

[Appendix A explains the foundational knowledge required if the reader is not familiar with software development.](#)

[Appendix B explains in detail what problem the macros attempt to solve.](#)

[Glossary](#)

[1. Introduction](#)

[Overview of Remote Type Reflection and Auto Address](#)

[Brief Purpose of the MemoryManager Class](#)

[2. Prerequisites](#)

[3. Understanding the Macros](#)

[Macro: READ_CLASS_FROM_REMOTE](#)

[Macro: READ_CLASS_TRemoteObject](#)

[Creates a TRemoteObject structure to store both the remote base address and a local copy of the data. elementPtr came from a nested type array. Our TRemoteObject is localPlayer and is able to be checked for null on a field from SDK::APlayerState.](#)

[Macro: READ_CLASS_FROM_FIELD](#)

[Macros: GET_PTR_FIELD and GET_VAL_FIELD](#)

[4. Using the MemoryManager Class](#)

[Initializing the MemoryManager](#)

[5. Type Reflection](#)

[Decltype\(\):](#)

[Offsetof\(\):](#)

[6. Handling TArray or Similar Structures](#)

[Appendix A](#)

[C++ & C Programming Languages](#)

[Appendix B](#)

[Memory & Pointers](#)

[C++ contiguous structured memory](#)

[GLOSSARY](#)

Appendix A explains the foundational knowledge required if the reader is not familiar with software development.

[GO TO APPENDIX A](#)

Appendix B explains in detail what problem the macros attempt to solve.

[GO TO APPENDIX B](#)

Glossary

[GO TO GLOSSARY](#)

1. Introduction

Overview of Remote Type Reflection and Auto Address

The purpose of the macros is to replace long and tedious code that is usually outdated with every change in the software development kit (SDK) with shorter code that will never have to be modified as long as the field names or types of the SDK do not change. Everytime a game updates and a new SDK is dumped, the memory spacing of types and fields change. Type reflection is when you can implicitly type a local variable at compile time or runtime. Remote in this context means a location in memory outside the current process's memory. Type means the type of data represented in memory; however it is required that our process has an exact replication of the types in the remote process for these macros to work. This lets the macros access memory of another process with type information on the data the macros retrieve. This sequence of actions is commonly called Type Reflection in programming communities because we mirror the type, aka we reflect the type. In order for our process to have exact replication of the types, we need an automatically generated software development kit(SDK) often referred to as a dump. SDK is a set of tools or code used for a specific project often providing a standardized method of data storage, organization, and functionality. A public project called [Dumper7](#) is available to generate SDK's for games made with Unreal Engine. My macros were created with the intention of using a SDK created by dumper7 but should be usable on any similar dump formats with little changes.

Brief Purpose of the MemoryManager Class

The **MemoryManager** class wraps system API calls to provide a structured interface for memory manipulation. It allows quick prototyping in code with the least amount of arguments per function. This of course can be written anyway you want but for my examples I have my own methodology which will include only the code required for type reflection.

2. Prerequisites

To understand and use the macros effectively, you should:

- Be comfortable with C++ syntax and programming concepts
- Understand how pointers and memory addresses work
- Have a Windows environment with administrator privileges
- Have a valid SDK generated for your specific target process

3. Understanding the Macros

The following macros simplify reading process memory by combining pointer arithmetic and function calls.

DISCLAIMER!!! Nested types are not fully supported for data reading, only pointer tracing.

(see next page)

Macro: **READ_CLASS_FROM_REMOTE**

Reads an object of type **SDKType** from a remote process. Where **SDKType** is literally the class type from the dumped SDK you have integrated into your project.

```
#define READ_CLASS_FROM_REMOTE(SDKType, remoteAddr, localVar) \
    struct __##localVar##_Wrapper { \
        uintptr_t remoteBase; \
        SDKType localCopy; \
    }; \
    __##localVar##_Wrapper localVar; \
    do { \
        localVar.remoteBase = (remoteAddr); \
        localVar.localCopy = memory.ReadMemory<SDKType>((remoteAddr)); \
    } while(0)
```

As an example of use:

```
GWorld = memory.ReadMemory<uintptr_t>(memory.baseAddress + GWorldOffset);
if (GWorld)
{
    std::cout << "\n\nWorld: " << std::hex << GWorld << "\n";

    READ_CLASS_FROM_REMOTE(SDK::UWorld, GWorld, klass);
}
```

Of course we need an address to start from and in the case of all Unreal Engine targets you must start with **GameWorld** which represents the game's current session effectively. It's the type **SDK::UWorld**. This type is a fully functional class with fields and functions although since we are remote, all we can use is the fields. Given the **GWorld** address is valid, we know there will be an instance of **SDK::UWorld** at that address and we want to read the entire class in one block. You may notice **klass** isn't declared but also isn't throwing a syntax error, that's because the macro creates a new object for use directly in scope of the macro. This object will have the type **SDK::UWorld** and be able to access all fields of the class. Any structs within this class will also be read and stored locally allowing for a more native access technique.

Macro: **READ_CLASS_TRemoteObject**

This macro is for type conversions and keeping the created structure in the caller's scope. An issue that may happen is a type is retrieved by another macro but you still need to read that object's address as its own type but the compiler will complain you can't convert a class to a raw address unless it was a pointer, we can't use pointers though for remote processes. This macro was an early approach to solve this issue. (see next page)

This struct is globally declared on your program so your runner threads can use it.

```
template<typename T>
struct TRemoteObject
{
    uintptr_t    remoteBase; // address in the remote process
    T           localCopy;  // your locally read copy of the remote data
};
```

Creates a `TRemoteObject` structure to store both the remote base address and a local copy of the data. `elementPtr` came from a nested type array. Our `TRemoteObject` is `localPlayer` and is able to be checked for null on a field from `SDK::APlayerState`.

```
// If valid, read that whole APlayerState object and do something with it:
if (elementPtr)
{
    READ_CLASS_TRemoteObject(SDK::APlayerState, elementPtr, localPlayer);
    if (localPlayer.localCopy.PawnPrivate)
    {
        READ_CLASS_FROM_FIELD(localPlayer, PawnPrivate, SDK::APawn, privPawn);
        if (privPawn.remoteBase)
        {

```

Macro: `READ_CLASS_FROM_FIELD`

Reads a nested object inside another structure. So this is a direct result of attempting to fix the nested type reflection issue discussed on the previous macro. I was able to derive the proper addresses from a nested type's field only using this "READ_CLASS_FROM_FIELD" macro. Attempting to retrieve a field ptr doesn't work on nested type classes. You also can't just retrieve the value because there isn't one and we are wanting a class read anyways.

`READ_CLASS_FROM_REMOTE` also does not work on nested type classes due to the scope requirement of being top down instead of only within the direct scope of the macro itself. Just in case we read null addresses, we have to test them and then use them if not null which is the reason for the scope top down.

```
#define READ_CLASS_FROM_FIELD(parentVar, fieldName, TargetSDKType, outVar) \
    TRemoteObject<TargetSDKType> outVar; \
    do { \
        uintptr_t __remotePtr = reinterpret_cast<uintptr_t>( \
            (parentVar).localCopy.fieldName \
        ); \
        outVar.remoteBase = __remotePtr; \
        outVar.localCopy = memory.ReadMemory<TargetSDKType>(__remotePtr); \
    } while(0)
```

(see next page)

Using the macro produces an object you can use at any point further down in the code due to the **TRemoteObject** structure. Notice on the Vector3 reading for location we are using hardcoded offsets? This is a limitation of the current version macros, I didn't have enough time before this paper was due to refine this nested type issue so although we can bounce around class pointers, we can't actually retrieve data with type reflection. [see disclaimer](#)

```
READ_CLASS_TRemoteObject(SDK::APlayerState, elementPtr, localPlayer);
if (localPlayer.localCopy.PawnPrivate)
{
    READ_CLASS_FROM_FIELD(localPlayer, PawnPrivate, SDK::APawn, privPawn);
    if (privPawn.remoteBase)
    {
        READ_CLASS_FROM_FIELD(privPawn, RootComponent, SDK::USceneComponent, rootcomp);
        if (rootcomp.remoteBase)
        {
            //0x128 RelativeLocation , we can't use macros. We have to hard code this part...
            Vector2 screen{};
            Vector3 enemyLocation = memory.ReadMemory<Vector3>(rootcomp.remoteBase + 0x128);
        }
    }
}
```

Macros: **GET_PTR_FIELD** and **GET_VAL_FIELD**

GET_PTR_FIELD is the same as if you manually wanted to take address A and add offset B for a result. The macro just allows the coder to use field names of SDK types, although it does allow a reflective object of the class to work. Typically calling READ_CLASS_FROM_REMOTE before is required.

Specifically GET_VAL_FIELD assumes we've already read the data from remote to local copy. This is important to know that VAL data can only be captured on non-nested typed objects and does not include nested types for the data or pointers. Just primitive types. Technically it's possible that simple custom structures can be captured when reading a class but I've seen it get mangled sometimes if any constructors were present on the remote structure. Do not blindly trust this macro to work on all structures, use with caution. [see disclaimer](#)

```
#define GET_PTR_FIELD(localVar, fieldName) \
    memory.ReadMemory<uintptr_t>( \
        localVar.remoteBase + offsetof(decltype(localVar.localCopy), fieldName) \
    )

#define GET_VAL_FIELD(localVar, fieldName) \
    ( localVar.localCopy.fieldName )
```

4. Using the **MemoryManager** Class

Initializing the MemoryManager

```
MemoryManager memory(TARGET_PROCESS_NAME);
```

The constructor is made to be as painless as possible for the coder. You just supply the process name you wanted to target. Everything else is handled in the background. Call it in your main function somewhere before doing anything else. Code for this class and the macros can be found here: <https://mega.nz/folder/Q9dDDBhC#1biAFR06JCODDiJepkCceQ>

5. Type Reflection

The macros are based on `decltype()` and `offsetof()` macros which already exist in c++.

Decltype():

A keyword from C++ 11 that types an expression at compile time without evaluating the expression. It can type reference as well if provided with double parenthesis around its intended member. Example:

```
int a = 5;
```

```
decltype((a)) c = a;
```

a is treated as int but has no type explicitly. c is the int reference of a.

This can be useful if you need to modify a by reference and then store it somewhere.

Decltype can use reference and const variables but auto cannot.

Offsetof():

A macro in the header `<cstddef>` that resolves the byte offset of a member within a structure or class. It calculates many bytes from beginning to the wanted member. If a member is a static data member, a bit-field, or a member function, the behavior is undefined. If the type is not PODType (plain old data type) then behavior is undefined also.

6. Handling TArray or Similar Structures

A unique approach to handling TArray or other custom structures to pointer arrays with extra metadata included is required when reading remote. The initial address on the field is actually the data pointer address within the structure. Below I have a rough example of how TArray is handled remotely. (see next page for step by step walkthrough of code image)

```
template<typename T>
```

```
struct TArray
```

```
{
```

```
    T* Data;
```

```
    int32 NumElements;
```

```
    int32 MaxElements;
```

```
}; //should be noted this is NOT how it appears in SDK but for raw addresses it's enough.
```

The object `localTArray` is a nested type of `TArray<APlayerState*>` which is impossible to directly process remotely because we can't dereference pointers remotely. All we can do is read what it points to or where the pointer is without type data or inheritance. As in, we only can use raw addresses. To deal with this we can read the `TArray`'s location in local copy which represents the remote location as a raw address. This is actually the address that represents the data pointer I spoke of earlier. The index count is available as a function call because the structure of `TArray` was picked up on the `GET_VAL_FIELD` luckily so we have the metadata already stored in a private member which is why I couldn't directly read it earlier. `remoteDataPtr` is the `Data*` and as such, will be iterated on until `i` equals index count metadata. `elementPtr` is the actual address pointed to by the data pointer in each index. This represents the `SDK::APlayerState` type.

```
READ_CLASS_FROM_REMOTE(SDK::AGameStateBase, gameState, klass_gamestatebase);
SDK::TArray<SDK::APlayerState*> localTArray = GET_VAL_FIELD(klass_gamestatebase, PlayerArray);
uintptr_t remoteDataPtr = *reinterpret_cast<uintptr_t*>(&localTArray);
std::vector<RenderData> newData;
for (int i = 0; i < localTArray.Num(); i++)
{
    // Read the i-th pointer (APlayerState*) from the remote array:
    uintptr_t elementPtr = memory.ReadMemory<uintptr_t>(
        remoteDataPtr + i * sizeof(uintptr_t)
    );

    // If valid, read that whole APlayerState object and do something with it:
    if (elementPtr)
    {
        READ_CLASS_TRemoteObject(SDK::APlayerState, elementPtr, localPlayer);
        if (localPlayer.localCopy.PawnPrivate)
        {
            READ_CLASS_FROM_FIELD(localPlayer, PawnPrivate, SDK::APawn, privPawn);
            if (privPawn.remoteBase)
```


Appendix A

C++ & C Programming Languages

A programming language is a structured text of instructions that allow a computer to perform whatever task is described in the instructions. The different universally agreed on standards for the structured text are referred to as languages. In the case of this paper, I am talking about C++ and C ([ISO c++](#)). ISO is an organization that revises C++ every five years to ensure international cooperation and standards. C++ was derived from C and most of the structured expressions are the same although it has more advanced features included as a basic functionality on the language itself. C++ / C language has a set of predetermined code that come with it referred to as header files or headers for short. These headers often contain mathematical formulas written in the language, text printing or formatting, and much more. C++ / C both come jam packed with functional pre-coded headers the programmer can just call upon in their own programs saving time. I code [C ANSI 1989 \(C89\)](#) combined with a mixture of C++ standards depending on my requirements.

C++ and C are compiled languages, meaning that to make the code into a usable program the computer can read, it translates the human readable text to binary. Binary consists of two numbers, one or zero. One represents on state or true state while zero represents off state or false. Computers can actually only understand binary, all programming languages will be translated into binary at some point before the computer can use it. There are many different kinds of compilers for C++ / C but I use Microsoft's Visual Studio Compiler for C++ / C. (MSVC) The reader doesn't have to understand what the differences between compilers actually are, just that they translate the same code differently sometimes which can cause undefined behavior.

C++ is a statically typed language which means it has predetermined types of data that are always checked for valid use cases before compilation begins. This greatly speeds up the coding process by allowing the coder to see error messages of which specific type is incorrect. Type checking is a very important part of C++ which is also the reason my macro struggles with a type nested into a type. This is covered in Appendix C.

Syntax refers to the specific structure the language follows.

For a more in depth and complete lesson on program structure of C++ 98 (first version approved by ISO) https://cplusplus.com/doc/oldtutorial/program_structure/

Currently C++23 is under review as of writing this document and the macros were written in a preview version of C++23 although it may be unnecessary depending on the compiler the reader uses.

Appendix B

The purpose of the macros is to replace long and tedious code that is usually outdated with every change in the SDK with shorter code that will never have to be modified as long as the field names or types of the SDK do not change. Everytime a game updates and a new SDK is dumped, the memory spacing of types and fields change. Sometimes only by a few bytes, others by hundreds or thousands of bytes. Traditional code would be using hard coded offsets that would need manual verification when using a new dump. These macros remove that need for manually updating offsets allowing for faster future development. The rest of appendix B is a detailed look at how the macros work.

Memory & Pointers

This section assumes you have read the “C++ & C” section above combined with the links to the language references to result in some basic understanding of coding.

Windows is based on 64 bit operator systems that use 8 byte segments to represent addresses in memory. 1 byte is 8 bits, this is where 64 bits comes from in the name. 8 byte data types commonly used when reading or writing memory addresses in C++ are:

The following are standard C++ types for 8 bytes. The leading “u” means unsigned versus types without “u” prefixing it are signed. Signed means the values are negative or positive. Unsigned means positive only but includes zero.

- `uintptr_t`
- `uint64_t`
- `int64_t`
- `long long`
- `unsigned long long`

The following are specific to Windows only and made by Microsoft to help standardize types on Windows operating systems. They are no different in actual size or meaning to the regular c++ types above except perhaps they have access to special functions.

- `LONG64`
- `ULONG64`
- `LONGLONG`
- `ULONGLONG`
- `QWORD`
- `DDWORD`
- `DWORD64`

The reader MUST understand basic variable assignments, pointers, classes, constructors, and functions to understand anything going forward.

The virtual memory used by every program is just a range of 8 byte addresses on 64 bit windows that are assigned to programs on an as needed basis. There's a complex set of operations that allow computers to create “virtual” address ranges for each program. Virtual

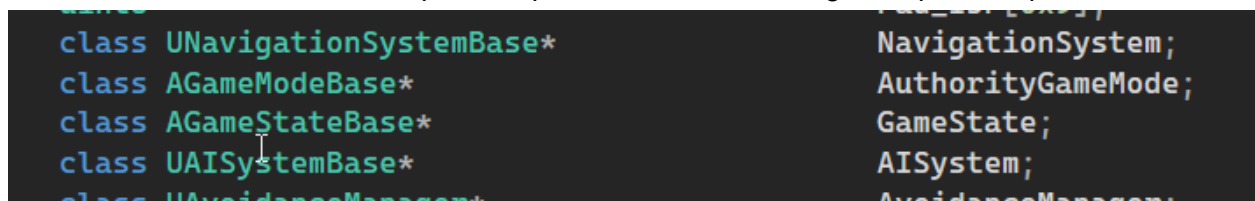
addresses are specific to each program and temporary. If the program closes, the range is free from use and will cease to contain data or exist in this context. This means ranges are reusable but not from the same module base address. A module base address is the location in physical memory containing the instance of the running program in question. Module often refers to DLL files but can also mean binary files.

In the header “Memory.h” linked on this document, the function `GetModuleBaseAddress()` will return the address in physical memory of the module specified by process ID. This is how the macro knows what base address to use because the result is saved in a variable specific to the object of the `MemoryManager` class during the constructor.

C does not have classes but it does have pointers, structures, custom types. C++ has everything C has plus classes and abstraction techniques. Classes are the primary type given to the objects created by the macros.

When dealing with memory it's common to read about using pointers to represent where in memory something is. All of the methods found in textbooks and higher education will fail to explain that pointers in another process are not relevant to your current process in any way. The coder cannot directly dereference a pointer of another process without some type reflection trickery. If our program makes a 1:1 copy of the other program's memory structures then assigns types to the data as we read it into buffers, our local program can use types as if in the other process. It is important to note that any writing to this data cannot be performed on this copied data because again, we are on a local cached version. Writing would require re-reading the location and writing into the other process.

To finalize the statements above on pointer dereferencing let's see an example. In this image the pointer to class “`AGameStateBase`” is seen declared as “`GameState`”. If we wanted to dereference the pointer to reveal the underlying class fields our external program needs to read the pointer address, then do another memory read to find what the pointer points to and start manually reading offsets from that address. This is extremely tedious and prone to errors due to memory padding being different depending on compilers and other variance. The macros allow the code to interact with remote process pointers similar to being local process pointers.



```
class UNavigationSystemBase*      NavigationSystem;
class AGameModeBase*             AuthorityGameMode;
class AGameStateBase*            GameState;
class UAISystemBase*             AISystem;
class UAvoidanceManager*         AvoidanceManager;
```

First the macro to read a class pointer's fields is called upon class `UWorld` which is where the field `GameState` resides. The macro creates the object “`klass`” which is typed “`SDK::UWorld`” and contains a cached image of the remote process memory containing the class.

`READ_CLASS_FROM_REMOTE(SDK::UWorld, GWorld, klass);`

Next the macro to retrieve pointers themselves from a cached class copy is used to set the variable `gameState` to what the pointer points to which is another 8 byte address.

```
auto gameState = GET_PTR_FIELD(klass, GameState);
```

Now unfortunately if you remember, the data we have is cached locally so if we read a pointer in our cache that is from the remote process, we can't read any real data or may get an error. As discussed earlier, remote pointers are not valid in local memory. To solve this, the macro for reading an entire class can again be called on the pointer. The macro will read from the remote process at this pointer's address to retrieve the entire class. It types the object called "klass_gamestatebase" as `SDK::AGameStateBase` as well.

```
READ_CLASS_FROM_REMOTE(SDK::AGameStateBase, gameState,  
klass_gamestatebase);
```

The final step will be retrieving a value from the local cache. See next section why `localTArray` gets populated with the pointer to the array, not the pointer to the first element.

```
SDK::TArray<SDK::APlayerState*> localTArray = GET_VAL_FIELD(klass_gamestatebase,  
PlayerArray);
```

C++ contiguous structured memory

Before explaining why it works, let's check out what structure is being read.

<https://dev.epicgames.com/documentation/en-us/unreal-engine/array-containers-in-unreal-engine>

`TArray` is a structure in Unreal Engine that can hold any type and serves as a robust array struct.

Here is a very simplistic outline of how remote process reading will see the `TArray` in memory.

```
struct TArray  
{  
    T* Data;  
    int32 count;  
    int32 max;  
};
```

This is literally just a struct that holds an array of pointers. What the pointers point to doesn't matter so much. Arrays and structs are stored contiguously in memory in the order they are coded. The following video is time-stamped to a part where "ThePrimeTime" youtuber explains the memory benefits of arrays vs lists. His example is javascript but the information about arrays is true for C++ as well. The same information about storing arrays in memory works for structures as well, because you can think of a structure as a box you put data inside. The data will not be stored outside that box.

<https://youtu.be/cvZArAipOjo?t=734>

This clearly shows why the local cached copy of memory can contain a full struct. Besides structs this appendix demonstrated that fields and classes may be read as well. By using the type names from the SDK directly on the code using the macros, we have created code immune to updates in the SDK unless an entire field or type name is changed locations or has been removed within the SDK.

GLOSSARY

1. **Remote** - Not within our own process. Something in another process. Not local to our context.
2. **Type** - How data is represented. Typically a type will be something built into the language such as an integer, float, double, char. Sometimes it can be custom made by a coder, in the case of the macros classes are used to type objects.
3. **Type Reflection** - Data is a type on the remote process memory block we are reading, the same exact type is defined 1:1 with our dump for the remote process. Our dump's type is used when reading remote memory to produce a local cache of the remote type. It only works if your dump is truly 1:1 with the remote process.
4. **Auto Address** - The automatic calculation of offsets from a specific address.
5. **Debugging** - To probe, analyze, and test code when attempting to fix unwanted behavior or broken code.
6. **SDK** - Software Development Kit. In the context of this paper it's mostly a dumped set of c++ headers containing runtime type information and the type's fields and functions.
7. **Compile time** - Compared to run time, this means when the compiler is translating the code into binary.
8. **Macro** - code that is expanded into other code. The definition of the macro is used by the preprocessor to substitute the macro name when found in code. It is effectively code generation at compile time.
9. **Value** - Meaning data. Something being represented in a field typically.
10. **Arbitrary Addresses** - Meaning the coder can pick any address they want within the scope of their project. It's not random, the address is within the remote process's memory.
11. **Class** - A user-defined type stored in memory similar to a structure. Used in object-oriented programming.

Karl-Bridge-Microsoft. (2022, May 14). *ReadProcessMemory function (memoryapi.h) - Win32 apps*. Microsoft Learn.

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory>

Karl-Bridge-Microsoft. (2019, August 23). *Tool Help Functions - Win32 apps*. Microsoft Learn.

<https://learn.microsoft.com/en-us/windows/win32/toolhelp/tool-help-functions>

offsetof - cppreference.com. (n.d.). <https://en.cppreference.com/w/cpp/types/offsetof>

C++ named requirements: PODType (deprecated in C++20) - cppreference.com. (n.d.).

https://en.cppreference.com/w/cpp/named_req/PODType

ThePrimeTime. (2023, June 29). *Why you should AVOID linked lists* [Video]. YouTube.

<https://www.youtube.com/watch?v=cvZArAipOjo>

Programming with C++ in Unreal Engine. (n.d.). dev.epicgames.com. Retrieved

February 7, 2025, from

<https://dev.epicgames.com/documentation/en-us/unreal-engine/programming-with-cplusplus-in-unreal-engine>

Complete documentation of working with Unreal Engine using C++.

Contains the article about TArray.

Wikipedia contributors. (2025, January 19). *Virtual memory*. Wikipedia.

https://en.wikipedia.org/wiki/Virtual_memory

GeeksforGeeks. (2025, January 15). *Virtual memory in operating system*.

GeeksforGeeks.

<https://www.geeksforgeeks.org/virtual-memory-in-operating-system/>

W3Schools.com. (n.d.). https://www.w3schools.com/cpp/cpp_classes.asp