Allocators: For the next 700 programming languages

Mike Brown mike@segfault.co.uk 19/06/2021

1. Foreword

I have reviewed and taken influence from several sources to come to the conclusions in this paper, and this paper owes a great deal to those resources. Zig, Oden, Dlang, C and C++, and most notably talks from Pablo Halpern, Alisdair Meredith and Andrei Alexandrescu.

This paper has code samples. These are to only highlight key ideas and concepts. None are recommended coding styles or practises.

A particular focus is made on C++, partly due to its popularity in the industry –common ground for discussion, and partly due to previous attempts to implement allocators and the historical lessons we have learnt.

As we are looking to improve on the current designs and design choices, there is a rather critical and negative light cast on C++. As an engineer, we need to always recognise and appreciate that while not perfect, these designs and features have provided some value to their users.

2. Introduction

This paper is intended to raise awareness to allocators for budding languages designers, specifically designers looking to design a Procedural System Programming Language[1]. We will discuss the overall shape of the problem, and to propose a framework for supporting custom and composable allocators.

Why have we Struggled?

It is always important to review the work from those who have travelled this path before us, to understand what issues they faced, and the reasons why. From the works I reviewed, I have one subjective opinion worth mentioning, a core motivation of this paper:

It is very easy to ignore allocators in the early to mid-term days of designing and implementing a language. It is possible to create very large and complex applications with just a basic memory management strategy. It is only when systems become large and complex enough for them to be a needed feature.

Allocation is the responsibility of a non-functional requirement – Memory Management, a property of your systems architecture. Systems properties transparently weave themselves into your systems behaviours.

The combination of these two points is at the heart of the problem. We leave designing the allocation strategies and framework too late. Other design choices often conflict, and we may now have a substantial existing code base – Because allocation is a system

[1] For this paper, System Programming Language is a language that allows you to access, manipulate, and control the position and layout of the underline bits.

property, they will influence all parts of your design choices. It becomes costly to alter and fix issues.

3. Shape of the Problem

Let's start by focusing on heap memory management and heap allocation. Heap allocation is contrasted with Stack allocation. Both are high level dynamic memory allocation strategies. Stack allocation will be discussed later.

To understand the responsibilities of heap allocation ("allocation" for the remainder of this section), we must start by quickly covering some conceptual types. Their design goals and usage effect how allocation will interact with them.

Primitive, and Composite Types

A Primitive Data type[i] is a basic building block type. E.g. `int`, `float`, `char`. They are built-in and provided by the language.

A Composite type[ii] is a type constructed from Primitive Data types, or other Composite types. In C, user-defined Composite types can be created with a `struct` definition.

While definitions have wondered over time, the roots in PODs[iii] and Data Records[iv] can help for us to understand their original design.

Composite types do not contain behaviour; they describe the *structure* of a blob of data. Recognising that Composite types are about structure is key to understanding their differences with Reference types.

Structural subtyping[v] is a feature that allows accessing a common subset structure in a Composite type. Figure 1 is an example of this.

Figure 1: Structural Subtyping

```
struct A {
    int value;
};

struct B: A {
    int double_value;
};

void foo(A& input) {
    input.value = 30;
    // We cannot keep the invariance
    // of value and double_value
    // input.double_value = 60;
}

int main() {
    B b;
    b.value = 10;
    b.double_value = 20;

    foo(b);
    return 0;
}
```

`foo()` only has a subset structural view of the data blob `b`. `foo()` has no way to alter or affect the `double_value` field of `b`.

Reference Types

Rather than a structural view of a type, A Reference type uses a behavioural view.

A Reference type provides a set of procedures that are used to interact with the underline data of that type.

In Figure 2, we have now tied `value` and `double_value` together with, which was our original intention. It no longer makes sense to separate them into separate structural definitions (Previously Figure 1 type `A` and `B`). If we want to set `m_value`, we do this via the `value()` procedure.

Figure 2: Behavioural View

```cpp
class C {
    int m_value;
    int m_double_value;
 public:
    C(int v) {
        m_value = v;
        m_double_value = v * 2;
    }
    void value(int v) {
        m_value = v;
        m_double_value = v * 2;
    }
};

int main() {
    C c(10);

    c.value(30);
    return 0;
}
```

Why is this important to Allocation?

Allocation is an action – functionality, procedures invoke allocation. Composite types do not contain procedures, but Reference types do contain their own procedures – Conceptually isolated and separate from the code the end-user programmer is writing to use that Reference type. And likely written by someone else entirely. This is the reason why we need to understand the differences of Composite and Reference types.

We will need a way to externally configure the allocation strategy of a Reference type.

4. Allocating Primitive & Composite types

Due to their structural focus, Composite type allocation is simpler. Given a blob of data of the correct size, overlay the structural shape over that blob, and you are ready to go.

Figure 3

```cpp
#include <cstdlib>

struct A { int value; };
struct B: A { int double_value; };

void foo(A& input) {
    input.value = 30;
}

int main() {
    // Using a C++ Raw Reference to
    // keep code similar to Figure 1
    B& b = *(B*)malloc(sizeof(B));
    b.value = 10;
    b.double_value = 20;

    foo(b);
    free(&b);
    return 0;
}
```

The highlighted lines in Figure 3 show the heap allocation mechanisms to the malloc allocation strategy used for Primitive and Composite types.

As a quick aside, malloc returns memory that is aligned for use as any type[2]. When we look at composing allocators, alignment will become a greater consideration.

Valid State

Procedures that modify a Composite type can take advantage in knowing the prior and future sequence of instructions – a Composite type can be

[2] Fundamentally aligned types - https://en.cppreference.com/w/c/memory/malloc

in a partially valid state while being processed.

In contrast, the procedure "stub" nature of a Reference type, which I will call methods, typically means the Reference type must always be in a valid state. The methods cannot know which or in which order they will be called in. In order to always be in a valid state allocation must be followed by construction (A call to the constructor method), an example of the C++ new mechanism is shown in Figure 4.

Figure 4

```
class C {
    int m_value;
    int m_double_value;
 public:
    C(int v) {
        m_value = v;
        m_double_value = v * 2;
    }
    void value(int v) {
        m_value = v;
        m_double_value = v * 2;
    }
};

int main() {
    C& c = *new C(10);

    c.value(30);
    delete &c;
    return 0;
}
```

Polymorphic Subtyping

Value & Reference Semantics

Because we now interact with the underline data via an opaque level of indirection, with methods, gives rise to the possibility of Reference Semantics.

5. Allocating Reference types

From here onwards, we will be discussing Reference types only. A quick reminder of their allocation requirements:

1. Reference types have isolated stubs, we need to inject allocation logic into them without knowledge of the underline procedure.

2. Reference types must be in a valid state throughout their lifetime.

Problems with C++ Allocation

C Pointers are not enough

C is the grandfather of many Systems Programming Languages. Every language I have read has been influenced by C in some way, even if its attempts to be a "Better C".

C does not have Reference types; it only has Primitive and Composite types.

6. Allocator Composability

(Extension) Reference Counted

`refCounted` is an extension allocator – Taking a base allocator strategy, this allocator will prefix all allocations with an atomic count representing the number of active references that point to this managed memory.

Treating reference counting as an extension to the allocator rather than to the ref handle type has the following advantages.

1. End-user programmer code is written with the same ref handle

type. Allocation strategy knowledge is not leaked into the codebase.
2. Location of the counts is handled by the allocator, if the allocator is single-threaded, the count operations do not need to be atomic.
3. Sharing of references is very common, the underline mechanisms needed to enable reference counting can also be used for debugging and error handling with non-reference counted allocation strategies.

7. Proposed Allocation Framework

[i] Primitive Data Type - https://en.wikipedia.org/wiki/Primitive_data_type
[ii] Composite Data Type - https://en.wikipedia.org/wiki/Composite_data_type
[iii] PODS - https://en.wikipedia.org/wiki/Passive_data_structure
[iv] Data Record - https://en.wikipedia.org/wiki/Record_(computer_science)
[v] Structural Subtyping - https://en.wikipedia.org/wiki/Composite_data_type#Primitive_subtype