**Паттерн Singleton (одиночка,синглет)**

**Назначение паттерна Singleton**

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Паттерн Singleton предоставляет такие возможности.

**Описание паттерна Singleton**

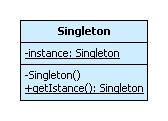
Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной - от запуска программы до ее завершения.
2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

**UML-диаграмма классов паттерна Singleton**



Паттерн Singleton часто называют усовершенствованной глобальной переменной.

**Реализация паттерна Singleton**

**Классическая реализация Singleton**

Рассмотрим наиболее часто встречающуюся реализацию паттерна Singleton.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // Singleton.h  class Singleton  {    private:      static Singleton \* p\_instance;      // Конструкторы и оператор присваивания недоступны клиентам      Singleton() {}      Singleton( const Singleton& );      Singleton& operator=( Singleton& );    public:      static Singleton \* getInstance() {          if(!p\_instance)              p\_instance = new Singleton();          return p\_instance;      }  };    // Singleton.cpp  #include "Singleton.h"    Singleton\* Singleton::p\_instance = 0; |

Клиенты запрашивают единственный объект класса через статическую функцию-член getInstance(), которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти. Впоследcтвии клиенты должны сами позаботиться об освобождении памяти при помощи оператора delete.

Последняя особенность является серьезным недостатком классической реализации шаблона Singleton. Так как класс сам контролирует создание единственного объекта, было бы логичным возложить на него ответственность и за разрушение объекта. Этот недостаток отсутствует в реализации Singleton, впервые предложенной Скоттом Мэйерсом.

**Singleton Мэйерса**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Singleton.h  class Singleton  {  private:      Singleton() {}      Singleton( const Singleton&);      Singleton& operator=( Singleton& );  public:      static Singleton& getInstance() {          static Singleton  instance;          return instance;      }  }; |

Внутри getInstance() используется статический экземпляр нужного класса. Стандарт языка программирования C++ гарантирует автоматическое уничтожение статических объектов при завершении программы. Досрочного уничтожения и не требуется, так как объекты Singleton обычно являются долгоживущими объектами. Статическая функция-член getInstance() возвращает не указатель, а ссылку на этот объект, тем самым, затрудняя возможность ошибочного освобождения памяти клиентами.

Приведенная реализация паттерна Singleton использует так называемую [отложенную инициализацию (lazy initialization)](http://cpp-reference.ru/articles/lazy-initialization/) объекта, когда объект класса инициализируется не при старте программы, а при первом вызове getInstance(). В данном случае это обеспечивается тем, что статическая переменная instance объявлена внутри функции - члена класса getInstance(), а не как статический член данных этого класса. Отложенную инициализацию, в первую очередь, имеет смысл использовать в тех случаях, когда инициализация объекта представляет собой дорогостоящую операцию и не всегда используется.

К сожалению, у реализации Мэйерса есть недостатки: сложности создания объектов производных классов и невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

**Улучшенная версия классической реализации Singleton**

С учетом всего вышесказанного классическая реализация паттерна Singleton может быть улучшена.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | // Singleton.h  class Singleton;  // опережающее объявление    class SingletonDestroyer  {    private:      Singleton\* p\_instance;    public:      ~SingletonDestroyer();      void initialize( Singleton\* p );  };    class Singleton  {    private:      static Singleton\* p\_instance;      static SingletonDestroyer destroyer;    protected:      Singleton() { }      Singleton( const Singleton& );      Singleton& operator=( Singleton& );     ~Singleton() { }      friend class SingletonDestroyer;    public:      static Singleton& getInstance();  };    // Singleton.cpp  #include "Singleton.h"    Singleton \* Singleton::p\_instance = 0;  SingletonDestroyer Singleton::destroyer;    SingletonDestroyer::~SingletonDestroyer() {      delete p\_instance;  }  void SingletonDestroyer::initialize( Singleton\* p ) {      p\_instance = p;  }  Singleton& Singleton::getInstance() {      if(!p\_instance)     {          p\_instance = new Singleton();          destroyer.initialize( p\_instance);      }      return \*p\_instance;  } |

Ключевой особенностью этой реализации является наличие класса SingletonDestroyer, предназначенного для автоматического разрушения объекта Singleton. Класс Singleton имеет статический член SingletonDestroyer, который инициализируется при первом вызове Singleton::getInstance() создаваемым объектом Singleton. При завершении программы этот объект будет автоматически разрушен деструктором SingletonDestroyer (для этого SingletonDestroyer объявлен другом класса Singleton).

Для предотвращения случайного удаления пользователями объекта класса Singleton, деструктор теперь уже не является общедоступным как ранее. Он объявлен защищенным.

**Использование нескольких взаимозависимых одиночек**

До сих пор предполагалось, что в программе используется один одиночка либо несколько несвязанных между собой. При использовании взаимосвязанных одиночек появляются новые вопросы:

* Как гарантировать, что к моменту использования одного одиночки, экземпляр другого зависимого уже создан?
* Как обеспечить возможность безопасного использования одного одиночки другим при завершении программы? Другими словами, как гарантировать, что в момент разрушения первого одиночки в его деструкторе еще возможно использование второго зависимого одиночки (то есть второй одиночка к этому моменту еще не разрушен)?

Управлять порядком создания одиночек относительно просто. Следующий код демонстрирует один из возможных методов.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | // Singleton.h  class Singleton1  {    private:      Singleton1() { }      Singleton1( const Singleton1& );      Singleton1& operator=( Singleton1& );    public:      static Singleton1& getInstance() {          static Singleton1 instance;          return instance;      }  };    class Singleton2  {    private:      Singleton2( Singleton1& instance): s1( instance) { }      Singleton2( const Singleton2& );      Singleton2& operator=( Singleton2& );      Singleton1& s1;    public:      static Singleton2& getInstance() {          static Singleton2 instance( Singleton1::getInstance());          return instance;      }  };    // main.cpp  #include "Singleton.h"    int main()  {      Singleton2& s = Singleton2::getInstance();      return 0;  } |

Объект Singleton1 гарантированно инициализируется раньше объекта Singleton2, так как в момент создания объекта Singleton2 происходит вызов Singleton1::getInstance().

Гораздо сложнее управлять временем жизни одиночек. Существует несколько способов это сделать, каждый из них обладает своими достоинствами и недостатками и заслуживают отдельного рассмотрения. Обсуждение этой непростой темы остается за рамками проекта. Подробную информацию можно найти в [3].

Несмотря на кажущуюся простоту паттерна Singleton (используется всего один класс), его реализация не является тривиальной.

**Результаты применения паттерна Singleton**

**Достоинства паттерна Singleton**

* Класс сам контролирует процесс создания единственного экземпляра.
* Паттерн легко адаптировать для создания нужного числа экземпляров.
* Возможность создания объектов классов, производных от Singleton.

**Недостатки паттерна Singleton**

* В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

**http://cpp-reference.ru/patterns/creational-patterns/singleton/**