Ампутируем глобальные объекты (?)

Есть одна страшная напасть, которая может просочиться в Ваш проект, а затем и в Ваш ум — желание использовать *глобальные объекты*. Что же за зверь этот *глобальный объект* и как с ним бороться? Давайте разбираться вместе.

Для начала нужно узнать зверя поближе. Самый простой способ создать глобальный объект — объявить *extern* переменную в заголовочном файле и создать её экземпляр в cpp:

---------------------

// header file

extern ResourceManager g\_resource\_manager;

// cpp file

ResourceManager g\_resource\_manager;

-----------------------

Более абстрактный подход — использование шаблона проектирования *одиночка.* [Шаблон одиночка](https://ru.wikipedia.org/wiki/Одиночка_(шаблон_проектирования)) гарантирует, что в однопоточном приложении будет единственный экземпляр класса с глобальной точкой доступа. Реализация на С++ позволяет выбрать несколько вариантов (!!!!!!!!! ссылки раз, два, три).

----

/////////////////////////////////////

class ResourceManager

{

private:

ResourceManager() = delete;

~ResourceManager() = delete;

public:

static ResourceManager& getInstance()

{

static ResourceManager instance;

return instance;

}

...

};

/////////////////////////////////////

Для многопоточного приложения можно использовать различные ~~костыли~~ механизмы для работы и с многопоточностью. Один из вариантов — предложенный в 2004 Майерсом и Александреску — использовать *Double-check locking*. Идея в том, что Вы проверяете валидность объекта в первом *if*-е и, если он не обнаружен, делаем *lock* и проверяем ещё раз. Выглядеть это будет так ([детальный разбор полётов](http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/)):

/////////////////////////

…

Singleton\* Singleton::getInstance() {

Singleton\* tmp = m\_instance;

... *// insert memory barrier*

**if** (tmp == **nullptr**) { // NULL еще с Майерсовских времен остался? )) - ну естественно =)

Lock lock;

tmp = m\_instance;

**if** (tmp == **nullptr**) {

tmp = **new** Singleton;

... *// insert memory barrier*

m\_instance = tmp;

}

}

**return** tmp;

}

...

/////////////////////////

Чем же так удобен шаблон, что ему уделяется столько внимания? Всё просто — он позволяет ~~вставить убийственную зависимость в Ваш код~~ использовать объект в любом месте программы:

///////////////////////////////

void PrepareResources(...)

{

ResourceManager::getInstance().Initialize ();

}

///////////////////////////////

Очень удобно и соблазн так сделать очень велик. Проблемы начинаются, когда нужно заменить часть системы не нарушив работы всего остального или же оттестировать код. В последнем случае, Вам придётся инициализировать все глобальные объекты, которые использует интересуемый метод. Более того, это очень осложняет замену поведения объекта на желаемое для тестов. Так же может создать проблем порядок разрушения, хотя на него в любом случае не стоит полагаться.

*Действительно, сложно обойтись без подобных объектов и ещё не видел кода, где глобальные переменные не используются вообще.*

В общем случае необходимо предпочитать использование контекстных переменных вместо глобальных. К примеру, если вам нужно отрисовать что-то и есть глобальный Renderer, то лучше его передать напрямую в метод *void Draw(Renderer& render\_instance)*, а не использовать глобальный *Render::Instance*(). Примеры подобных вещей и их решения можно подсмотреть в [очень интересном блоге](https://blog.molecular-matters.com/2011/11/11/singleton-is-an-anti-pattern/). Мы же рассмотрим случай, когда вы не можете представить, как избавиться от глобального объекта, либо это переходной этап к вырезанию оных, что может понадобиться в реальных проектах.

Для начала постановка задачи:

1. К объекту должен быть доступ из всех частей программы. Это может понадобиться, чтобы не ломать сразу весь код.
2. Все "глобальные" объекты должны храниться централизованно — для простоты поддержки.
3. Добавление/удаление объекта должно быть во время исполнения — глобальный объект на определённое время (такую проблему решает ленивая инициализация Singleton`а — когда конструктор/деструктор публичные и в коде управляется временем жизни объекта.
4. Возможность добавлять заменять глобальные объекты в зависимости от контекста — реальный запуск или тестирование.

Для централизованного хранения подойдёт вектор объектов типа GlobalObjectBase – все наследники, соответственно, могут быть туда добавлены либо удалены.

class GlobalObjectBase

{

public:

virtual ~GlobalObjectBase() {}

};

Сам GlobalObjectBase будет своеобразной меткой для определения глобального объекта.

Теперь нужен класс, который позволит хранить, добавлять, удалять и, естественно, использовать объекты. К сожалению, его придётся сделать глобальным одним из способов, который Вам понравится больше.

================================================

class GlobalObjectsStorage

{

private:

using ObjPtr = std::unique\_ptr<GlobalObjectBase>;

std::vector<ObjPtr> m\_dynamic\_globals;

private:

GlobalObjectBase\* GetGlobalObjectImpl(size\_t i\_type\_code) const

{

auto it = std::find\_if(m\_dynamic\_globals.begin(), m\_dynamic\_globals.end(), [i\_type\_code](const ObjPtr& obj)

{

return typeid(\*obj).hash\_code() == i\_type\_code;

});

if (it == m\_dynamic\_globals.end())

{

// здесь можно добавить ассерт о том, что что-то пошло не так return nullptr;

}

return it->get();

}

void AddGlobalObjectImpl(std::unique\_ptr<GlobalObjectBase> ip\_object)

{

m\_dynamic\_globals.push\_back(std::move(ip\_object));

}

void RemoveGlobalObjectImpl(size\_t i\_type\_code)

{

auto it = std::find\_if(m\_dynamic\_globals.begin(), m\_dynamic\_globals.end(), [i\_type\_code](const ObjPtr& obj)

{

return typeid(\*obj).hash\_code() == i\_type\_code;

});

if (it != m\_dynamic\_globals.end())

m\_dynamic\_globals.erase(it);

}

public:

GlobalObjectsStorage() {}

template <typename ObjectType>

void AddGlobalObject()

{

AddGlobalObjectImpl(std::make\_unique<ObjectType>());

}

template <typename ObjectType>

ObjectType\* GetGlobalObject() const

{

return static\_cast<ObjectType\*>(GetGlobalObjectImpl(typeid(ObjectType).hash\_code()));

}

template <typename ObjectType>

void RemoveGlobalObject()

{

RemoveGlobalObjectImpl(typeid(ObjectType).hash\_code());

}

};

================================================

Для создания работы с данным видом объектов достаточно их типа, поэтому интерфейс GlobalObjectsStorage составляют шаблонные методы, которые передают нужные данные реализации. Использовать можно как std::vector, так и std::unordered\_map, но std::vector выигрывает в производительности примерно на 10% (стоит, наверное, сделать тесты и добавить картинки/таблицы).

Пользоваться подобным хранилищем можно уже в таком виде.

class ResourceManager : public GlobalObjectBase

{

public:

void Initialize() {}

};

void Test()

{

static GlobalObjectsStorage g\_storage;

// делаем объект "глобальным"

g\_storage.AddGlobalObject<ResourceManager>();

// используем

g\_storage.GetGlobalObject<ResourceManager>()->Initialize();

// и удаляем

g\_storage.RemoveGlobalObject<ResourceManager>();

}

Так-с, уже можно худо-бедно использовать. То есть пункты 1, 2 и 3 выполнены. Теперь последний — контекстно зависимое хранилище глобальных объектов. Для этого следует добавить класс-родитель для хранилища (перенеся туда шаблонные методы) и сделав виртуальными методы имплементации.

class ObjectStorageBase

{

private:

virtual GlobalObjectBase\* GetGlobalObjectImpl(size\_t i\_type\_code) const = 0;

virtual void AddGlobalObjectImpl(std::unique\_ptr<GlobalObjectBase> ip\_object) = 0;

virtual void RemoveGlobalObjectImpl(size\_t i\_type\_code) = 0;

public:

template <typename ObjectType>

void AddGlobalObject()

{

AddGlobalObjectImpl(std::make\_unique<ObjectType>());

}

template <typename ObjectType>

ObjectType\* GetGlobalObject() const

{

return static\_cast<ObjectType\*>(GetGlobalObjectImpl(typeid(ObjectType).hash\_code()));

}

template <typename ObjectType>

void RemoveGlobalObject()

{

RemoveGlobalObjectImpl(typeid(ObjectType).hash\_code());

}

};

class DefaultObjectsStorage : public ObjectStorageBase

{

private:

using ObjPtr = std::unique\_ptr<GlobalObjectBase>;

std::vector<ObjPtr> m\_dynamic\_globals;

private:

virtual GlobalObjectBase\* GetGlobalObjectImpl(size\_t i\_type\_code) const override

{ … }

virtual void AddGlobalObjectImpl(std::unique\_ptr<GlobalObjectBase> ip\_object) override

{ … }

virtual void RemoveGlobalObjectImpl(size\_t i\_type\_code) override

{ … }

public:

DefaultObjectsStorage() {}

};

void Test()

{

static std::unique\_ptr<ObjectStorageBase> gp\_storage(new DefaultObjectsStorage());

// делаем объект "глобальным"

gp\_storage->AddGlobalObject<ResourceManager>();

// используем

gp\_storage->GetGlobalObject<ResourceManager>()->Initialize();

// и удаляем

gp\_storage->RemoveGlobalObject<ResourceManager>();

}

Данный класс можно сделать статическим членом главного состояния приложения, либо использовав Singleton. Главное, что должно соблюдаться – возможность установить нужную реализацию интерфейса ObjectStorageBase. Один из вариантов – сделать класс либо пространство имён Core, который будет хранить в себе указатель на текущий ObjectStorageBase.

class Core

{

private:

static std::unique\_ptr<ObjectStorageBase> mp\_go\_storage;

...

public:

static void SetGlobalObjectStorage(std::unique\_ptr<ObjectStorageBase> ip\_getter)

{

mp\_go\_storage = std::move(ip\_getter);

}

static ObjectStorageBase\* GetGlobalObjectStorage() { return mp\_go\_storage.get(); }

template <typename ObjectType>

static ObjectType\* GetGlobalObject()

{

return mp\_go\_storage->GetGlobalObject<ObjectType>();

}

Последние штрихи. Понятно, что производительность от подобной обёртки будет страдать. В данном случае достаточно сильно. Если посмотреть профайлер (нужен скрин), то самое долгое место в поиске занимает typeid(\*obj).hash\_code(). Предполагаемое узкое место, так как добыча данных о типе во время исполнения тратит ценное процессорное время. Для теста производительности было создано десять различных типов. Сначала они использовались напрямую как глобальный объект, затем через DefaultObjectsStorage. Финальный тестовый код можно посмотреть [здесь](https://github.com/TraurigeNarr/SupportSDK/blob/master/Samples/PerformanceTests/PerformanceTests/GlobalObjcetGetterTests.cpp). Результат для *1 000 000* вызовов*.*



Текущий код работает медленнее обычного глобального объекта почти в 18 раз! Срочно исправлять! Для этого добавляем в базовый класс глобальных объектов – GlobalObjectBase – методы для хранения хеша типа.

class GlobalObjectBase

{

private:

size\_t m\_hash\_code;

public:

virtual ~GlobalObjectBase() {}

size\_t GetTypeHashCode() const { return m\_hash\_code; }

void RecalcHashCode() { m\_hash\_code = typeid(\*this).hash\_code(); }

};

Так же стоит поменять метод ObjectStorageBase::AddGlobalObject и DefaultObjectsStorage:: GetGlobalObjectImpl

class ObjectStorageBase

{

…

public:

template <typename ObjectType>

void AddGlobalObject()

{

auto p\_object = std::make\_unique<ObjectType>();

p\_object->RecalcHashCode();

AddGlobalObjectImpl(std::move(p\_object));

}

…

};

class DefaultObjectsStorage : public ObjectStorageBase

{

…

private:

virtual GlobalObjectBase\* GetGlobalObjectImpl(size\_t i\_type\_code) const override

{

auto it = std::find\_if(m\_dynamic\_globals.begin(), m\_dynamic\_globals.end(), [i\_type\_code](const ObjPtr& obj)

{

return obj->GetTypeHashCode() == i\_type\_code;

});

if (it == m\_dynamic\_globals.end())

{

// здесь можно добавить ассерт о том, что что-то пошло не так

return nullptr;

}

return it->get();

}

…

};

С этими изменениями результаты работы будут приятнее – отличия в 3 раза, а не в 18!

Выводы:

Данная оболочка позволяет свести в одно место контроль за глобальными объектами, что, в свою очередь, даст возможность тестировать и изменять методы, завязанные на глобальные объекты, изменять поведение для тестов и мягче совершить переход к новой архитектуре.