VGP336 - Memory Management

Instructor: Peter Chan

What we talked about so far ...

Back in the days, developers used an OO hierarchy for managing game objects.

The advantages include:

- OOP is that it is intuitive
- Allows big teams to collaborate easily due to proper encapsulation
- Easy code reuse and extensibility via inheritance and polymorphism

However, OOP tend to result in large and rigid class hierarchy which is hard to modify and maintain. Heavy use of virtual implies low performance. Plus multiple inheritance can be nasty.

OOP is Intuitive

My C++ instructor used to tell me to circle all the **nouns** and ***verbs*** in my assignments. **Nouns** then become classes, while ***verbs*** become functions. It does not cover everything, but serves as a good starting point for any project.

Example:

Your assignment is to implement a game where you control a **hero** and ***attack*** **enemies** with **weapons** and ***collect*** **gold**, you beat the game if you ***kill*** the **final boss**.

Multiple Inheritance ...

The major turning point is that people really want to avoid having to deal with multiple inheritance, or sometimes referred as the **dreaded diamond**.

Instead, people started to split base classes into smaller parts so they can form new inheritance chain to avoid multiple parent classes.

They also started using the “has-a” composition pattern instead of “is-a” pattern to avoid deep hierarchy and virtual calls.

This eventually leads to what is known as the **component-based architecture**.

Common Memory Related Problems

Before we jump into implementing our component-based system. First we want to focus on tackling some common memory related problems in software development. These include:

- Memory Leak
- Memory Stomp
- Dangling Pointers
- Stack Overflow (however, this is likely an algorithm level problem)
- Memory Fragmentation

Memory Leak

A running process has a limited amount of memory that they can request from the operating system. Memory leak occurs when the process has allocated memory and fail to return ownership back to the system.

This means the available space will reduce overtime and eventually lead to the system throwing a std::bad_alloc exception and crashes the process.

Memory leak typically happens when the programmer forget to free/delete allocations or is releasing them inappropriately.

Example

```
for (size_t i = 0; i < 1000; ++i)
{
    int* intArray = new int[1000000];
    intArray[0] = 42;
    intArray[10] = 7;
    printf("%d: intArray[0] is %d\n", i, intArray[0]);
    printf("%d: intArray[10] is %d\n", i, intArray[10]);
    printf("***\n");
    delete intArray;    // <-- Wrong deletion here
}
```

Memory Stomp

Memory stomp occurs when you write to a memory location that is used by something else.

Depending on where you are stomping memory, different errors can occur including Access Violation exception, Stack Corruption, or worse yet, random Runtime Logic Errors.

Typically, memory stomps can occur when the programmer is writing to out-of-bound memory locations, or writing to dangling pointers.

Example

```
int intArray[10];
int index = 0;
while (true)
{
    intArray[index] = 42;
    printf("intArray[%d] is now 42.\n", index);
    ++index;
}
```

Dangling Pointer

Dangling pointers are pointers that reference memory locations where the original object no longer exists.

This is usually due to bad ownership management. In this context, ownership means “who is responsible for clean up”.

Dangling pointers can happen subtly. For example, you may be access an array that has already been deleted, or you are holding on to an object pointer passed to you where the object has been destroyed.

Example

```
int* intArray = new int[10];
for (size_t i = 0; i < 10; ++i)
    intArray[i] = 42;
delete[] intArray;
for (size_t i = 0; i < 10; ++i)
    printf("intArray[%d] = %d\n", i, intArray[i]);
```

Stack Overflow

Every time you call a function, a stack frame is pushed onto your stack memory. The stack frame holds information including: local variables used by the function, the return address when you exit the function, and function arguments.

However, there is a fixed size limit to your stack memory. Therefore, if you call too many functions, you may encounter a stack overflow and the process will crash.

Stack overflow typically occurs when you have deep recursion or when you have bad loop conditions.

Example

```
int Fibonacci(int i) {
    if (i == 0 || i == 1)
        return 1;
    return Fibonacci(i - 1) + Fibonacci(i - 2);
}

void main() {
    printf("Fibonacci(5) is %d\n", Fibonacci(5));
    printf("Fibonacci(-5) is %d\n", Fibonacci(-5));
}
```

Memory Fragmentation

Memory fragmentation occurs when large chunks of available memory for your process are being split up by tiny allocations and become unusable.

This means that although you have enough free space in total, your allocation will fail with std::bad_alloc exception since the available memory are scattered.

This usually happens when you have repeated allocations and deallocations of different sizes over an extended run time.

Example

```
std::vector<int*> intPointers;
while (true)
{
    int random = rand() % 100;
    if (random > 50)
    {
        if (random > 80)
            intPointers.push_back(new int[100000000]);
        else
            intPointers.push_back(new int[1000]);
    }
    else if (!intPointers.empty())
    {
        int randomIndex = rand() % intPointers.size();
        delete[] intPointers[randomIndex];
        intPointers[randomIndex] = intPointers.back();
        intPointers.pop_back();
    }
}
```

How do we deal with them?

There are many solutions available to tackle these memory problems. For examples, developers may choose to implement **custom memory allocators** so they can monitor and manage allocations directly.

Other techniques include adding **memory markers** and **memory barriers** to detect dangling pointers and out-of-bound access.

The C++ standard also provides their own solutions include std::unique_ptr, std::shared_ptr and std::weak_ptr.

Example - Memory Barrier

```
int* SafeIntAlloc(size_t size)
{
    // Allocate 3 extra integers, two before the array
    // and one after. We will store the hex-word 0xbeefbeef
    // as markers of the begin and end of the allocation,
    // as well as the request size.
    // e.g.
    // [bfbf][size][ ..... int array ..... ][bfbf]
    int* memory = (int*)malloc(sizeof(int) * (size + 3));
    memory[0] = 0xbeefbeef;
    memory[1] = size;
    memory[size + 2] = 0xbeefbeef;
    return memory + 2;
}
```

Example - Memory Barrier

```
void SafeIntDelete(int* memory)
{
    try {
        // Check if our markers are modified, if so, the array
        // was stomped and we throw an execption
        if (memory[-2] != 0xbeefbeef || memory[memory[-1]] != 0xbeefbeef) {
            throw std::bad_alloc();
        }
    } catch (std::exception e) {
        ASSERT(false, "Memory fence overwritten!");
    }
    free(memory - 2);
}
```

Example - Memory Barrier

Usage:

```
//int* intArray = new int[100];
int* intArray = SafeIntAlloc(100);
for (size_t i = 0; i <= 100; ++i)
    intArray[i] = 42;
SafeIntDelete(intArray);
//delete[] intArray;
```

Custom Allocators and Handles

For our engine, we will be implementing the following memory management mechanism:

- BlockAllocator
- TypedAllocator
- ArenaAllocator
- Handles

BlockAllocator

- Mixing large and small allocations can cause memory fragmentation
- Meant for small blocks (e.g. anything up to 64k)
- Avoids memory fragmentation
- Constant time allocation and deallocation by tracking freeslots
- C-style interface (i.e. malloc/free)

malloc versus new

malloc	new
C-style, requires **free**	C++ style, requires **delete** or **delete[]**
typeless, meaning you need to specify size and returns void*	type aware, meaning you just need to specify count and no casting
allocates memory from the heap	allocates memory from the heap, then calls the constructor (**delete** will perform the reverse)

Placement New

Placement new is a variation new operator in C++. Normal new operator does two things : (1) Allocates memory (2) Constructs an object in allocated memory.

Placement new allows us to separate above two things. In placement new, we can pass a preallocated memory and construct an object in the passed memory.

Example:

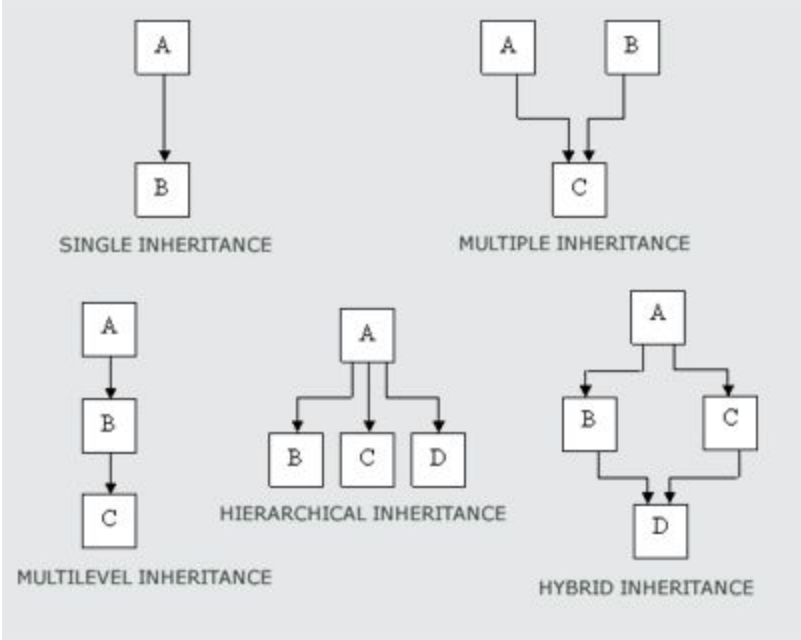
```
char* ptr = malloc(sizeof(T));   // allocate memory
T* tptr = new (ptr) T;           // construct in allocated storage ("place")
tptr->~T();                      // destruct
free(ptr);                       // deallocate memory
```

Public, Protected and Private Inheritance

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Source: https://www.geeksforgeeks.org/inheritance-in-c/

Types of Inheritance



Source: http://www.cppforschool.com/tutorial/inheritance.html

TypedAllocator

- Privately inherit from BlockAllocator since we want to use Allocate and Free internally, but do not want to expose them to the user
- This also means user cannot use a BlockAllocator* to point at it
- C++ style interface (i.e. new/delete)
- Calls constructor and destructor automatically
- Can support overloaded constructors via Variadic Template and Perfect Forwarding

References:
https://en.cppreference.com/w/cpp/language/parameter_pack
https://en.cppreference.com/w/cpp/utility/forward