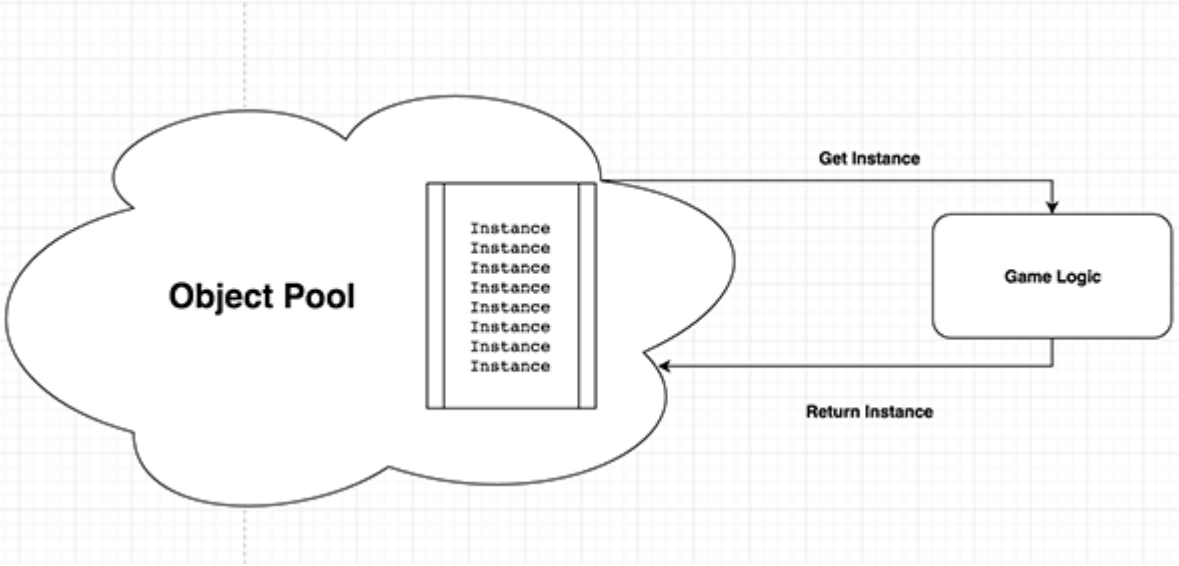
Object Pool

Кирил Сидоров

Object Pool



Реклама!

Не вистачає **часу** на створення нових об’єктів?

Візьми з пулу прямо зараз і користуйся*!

* об’єкт потрібно повернути після використання

Simple Pool

```cpp
template <typename T>
class SimplePool
{
public:
    SimplePool(const int size);
    ~SimplePool();

    // Delete copying and moving
    SimplePool(const SimplePool&) = delete;
    SimplePool(SimplePool&&) = delete;
    SimplePool& operator=(const SimplePool&) = delete;
    SimplePool& operator=(SimplePool&&) = delete;

    void push(T* object);
    T* pull();

private:
    T** _objects = nullptr;
    int _maxSize = 0;
    int _currentSize = 0;
};
```

Особливості:

- створює об’єкти зараніше
- видаляє об’єкти при знищенні
- видає об’єкти
- приймає об’єкти

Simpler Pool

```cpp
template <typename T>
class SimplerPool
{
public:
    SimplerPool(const int size);
    ~SimplerPool();

    // Delete copying and moving
    SimplerPool(const SimplerPool&) = delete;
    SimplerPool(SimplerPool&&) = delete;
    SimplerPool& operator=(const SimplerPool&) = delete;
    SimplerPool& operator=(SimplerPool&&) = delete;

    void push(unique_ptr<T> object);
    unique_ptr<T> pull();

private:
    unique_ptr<T>* _objects;
    int _maxSize = 0;
    int _currentSize = 0;
};
```

Ідіома RAII

Resource Acquisition Is Initialization

```cpp
class RAII
{
    void* _object = nullptr;

public:
    RAII(void* object) : _object(object) {}
    ~RAII() { delete _object; }
};
```

Все те ж саме, але:

- захист від витоку пам’яті
- захист від неправильного використання

Simpler Pool

```
template <typename T>
class SimplerPool
{
public:
    SimplerPool(const int size);
    ~SimplerPool();

    // Delete copying and moving
    SimplerPool(const SimplerPool&) = delete;
    SimplerPool(SimplerPool&&) = delete;
    SimplerPool& operator=(const SimplerPool&) = delete;
    SimplerPool& operator=(SimplerPool&&) = delete;

    void push(unique_ptr<T> object);
    unique_ptr<T> pull();

private:
    vector<unique_ptr<T>> _objects;
};
```

Ідіома RAII

Resource Acquisition Is Initialization

```
class RAII
{
    void* _object = nullptr;

public:
    RAII(void* object) : _object(object) {}
    ~RAII() { delete _object; }
};
```

Все те ж саме, але:

- захист від витоку пам’яті
- захист від неправильного використання
- не потрібно “вручну” нічого видаляти

Simplest Pool

Недолік SimplerPool – потрібно повертати об’єкт назад.

Як вирішити?
RAII “під іншим соусом”

```
class RAII
{
    void* _object = nullptr;

public:
    RAII(void* object) : _object(object) {}
    ~RAII() { delete _object; }
};
```

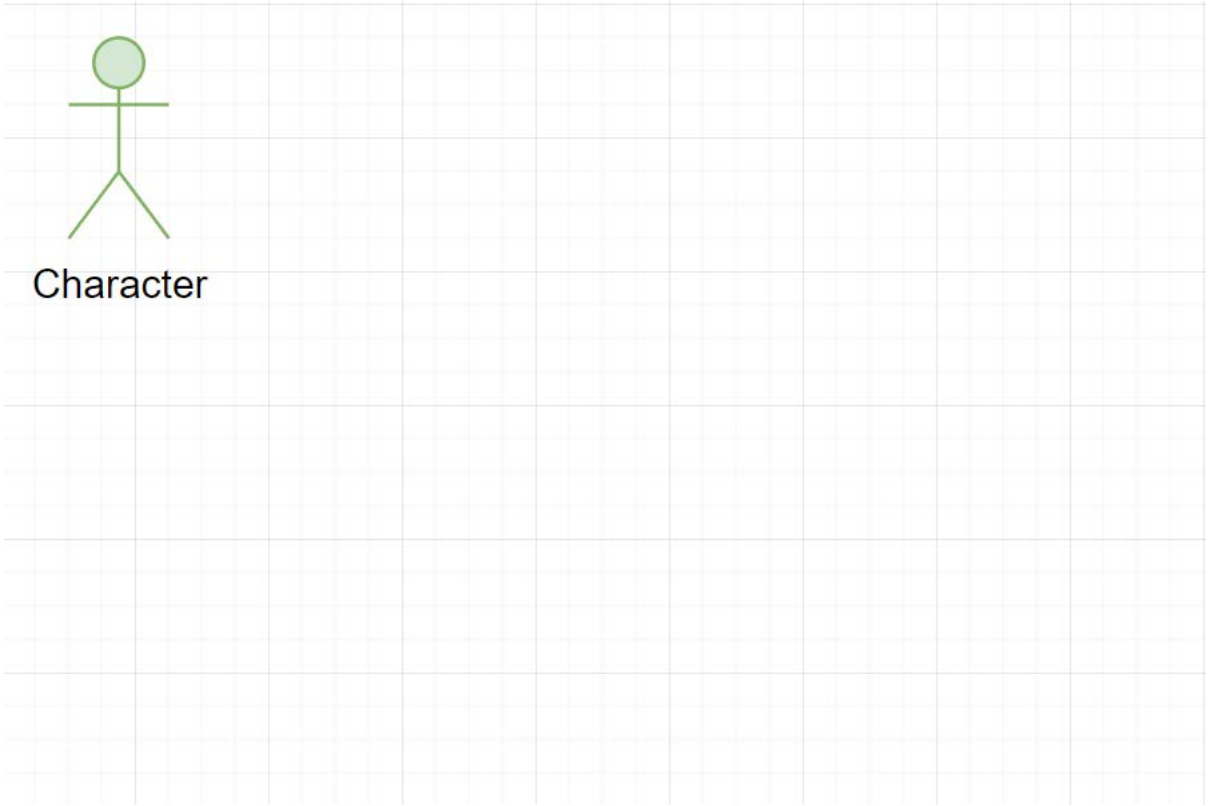
```
class OurRAII
{
    void* _object = nullptr;
    Pool& _pool;

public:
    OurRAII(void* o, Pool& p) : _object(o), _pool(p) {}
    ~OurRAII() { _pool.push(_object); }
};
```

Simplest Pool

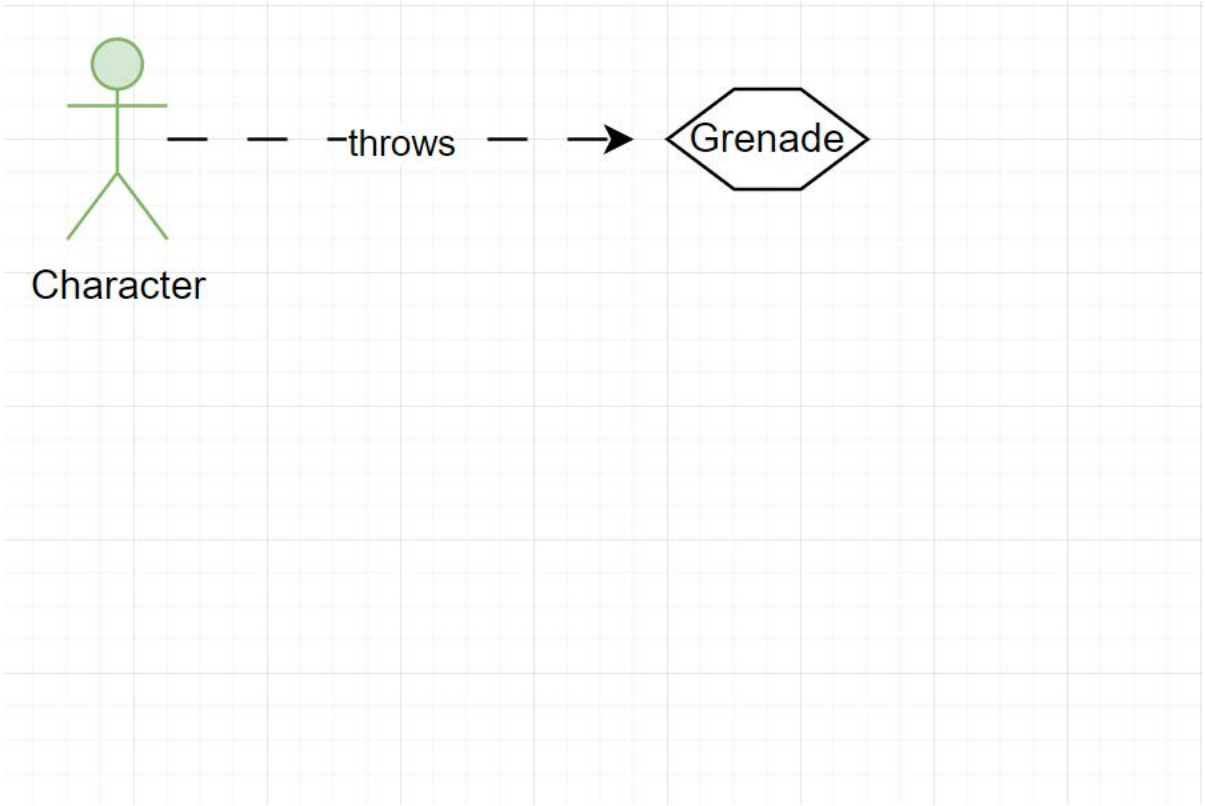
```cpp
template <typename T>
class SimplestPool
{
public:
    class Proxy { ... };

    SimplestPool(const int size);
    ~SimplestPool();

    // Delete copying and moving
    SimplestPool(const SimplestPool&) = delete;
    SimplestPool(SimplestPool&&) = delete;
    SimplestPool& operator=(const SimplestPool&) = delete;
    SimplestPool& operator=(SimplestPool&&) = delete;

    void push(unique_ptr<T> object);
    Proxy pull();

private:
    vector<unique_ptr<T>> _objects;
};
```

```cpp
class Proxy
{
private:
    friend class SimplestPool<T>;
    Proxy(SimplestPool<T>& pool, unique_ptr<T> obj = nullptr);

    SimplestPool<T>& _pool;
    unique_ptr<T> _object;

public:
    ~Proxy();

    Proxy(Proxy&&) = default;
    Proxy& operator=(Proxy&&) = default;

    T* operator->();
    T& operator*();

    operator bool() const;

    Proxy(const Proxy&) = delete;
    Proxy& operator=(const Proxy&) = delete;
};
```

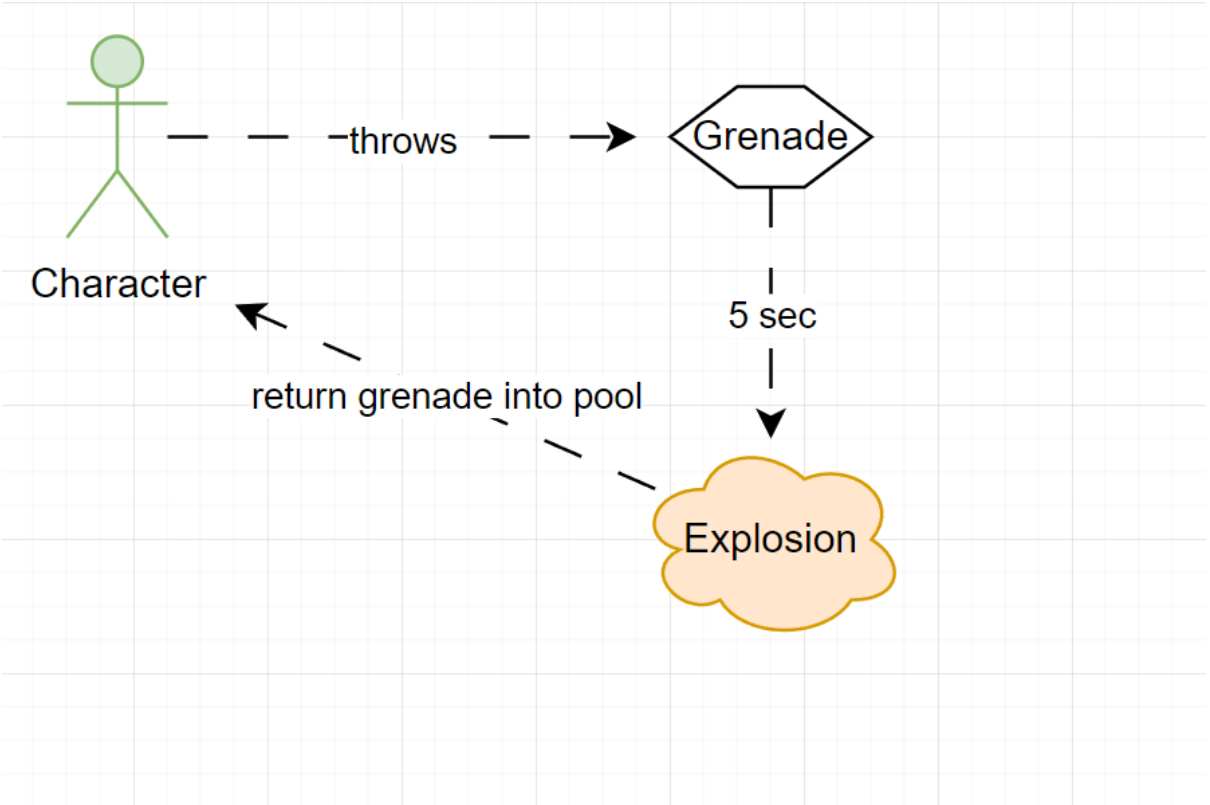
Bug with Grenades



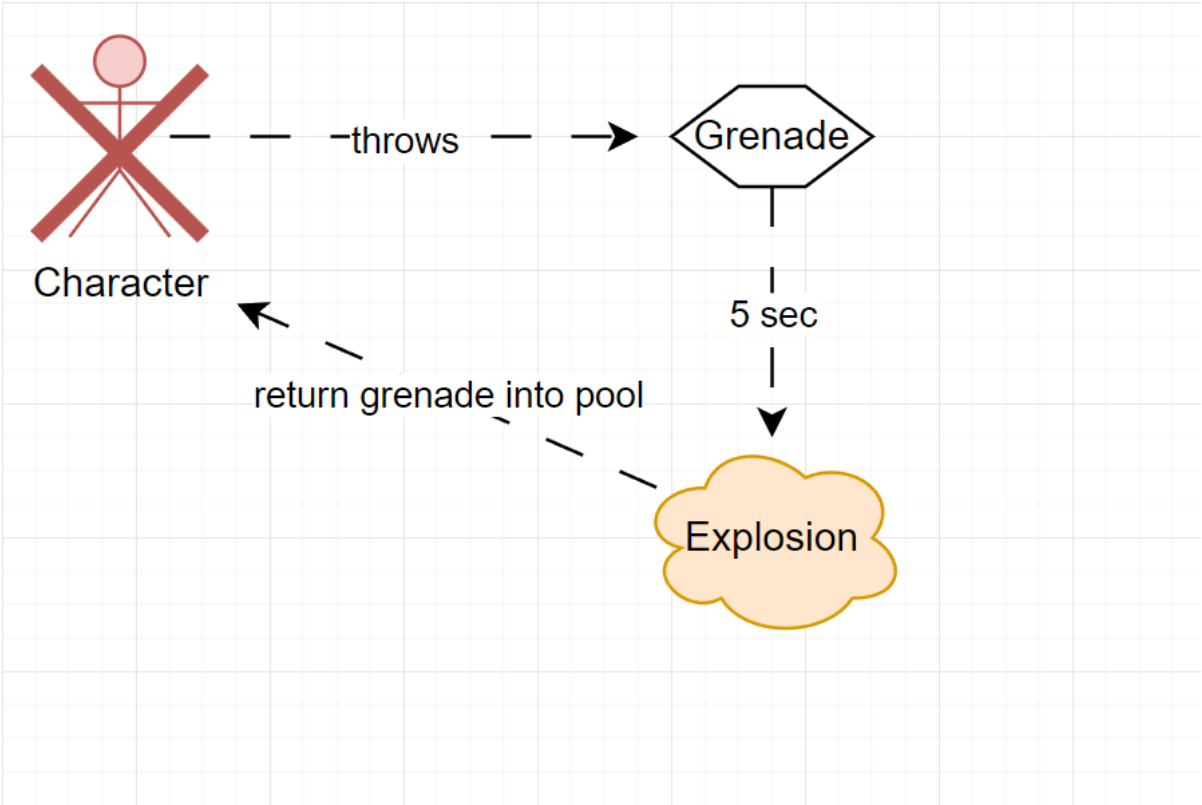
Bug with Grenades



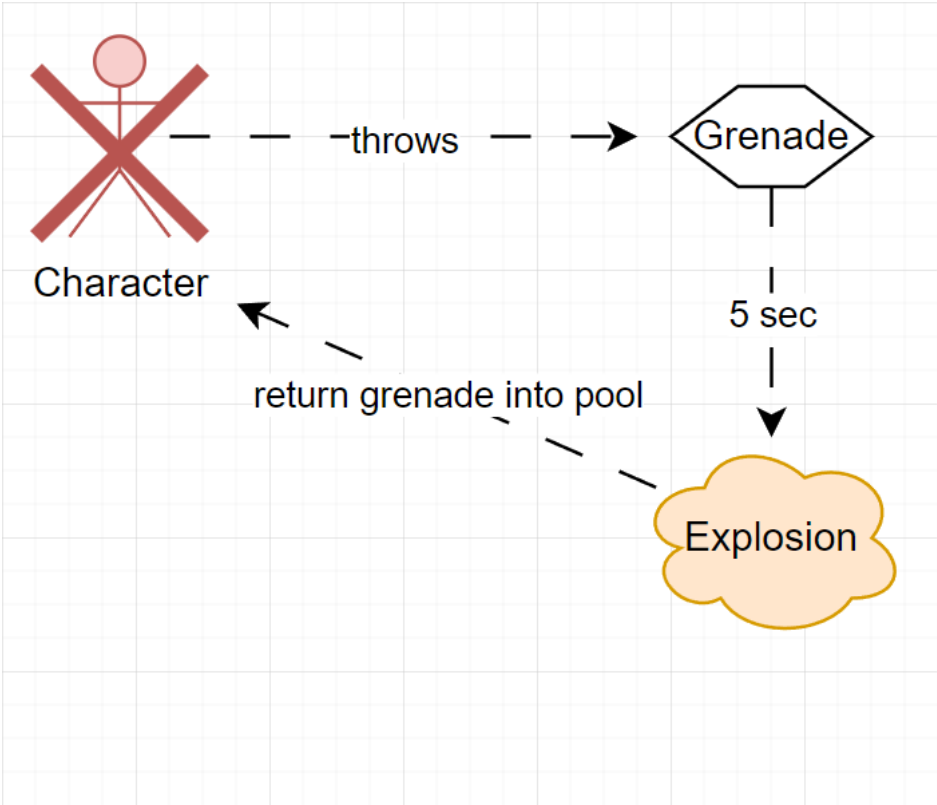
Bug with Grenades



Bug with Grenades



Bug with Grenades



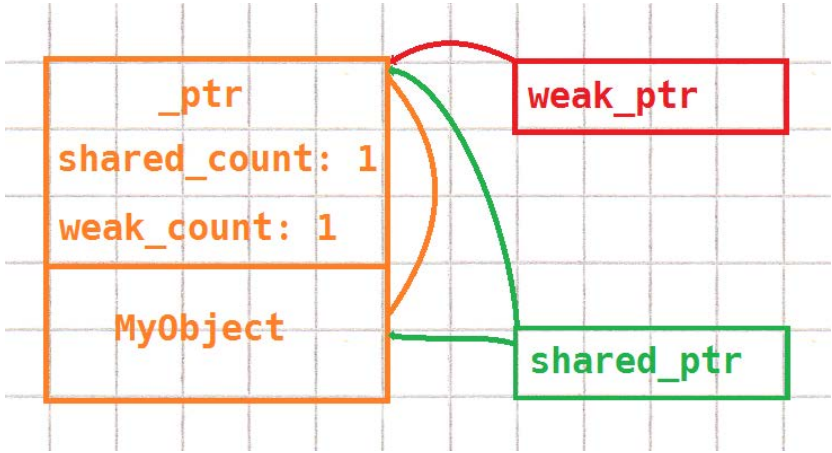
Is safety achievable?

Потрібен механізм перевірки існування пулу!

Варіанти:

1. Зберігати в Pool масив Proxy і сповіщати їх

1. Використовувати smart pointers



Shared From This

```cpp
template <typename T>
class SharedFromThis
{
private:
    weak_ptr<T> _weakThis;

public:
    shared_ptr<T> asShared();
    weak_ptr<T> asWeak();

    template <typename... Args>
    static shared_ptr<T> create(Args&&... args);

    ~SharedFromThis() = default;

    // Delete copying and moving
    SharedFromThis(const SharedFromThis&) = delete;
    SharedFromThis(SharedFromThis&&) = delete;
    SharedFromThis& operator=(const SharedFromThis&) = delete;
    SharedFromThis& operator=(SharedFromThis&&) = delete;

protected:
    // Intentionally hide constructor, so that only create() can be used
    SharedFromThis() = default;
};
```

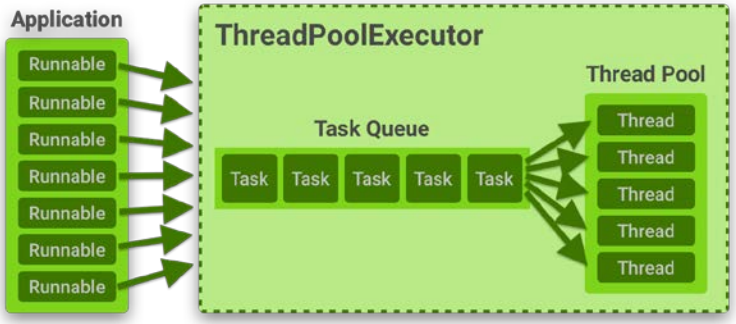
Safe Pool

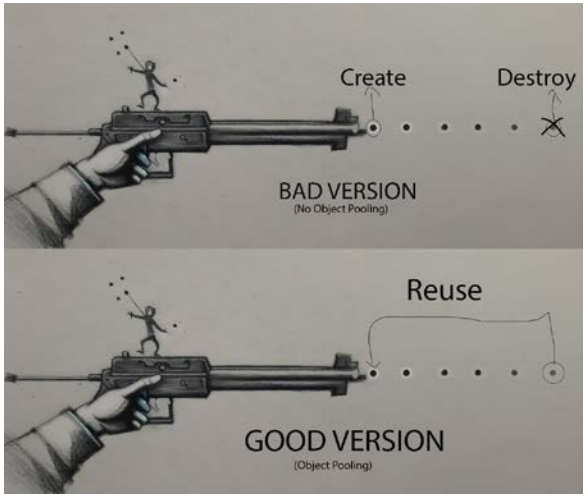
```cpp
template <typename T>
class SafePool : public SharedFromThis<SafePool<T>>
{
    friend class SharedFromThis<SafePool<T>>;

public:
    class Proxy { ... };

    ~SafePool();

    // Delete copying and moving
    SafePool(const SafePool&) = delete;
    SafePool(SafePool&&) = delete;
    SafePool& operator=(const SafePool&) = delete;
    SafePool& operator=(SafePool&&) = delete;

    void push(unique_ptr<T> object);
    Proxy pull();

protected:
    SafePool(const int size);

private:
    vector<unique_ptr<T>> _objects;
};
```

```cpp
class Proxy
{
private:
    friend class SafePool<T>;
    Proxy(weak_ptr<SafePool<T>> pool, unique_ptr<T> obj = nullptr);

    weak_ptr<SafePool<T>> _pool;
    unique_ptr<T> _object;

public:
    ~Proxy();

    Proxy(Proxy&&) = default;
    Proxy& operator=(Proxy&&) = default;

    T* operator->();
    T& operator*();

    operator bool() const;

    Proxy(const Proxy&) = delete;
    Proxy& operator=(const Proxy&) = delete;
};
```

Приклади використання

- Пул потоків (thread pool)
- Пул снарядів
- Пул персонажів
- Пул цілих чисел у Java
- …



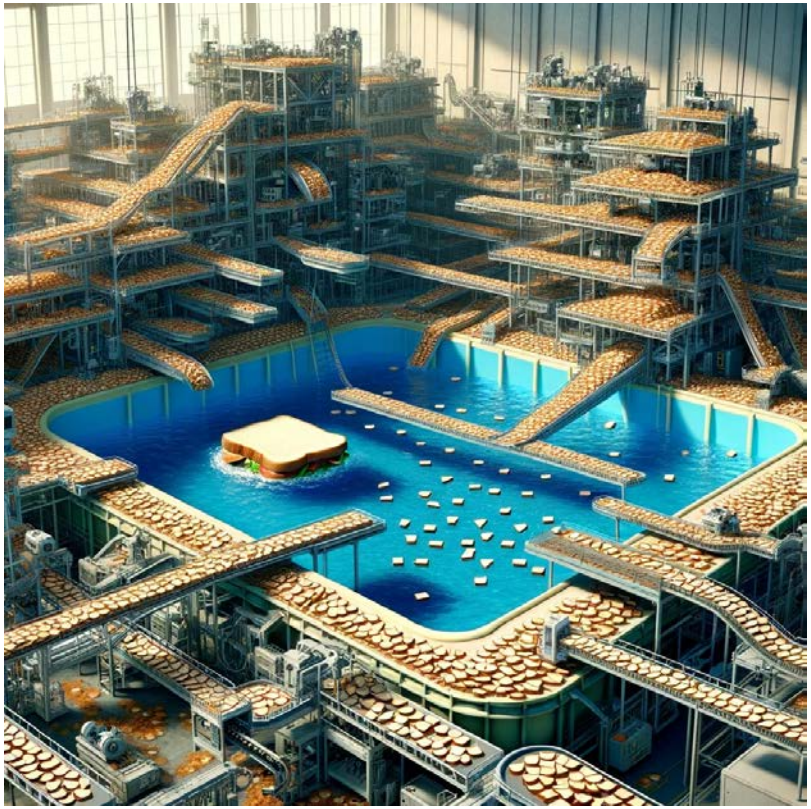


Недоліки

- Використання конструкторів/деструкторів обмежене
- Потрібен жорсткий контроль за очищенням об’єкта
- Збільшення використання пам’яті
- Довша ініціалізація програми

Про що варто пам’ятати?

- Дрібні об’єкти може бути “дешевше” створювати одразу
- Потрібен профайлінг і заміри продуктивності
- Багатопотоковість
- Паттерн завжди можна і треба змінювати для своїх цілей



Ідея розширення: стратегія обробки нестачі

- Повернути nullptr?
- Створити новий об’єкт і повернути його?
- Наповнити пул об’єктами повністю?
- Мати “запасний” об’єкт для цього?

Інші ідеї

- Адаптивний розмір пула
- Ініціалізатор об’єкта
- Time-slicing наповнення пула
- Використати placement new для наповнення пула

Дякую за увагу!

Критика? Запитання? Погрози?