TrecPointers: AnaGame’s smart pointers

TrecPointers are smart pointers used by AnaGame to manage memory. The smart pointer utilizes a Table to hold actual references to objects. Smart Pointers in general use a counter and delete the object once the counter reaches 0.

Originally, AnaGame used raw pointers. However, memory concerns (chiefly memory leaks) prompted the use of smart pointers. Initially, the std::shared\_ptr from the Standard C++ Library was used. However, there was the issue that occurs in this piece of code.

int\* ptr\_i = new int(3);

std::shared\_ptr<int> sh\_ptr\_i1 = ptr\_i;

std::shared\_ptr<int> sh\_ptr\_i2 = ptr\_i; // Error

std::shared\_ptr<int> sh\_ptr\_i3 = sh\_ptr\_i1;

The way the creation of the second pointer is an error is that there are now two smart pointers with 2 separate counters referencing the same object. Therefore, if one of those smart pointers is assigned something else (or when the first one goes out of scope), the counter will reach 0 and the object will be deleted – even though there is still another smart pointer referencing it.

(The final line works as the two smart pointers share the same counter as well).

Therefore, AnaGame provides the TrecPointer, a special type of smart pointer where pointing to the same object means using the same counter as well – regardless of how a given TrecPointer was assigned its reference.

int\* ptr\_i = new int(3);

TrecPointer<int> sh\_ptr\_i1 = ptr\_i;

TrecPointer<int> sh\_ptr\_i2 = ptr\_i; // not an Error

TrecPointer<int> sh\_ptr\_i3 = sh\_ptr\_i1;

## Files to look at (TrecLib)

|  |  |
| --- | --- |
| File Name | Purpose |
| TrecPointerI (h/cpp) | Holds the table of references as well as internal functions used to manage the table |
| TrecPointerBase (h/cpp) | Base class for the TrecPointer, which calls the functions in the TrecPointerI files |
| TrecPointer (h) | Holds an index to (and a copy of) the reference in the reference table |
| TrecComPointer (h) | Variant of the TrecPointer intended to work with COM objects (i.e. DirectX objects that you call “Release()” on instead of deleting) |

## Table of References (Stored by TrecLib library)

Rather than having each TrecPointer store their own copy of the reference (although they certainly do), the references are stored in a table hosted by the TrecLib library and is only accessible to TrecPointers through the base class declared and implemented in TrecLib. The table consists of the following:

|  |  |
| --- | --- |
| **Data Type** | **Description** |
| Void\* pointer | The actual reference |
| Unsigned short count | How many references are out there |
| Unsigned char timeCount | A checking mechanism for validity |

* Pointer: The actual reference itself, stored as a void pointer in order to support any type being used without the need for inline references.
* Count: used to track the number of reference stored. Because there is one counter stored in the table, AnaGame’s TrecPointer is able to avoid the multi-counter problem possible in other smart-pointer implementations[[1]](#footnote-1).
* timeCount: Used to check whether or not a TrecPointer is out of date. This feature allows TrecPointers to be able to delete their reference even if other TrecPointers still reference that object. Other TrecPointers that are “dangling” will switch to NULL before being used (do a NULL check)

# TrecPointer Operations

There are standard operations that can be applied to the TrecPointer, each of which have correlating operations done on the Reference Table.

## Add Reference by Pointer

Best Case Running Time: Ω(1), where pointer is already in the table and is amongst the first elements

Worst Case Running Time: O(n), where n is the number of references in the table

1. A TrecPointer is provided a raw pointer, either through it’s constructor or through the overloaded assignment (‘=’) operator.
2. TrecPointer calls its base class method “init(void\*, const char\*)” with the raw pointer as a parameter.
3. The Base Pointer Calls the method “IncrementCount(void\*, const char\*)”
4. “IncrementCount” calls the function “InsertNewReference(void\*, int&)”
5. InsertNewReference scans the table to see if the pointer is already stored. If it is, the index of that pointer, as well as the *timeCount,* are returned.
6. If the pointer is not in the table, then a queue will provide an available index to store it in, or a new slot is added to the end of the table.
7. Once a slot is taken, it is populated with the pointer, a randomly generated *timeCount*, and a counter value of 1.
8. Return to TrecPointer

## Add Reference by Index

Running Time: Θ(1)

1. TrecPointer is provided a TrecPointer, either through the constrictor or the overloaded assignment operator
2. Base Pointer’s “SetTrecPointer(TrecPointerBase&)” is called, which simply sets the index and *timeCount* to that of the copied pointer.
3. Base Pointer’s “IncrementCount(const char\*)” is called
4. “IncrementCount” calls the “InsertNewReference(UINT, UCHAR&)” function.
5. InsertNewReference finds the entry referenced by the index. If there is a pointer to it, the counter is incremented and *true* is returned. Otherwise, *false* is returned.

## Retrieving the Raw Pointer

Running Time: Θ(1)

1. TrecPointer’s “->” operator or get() method is called, either to perform an operation on the object or perform a null check
2. The Base Class’s “Get” method is called.
3. If the index is less than 0, null is returned.
4. The FUNCTION “Get(UINT, UCHAR)” is called
5. The following is returned to the TrecPointer based off of the following conditions:
   1. Index is larger than table 🡪 Return null
   2. The timeCount provided does **not** match with what’s in the table[[2]](#footnote-2) 🡪 return null
   3. The two timeCount’s match 🡪 return the pointer
6. If the TrecPointer is provided a null pointer, report null to the caller. Otherwise, cast it to the designated type

## Dereferencing TrecPointer

Running Time: Θ(1)

Occurs when an active TrecPointer is deleted or is assigned a new reference

1. The Base Pointer’s “DecrementCount()” method is called.
2. The Base Pointer calls the “DeReference(UINT, UCHAR)”
3. DeReference does index checking and time-Checking. If neither if them check out, null is returned.
4. If the checks passed, the counter is decremented. If the counter reaches 0, then the “Delete(UINT, UCHAR)” function is called.
5. Either null is returned automatically or Delete returns a value.
6. If the TrecPointer gets a value returned, it deletes the object it was pointing to (or “Releases” it in the case of TrecComPointers).
7. The TrecPointer is now released from the old reference and can be used for a new reference.

## Deleting a TrecPointer

Running Time: Θ(1)

This occurs either when Dereferencing causes the counter to reach 0, or the TrecPointer’s “Delete()” method is called. Because there are two major pathways for this process to occur.

First Pathway: Steps 1-4 in “Dereferencing TrecPointer”

Second Pathway:

1. TrecPointer’s Delete() method is called
2. Base Pointer’s “NukeCount(const char\*)” is called
3. NukeCount calls the “Delete(UINT, UCHAR)” function

The Delete process

1. The current reference is stored in a local variable.
2. The Entry is marked as having a null reference and the counter is reset to 0.
3. The Index of the Entry is added to a queue to speed up the process of adding a new reference in “Add Reference by Pointer”.
4. The pointer to delete is returned
5. The TrecPointer involved casts the returned pointer (unless it’s null) and deletes it.
6. The TrecPointer is now released from the old reference and can be used for a new reference.

Note: If you’re going to delete an object/variable referenced in the table, DO NOT delete if through a raw pointer. Use the TrecPointer’s Delete() method for that purpose.

# Dos/Don’ts of TrecPointer

## Heap vs. stack

// DON'T

int i = 3;

TrecPointer<int> tc\_ptr\_i(&i); // I is on the stack,

// DO

int \*i = new int(3);

TrecPointer<int> tc\_ptr\_i(i); // I points to the heap

The TrecPointer was built to support automatic deleting whenever the counter reached 0. For objects in the heap (meaning allocated via *new* or some factory method/function), this makes sense as there is no mechanism through which the objects will be deleted on their own.

The stack, on the other hand, variables are popped whenever they go out of scope. If the variable is an object, then that object’s destructor will be invoked automatically. This means that the memory referenced could be used for something new by the time the TrecPointer gets around to *deleting* an object that’s no longer present.

Note: The TrecPointer currently has no mechanism to check for that, or know when an object was deleted because it went out of scope.

## Deleting a pointer

The TrecPointer introduced the ability to manually delete an object owned by the TrecPointer, even if other TrecPointer’s are owning them. This is because the raw pointer and counter is stored in one place and the system has a mechanism to catch dangling TrecPointers – even if the entry they referenced is refilled.

However, there are rules on how to manage deletion – including a *don’t* that applies to any smart pointer, TrecPointer or otherwise.

// DON'T

int \*i = new int(3);

TrecPointer<int> tc\_ptr\_i(i);

delete i; // TrecPointer has no way of knowing this happened

// DO

int \*i = new int(3);

TrecPointer<int> tc\_ptr\_i(i);

i = nullptr;

tc\_ptr\_i.Delete(); // Any other TrecPointer will find out this happened

// Just be sure to nullify any raw pointer that remains

// Okay

int \*i = new int(3);

delete i; // No smart pointer was assigned this value, so no conflict occurs

The reason for using the Delete() method is because dangling TrecPointer’s count on the table to update them to the change. The Delete method updates the table prior to performing deletion, so any dangling TrecPointers will reset to null upon future access. However, performing deletion on a raw pointer will NOT update the table so any dangling TrecPointers will REMAIN dangling. (This is true for any smart pointer that I’m aware of).

## Using TrecPointer’s vs. other smart pointers

If working on AnaGame, use the TrecPointer. However, if you use a different smart pointer, make sure to stick with that smart pointer. There is no mechanism through which the TrecPointer’s counter can sync with that of an external smart-pointer.

// DON'T

int \*i = new int(3);

TrecPointer<int> tc\_ptr\_i(i);

std::shared\_ptr<int> sh\_ptr\_i(i); // Danger! There are now two different smart pointer types,

// each with their own counter.

// DO 1

int \*i = new int(3);

TrecPointer<int> tc\_ptr\_i(i);

TrecPointer<int> tc\_ptr\_i2(i); // Safe! The two TrecPointers share the same counter

The “Do” in the above example leads back to why I created the TrecPointer in the first place. Before the TrecPointer, I was attempting to use the standard shared\_ptr and found out the hard way that the “Do” above was an error for the smart pointer I was using.

// DON'T

int \*i = new int(3);

std::shared\_ptr<int> sh\_ptr\_i(i);

std::shared\_ptr<int> sh\_ptr\_i2(i); // Danger! the shared\_ptr creates it's own counter

// Okay

int \*i = new int(3);

std::shared\_ptr<int> sh\_ptr\_i(i);

std::shared\_ptr<int> sh\_ptr\_i2(sh\_ptr\_i); // Okay! the second pointer uses the counter of the first

// Or Use TrecPointers (This example is why I created the TrecPointer in the first place)

Rather than conform to the rule imposed (AnaGame was still large at the time and making sure this rule was followed would have been a nightmare), I opted to design the TrecPointer instead. While it was possible to minimize violations of the rule, I preferred to reduce the consequence of a memory error to a bottleneck of O(n).

# To-Do’s:

Yes, the TrecPointer is far from perfect and there are potential changes that could be made. For the moment, I’m happy with the TrecPointer in its current form. However, there are still issues present that I’d like resolved before AnaGame can be declared version 1.0.

1. Better handling of the stack
   1. TrecPointer has no way of knowing if pointer points to the stack or not
   2. Should AVOID inline assembly as AnaGame should be made available in 64-bit
   3. Probably best mechanism is for TrecPointer to hold a heap-copy of a stack object/variable.
2. Safe Removal of the TrecPointer’s local pointer copy
   1. Currently, the pointer is held by the table, but it’s also held by any TrecPointer pointing to it
   2. This creates more memory overhead than what should be necessary
   3. Although they don’t appear to be needed once present, removing them causes issues to arise.
   4. Bug is probably in the “TrecPointerI.cpp” file. “InsertNewReference(UINT index, UCHAR& tc)” is a likely culprit
3. Usability improvements
   1. When doing a null check on a shared\_ptr, simply putting it in an if-statement is enough
   2. For the TrecPointer, you have to call the “get()” method in the if-statement
4. Efficient type management
   1. There could come a time when a TrecPointer is accessed after being dangling for an extended period of time.
   2. While extremely unlikely, it is NOT impossible for a table entry to be updated 256 times that it’s timeCount matches that of a dangling TrecPointer.
   3. If the TrecPointer system can catch a type mismatch in this scenario, it can add another layer of defense
   4. Should probably wait until AnaGame’s own type management system is built up
5. Memory security around the Table should be added
   1. If malware ever accesses the TrecPointer table, either knowingly or randomly, it could compromise an AnaGame process
   2. Need to find some way so that only code from the TrecLib DLL can access it.
   3. Currently the focus is on Windows, though given I hope to one day port AnaGame to Linux/OS X/etc. feel free to offer a mechanism for those platforms as well

1. Provided you don’t ever store the same reference in a different smart-pointer type. The TrecPointer counter will not include the other reference – nor will the other smart-pointer be aware of the TrecPointer’s. [↑](#footnote-ref-1)
2. This most likely occurred because a TrecPointer’s Delete method was called. This sets the entry to null and the entry was later used by a new reference (which increments the timeCount) [↑](#footnote-ref-2)