально видимым. Таким образом кто угодно и где угодно может наложить на него свою лапу. Целиком классическая реализация выглядит следующим образом:

class FileSystem

{

public:

static FileSystem& instance()

{

// Lazy initialize.

if (instance\_ == NULL) instance\_ = new FileSystem();

return \*instance\_;

}

private:

FileSystem() {}

static FileSystem\* instance\_;

};

Статический член instance\_ хранит экземпляр класса, а приватный конструктор обеспечивает то что этот экземпляр единственный. Публичный и статический метод instance() предоставляет доступ к экземпляру для всей остальной кодовой базы. А еще он отвечает за создание экземпляра синглтона методом ленивой инициализации, т.е. в момент первого вызова.

Современный вариант выглядит следующим образом:

class FileSystem

{

public:

static FileSystem& instance()

{

static FileSystem \*instance = new FileSystem();

return \*instance;

}

private:

FileSystem() {}

};

С++11 гарантирует, что инициализация локальной статической переменной происходит только один раз, даже в случае конкурентного доступа. Поэтому, при условии что у вас есть поддерживающий C++11 компилятор, такой код является потоково-безопасным. А вот первый пример — нет.

**4. Зачем мы его используем**

Похоже у нас есть победитель. Наша обертка над файловой системой доступна всегда и везде и ее не нужно передавать с помощью каких либо ухищрений. Сам класс достаточно умен для того чтобы не позволить нам по ошибке устроить месиво из кучи его экземпляров. А вот и еще несколько приятных особенностей:

**Экземпляр не будет создан если его никто не захочет использовать.** Экономия памяти и циклов процессора — это всегда хорошо. Благодаря тому что синглтон инициализируется при первом вызове, его экземпляр не создастся если никто в игре к нему не обратится.

**Инициализация во время выполнения.** Очевидной альтернативой Синглтону является класс ос статическими переменными членами. Мне нравятся простые решения, да и использование статических классов вместо синглтона возможно. Однако у статических членов есть одно ограничение: автоматическая инициализация. Компилятор инициализирует статические переменные до вызова main() . Это значит, что они не могут использовать информацию, которая будет известна только после того как программа запустится и начнет работать (например, когда будет загружена файл настроек). А еще это значит, что они не могут полагаться друг на друга — компилятор не гарантирует очередности в которой относительно друг друга статические переменные будут инициализированы.

Ленивая инициализация решает обе эти проблемы. Синглтон будет инициализирован настолько поздно, насколько возможно, так что ко времени его создания нужная информация уже будет загружена. И если это не приводит к циклической зависимости, один синглтон может ссылаться при инициализации на другой.

**У вас может быть подкласс синглтон.** Это очень мощная, но редко используемая возможность. Скажем например, что мы хотим сделать свою обертку над файловой системой кросс-платформенной. Чтобы это заработало нам нужно сделать интерфейс файловой системы абстрактным с подклассами, которые будут реализовывать интерфейсы для каждой платформы. Вот базовый класс:

class FileSystem

{

public:

virtual ~FileSystem() {}

virtual char\* readFile(char\* path) = 0;

virtual void writeFile(char\* path, char\* contents) = 0;

};

Затем мы определяем производные классы для нескольких платформ:

class PS3FileSystem : public FileSystem

{

public:

virtual char\* readFile(char\* path)

{

// Use Sony file IO API...

}

virtual void writeFile(char\* path, char\* contents)

{

// Use sony file IO API...

}

};

class WiiFileSystem : public FileSystem

{

public:

virtual char\* readFile(char\* path)

{

// Use Nintendo file IO API...

}

virtual void writeFile(char\* path, char\* contents)

{

// Use Nintendo file IO API...

}

};

А теперь превращаем FileSystem в синглтон:

class FileSystem

{

public:

static FileSystem& instance();

virtual ~FileSystem() {}

virtual char\* readFile(char\* path) = 0;

virtual void writeFile(char\* path, char\* contents) = 0;

protected:

FileSystem() {}

};

Хитрость здесь заключается в создании экземпляра:

FileSystem& FileSystem::instance()

{

#if PLATFORM == PLAYSTATION3

static FileSystem \*instance = new PS3FileSystem();

#elif PLATFORM == WII

static FileSystem \*instance = new WiiFileSystem();

#endif

return \*instance;

}

С помощью простого переключателя компилятора мы связываем обертку файловой системы с конкретным типом. И теперь вся наша кодовая база может получать доступ к файловой системе через FileSystem::instance() не привязываясь ни к какому платформозависимому коду. Вместо этого получившаяся связность инкапсулируется внутри реализации самого класса FileSystem.

**5. Почему мы можем пожалеть, что стали его использовать**

**Это глобальная переменная**

Важнейшим усвоенным уроком насчет глобальных переменных было то, что использовать глобальные переменные — плохо. По нескольким причинам:

**Они делают код менее понятным**. Допустим мы ищем баг в функции, написанной кем-то другим. Если функция не затрагивает никаких глобальных состояний, мы можем ограничиться рамками функции и просто изучить какие аргументы она получает.

А теперь представьте что прямо в середине функции происходит обращение к SomeClass::getSomeGlobalData() . Теперь для того, чтобы понять что происходит, нам нужно прошерстить всю кодовую базу и выяснить, кто еще работает с этими глобальными данными. И у вас конечно нет причин ненавидеть глобальные переменные, пока однажды вам не придется просмотреть миллион строк кода в три часа утра в поисках того глючного вызова, который все таки записывает в статическую глобальную переменную некорректное значение.

**Они усиливают связность**. Новый программист в вашей команде конечно еще не знаком с вашей прекрасной, легко поддерживаемой архитектурой игры, но ему нужно выполнить задание: добавить проигрывание звуков, когда камни падают на землю. Мы с вами хорошо понимаем, что нам нужно любой ценой не допускать лишних связей между физической и аудио подсистемами, но новичок просто хочет выполнить свое задание. К нашему несчастью экземпляр AudioPlayer имеет глобальную область видимости. И вот после добавления всего одного #include вся ранее возведенная архитектура рушится.

Если бы экземпляр аудио плеера не был бы объявлен глобальным, добавление #include с его заголовочным файлом так ничего бы и не дало. Этот факт сам по себе четко сказал бы новичку, что эти модули не должны ничего знать друг о другие и ему нужно найти другой способ решения проблемы. Управляя доступом к экземпляру вы управляете связностью.

**Они не конкурентно-дружественны**. Деньки, когда игра работала на одноядерном процессоре уже сочтены. Современный код должен по крайней мере корректно работать в многопоточной системе, даже если он не использует все ее преимущества. Когда мы делаем что либо глобальным, у нас образуется область памяти, видимая всеми потоками. И каждый поток может к ней обратится не зная, что возможно с этой памятью уже работает кто-то еще. Это приводит к блокировкам (deadlocks), гонкам за доступ (race conditions) и другим страшным багам синхронизации.

**Он решает две проблемы, даже если у вас всего одна**

Слово "и" в описании Синглтона от Банды Четырех выглядит немного странным. Решает этот шаблон одну или сразу две проблемы? Что, если у нас только одна из этих проблем? Обеспечение наличия всего одной копии экземпляра может быть полезно, но кто сказал, что мы хотим, чтобы кто угодно мог пользоваться этим экземпляром? И наоборот, глобальная видимость может быть полезной, но хочется иметь возможность иметь множество экземпляров.

Вторая из проблем — удобство доступа — обычно и побуждает нас к использованию шаблона Синглтон. Представим себе класс журналирования — логгер. Большинство модулей в игре только выиграют от возможности легко добавлять в лог диагностическую информацию. В то же время передача единственного экземпляра логгера в каждый класс будет только перегружать код и отвлекать от его основного назначения.

Очевидным решением является преобразования класса Log в синглтон. В результате каждая функция может получить экземпляр класса напрямую. Но в то же самое время у нас появляется новое ограничение. Внезапно мы лишаемся возможности иметь больше одного класса логгера.

По началу это и не проблемы вовсе. Мы пишем всего в один лог файл и нам все-равно не нужно больше одного экземпляра. А дальше, в процессе разработки у нас появляются новые трудности. Каждый член команды использует логгер в своих нуждах и вскоре лог файл превращается в месиво данных. Программистам приходится пролистывать полотна сообщений, чтобы найти интересующее.

Мы могли бы решить эту проблему добавив возможность писать в несколько лог файлов. Для этого можно завести отдельный логгер для всех областей игры: сети, пользовательского интерфейса, аудиоподстистемы, геймплея. Но мы не можем этого сделать. Класс Log не только не позволяет нам больше иметь несколько экземпляров класса, но и налагает такое ограничение на все свои вызовы:

Log::instance().write("Some event.");

Теперь, чтобы вернуть себе возможность иметь одновременно несколько экземпляров логгера (как было изначально), им нужно изменить не только сам класс, но и все участки кода, где он используется. Наш удобный доступ перестал быть удобным.

**Ленивая инициализация отнимает у вас контроль над происходящим**

Инициализация системы требует времени: на выделение памяти, на загрузку ресурсов и т.д. Если инициализация аудио подсистемы требует несколько сотен миллисекунд, нам нужно иметь возможность контролировать, когда эта инициализация произойдет. Если мы позволим ей лениво инициализироваться при первом проигрывании звука — это произойти в середине напряженной игры и вызовет неслабое падение FPS и заикание геймплея.

Кроме этого, играм обычно требуется надежный контроль размещения объектов в куче чтобы избежать фрагментации памяти. Если аудио система выделяет при инициализации область в памяти из кучи, нам нужно понимать когда произойдет инициализация, чтобы проконтролировать, где в куче будет размещена эта область.

Из-за двух этих проблем, большинство игр, которые я видел, не полагаются на ленивую инициализацию. Вместо этого они реализуют Синглтон следующим образом:

class FileSystem

{

public:

static FileSystem& instance() { return instance\_; }

private:

FileSystem() {}

static FileSystem instance\_;

};

Таким образом, мы решаем проблему ленивой инициализации, но делаем это за счет потери некоторых особенностей синглтона, которые делают его лучше, чем простая глобальная переменная. После добавление статического экземпляра мы больше не можем использовать полиморфизм и класс должен стать конструируемым во время статической инициализации. И еще мы теперь не можем освободить память, занимаемую экземпляром когда он станет не нужным.

Вместо создания синглтона, все что мы получили — это простой статический класс. Это не обязательно плохо, но если нам нужен просто статический класс, то почему бы вообще не избавиться от метода instance() и пользоваться статической функцией напрямую? Вызов Foo::bar() проще чем Foo::instance().bar() и к тому же яснее показывает что мы имеем дело со статической памятью.

**6. Что можно сделать вместо этого**

**Подумайте нужен ли вам класс вообще**

Большинство синглтонов, которые я видел в играх, были "менеджерами": это были просто такие туманные классы, созданные только для того, чтобы нянчиться с другими объектами. Я помню кодовые базы, где практически у каждого класса был свой менеджер: Монстр, Менеджер монстров, Частица, Менеджер частиц, Звук, Менеджер звуков, Менеджер менеджеров. Иногда в имени встречались слова "система" или "движок", но сама суть не менялась. Иногда классы смотрители конечно полезны, но зачастую они просто демонстрируют незнание ООП . Полюбуйтесь на эти два надуманных класса:

class Bullet

{

public:

int getX() const { return x\_; }

int getY() const { return y\_; }

void setX(int x) { x\_ = x; }

void setY(int y) { y\_ = y; }

private:

int x\_, y\_;

};

class BulletManager

{

public:

Bullet\* create(int x, int y)

{

Bullet\* bullet = new Bullet();

bullet->setX(x);

bullet->setY(y);

return bullet;

}

bool isOnScreen(Bullet& bullet)

{

return bullet.getX() >= 0 &&

bullet.getX() < SCREEN\_WIDTH &&

bullet.getY() >= 0 &&

bullet.getY() < SCREEN\_HEIGHT;

}

void move(Bullet& bullet)

{

bullet.setX(bullet.getX() + 5);

}

};

Может этот пример и довольно туп, но я видел достаточно много кода, архитектурная сущность которого после небольшого упрощения сводилась к тому же. Если посмотреть на код выше, вполне логично предположить что BulletManager можно сделать синглтоном. В конце концов все где встречаются пули, будет нуждаться и в их менеджере, так что сколько еще экземпляров менеджера нам нужно?

Правильный ответ — ниcколько. И вот как мы решим проблему "синглтона" для нашего класса менеджера:

class Bullet

{

public:

Bullet(int x, int y) : x\_(x), y\_(y) {}

bool isOnScreen()

{

return x\_ >= 0 && x\_ < SCREEN\_WIDTH &&

y\_ >= 0 && y\_ < SCREEN\_HEIGHT;

}

void move() { x\_ += 5; }

private:

int x\_, y\_;

};

Вот и все. Нет менеджера — нет проблемы. Плохо спроектированные синглтоны — это чаще всего просто "вспомогательные" классы, добавляющие функциональность к другим классам. Везде где это возможно просто переносите поведение внутрь самого класса. В конце концов ООП именно о том чтобы позволить классу заботиться о себе самостоятельно.

**Ограничение класса единственным экземпляром**

Это ровно половина того, что дает нам шаблон Синглтон. И как в нашем примере с файловой системой, такое ограничение на наличие единственного экземпляра класса может быть критичны. Однако из этого совсем не следует, что мы обязательно желаем предоставлять публичный, глобальный доступ к экземпляру. Нам может быть удобнее ограничить доступ к некоторым областям кода или даже сделать его приватным для отдельного класса. В таком случае предоставление глобального доступа только ослабляет архитектуру.

Нам может понадобиться обеспечить существование единственного экземпляра без предоставления глобального доступа. Есть несколько вариантов решения. Вот один из них:

class FileSystem

{

public:

FileSystem()

{

assert(!instantiated\_);

instantiated\_ = true;

}

~FileSystem() { instantiated\_ = false; }

private:

static bool instantiated\_;

};

bool FileSystem::instantiated\_ = false;

Класс позволяет кому угодно вызвать его конструктор, но по прежнему содержит проверку и выдает ошибку если вы попытаетесь создать второй экземпляр. И когда правильный код создаст экземпляр первым, мы можем быть уверены что никто больше другой экземпляр не создаст и доступа к нему не получит. Класс обеспечивает существование себя в единственном экземпляре, но не навязывает нам того как с ним нужно обращаться.

Недостатком этого примера является то, что проверка на существование экземпляра выполняется во время выполнения. А вот шаблон Синглтон самой своей сущностью гарантирует существование только одного экземпляра класс уже на этапе компиляции.

**Удобство доступа к экземпляру**

Удобство доступа — одна из причин, почему мы выбираем синглтон. С его помощью очень просто получить доступ к объекту, который может понадобиться нам в самых различных местах. К сожалению эта простота имеет свою цену: доступ к объекту слишком просто получить даже оттуда, откуда нам не хотелось бы.

Общее правило такое: нам желательна как можно более узкая видимость переменных, но без ущерба для функциональности. Чем уже видимость объекта, тем реже нам нужно задумываться о его влиянии на другие части кода в работе. Прежде чем применять подход дробовика в виде объекта синглтона с глобальной областью видимости, давайте посмотрим какие еще варианты доступа кодовой базы к объекту у нас есть:

**Прямая передача.** Простейшее решение и в некоторых случаях лучшее. Мы просто передаем объект в качестве аргумента в функцию, которая в нем нуждается. Не стоит сгоряча отбрасывать этот подход за его громоздкость.

Представьте себе функцию рендеринга объекта. Чтобы отрендерить объект, нужно получить доступ к объекту, представляющему собой устройство видеовывода и поддерживающему состояния рендера. Очень часто все это передается целиком в функции рендеринга через параметр с именем context или подобным.

С другой стороны, некоторые объекты не соответствуют контексту метода. Например, функции управляющей AI полезно иметь возможность писать в лог, но это совсем не основная ее задача. Поэтому передача в эту функцию объекта Log в качестве аргумента будет смотреться по крайней мере странно и в таком случае лучше выбрать другой вариант.

**Получение из базового класса.** Во многих играх архитектура представляет собой неглубокую, но достаточно ветвистую иерархию. Зачастую всего с одним уровнем наследования. Например, у вас может быть базовый класс GameObject , от которого наследуются классы для каждого врага или объекта в игре. При такой архитектуре большая часть игрового кода обитает в "листьях" унаследованного класса. Это значит что у всех классов есть доступ к одному и тому же базовому классу GameObject. Мы можем воспользоваться этим преимуществом:

class GameObject

{

protected:

Log& getLog() { return log\_; }

private:

static Log& log\_;

};

class Enemy : public GameObject

{

void doSomething()

{

getLog().write("I can log!");

}

};

При этом никто за пределами GameObject не может получить доступ к его объекту Log, а любая другая унаследованная сущность могут, с помощью getLog().

**Получить через другой объект, который уже является глобальным**. Цель — убрать вообще все глобальные состояния, конечно похвальна, но навряд ли практична. Большинство кодовых баз обязательно имеет хотя бы несколько глобальных объектов, например объекты Game или World , представляющие общее состояние игры. Вы можете обратить это себе на пользу и уменьшить количество глобальных объектов, нагрузив уже существующие. Вместо того чтобы делать синглтон из Log, FileSystem и AudioPlayer можно поступить таким образом:

class Game

{

public:

static Game& instance() { return instance\_; }

// Functions to set log\_, et. al. ...

Log& getLog() { return \*log\_; }

FileSystem& getFileSystem() { return \*fileSystem\_; }

AudioPlayer& getAudioPlayer() { return \*audioPlayer\_; }

private:

static Game instance\_;

Log \*log\_;

FileSystem \*fileSystem\_;

AudioPlayer \*audioPlayer\_;

};

Здесь глобальным объектом является только World . Функции могут получить доступ к другим системам через него:

Game::instance().getAudioPlayer().play(VERY\_LOUD\_BANG);

Если позднее архитектуру придется изменить и добавить несколько экземпляров World ( например для стримминга или тестовых целей) Log , FileSystem и AudioPlayer останутся незатронутыми — они даже разницы не заметят. Есть и недостаток — в результате гораздо больше кода будет завязано на сам класс World . Если классу просто нужно проиграть звук, наш пример все равно требует от него знания о World для того, чтобы получить аудио плеер.

Выходом может быть гибридное решение. Код, уже знающий о World может получать через него прямой доступ к AudioPlayer. А код, который о нем не знает может получать доступ к AudioPlayer с помощью другого решения, о которых мы уже говорили.