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 (а главное нужно ли?) с ним бороться? Давайте разбираться вместе.

Для начала нужно узнать зверя поближе. Существует масса способов создать глобальный объект. Самый простой — 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).

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 чуть ли не 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.

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/).

Однако совсем без глобальных объектов жизнь тоже не мила. Если нужен доступ к настройкам или прототипам, то к каждому объекту не прицепишь все нужные контейнеры, фабрики и классы настроек. Этот случай мы и будем рассматривать.

Для начала постановка задачи:

1. К объекту должен быть доступ из всех частей программы.
2. Все "глобальные" объекты должны храниться централизованно — для простоты поддержки.
3. Возможность добавлять заменять глобальные объекты в зависимости от контекста — реальный запуск или тестирование.
4. (?) 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.

Интересное решение было подсмотрено в [недрах](https://github.com/CRYTEK-CRYENGINE/CRYENGINE/blob/release/Code/CryEngine/CryCommon/CrySystem/ISystem.h) CryEngine (смотреть структурку SSystemGlobalEnvironment), где глобальные объекты завёрнуты в одну структуру и являются указателями на абстрактные сущности, которые инициализируются в нужный момент в нужном месте программы. Никаких дополнительных накладных расходов, никаких лишних надстроек, контроль за типом во время компиляции – красота! Всё, можно расходится.

Но есть одно «но» - CryEngine достаточно старый и уже годами обточенный проект, где все интерфейсы уже устаканились, а новое прикручивается подобно тому, что уже существует в проекте. Есть и другой случай – проект ещё молодой, бурно развивающийся и строгих интерфейсов в нём нету. Более того, функционал ещё и меняется постоянно, что сподвигнет вносить правки в интерфейсе достаточно часто. Для этого можно попробовать подняться на уровень выше и посмотреть на проблему под другим углом – создать хранилище глобальных объектов, которые будут отнаследованы от базового GlobalObjectBase. К сожалению, расходов во время исполнения не избежать, но попробовать подобный вариант, безусловно, стоит!

Базовый класс будет являться своеобразной меткой, что это объект глобальный и в хранилище будет храниться вектор объектов данного класса:

class GlobalObjectBase

{

public:

virtual ~GlobalObjectBase() {}

};

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

{ … }

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

{ … }

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

{ … }

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.

Итак, первый тест-драйв – работает!

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

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>();

}

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

Но это ещё не всё – подменять объекты для разных контекстов нельзя. Исправляем. 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>();

}

### Всё ли у нас хорошо?

Понятно, что производительность от подобной обёртки будет страдать. В данном случае достаточно сильно. Профайлер подсказывает, что 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! Это катастрофа!

Один из способов решить проблему, добавить 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.

Дополнительно сохраняем статически данные о типе в шаблонной функции родительского класса ObjectStorageBase::GetGlobalObject.

class ObjectStorageBase

{

…

public:

template <typename ObjectType>

void AddGlobalObject()

{

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

p\_object->RecalcHashCode();

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

}

template <typename ObjectType>

ObjectType\* GetGlobalObject() const

{

static size\_t type\_hash = typeid(ObjectType).hash\_code());

return static\_cast<ObjectType\*>(GetGlobalObjectImpl(type\_hash);

}

…

};

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;

}

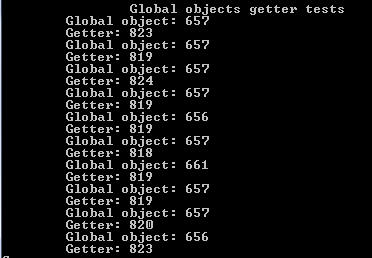
return it->get();

}

…

};

Данный трюк позволит нам серьёзно уменьшить время поиска нужного объекта и отличие будет уже не в 18 раз, а в 1.25, что уже терпимо.



Данная обёртка может использоваться, как в процессе рефакторинга кода, так и в работе над юнит-тестами. Теперь, вместо инициализации глобальных объектов их можно унаследовать от GlobalObjectBase и сделать виртуальными те методы, которые нужны для класса под тестированием. Во время теста же подменяем хранилище глобальных объектов, которое возвращает нам уже подсадной объект, который, в свою очередь, возвращает нужные нам значения. Такой подход позволяет точечно изменяя код в немногих местах сделать его тестируемым.