C++ memory model

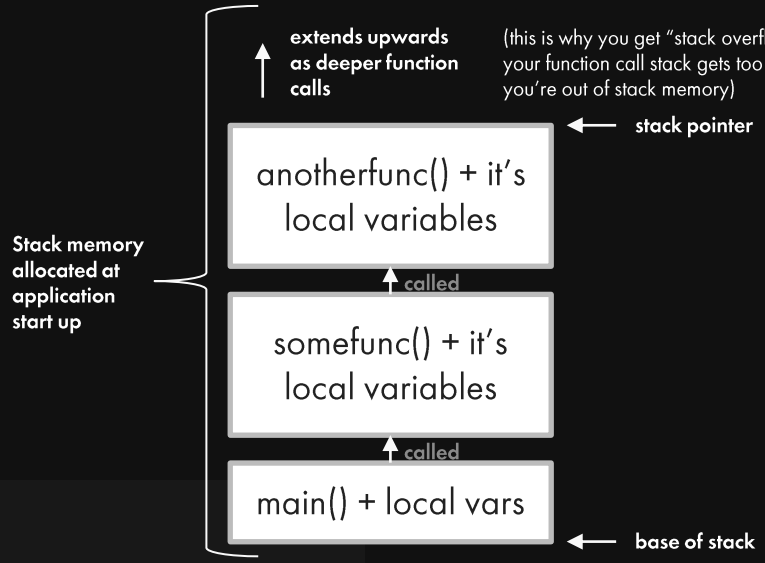
Overview

- Quick refresher on stack vs heap and C# memory models
- How does C++ differ?
- Everything is a block of memory – think of it that way
- Memory allocation at the assembly level
- How things get copied when passed into functions
- Copy & move constructors, copy and move assignment operators

Stack vs heap refresher

Stack

- A.k.a. call stack, memory for local vars
- Stack size of func. known at compile time



Heap

- Dynamically allocated at runtime
- malloc(), new operator, etc.
- Can dynamically allocate memory of any size, as long as the OS has enough memory available

C# refresher

Memory model is defined by the **type**

Classes

- Allocated to the heap (dynamic memory)
- Never on the stack

- Always nullable at runtime*
- Always passed by reference to functions

Structs and value types

- Allocated on the stack when local variables or parameters
- Can be part of the heap if inside a class object that is allocated on the heap
- Never nullable
- Always passed by value (copied)

C++ memory model

Memory model is defined by the **usage**

- Only difference between classes and structs is structs have public members by default
- Otherwise exactly the same

- Usage defines where memory is allocated, whether things are passed by value or ref...

C++ memory model

- Size of classes is the size of all their members added up
- Note that we use the same type in two different ways and this is totally fine, because usage defines behaviour not the type itself.

```cpp
#include <string>

// The size of MyClass is 32 bytes - the same size
// as std::string because that is the only field.
class MyClass
{
public:
    std::string MyStr;
};

int main()
{
    // When main() is called by the OS, it will reserve
    // enough space on the stack to fit the entirety of
    // an FMyClass.
    MyClass StackAllocated;

    // The OS will also reserve enough space for a
    // pointer to MyClass; 8 bytes on a 64-bit machine.
    MyClass* HeapAllocated = new MyClass();

    // Note that we use the same type (MyClass) in two
    // different ways.

    delete HeapAllocated; // No garbage collection in C++ :)
    return 0;
}
```

C++ memory model

- Just wanted to note this is also valid.
- Pointers and references can point anywhere – please don't associate them only with heap allocated memory.
- In fact we will use references to ensure we don't do unnecessary copies on function calls!

```cpp
int main()
{
    // When main() is called by the OS, it will reserve
    // enough space on the stack to fit the entirety of
    // an FMyClass.
    MyClass StackAllocated;

    // This is also valid, because pointers can point anywhere.
    MyClass* PointerToStack = &StackAllocated;

    // Same thing except as a reference - note that we don't need
    // to take the address on the right-hand side.
    MyClass& ReferenceToStack = StackAllocated;

    // Nothing to delete now because we didn't use new().
    return 0;
}
```

Everything is just blocks of memory

Everything is just blocks of memory

- Objects you allocate on the heap are just blocks of memory
- Local variables are just blocks of memory (on the call stack)

- The sooner you think of everything as just blocks of memory, the easiest C++ will be

- Avoid thinking about “objects” in the OOP sense, this does not help you in C++

Godbolt example showing how memory is allocated

https://godbolt.org/z/a8cTKjard
Initially not allocating FTestClass anywhere, so we don't see allocations for it

STEP 2: Stack allocation - Comment out FTestClass variable declaration:

- Default constructors/destructors no longer removed by linker – these call constructors/destructors for all member fields
- Constructor and destructor gets called against the stack pointer (see "48" and "rsp"/"rbp"). This is how we're allocating FTestClass on the stack – main moves the stack pointer up and then runs the constructor against the *48 bytes of memory* it reserved.

Note: FTestClass is actually 36 bytes in size, but the compiler reserves extra space for debugging/metadata.

Godbolt example showing how memory is allocated

STEP 3: Heap allocation - Comment out FTestClass heap allocation

- Again, default constructors now present
- TestClassOnHeap calls “operator new(uint64)” with “40” as the argument to allocate *40 bytes of memory* on the heap. Pointer to allocated memory is stored in “rax” register.
- Constructor gets called against the new block of memory.

- *Note: No destructor call because we didn’t call delete!* There is no garbage collection here, so we are responsible for calling delete when we’re done with the memory.

Godbolt example showing how memory is allocated

- STEP 4: Non-default constructor

  Currently we see “xmm0” references – this is where the compiler is zeroing out the memory for the class when it was allocated because it has a *default* constructor.

  If we create an empty parameter-less constructor, this replaces the default constructor and does nothing.

  Empty parameter-less constructor and default constructors are different. If you need to declare a default constructor, it looks like this instead:

  ```
  FTestClass() = default;
  ```

Godbolt example showing how memory is allocated

- STEP 5: Delete memory from heap

  This now calls the destructor. If your class allocated it's own heap memory, you'd want to free that memory in the class's destructor.

  After that the compiler calls "operator delete(void*, uint64)" which tells it to release the memory at the given pointer (with the given length in bytes).

Additional learning: to learn how the stack pointer and registers change as we make calls in C++, check out the "Expert level debugging" talk at GCAP 2022 when it goes online.

More Godbolt examples

https://godbolt.org/z/1rh69Pb5s

STEP 1: Let's find out the address of things

- Note how M and FirstInt are the same address!
- Again: there is nothing more to classes than just blocks of memory, and since FirstInt is the first thing inside the block of memory allocated to M it's the same address.
- SecondInt is 4 bytes after FirstInt because integers are 32-bits (4 bytes) in size.
- Can you guess what the memory address of ThirdInt is? ☺

More Godbolt examples

STEP 2: Pointer math time

- Can we use pointer math to read SecondInt without actually referencing it?
- Again: there is nothing more to classes than just blocks of memory, and since FirstInt is the first thing inside the block of memory allocated to M it's the same address.
- SecondInt is 4 bytes after FirstInt because integers are 32-bits (4 bytes) in size.

Note: We can still access SecondInt via pointer math even if we make it private in the header. This is because the computer doesn't care at runtime – it's still your application's memory.

You should avoid doing this obviously, but sometimes I've used it in the past to get at engine internals 😏

How does copying work?

If usage defines whether something is stack/heap and by-value/by-reference, does this **mean that we can copy anything?**

By default, yes. The compiler will create a default copy constructor for every* class and struct.

https://godbolt.org/z/56bj5vYfc

Every* class and struct

Ok not every class and struct.

Some types are classified as a “trivial type”.

These are types that you can copy just by copying the whole memory block. i.e. none of the types within the class are non-trivial (require calling a copy constructor).

Copying the memory block is much faster than calling copy constructors all the way down, so the compiler will choose to do this if it can.

Means you won’t see a copy constructor called in the assembly for trivial types.

Every* class and struct

Trivial. All integers and pointers are trivial so FTrivialClass is as well.
https://godbolt.org/z/zxWWP8Msf

```
class FTrivialClass {
public:
    void* P;
    int I;
    long L;
};
```

Non-trivial. Std::string is non-trivial (has a non-default copy constructor), so FNonTrivialClass is non-trivial as well.

```
class FNonTrivialClass {
public:
    std::string S;
    int I;
};
```

Default copy constructor is often incorrect

Default copy constructor does not understand raw pointers.

Will not copy any heap memory that you'd allocated with "new" – it will just copy the pointer.

That makes it hard to correctly know when to "delete".

https://godbolt.org/z/evvoEjPhP

Tip: Always explicitly set constructors

Define or delete all the constructors and destructors in your type:

https://godbolt.org/z/a7va8xKb5

If you have a raw pointer in your type **you must define or delete copy and move constructors**.

Unreal has a macro for deleting copy and move:

```
UE_NONCOPYABLE(FTestClass);
```

```
class FTestClass
{
public:
    // explicitly use default constructor/destructor.
    FTestClass() = default;
    ~FTestClass() = default;

    // prevent copies or moves by deleting
    // the default copy constructor and
    // default move constructor.
    FTestClass(const FTestClass&) = delete;
    FTestClass(FTestClass&&) = delete;
};

int main()
{
    FTestClass A;

    // now the compiler won't let us do this.
    FTestClass B = A;

    // it also won't let us do this (copy into
    // an existing object).
    B = A;

    return 0;
}
```

REDPOINT

What is the move constructor?

Copy constructor = copy the contents of target object to a new one

Move constructor = move the contents of target object to a new one, and prevent target from freeing resources (which we just “stole”)

If move constructor is not deleted, you’ll also have these:

Copy assignment operator = copy the contents of target object into this one

Move assignment operator = move the contents of target object into this one, and prevent target from freeing resources (which we just “stole”)

User-defined copy and move semantics

If you set a user-defined move constructor or don't delete it, you must also define at least a copy assignment operator (and sometimes move assignment):

https://godbolt.org/z/fxdE4Tj74

std::move (or MoveTemp in Unreal) tells the compiler to use move semantics instead of copy semantics, which can reduce allocations.

```
// implement copies and moves.
FTestClass(const FTestClass& Other)
{
    // use it to initialize A.

    // dereference operator * gets us the actual
    // value of the int instead of the pointer.
    this->A = new int(*Other.A);
}
FTestClass(FTestClass&& Other)
{
    // steal the other's pointer into us.
    this->A = Other.A;
    // set the other's A to nullptr so it won't
    // free the int when it's destructor runs
    Other.A = nullptr;
    // after this only we have the pointer.
}
```

```
FTestClass& operator=(const FTestClass& Other)
{
    if (this->A != nullptr)
    {
        // make sure we free up memory if we no
        // longer need it.
        delete this->A;
    }
    this->A = new int(*Other.A);
    return *this; // returning this is expected behaviou
}
```

```
FTestClass& operator=(FTestClass&& Other)
{
    if (this->A != nullptr)
    {
        // make sure we free up memory if we no
        // longer need it.
        delete this->A;
    }
    this->A = Other.A;
    // set the other's A to nullptr so it won't
    // free the int when it's destructor runs
    Other.A = nullptr;
    return *this;
}
```

REDPOINT TECHNICALLY GAMES

THE SUPER IMPORTANT SUMMARY SLIDE

- Everything is a block of memory. *Everything is a block of memory.*
  Everything is a block of memory.

- Always define or delete copy and move constructors to avoid unexpected behaviour. `UE_NONCOPYABLE(T)` is your friend!

- If you have a user-defined move constructor, you must also implement the copy assignment operator. Implement the move assignment operator as well if you want those semantics.

- Expect it to take a while to get into the mindset of “everything is a block of memory”. It’s different to most if not all other programming languages you’re used to.