Apex.AI

Practical memory pool based allocators for Modern C++

by Misha Shalem
misha.shalem@apex.ai

- CppCon 2019:

  Safe Software for Autonomous Mobility With Modern C++

  by Andreas Pasternak

- Quote:

  “Memory pools and allocators are only one piece of the solution”

Today we going to talk about this one piece in (more) depth

Memory allocations in real-time safety-critical environment

- The solution should be ***safe*** and ***certifiable***
- What does it mean practically for C++ memory allocations?

We asked an independent 3rd party safety assessor and the answer was

“It should comply to Autosar C++ 14 Coding Guidelines regarding memory allocations”

Autosar C++ guidelines

Rule A18-5-5 (required, implementation, partially automated)

Memory management functions shall ensure the following:

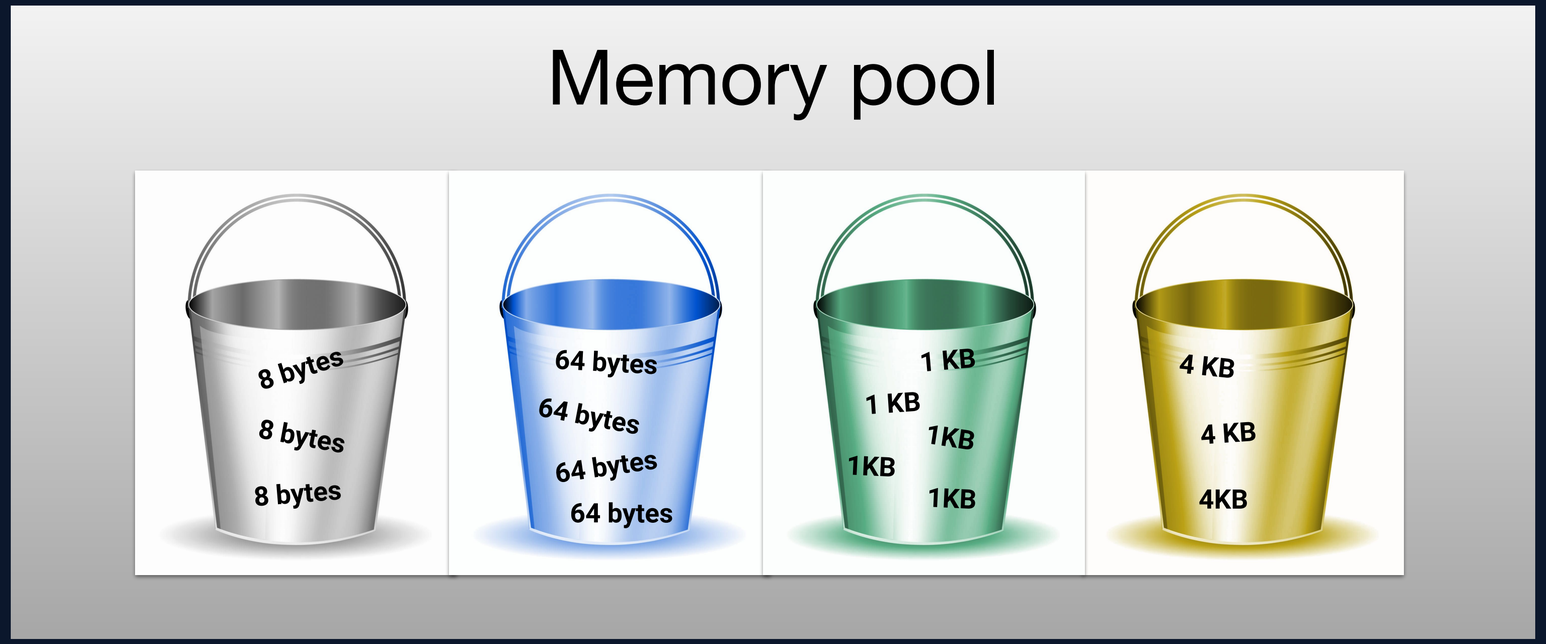
- ***deterministic behavior resulting with the existence of worst-case execution time***
- ***avoiding memory fragmentation***
- avoid running out of memory
- avoiding mismatched allocations or deallocations
- ***no dependence on non-deterministic calls to kernel***

Rule A18-5-7 (required, implementation, non-automated)

If non-realtime implementation of dynamic memory management functions is used in the project, then ***memory shall only be allocated and deallocated during non-realtime program phases***

Memory pool

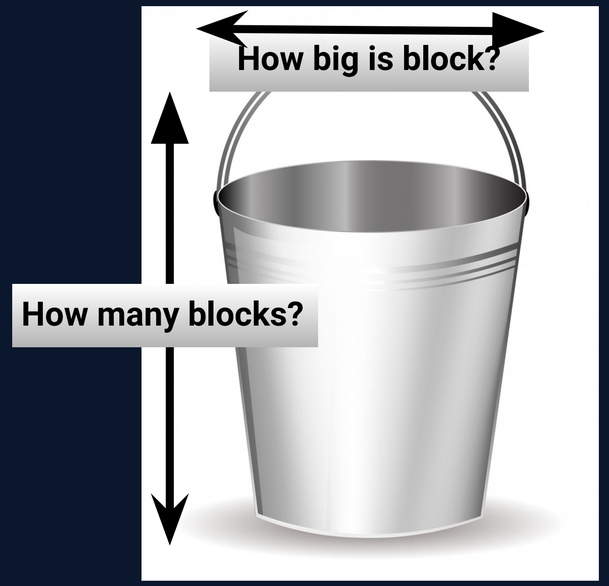
- Memory pools with fixed block sizes is a solution which addresses these issues:
  - It provides constant time (de)allocations in a preallocated buffer of memory
  - It avoids memory fragmentation with accurately-sized buckets
  - It is easy to implement, understand and reason about



Memory pools and C++

There is a problem though...

The developer needs to know dimensions of all buckets in advance




```
BucketSize = BlockSize * BlockCount
```

Memory pools and C++

- Maximum block count depends on the runtime
- **But block sizes should be known at compile time**

However...

C++ is all about high-level abstractions which hide the real sizes of allocations

Examples: `std::map`, `std::list`, `std::promise`, `std::shared_ptr`

All these can be parametrized with a user-provided allocator but they never actually use it directly

Memory pools and C++

There is good news though

Even when the user-provided allocator is not used, a related (“rebound”) one usually is

And allocators are always familiar with the real types they allocate for

```
template<typename T>
class allocator {
  // …
  template<typename U>
  struct rebind {
    using other = allocator<U>;
  };
   // …
  using pointer = T*;
   // …
  pointer allocate(size_type n, const void * hint = 0) {
     // has to allocate n * sizeof(T) bytes
  }
  // ...
};
```

Goal

It should be possible to use any code which adopts standard C++ allocator model without modifications but allocate from a memory pool under the hood

Implementation

Components:

1. A memory pool
2. An allocator which allocates from a memory pool
3. A way to generate bucket definitions according to the actual allocations

Memory pool implementation

Components:

1. ***A memory pool***
2. An allocator which allocates from a memory pool
3. A way to generate bucket definitions according to the actual allocations

Memory pool implementation

- A minimal implementation suffices as an example:
  - Some error checks are skipped for better readability
  - The implementation is not thread-safe
  - Default alignment is assumed
- However, multiple instances can be defined in the same application
  - Safety critical apps usually require separation of memory pools
  - A pool per thread would allow to avoid locking altogether

Bucket implementation

Bucket

```
class bucket  {
public:
    const std::size_t BlockSize;
    const std::size_t BlockCount;

    bucket(std::size_t block_size, std::size_t block_count);
    ~bucket();
    // Tests if the pointer belongs to this bucket
    bool belongs(void * ptr) const noexcept;
    // Returns nullptr if failed
    [[nodiscard]] void * allocate(std::size_t bytes) noexcept;
    void deallocate(void * ptr, std::size_t bytes) noexcept;

private:
    // Finds n free contiguous blocks in the ledger and returns the first block's index or BlockCount on failure
    std::size_t find_contiguous_blocks(std::size_t n) const noexcept;
    // Marks n blocks in the ledger as "in-use" starting at 'index'
    void set_blocks_in_use(std::size_t index, std::size_t n) noexcept;
    // Marks n blocks in the ledger as "free" starting at 'index'
    void set_blocks_free(std::size_t index, std::size_t n) noexcept;

   // Actual memory for allocations
   std::byte* m_data{nullptr};
   // Reserves one bit per block to indicate whether it is in-use
   std::byte* m_ledger{nullptr};
};
```

Construction/Destruction

```
bucket::bucket(std::size_t block_size, std::size_t block_count)
  : BlockSize{block_size}
  , BlockCount{block_count}
{
  const auto data_size = BlockSize * BlockCount;
  m_data = static_cast<std::byte*>(std::malloc(data_size));
  assert(m_data != nullptr);
  const auto ledger_size = 1 + ((BlockCount - 1) / 8);
  m_ledger = static_cast<std::byte*>(std::malloc(ledger_size));
  assert(m_ledger != nullptr);
  std::memset(m_data, 0, data_size);
  std::memset(m_ledger, 0, ledger_size);
}

bucket::~bucket() {
  std::free(m_ledger);
  std::free(m_data);
}
```

bucket::allocate()

```
void * bucket::allocate(std::size_t bytes) noexcept {
  // Calculate the required number of blocks
  const auto n = 1 + ((bytes - 1) / BlockSize);

  const auto index = find_contiguous_blocks(n);
  if (index == BlockCount) {
    return nullptr;
  }

  set_blocks_in_use(index, n);
  return m_data + (index * BlockSize);
}
```

bucket::deallocate()

```
void bucket::deallocate(void * ptr, std::size_t bytes) noexcept {
    const auto p = static_cast<const std::byte *>(ptr);
    const std::size_t dist = static_cast<std::size_t>(p - m_data);
    // Calculate block index from pointer distance
    const auto index = dist / BlockSize;
    // Calculate the required number of blocks
    const auto n = 1 + ((bytes - 1) / BlockSize);
    // Update the ledger
    set_blocks_free(index, n);
}
```

Memory pool implementation

How to configure the pools?

- A memory pool instance is merely a collection of buckets
- Each bucket has two properties: `BlockSize` and `BlockCount`
- A collection of these properties per bucket will define an instance of the pool

- What would a collection of parameters look like?
- How would we define more than one pool instance?

```
// The default implementation defines a pool with no buckets
template<std::size_t id>
struct bucket_descriptors {
    using type = std::tuple<>;
};
```

Specializing instances

```
struct bucket_cfg16 {
    static constexpr std::size_t BlockSize = 16;
    static constexpr std::size_t BlockCount = 10000;
};

struct bucket_cfg32{
    static constexpr std::size_t BlockSize = 32;
    static constexpr std::size_t BlockCount = 10000;
};

struct bucket_cfg1024 {
    static constexpr std::size_t BlockSize = 1024;
    static constexpr std::size_t BlockCount = 50000;
};

template<>
struct bucket_descriptors<1> {
    using type = std::tuple<bucket_cfg16, bucket_cfg32, bucket_cfg1024>;
};
```

Creating the memory pool(s)

```
template<std::size_t id>
using bucket_descriptors_t = typename bucket_descriptors<id>::type;

template<std::size_t id>
static constexpr std::size_t bucket_count = std::tuple_size<bucket_descriptors_t<id>>::value;

template<std::size_t id>
using pool_type = std::array<bucket, bucket_count<id>>;

template<std::size_t id, std::size_t Idx>
struct get_size
: std::integral_constant<std::size_t,
      std::tuple_element_t<Idx, bucket_descriptors_t<id>>::BlockSize>{};

template<std::size_t id, std::size_t Idx>
struct get_count
: std::integral_constant<std::size_t,
      std::tuple_element_t<Idx, bucket_descriptors_t<id>>::BlockCount>{};
```

Creating the memory pool(s)

```
template<std::size_t id, std::size_t... Idx>
auto & get_instance(std::index_sequence<Idx...>) noexcept {
  static pool_type<id> instance{{{get_size<id, Idx>::value, get_count<id, Idx>::value} ...}};
  return instance;
}

template<std::size_t id>
auto & get_instance() noexcept {
  return get_instance<id>(std::make_index_sequence<bucket_count<id>>());
}
```

How to choose the right bucket?

How about simply going through the buckets until a big enough is found?

(Assuming they are sorted by block size)

memory_pool::allocate()

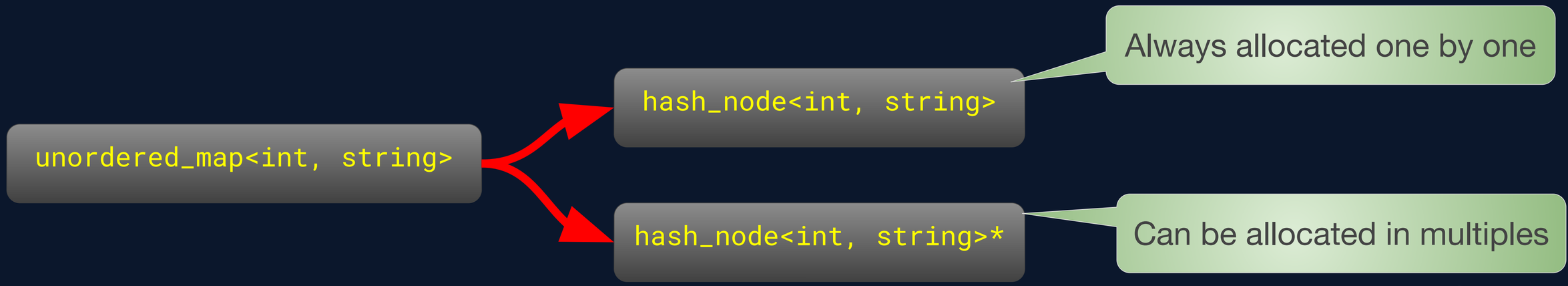
Naive “First Fit” strategy

```
// Assuming buckets are sorted by their block sizes
template<std::size_t id>
[[nodiscard]] void * allocate(std::size_t bytes) {
  auto & pool = get_instance<id>();

  for (auto & bucket : pool) {
    if(bucket.BlockSize >= bytes) {
      if(auto ptr = bucket.allocate(bytes); ptr != nullptr) {
         return ptr;
      }
    }
  }

  throw std::bad_alloc{};
}
```

How to choose the right bucket?

This will work for the types which always allocate memory for their objects one by one (like `std::map`) but not for those which allocate memory for multiple objects at once (like `std::unordered_map`):

- There may simply be no bucket big enough
- A “wrong” bucket can be chosen which will be “stolen” from another type



`memory_pool::allocate()`

Closer to a real one

When allocating from a bucket it is unknown whether the allocation is for one or multiple objects — only the size in bytes is given

N.B. This also mimics how `std::pmr::memory_resource` defines the allocation interface

memory_pool::allocate()

Closer to a real one

One way to solve this problem would be to find which allocation would lead to the least wasted memory and will take the least amount of blocks

```
struct info {
  std::size_t index{0}; // Which bucket?
  std::size_t block_count{0}; // How many blocks would the allocation take from the bucket?
  std::size_t waste{0}; // How much memory would be wasted?

  bool operator<(const info & other) const noexcept {
    return (waste == other.waste) ? block_count < other.block_count : waste < other.waste;
  }
};
```

memory_pool::allocate()

```
template<std::size_t id>
[[nodiscard]] void * allocate(std::size_t bytes) {
  auto & pool = get_instance<id>();
  std::array<info, bucket_count<id>> deltas;
  std::size_t index = 0;
  for (const auto & bucket : pool) {
    deltas[index].index = index;
    if (bucket.BlockSize >= bytes) {
      deltas[index].waste = bucket.BlockSize - bytes;
      deltas[index].block_count = 1;
    } else {
      const auto n = 1 + ((bytes - 1) / bucket.BlockSize);
      const auto storage_required = n * bucket.BlockSize;
      deltas[index].waste = storage_required - bytes;
      deltas[index].block_count = n;
    }
    ++index;
  }

  sort(deltas.begin(), deltas.end()); // std::sort() is allowed to allocate

  for (const auto & d : deltas)
    if (auto ptr = pool[d.index].allocate(bytes); ptr != nullptr)
      return ptr;

  throw std::bad_alloc{};
}
```

How about fragmentation?

Rule A18-5-5 (required, implementation, partially automated)

Memory management functions shall ensure the following:

- *deterministic behavior resulting with the existence of worst-case execution time*
- ***avoiding memory fragmentation***
- *avoid running out of memory*
- *avoiding mismatched allocations or deallocations*
- *no dependence on non-deterministic calls to kernel*

Haven’t we just reintroduced (local) memory fragmentation?

It depends on the logic of the users

They should be mindful of allocation/deallocation patterns and keep memory pools separated

memory_pool::deallocate()

```
template<std::size_t id>
void deallocate(void * ptr, std::size_t bytes) noexcept {
  auto & pool = get_instance<id>();

  for (auto & bucket : pool) {
    if (bucket.belongs(ptr)) {
      bucket.deallocate(ptr, bytes);
      return;
    }
  }
}
```

Explicit initialization of the memory pool

```
template<size_t id> // Is there a specialization for this id?
constexpr bool is_defined() noexcept {
    return bucket_count<id>!= 0;
}

template<size_t id>
bool initialize() noexcept {
    (void) get_instance<id>();
    return is_defined<id>();
}
```

Allocator implementation

Components:

1. A memory pool
2. ***An allocator which allocates from a memory pool***
3. A way to generate bucket definitions according to the actual allocations

Allocator implementation

The allocator:

- Implements standard C++ allocator model compatible with `std::allocator_traits`
- Is bound to a specific memory pool instance in compile time
- Transparently allocates from another resource until memory pool is defined *(development)*
- Facilitates definitions of the buckets *(testing and production)*

Usage scenarios

Using the same memory pool for nested objects:

```
static constexpr auto AllocId = 1;
using list = std::list<int, static_pool_allocator<int, AllocId>>;
using vector = std::vector<list, static_pool_allocator<list, AllocId>>;
// Both vector and list allocate from the pool with id 1
vector v;
v.emplace_back(5u, 42); // vector is [{ 42, 42, 42, 42, 42}]
```

Using separate memory pools for nested objects:

```
static constexpr auto ListAllocId = 1;
static constexpr auto VectorAllocId = 2;
using list = std::list<int, static_pool_allocator<int, ListAllocId>>;
using list_allocator = typename list::allocator_type;
using vector_allocator = static_pool_allocator<list, VectorAllocId>;
using scoped_adapter = std::scoped_allocator_adaptor<vector_allocator, list_allocator>;
using vector = std::vector<list, scoped_adapter>;
// List and vector allocate from different memory pools (1 and 2 respectively)
vector v;
v.emplace_back(5u, 42); // vector is [{ 42, 42, 42, 42, 42}]
```

Static pool allocator

The implementation skips the boilerplate code which is identical to

`std::pmr::polymorphic_allocator`

Static pool allocator

```
template<typename T = std::uint8_t, std::size_t id = 0>
class static_pool_allocator {
public:
    template<typename U>
    struct rebind { using other = static_pool_allocator<U, id>; };

    static_pool_allocator() noexcept : m_upstream_resource{pmr::get_default_resource()} {}
    static_pool_allocator(pmr::memory_resource * res) noexcept : m_upstream_resource{res} {}

    template<typename U>
    static_pool_allocator(const static_pool_allocator<U, id> & other) noexcept
          : m_upstream_resource{other.upstream_resource()} {}

    template<typename U>
    static_pool_allocator & operator=(const static_pool_allocator<U, id> & other) noexcept {
         m_upstream_resource = other.upstream_resource();
         return *this;
    }

    static bool initialize_memory_pool() noexcept { return memory_pool::initialize<id>(); }

private:
    pmr::memory_resource * m_upstream_resource;
};
```

static_pool_allocator::allocate()

```
pointer allocate(size_type n, const void * = 0) {
  // Reminder: here T is the type which we actually allocate for
  if constexpr (memory_pool::is_defined<id>()) {
    return static_cast<T*>(memory_pool::allocate<id>(sizeof(T) * n));
  } else if (m_upstream_resource != nullptr) {
    return static_cast<T*>(m_upstream_resource->allocate(sizeof(T) * n, alignof(T)));
  } else {
    throw std::bad_alloc{};
  }
}
```

How to collect the type sizes?

- *Could we just log the sizes of `T` in the runtime?*
  - **Does not necessarily reflects 100% line coverage**
- *What if we were to make sure the line coverage is 100%?*
  - **It could work but it can be impossible or unpractical to do for a real application**
  - **It would lack the call site information which would limit the insight users can get about where and why allocations happen, making the debugging and reasoning harder**

It would be best to register all calls to `allocate()` instantiated for every type `T` in the compiled code, regardless of the runtime flow, as well as the call path which led to these allocations

Injecting Instrumentation Code

```
namespace instrument {
  template<std::size_t id, typename T, std::size_t size>
  void type_reg() {}
}
```

```
pointer allocate(size_type n, const void * = 0) {

  instrument::type_reg<id, T, sizeof(T)>();

  if constexpr (memory_pool::is_defined<id>()) {
    return static_cast<T*>(memory_pool::allocate<id>(sizeof(T) * n));
  } else if (m_upstream_resource != nullptr) {
    return static_cast<T*>(m_upstream_resource->allocate(sizeof(T) * n, alignof(T)));
  } else {
    throw std::bad_alloc{};
  }
}
```

static_pool_allocator::deallocate()

```
void deallocate(T * ptr, size_type n) {
  if constexpr (memory_pool::is_defined<id>()) {
    memory_pool::deallocate<id>(ptr, sizeof(T) * n);
  } else if (m_upstream_resource != nullptr) {
    m_upstream_resource->deallocate(ptr, sizeof(T) * n, alignof(T));
  } else {
    assert(false);
  }
}
```

Allocation analyzer implementation

Components:

1. A memory pool
2. An allocator which allocates from a memory pool
3. ***A way to generate bucket definitions according to the actual allocations***

Allocation analyzer implementation

In order to collect the information injected by the allocator we need to:

- Compile the code with `clang` into LLVM bitcode
- Implement a custom LLVM pass(es) which would be able to:
  - Compose the list of all unique allocation types per pool instance
  - Record the call graph for every allocation (for debug purposes)
  - Generate bucket and memory pool definitions
- Run the pass on the bitcode using the LLVM `opt` tool

Compile the code with clang into LLVM bitcode

- The code should be compiled with:
  - `-g` to have the *file:line* information for call graphs
  - `-O0` to prevent optimizing out our instrumentation calls
  - `-emit-llvm` to compile into the bitcode
  - Probably `-DNDEBUG` to get the *`Release`* versions of data structures

Then the bitcode files can be (optionally) linked together with `llvm-link`

Printing the list of all unique allocations

```
class AllocListPass : public llvm::FunctionPass {
public:
  static char ID;
  AllocListPass() : llvm::FunctionPass(ID) {}

  bool runOnFunction(llvm::Function & f) override {
    const auto pretty_name = boost::core::demangle(f.getName().str().c_str());
    static const std::regex call_regex{R"(void instrument::type_reg<([^,]+),(.+),([^,]+)>\(\))"};

    std::smatch match;
    if (std::regex_match(pretty_name, match, call_regex)) {
      if (match.size() == 4) {
        const auto pool_id = std::atoi(match[1].str().c_str());
        const auto type = match[2].str();
        const auto size = std::atoi(match[3].str().c_str());
        std::cout  << "Pool ID: " << pool_id << ", Size: " << size << ", Type: " << type << "\n";
      }
    }
    return false; // does not alter the code, a read-only pass
  }
};

char AllocListPass::ID = 0;
static llvm::RegisterPass<AllocListPass> dummy("alloc-list", "This pass lists memory pool allocations");
```

Analyzing call graph for every allocation

```
0  // file: /home/program.cpp
1  int main() {
2    f();
3    return 0;
4  }
5
6  void f() {
7    x();
8  }
9
10 void x() {
11   std::list<int, static_pool_allocator<int, 3>> lst;
12   lst.push_back(1);
13   // ...
14 }
```

- This time we use `llvm::ModulePass`
- Run a ***depth-first search*** starting from entry points (e.g. `main()`)
- Every time `type_reg<>` is discovered:
  - Register allocation type
  - Print the the call graph for allocation
- Check for recursions along the way and drop these branches
- Finally, the list of unique allocations can be translated into a header file with memory pool specializations

```
Call graph for: Pool ID: 3, Size: 24, Type:  std::__1::__list_node<int, void*>:

1. static_pool_allocator<std::__1::__list_node<int, void*>, 3ul>::allocate(unsigned long, void const*) called at /usr/include/c++/v1/memory:1547
2. std::__1::allocator_traits<static_pool_allocator<std::__1::__list_node<int, void*>, 3ul>>::allocate(static_pool_allocator<std::__1::__list_node<int, void*>, 3ul>&,
unsigned long) called at /usr/include/c++/v1/list:1079
3. std::__1::list<int, static_pool_allocator<int, 3ul>>::__allocate_node(static_pool_allocator<std::__1::__list_node<int, void*>, 3ul>&) called at
/usr/include/c++/v1/list:1569
4. std::__1::list<int, static_pool_allocator<int, 3ul>>::push_back(int const&) called at /home/program.cpp:12
5. x() called at /home/program.cpp:7
6. f() called at /home/program.cpp:2
```

Call graph caveats

The call graph might be incomplete because of:

- Virtual calls
- Calls through a function pointer
- Inline assembler

Both approaches can be combined by completing the list with allocations which are not in the call graph (“orphaned”) as a compromise solution

Generating memory pool definitions

```
opt -load alloc-analyzer.so -alloc-analyze -gen-hdr my_defs.hpp -entry-point "main"< home/program.bc -o /dev/null
```

```cpp
// file: my_defs.hpp
struct bucket_cfg24 {
    static constexpr std::size_t BlockSize = 24;
    static constexpr std::size_t BlockCount = 10000;
};

/* More buckets for memory pool 3 */

template<>
struct bucket_descriptors<3> {
    using type = std::tuple</* more buckets */ bucket_cfg24 /* more buckets */>;
};

/* Other buckets and memory pool specializations */
```

Include, recompile…

Summary

- Memory allocations is a fundamental part of C++
- Memory allocations have to be deterministic to be usable in safety-critical applications
- A memory pool with fixed bucket sizes can be a solution but it is complex to use with the C++ allocator model
- An instrumenting allocator combined with an analyzing tool can be a certifiable solution for this problem

Thank you!

misha.shalem@apex.ai