**Smart Pointers**

\* \* \*

**Abstract**

When it comes to coding in C++, memory management can break the difference between your code usability and readability. If memory management isn’t taken into account, it could potentially hinder your code and make simplicity into chaos. There are several components that can aid a programmer in properly deleting objects to maximize the memory capability and one of the most important ones in C++ is Smart Pointers. Like its predecessor, the raw pointer, smart pointers offer the same functions with an added property, which is the ability to automatically delete objects when they become out of scope or deallocated. Smart pointers were created to deal with these kind of issues effectively while relieving the coder from doing the manual work of memory management. This paper will dwell deeper into smart pointers and its subclasses to properly demonstrate their different functions and determine their efficacy in the C++ programming language.

***Keywords***

*Smart Pointers, Dynamic Allocation, Shared Pointers, Unique Pointers,*

**Introduction**

As mentioned previously, smart pointers are useful when it comes to managing resources, fixing memory leaks, and proper use of memory. When coding, a lot of pointer issues may arise because programmers themselves must keep in mind where the pointer context is in at all times. Forgetting something may result in a bug. Smart Pointers come in handy because all the details are automatically fixed without any manual performance. Issues like going out of scope, forgetting to initialize pointers, or

deletion function calls are all handled under the hood. In C++ a smart pointer acts as a wrapper class over a pointer with the operator \* and -> overloaded. The libraries provide smart pointers in three different forms: unique\_ptr, shared\_ptr, and weak\_ptr that will be discussed further on as well as garbage collection and how unused data is handled and deleted.

**Method**

*Unique Pointers*

A Unique Pointer is a smart pointer that owns and manages an object through a pointer and gets rid of that object as soon as the unique\_ptr goes out of scope. As the name implies, a unique pointer can hold only one copy of the object at a time and can be initialized with a pointer upon its creation or assigned to one later on.

A Unique Pointer possesses two important components: a stored pointer and a stored deleter. A stored pointer is the pointer to the object its handling and can be altered by an assignment operation or by calling the reset member. Additionally, it can be accessed for reading by utilizing the get or release members. The stored deleter component is a callable object that can take an argument of the same type just like the stored pointer. It can be modified through an assignment operation and accessed using the get\_deleter member.

Member types of a unique\_ptr as are follow:

|  |  |
| --- | --- |
| **Member Type** | **Definition** |
| element\_type | The type of object being managed |
| deleter\_type | The type of the *stored deleter*. Default: [default\_delete](http://www.cplusplus.com/default_delete)<T> |
| pointer | The pointer type |

*Weak Pointers*

Weak Pointers provide the solution to specific problems when working with shared pointers. The weak\_ptr, henceforth referred to as WP, is a type of smart pointer that provides a solution to problems associated with accessing an object that may or may not be "alive". This is the basic problem that WPs solve. Here we explain what a WP is, what a

WP does and how one works.

What is a weak\_ptr? A WP is itself an object that stores a reference to an object pointed to by a shared pointer. The WP is a tool for programmers to safely access objects that may not exist. Additionally, the WP is a tool to break a reference cycle that causes memory leaks.

What does a WP do? A WP has six methods that provide information about the object it points to and about the shared pointers that point to an object.

Some of the characteristics of WPs are 1. WPs only work in conjunction with shared pointers 2.WPs do not increase the number of strong references to the object it points to. 3. WPs have no ability to access the object it points to. A WP's main job is to give information about the shared pointers and the object shared pointers point to.

|  |  |
| --- | --- |
| reset | Releases ownership of the object being pointed to (managed object). |
| swap | Swaps a managed object for a different managed object. |
| use\_count | Returns the number of shared\_ptr objects that point to an object. |
| expired | Checks to see if the object being pointed to exists. |
| lock | Creates a shared\_ptr that points to the object. |
| owner\_before | Provides owner based ordering of WPs. |

How does a WP work? We know multiple shared pointers point to the same object. A danger exists if a program tries to access an object pointed to by a shared\_ptr if the object no longer exists in memory. It is sometimes difficult to keep track of numerous shared pointers. A WP works by calling its member functions on shared pointers such as, “expired”. This member function checks to see if the object being pointed to still exists. This can inform the program if it is safe to access the managed object. Additionally, a WP forms a “weak” reference to an object. This can prevent memory leaks by allowing the programmer to prevent strong references cycles by using a WP(creates a weak reference) instead of shared pointer (creates a strong reference).

*Shared\_ptr*

A shared pointer is a special type of smart pointer that keeps track of how many shared pointers are sharing the resource. Internally, it uses two pointers: one pointing to the resource that it owns and the other pointing to a dynamically allocated object that keeps track of how many shared\_ptr are pointing to a single resource. Objects of the shared\_ptr type have the ability to take ownership of a pointer and share that ownership. The reference counter gets incremented whenever a new pointer is pointed to a resource and it gets decremented when either the resource’s destructor is called, the last remaining shared pointer to the resource is reassigned to a new pointer, or when the pointer reaches out of the scope. In the case that the resource gets destroyed, its memory will also be deallocated as well.

Shared pointers are used when you need to use multiple smart pointers that share ownership of a resource. Since shared pointers automatically deallocate the resource that it points to, it is especially useful for instances where the lifetime of a resource and its ownership may vary.

*Garbage Collector vs Reference Counting*

In the early days of programming, programmers faced many issues concerning memory such as slow processing speed and little memory space to work with. When an object is created and a variable is used to refer to it. Upon termination, only the variable is destroyed. This means that only a reference to the object is deleted, but the actual object still lives in memory. Roughly describing, if there happens to be too many of these rogue objects in memory, the computer will be resource starved and will be unable to perform any more instructions. Programmers came up with ways to solve this problem with the most common method being manual memory management (MMM). This allows us to keep track of each object and release them at the right time.

Nowadays, we have automatic memory management (AMM) in modern language. The two most commonly used is Garbage Collection (GC) and Automatic Reference Counting (ARC). While GC is used by languages such as Java, JavaScript, C#, etc., ARC is used by Objective C, Swift, C++ smart pointers, etc.

GC is used at runtime in the background. It detects inactive objects and removes them from memory. An object is considered inactive when it is unreachable, or, simply put, there are no references to the object. ARC on the other hand is deployed automatically as a compile mechanism at compile time. It injects memory management calls such as retain and release on objects as the reference point fluctuate. A reference count indicates when an object can be destroyed. As more variables are referenced to the object, the reference count goes up. As all the variables are being nullified, the reference count goes down. Finally, when the reference count reaches zero, the object’s memory is deallocated.

These two methods both have their strengths and weaknesses:

|  |  |  |
| --- | --- | --- |
|  | **Strengths** | **Weaknesses** |
| ARC | +Local, incremental processes  +reclaims memory immediately | - Unable to handle retained cycles(circular object referencing) |
| GC | + Don’t need to worryabout adding “free” or “delete” statements to code   + Never have bugs related to dangling references/pointers (double frees, etc.) | -Random(indeterminate) deallocation   - Cannot implement own class destructor   - CG algorithms takes a lot of memories to execute compared to ARC |

Performance wise, there is no clear winner. They both have their own advantages and disadvantages.

**Results & Conclusions**

In 2011, the three types of smart pointers were introduced and by the year 2012, many C++ developers understood their importance which were followed rapidly by proceedings to remove the use of raw pointers in open source libraries. Nowadays, many people who learn C++ would not know the existence of raw pointers unless they’ve been working with the language for a while. Smart Pointers should always be preferred over raw pointers to make a code safer and with better readability and modification capabilities.

As seen in the coding examples on github, smart pointers are a way to represent ownership semantics. While they do not cover the entirety of efficient resource management, they do, however, handle the most common problematic cases that arise day to day.

**Discussion**

Several problems our team encountered through this project was the gathering of necessary information to understand the topic at its fullest and overall coordination when using GitHub. As many of us were not experienced with GitHub, we were learning how to fork and merge different pieces of code together, while writing proper documentation so that the viewers could understand. Searching and gathering the sources where some did not provide the information we were looking for was also a persistent issue. Additionally, we ran over a few issues when trying to run Valgrind, a third party memory profiler, in order to demonstrate a memory leak. Our main concern was understanding the output of Valgrind and proving that it was an actual memory leak. We looked online for further Valgrind examples to use that as comparison points.

**References**

* Johnston,Phillip.“EmbeddedArtistry.” *EmbeddedArtistry*, 4 Jan. 2017, https://embeddedartistry.com/blog/2016/9/19/rfr0r76r0ovd0gk574kfsldxfbklgs.
* “Internal Pointers.” *Internal Pointers*, 23 Oct. 2018, <https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c>.
* Guron, Punit. “IBM Community.” *IBM Community*, 17 Nov. 2012, <https://www.ibm.com/developerworks/community/blogs/6e6f6d1b-95c3-46df-8a26-b7efd8ee4b57/entry/detect_memory_leaks_with_memcheck_tool_provided_by_valgrind_part_i8?lang=en>.
* Gupta, Himanshu. “Geeks for Geeks.” *Geeks for Geeks*, <https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/>.
* Boccara, Jonathan. “Fluent C .” *Fluent C*, 22 Aug. 2017, https://www.fluentcpp.com/2017/08/22/smart-developers-use-smart-pointers-smart-pointers-basics/.