

Gestión de la Memoria Dinámica (Object Pool, Allocators, etc.)

TPV 2
Samir Genaim

UNIVERSIDAD
COMPLUTENSE
DE MADRID
Lecture Notes
Samir Genaim
TPV2
Facultad de Informática
Universidad Complutense de Madrid

Lo que queremos conseguir ...

- ★ Pedir memoria contigua al principio del programa, usarla para la construcción de objetos y liberarla al final.
- ★ Ventajas: minimizamos la comunicación con el sistema operativo (`new/delete`) y mantenemos los objetos en memoria contigua para minimizar "cache misses".
- ★ Vamos a ver 2 versiones de object pools:
 - Object Pool I: crear (y inicializar) todos los objetos al principio del programa y simplemente reusarlos durante la ejecución, sin ejecutar la destructora cada vez que "liberamos" un objeto.
 - Object Pool II: sólo pedir la memoria al principio, pero la construcción y destrucción de los objetos se hace durante la ejecución en esa memoria. Liberar toda la memoria al final.

Object Pool (Type I)



Lecture Notes
Samir Genaim
TPV2
Facultad de Informática
Universidad Complutense de Madrid

Contexto y Objetivo

- ◆ Queremos hacer algún efecto en un juego que consiste en dibujar muchos objetos del mismo tipo:
 - ✓ disparar muchas balas
 - ✓ una explosión con muchas chispas
 - ✓ introducir muchos asteroides
 - ✓ etc.
- ◆ Cada uno de los objetos se representa con **struct** o **class** para llevar su información, operaciones, etc.
- ◆ Después de hacer el efecto deseado, los objetos "mueren" (no tienen ningún efecto en el juego)
- ◆ La cuestión es cómo vamos a crear estos clases?

Crear cuando lo necesitas

- ♦ Crear los objetos cada vez que necesitamos hacer el efecto y liberar la memoria después
- ♦ Puede afectar negativamente al juego en términos de tiempo de ejecución
 - ✓ Muchas peticiones al sistema operativo para pedir y liberar memoria dinámica — puede ser muy costoso en algunas plataformas
 - ✓ Fragmentación de la memoria: los objetos viven en distintas partes de la memoria algo que incrementa el número de "**cache miss**"
 - ✓ Siempre hay que construir/destruir los objetos usando la constructora/destructora — puede ser muy costoso en algunos casos

Crear una vez y reusar

- ◆ La idea principal del object pool (type I) es:
 - ✓ Crear los objetos al principio (p.ej., un array) y a lo mejor inicializarlos con alguna información
 - ✓ Pedir al pool un objeto cuando necesitas uno y el pool te devuelve uno **no usado**
 - ✓ Cuando dejas de usar el objeto, devuélvelo al pool para poder reutilizarlo

Object Pool: usando un array

```
template<typename T, std::size_t SIZE>
class ObjectPool {
public:
    ObjectPool() :
        pool_(), // 
        size_(SIZE), //
        used_(), //
        lastUsed_(SIZE - 1) {
    }

    virtual ~ObjectPool() {
    }

    ...

private:
    T pool_[SIZE];
    std::size_t size_;
    std::bitset<SIZE> used_;
    std::size_t lastUsed_;
};
```

El tipo de objetos y el tamaño son parámetros del template

Usando un array para los objetos del pool. T es una clase/struct y tiene que tener constructora por defecto.

Un bitset para marcar los usados

Indice del último objeto usado (para la búsqueda circular)

Object Pool: alloc y free

```
T* alloc() {  
    auto i = lastUsed_ + 1 % size_;  
    while (used_[i] && i != lastUsed_)  
        i = i + 1 % size_;  
    if (!used_[i]) {  
        used_[i] = true;  
        return pool_+i;  
    } else return nullptr;  
}
```

```
void free(T *p) {  
    auto idx = p-pool_;  
    assert(idx >= 0 && idx < size_);  
    used_[idx] = false;  
}
```

Buscar un objeto no usado (búsqueda circular)

Si hay uno, lo marca como usado y devuelve un puntero correspondiente, en otro caso devuelve nullptr

Marca el objeto como no usado

Object Pool: acceso a todos

A veces necesitamos inicializar todos los objetos al principio con alguna información (que no se puede hacer en la constructora por defecto) en lugar de hacerlo cada vez que usamos el objeto.

```
T& operator[](std::size_t idx) {  
    assert(idx >= 0 && idx < size_);  
    return &pool_ + idx;  
}  
  
std::size_t size() {  
    return size_;  
}  
  
iterator begin() {  
    return pool_;  
}  
  
iterator end() {  
    return pool_ + size_;  
}
```

Se puede dar acceso por indice (aunque el objeto no está en uso), así se puede recorrer a todos usando un bucle 'for'

Se puede definir un iterator y usar un bucle for-each. En este caso es suficiente usar el puntero como iterator:
using iterator = T*;

Object Pool: un array dinámico

```
template<typename T>
class ObjectPool {
public:
    ObjectPool(std::size_t size) :  
        pool_(new T[size]),  
        size_(size),  
        used_(size),  
        lastUsed_(size - 1) {  
}  
  
    virtual ~ObjectPool() {  
        delete [] pool_;  
    }  
  
private:  
    T *pool_;  
    std::size_t size_;  
    std::vector<bool> used_;  
    std::size_t lastUsed_;  
};
```

Sólo el tipo de los objetos, el tamaño lo pasamos por la constructora.

Usando un array dinámico para los objetos del pool. Lo liberamos en la destructora

No se puede usar std::bitset, porque requiere el tamaño durante la compilación. Usamos std::vector<bool>, que normalmente es especialización de std::vector para el tipo bool y usa representación basada en bits como std::bitset. También se puede usar utils/DynamicBitSet

Object Pool (Type II)



Lecture Notes
Samir Genaim
TPV2
Facultad de Informática
Universidad Complutense de Madrid

Lo que queremos conseguir ...

- ★ Pedir memoria contigua al principio del programa, pero sin crear los objetos (por ejemplo, memoria suficiente para 100 objetos de tipo A).
- ★ Crear un objeto (y ejecutar su constructora) usando la memoria que ya tenemos asignada.
- ★ Destruir un objeto (ejecutar su destructora) sin devolver la memoria al sistema operativo (para reusar esa memoria).
- ★ Liberar la memoria asignada al final del programa.
- ★ La diferencia del concepto anterior es que los objetos se crean y destruyen durante la ejecución.

Qué hacen new y delete

- ◆ Cuando ejecutamos “`p = new A(...)`”, estamos haciendo 2 operaciones
 - ✓ **Allocate**: pedir al sistema operativo memoria (contigua) de tamaño `sizeof(A)`
 - ✓ **Construct**: inicializar dicha memoria con la información de `A` y ejecutar su constructora
- ◆ Cuando ejecutamos “`delete p`”, estamos haciendo 2 operaciones
 - ✓ **Destruct**: ejecutar la destructora del objeto al que señala `p`
 - ✓ **Deallocate**: devolver la memoria al sistema operativo.

Allocation y Construction

Para gestionar la memoria en un programa, necesitamos un mecanismo que nos permite separar allocation y construction:

✓ **Allocate:** `_VSTD::__libcpp_allocate(n,alignof(T))` pide al sistema operativo **n bytes** de memoria contigua, devuelve un puntero **void*** - se puede usar `malloc(n)` como en C.

✓ **Construct:** `::new(p) A(...)` construye un objeto de tipo **A** usando la memoria a donde señala **p**

alignof(T) es para que use una dirección que minimiza los accesos suponiendo que vamos a usar esa memoria para almacenar objetos de tipo T (ya lo hemos visto con el tema de packing, padding, etc).

Destruction y Deallocation

... también necesitamos un mecanismo que nos permite separar destruction y deallocation:

- ✓ **Destruct:** `p->~T()` llama a la destructora del objeto al que señala p (p es de tipo T^* , pero el objeto puede ser de sub-tipo de T, llamaría a la destructora correcta si es **virtual**)
- ✓ **Deallocate:** `VSTD::libc++_deallocate(p,n,alignof(T))` libera (devuelve) n bytes a partir de la dirección donde señala p – se puede usar `free(p)` como en C. El objetivo de usar `alignof(T)` es como antes ...

Ejemplo I

```
A *p = static_cast<A*>(malloc(sizeof(A)));
```

```
::new (p) A(1);
```

```
p->~A();
```

```
::new (p) A(4);
```

```
p->~A();
```

```
free(p);
```

Pedir memoria de tamaño sizeof(A)

Construir una instancia de A en la memoria a donde señala p

Destruir el objeto

Destruir el objeto

Construir otra instancia en la misma memoria

Liberar la memoria – NO EJECUTA LA DESTRUCTORA, SÓLO LIBERA LA MEMORIA

Ejemplo II

```
A *p = static_cast<A*>(malloc(10*sizeof(A)));
::new (p) A(1);
::new (p+1) A(4);
::new (p+2) A(3);
```

Pedir memoria para
10 objetos de tipo A

```
cout << p[0].getN() << endl;
cout << (p+1)->getN() << endl;
cout << p[2].getN() << endl;
```

Construir 3 instancias

```
p[1].~A();
(p+2)->~A();
```

Destruir

Usar las instancias, como
objetos o punteros a
objetos

```
::new (p+1) A(55);
cout << p[1].getN() << endl;
```

Reusar memoria

```
p[0].~A();
p[1].~A();
```

```
free(p);
```

Liberar la memoria – NO EJECUTA LA
DESTRUCTORA, SÓLO LIBERA LA MEMORIA

Object Pool (Type II)

```
template<typename T>
class ObjectPool {
public:
    ObjectPool(std::size_t size) :  
        size_(size),  
        lastUsed_(size - 1),  
        used_(size) {  
        pool_ = static_cast<T*>(malloc(size_* sizeof(T)));  
    }  
  
    virtual ~ObjectPool() {  
        ::free(pool_);  
    }  
  
private:  
    std::size_t size_;  
    T *pool_;  
    std::vector<bool> used_;  
    std::size_t lastUsed_;  
};
```

Pedir toda la memoria necesaria al principio

Liberar la memoria al final, usamos `::free` para distinguirlo del `free` de la clase `ObjectPool`

`vector<bool>` para indicar que posiciones en `pool_` están ocupados.

Object Pool (Type II)

```
template<typename ...Ts>
T* alloc(Ts &&...args) {
    auto i = lastUsed_ + 1 % size_;
    while (used_[i] && i != lastUsed_)
        i = i + 1 % size_;
    if (!used_[i]) {
        used_[i] = true;
        ::new (pool_ + i) T(std::forward<Ts>(args)...);
        return pool_ + i;
    } else return nullptr;
}
```

```
void free(T *p) {
    auto idx = p - pool_;
    assert(used_[idx]);
    p->~T();
    used_[idx] = false;
}
```

Buscar posición libre
y construir el objeto
en esa posición

Destruir el objeto y marcarlo
como no usado

malloc/free memoria C++

En C++, en lugar de usar

```
malloc(size_ * sizeof(T))  
free(pool_);
```

Mejor usar las siguientes instrucciones porque tienen en cuenta el "alignment" de los datos en la memoria

```
_VSTD::__libcpp_allocate(size_ * sizeof(T),  
                           _LIBCPP_ALIGNOF(T));  
  
_VSTD::__libcpp_deallocate((void*)pool_,  
                           size_ * sizeof(T),  
                           _LIBCPP_ALIGNOF(T));
```

Usando std::allocator

```
template<typename T>
class ObjectPool {
public:
    ObjectPool(std::size_t size) :
        size_(size),
        lastUsed_(size - 1),
        used_(size),
        alloc_() {
        pool_ = alloc_.allocate(size_);
    }

    virtual ~ObjectPool() {
        alloc_.deallocate(pool_, size_);
    }

private:
    ...
    std::allocator<T> alloc_;
};
```

En C++ se puede usar
std::allocator<T>

Es como usar
__VSTD::__libc++_allocate
__VSTD::__libc++_deallocate

Resumen (Type III)

- ◆ Siempre usar std::allocator (o otro allocator) para las operaciones: allocate y deallocate
- ◆ Hay que ajustar el máximo numero de objetos que se puede crear (el tamaño de **pool**) a las necesidades del programa, no pedir más memoria de lo que necesitas.
- ◆ Se puede cambiar el tamaño del pool durante la ejecución si necesitamos más, pero ejemplo la clase std::vector lo hace (**redimensiona** y mueve los objetos a la nueva memoria – impleméntalo para MyVector)
- ◆ Se puede hacer **ObjectPool** para gestionar varios tipos de objetos a la vez, pero esto no es trivial si los objetos tienen distintos tamaños