Curso: C++ Moderno
Período: 1º semestre/2017
Aluno: Arthur Nunes de Paiva Santos Queiroz

Lista de exercícios Módulo 08: Gerenciamento de memória

Questões

Questão 1

Qual é o problema do smart pointer std:auto_ptr de C++98/03?

std:auto_ptr se comporta como um std::unique_pointer, mas sua semântica de transferência não é intuitiva, pois ele é mutado quando é atribuído à outra variável. std::unique_ptr, por outro lado, não é copiável, e sua posse tem que ser transferida explicitamente.

Fonte: http://stackoverflow.com/questions/3697686/why-is-auto-ptr-being-deprecated

Questão 2

Implemente a classe smart_ptr estudada nos slides. Verifique se ela funciona de acordo com sua expectativa (você pode, por exemplo, adicionar std::cout nos métodos para fins de debug).

smart_pointer.h

```
#ifndef SMART_PTR_H
#define SMART_PTR_H

#include <iostream>

template <class T>
class smart_ptr
{
public:

smart_ptr(T* ptr)
: ptr_(ptr){
   std::cout << "Created smart_ptr" << std::endl;
}
```

```
~smart_ptr() {
   delete ptr_;
   std::cout << "Destroyed smart_ptr" << std::endl;
}

T* get(){
   return ptr_;
}

T* release(){
   T* t = ptr_;
   ptr_ = nullptr;
   return t;
}

private:
   T* ptr_;
};


#endif // SMART_PTR_H
```

main.cpp

```
#include <iostream>
#include "smart_ptr.h"
#include <string>

int main()
{
        smart_ptr<std::string> ptr(new std::string("Hello world"));

   return 0;
}
```

Questão 3

Visto que std::unique_ptr pode armazenar um deleter em seu segundo parametro template, como é possivel que ele não traga overhead de tamanho em relação a um ponteiro nativo?

Isso ocorre porque o deleter é armazenado como um membro de std::unique_ptr da seguinte forma:

```
typedef std::tuple<typename _Pointer::type, _Dp>  __tuple_type;
__tuple_type  _M_t;
```

Sendo _Dp uma classe vazia:

```
typename _Dp = default_delete<_Tp>
```

Uma vez que std::tuple implementa (dependendo do compilador utilizado) "empty base class optimization", se o ponteiro for instanciado com o deleter padrão, o tamanho da tuple corresponde apenas ao tamanho do raw pointer.

Fonte: http://stackoverflow.com/questions/13460395/how-can-stdunique-ptr-have-no-size-overhead
http://www.stroustrup.com/bs_faq2.html#sizeof-empty
https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01404.html
https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-api-4.5/a01066_source.html

Questão 4

Investigue as vantagens/desvantagens de std: :unique_ptr com arrays. Compare com std::vector.

O uso de std::unique_ptr<T[]> não apresenta vantagens expressivas sobre o uso de std::vector, sendo usado apenas como último recurso quando o uso de std::vector não for possível.

Fonte: http://stackoverflow.com/a/16711846/702828

Questão 5

Qual o problema do codigo A?

A função f cria um novo std::shared_ptr a partir do raw pointer do primeiro smart_ptr, de forma que ambos tentaram controlar o ciclo de vida do pointeiro sem ter conhecimento um do outro, causando erros por conta de deleções adicionais.

Questão 6

Cria um std:unique_ptr e um std:shared_ptr que realizam um log perante a destruição do objeto.

unique_ptr.cpp

```
#ifndef UNIQUE_PTR_H
#define UNIQUE_PTR_H

#include <iostream>

template <class T>
class unique_ptr
{
public:

unique_ptr(T* ptr)
: ptr_(ptr){
   std::cout << "unique_ptr<T>: construtor" << std::endl;
}

~unique_ptr() {
         if(ptr_){
      std::cout << "unique_ptr<T>: internal pointer deleted" << std::endl;
         }
   delete ptr_;
   std::cout << "unique_ptr<T>: destructor" << std::endl;
}

unique_ptr<T>(const unique_ptr<T>& ptr) = delete;
void operator=(const unique_ptr<T>& ptr) = delete;

unique_ptr(unique_ptr<T>&& ptr) : ptr_(ptr.ptr_)
{
   ptr.ptr_ = nullptr;
   std::cout << "unique_ptr<T>: transfer constructor" << std::endl;
}

unique_ptr<T>& operator=(unique_ptr<T>&& ptr){
```

```
    ptr_ = ptr.ptr_;
    ptr.ptr_ = nullptr;
    std::cout << "unique_ptr<T>: transfer assignment" << std::endl;
    return *this;
}

T* get(){
    return ptr_;
}

T* release(){
    T* t = ptr_;
    ptr_ = nullptr;
    return t;
}

private:
    T* ptr_;
};


#endif // UNIQUE_PTR_H
```

shared_ptr.cpp

```
#ifndef SHARED_PTR_H
#define SHARED_PTR_H

#include <iostream>

template <class T>
class shared_ptr
{

public:

shared_ptr(T* ptr)
: ptr_(ptr), counter_(new int(0)){
    std::cout << "shared_ptr<T>: constructor" << std::endl;
    increase();
}

~shared_ptr() {
    std::cout << "shared_ptr<T>: destructor" << std::endl;
    decrease();
}

shared_ptr(const shared_ptr<T>& ptr) : ptr_(ptr.ptr_), counter_(ptr.counter_){
    std::cout << "shared_ptr<T>: copy constructor" << std::endl;
    increase();
}

shared_ptr<T>& operator=(const shared_ptr<T>& ptr){
    std::cout << "shared_ptr<T>: assignment" << std::endl;
    if(this!=&ptr)
    {
        ptr_ = ptr.ptr_;
        counter_ = ptr.counter_;
```

```
        increase();
    }
}

shared_ptr(shared_ptr<T>&& ptr) : ptr_(ptr.ptr_), counter_(ptr.counter_)
{
    ptr.ptr_ = nullptr;
    ptr.counter_ = nullptr;
    std::cout << "shared_ptr<T>: transfer constructor" << std::endl;
}

shared_ptr<T>& operator=(shared_ptr<T>&& ptr){
    ptr_ = ptr.ptr_;
    counter_ = ptr.counter_;
    ptr.ptr_ = nullptr;
    ptr.counter_ = nullptr;
    std::cout << "shared_ptr<T>: transfer assignment" << std::endl;
    return *this;
}

void increase(){
    if(counter_) {
        (*counter_)++;
        std::cout << "shared_ptr<T>: counter incremented = " << *counter_ << std::endl;
    }
}

void decrease(){
    if(!counter_){
        return;
    }

    --(*counter_);
    std::cout << "shared_ptr<T>: counter decremented = " << *counter_ << std::endl;

    if(!*counter_){
        delete ptr_;
        delete counter_;
        ptr_ = nullptr;
        counter_ = nullptr;
        std::cout << "shared_ptr<T>: internal pointer deleted" << std::endl;
    }
}

T* get(){
    return ptr_;
}

private:
    T* ptr_ = nullptr;
    int* counter_ = nullptr;
};


#endif // SHARED_PTR_H
```

main.cpp

```cpp
#include <iostream>
#include "unique_ptr.h"
#include "shared_ptr.h"
#include <string>

int main()
{
	unique_ptr<std::string> uptr(new std::string("Hello world"));
	//unique_ptr<std::string> uptr2(uptr);	// exception, copy not allowed
	//unique_ptr<std::string> uptr3 = uptr;	// exception, copy not allowed
	unique_ptr<std::string> uptr4(std::move(uptr));  // ptr2 gets empty
	unique_ptr<std::string> uptr5 = std::move(uptr4);	// ptr3 gets empty

	shared_ptr<std::string> ptr(new std::string("Hello world"));
	shared_ptr<std::string> ptr2(ptr);
	shared_ptr<std::string> ptr3 = ptr;
	shared_ptr<std::string> ptr4(std::move(ptr2));	// ptr2 gets empty
	shared_ptr<std::string> ptr5 = std::move(ptr3);  // ptr3 gets empty

   return 0;
}
```

Questão 7

Implemente sua propria my::make_unique de maneira equivalente a std::make_unique.

main.cpp

```cpp
#include <iostream>
#include <string>
#include <memory>

// http://stackoverflow.com/questions/17902405/how-to-implement-make-unique-function-in-c11
// http://stackoverflow.com/questions/7038357/make-unique-and-perfect-forwarding
// https://herbsutter.com/gotw/_102/
// http://stackoverflow.com/questions/27594731/what-are-the-6-dots-in-template-parameter-packs
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
   return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

int main()
{
   auto ptr = make_unique<std::string>("Hello world");

   return 0;
}
```

Questão 8

Explique o erro de compilação do codigo B e corrija-o de duas maneiras diferentes.

Std::vector::push_back() realiza a cópia do argumento passado e std::unique_ptr não permite cópias.

O problema pode ser resolvido de duas maneiras:

- Movendo-se o objeto std::unique_ptr para dentro do vetor.
- Utilizando um std::shared_ptr no lugar de std::unique_ptr.

main.cpp

```cpp
#include <iostream>
#include <string>
#include <memory>
#include <vector>

struct A{};

// void e() {
//         std::vector<std::unique_ptr<A>> v;
//         std::unique_ptr<A> up(new A);
//         v.push_back(up);
//         v.clear();
// }

void f() {
        std::vector<std::unique_ptr<A>> v;
        std::unique_ptr<A> up(new A);
        v.push_back(std::move(up));
        v.clear();
}

void g() {
        std::vector<std::shared_ptr<A>> v;
        std::shared_ptr<A> up(new A);
        v.push_back(up);
        v.clear();
}

int main()
{
        f();
        g();


   return 0;
}
```

Questão 9

Implemente um pequeno rastreador de memória sobrescrevendo operator new e delete globalmente.

main.cpp

```cpp
#include <iostream>
#include <string>
#include <memory>
#include <map>
#include <vector>
```

```
// http://www.dreamincode.net/forums/topic/73544-how-to-determine-the-size-of-what-a-void-pointer-is-pointing-at/
// http://stackoverflow.com/questions/8186018/how-to-properly-replace-global-new-delete-operators

// int memory_usage = 0;
// std::map<void*, size_t> allocated_objects;

void* operator new (size_t n) {
	void* p = malloc(n);

	std::cout << "new: memory = " << n << std::endl;

	// The code below throws a segmentation fault due to stackoverflow.
	// memory_usage += n;
	// allocated_objects[p] = n;
	// std::cout << "new: memory usage = " << memory_usage << std::endl;

	return p;
}

void operator delete(void* p) {
	free(p);

	std::cout << "delete" << std::endl;

	// memory_usage-=allocated_objects[p];
	// allocated_objects.erase(p);
	// std::cout << "delete: memory usage = " << memory_usage << std::endl;
}


struct A{
	int value1;
};

struct B{
	int value1;
	int value2;
};

struct C{
	int value1;
	int value2;
	int value3;
};


int main()
{
	A* a = new A();
	B* b = new B();
	C* c = new C();
	delete(a);
	delete(b);
	delete(c);

   return 0;
}
```

Questão 10

Implemente um pool de memória para 1000 objetos de um tipo T, partindo de MemoryPool.

MemoryPool.h

```
#ifndef MEMORY_POOL_H
#define MEMORY_POOL_H

#include <vector>
#include <new>
#include <iterator>
#include <iostream>

template <class T>
class MemoryPool
{
private:
	static constexpr size_t POOL_SIZE = 1000;
	std::vector<size_t> freed_;
	size_t next_ = 0;
	T pool_[POOL_SIZE];

public:
	MemoryPool(){}
	virtual ~MemoryPool(){}

	void* allocate(size_t){
		int index = 0;

		if(freed_.empty()){

			if(next_ >= POOL_SIZE){
				throw std::bad_alloc();
			}

			index = next_++;

			std::cout << "MemoryPool.allocate: using new memory." << std::endl;

		} else{

			// http://stackoverflow.com/questions/12600330/pop-back-return-value
			index = freed_.back();
			freed_.pop_back();

			std::cout << "MemoryPool.allocate: using freed memory." << std::endl;

		}

		std::cout << "MemoryPool.allocate: index = " << index << std::endl;

		void* p = &pool_[index];
		return p;

	}

	void release(void* p){
		freed_.push_back( std::distance(pool_, (T*)p));
	}
```

```
};

#endif // MEMORY_POOL_H
```

main.cpp

```
#include <iostream>
#include "MemoryPool.h"
#include <vector>

struct A{
        int value_ = 0;
        A(){}
        A(int value) : value_(value){

        }
        virtual ~A(){};
};

void* operator new (size_t n, MemoryPool<A>* pool) {
        std::cout << "new: custom memory pool."<< std::endl;
        void* p = pool->allocate(n);

        return p;
}

void operator delete (void* p, MemoryPool<A>* pool) {
        std::cout << "delete: custom memory pool."<< std::endl;
        pool->release(p);
}

MemoryPool<A> pool;

int main()
{
        std::vector<A*> instances;

        for (int i = 0; i < 10; i++){;
                instances.push_back(new (&pool) A(i));
                std::cout << "\tA.value = " << instances[i]->value_ << std::endl;
        }

        for (int i = 0; i < 10; i+=2){
                std::cout << "\tdeleting i = " << i << ", A = " << instances[i]->value_ <<
std::endl;
                operator delete(instances[i], &pool);
        }

        for (int i = 0; i < 10; i+=2){
                instances[i] = new (&pool) A(i+10);
                std::cout << "\tA.value = " << instances[i]->value_ << std::endl;
        }

        std::cout << "RESULT:" << std::endl;
        for (int i = 0; i < 10; i++){;
                std::cout << "\tA.value = " << instances[i]->value_ << std::endl;
        }
```

```
    return 0;
}
```

Questão 11

Mostre como o "problema" de visibilidade de nomes de C++ se reproduz no caso de sobrescrita dos operator new e delete.

A resolução dos operadores new/delete é realizada em tempo de linkagem. Apesar de das regras de linkagem de bibliotecas tipo “*.so” especificarem que o símbolo utilizado é aquele da biblioteca carregada primeiro, o padrão determina que operadores new/delete definidos pelo usuário substituem as implementações default.

O compilador implementa este comportamento marcando as funções padrões como referências "fracas", que são sobrescritas pelo linker se outro simbolo com o mesmo nome for encontrado.

Fonte: http://stackoverflow.com/questions/37041819/without-root-access-run-r-with-tuned-blas-when-it-is-linked-with-reference-blas/37064043#37064043

http://stackoverflow.com/a/8186116/702828

http://stackoverflow.com/questions/37145235/c-custom-global-new-delete-overriding-system-libraries