

# C++ Programming for the Heap-Deprived

Asaf Helfer

 BrightSource Energy

Core C++ 2019

# Agenda

- When do we use heap allocations?
- Why not to use heap allocations?
- Use cases and no-heap replacements

# !Agenda

- Containers of unknown sizes
  - Memory pools
  - Allocators
- Thread safety

# Storage Duration

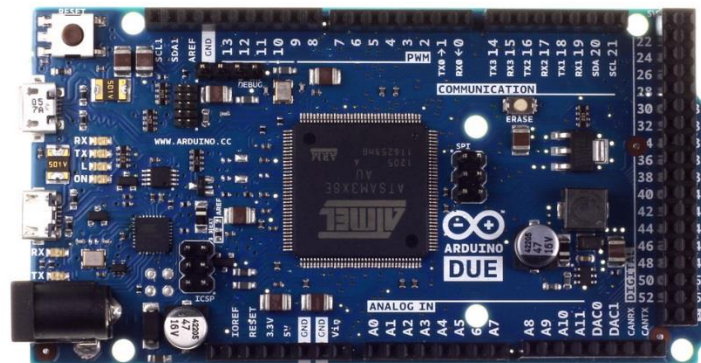
- **Static:**
  - Duration is entire program lifetime. Address is set at link time.
- **Automatic:**
  - aka stack. Automatic variables. Address depends on flow.
- **Dynamic:**
  - aka heap. Manually managed. Address depends on flow.
- **(Thread – not relevant for this talk)**

# Why Use Dynamic Allocations?

- Decide on required memory size at runtime
- Separate allocation and initialization context from object lifetime

# Memory Usage in Micro Controllers

- Direct access to memory – a single memory space
- Define memory sections manually
- Define your own stack(s) memory area
- Define exact address locations of some data items
- RAM is usually very limited



# Problems with using the Heap

- Why not to allocate?
  - No determinism
    - We want data addresses to be known in advance
  - Memory fragmentation
  - Runtime failures
    - We don't want memory-related runtime failures
  - Runtime performance

# Memory Pools

- A good solution, but not for this problem
- Might solve fragmentation if designed correctly
- Does not help with determinism and runtime failures



# C++ Standard

- Classes which are guaranteed not to use dynamic memory allocation:
  - Optional

When an instance of `optional<T>` *contains a value*, it means that an object of type `T`, referred to as the optional object's *contained value*, is allocated **within the storage** of the optional object. **Implementations are not permitted to use additional storage, such as dynamic memory**, to allocate its contained value.

## 23.6.3 Class template optional

- Variant

When an instance of `variant` holds a value of alternative type `T`, it means that a value of type `T`, referred to as the variant object's *contained value*, is allocated **within the storage** of the variant object. **Implementations are not permitted to use additional storage, such as dynamic memory**, to allocate the contained value.

## 23.7.3 Class template variant

# Use case 1: A global application object

- Ideal – value semantics, app is on stack

```
int main()
{
    MyBigApplication app;
    app.run();
}
```



# Use case 1: A global application object

- Next best thing: Wrap memory allocation with value semantics, app is on heap

```
int main()
{
    auto app = std::make_unique<MyBigApplication>();
    app->run();
}
```

# Use case 1: A global application object

- With no heap allocation

```
int main()
{
    auto app = ???;
    app->run();
}
```

Where should we store app?

# Use case 1: A global application object

- We want to separate storage duration from object lifetime
- Storage will:
  - Have static duration
  - Be initialized dynamically

```
int main()
{
    static MyBigApplication app;
    app->run();
}
```

# Use case 1: A global application object

- A little too static?

```
int main()
{
    if (isNewerHardware())
    {
        static MyBigApplication app(port1, port2, port3);
        app.run();
    }
    else
    {
        static MyBigApplication app(port1);
        app.run();
    }
}
```

Memory will be allocated for both instances

# Use case 1: A global application object

- Use a static/global variable of type Lazy

```
static Lazy<MyBigApplication> s_app;

int main()
{
    MyBigApplication* app;
    if (isNewerHardware())
    {
        app = &s_app.construct(port1, port2, port3);
    }
    else
    {
        app = &s_app.construct(port1);
    }
    app->run();
}
```

# C++11: `std::aligned_storage`

```
template< std::size_t Len, std::size_t Align = /*default-alignment*/ >  
struct aligned_storage;
```

- Usage:

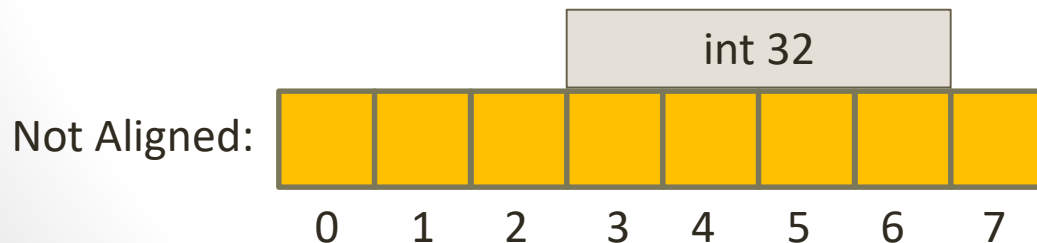
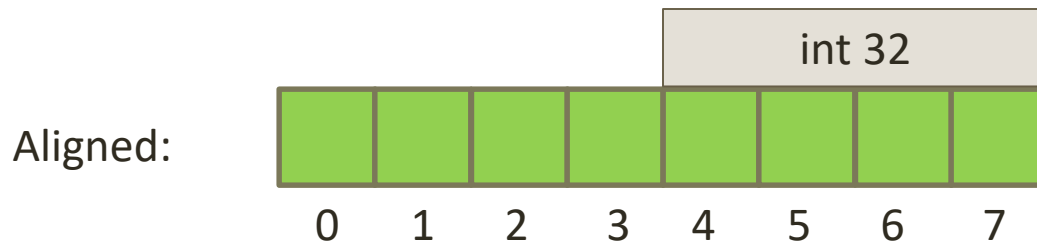
```
std::aligned_storage<8, 4>::type buffer;
```

- Provides an uninitialized storage
- Actual size and alignment are implementation defined
- C++14: `std::aligned_storage_t = std::aligned_storage::type`



# Memory Alignment

- Modern CPUs can't access memory at any address
  - Address must be aligned with the CPU word size
- Alignment of N means the memory address is divisible by N
- Accessing with the wrong alignment could cause a performance hit or even a system crash, depending on the CPU



# Memory Alignment in C++

- Each type has its own requirement for alignment, depending on hardware
- Alignment is always a power of 2
- Compilers will take care of that for you. If you let them.
  - `#pragma pack` doesn't let them
- C++11 added two keywords: `alignof`, `alignas`

# Uninitialized Storage for the Unlucky

- Pre-C++11

```
template <class T>
struct Storage
{
    uint64_t buffer[sizeof(T) / 8 + 1];
};
```

# Placement New

- Can be used to construct an object in a pre-allocated memory block
- For example:

```
struct A { ... };  
  
alignas(A) char buffer[sizeof(A)];  
  
// No allocation, only in-place construction  
A* a = new (buffer) A();  
  
// Explicit destruction  
a->~A();
```

# Lazy Initialization

```
template <class T>
class Lazy
{
    std::aligned_storage_t<sizeof(T), alignof(T)> m_storage;
    bool m_isInitialized = false;
public:
    ~Lazy() { destruct(); }

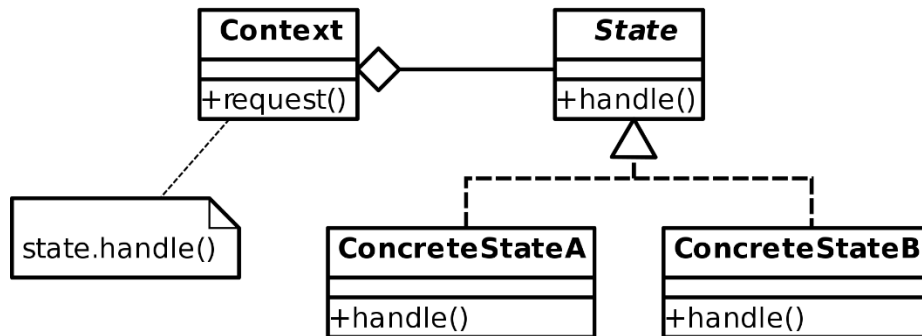
    template <typename... Args> T& construct(Args&&... args)
    {
        auto obj = ::new (&m_storage) T(std::forward<Args>(args)...);
        m_isInitialized = true;
        return *obj;
    }

    void destruct()
    {
        if (m_isInitialized)
        {
            get().~T();
            m_isInitialized = false;
        }
    }

    T& get() { return *reinterpret_cast<T*>(&m_storage); }
};
```

# Use case 2: State Pattern

- Structure:



# Use case 2: State Pattern

- We only need a single state object at a time
- **Heap-based** implementation:

```
// Given:
struct BaseState {};
struct State1 : public BaseState {};
struct State2 : public BaseState {};
struct State3 : public BaseState {};
struct State4 : public BaseState {};
struct State5 : public BaseState {};

// Use (within a context object):
std::unique_ptr<BaseState> currentState;
currentState = std::make_unique<State1>(*this);

currentState->event();
```

# Use case 2: State Pattern

- Could maybe use `std::variant`, but It's C++17 – compilers for microcontrollers are usually behind
- No heap – need a storage:

```
std::aligned_storage_t<sizeof(???), alignof(???)> stateBuffer;
```

- How big?



# Use case 2: State Pattern

- Find maximum size for allowed types – tail recursion:

```
template <typename First, typename... Args>
struct MaxSize
{
    static const std::size_t Size =
        MaxSize<First>::Size > MaxSize<Args...>::Size ?
        MaxSize<First>::Size :
        MaxSize<Args...>::Size;
};
```

```
template <typename First>
struct MaxSize<First>
{
    static const std::size_t Size = sizeof(First);
};
```

```
// C++14
template <typename... Args>
constexpr std::size_t MaxSize_v = MaxSize<Args...>::Size;
```

# Use case 2: State Pattern

- MaxSize for the Unlucky (pre-C++11):

```
template <typename T1, typename T2 = void, typename T3 = void,
         typename T4 = void, typename T5 = void>
class MaxSizeLegacy
{
    static const std::size_t TailSize =
        MaxSizeLegacy<T2, T3, T4, T5>::Size;
public:
    static const std::size_t Size =
        (MaxSizeLegacy<T1>::Size > TailSize) ?
        MaxSizeLegacy<T1>::Size :
        TailSize;
};

template <typename Type>
struct MaxSizeLegacy<Type, void, void, void, void>
{
    static const std::size_t Size = sizeof(Type);
};
```

# Use case 2: State Pattern

- And similarly for alignment:

```
template <typename First, typename... Args>
struct MaxAlign
{
    static const std::size_t Align =
        MaxAlign<First>::Align > MaxAlign<Args...>::Align ?
        MaxAlign<First>::Align :
        MaxAlign<Args...>::Align;
};

template <typename First>
struct MaxAlign<First>
{
    static const std::size_t Align = alignof(First);
};

// C++14
template <typename... Args>
constexpr std::size_t MaxAlign_v = MaxAlign<Args...>::Align;
```

# Use case 2: State Pattern

- And usage (assuming we know all types):

```
template <class BaseClass, class... DerivedClasses>
class GenericHierarchyFactory
{
    std::aligned_storage_t<
        MaxSize_v<DerivedClasses...>,
        MaxAlign_v<DerivedClasses...>
    > m_buffer;

    ...
};
```

# Use case 2: State Pattern

```
template <class BaseClass, class... DerivedClasses>
class GenericHierarchyFactory
{
    // std::aligned_storage_t<...> m_buffer;
    BaseClass* m_currentObject = nullptr;
public:
    template <class Derived, typename... ConstructionParams>
    BaseClass& construct(ConstructionParams&&... params)
    {
        static_assert(sizeof(Derived) <= sizeof(decltype(m_buffer)),
            "Derived class is too big for buffer");
        static_assert(alignof(Derived) <= alignof(decltype(m_buffer)),
            "Derived class is misaligned for buffer");

        auto createdObject = ::new (&m_buffer)
            Derived(std::forward<ConstructionParams>(params)...);
        m_currentObject = static_cast<BaseClass*>(createdObject);
        return *m_currentObject;
    }
    ...
};
```

# Use case 2: State Pattern

```
template <class BaseClass, class... DerivedClasses>
class GenericHierarchyFactory
{
    ...
    void destruct()
    {
        if (m_currentObject)
        {
            m_currentObject->~BaseClass();
            m_currentObject = nullptr;
        }
    }
};
```

Why do we need to store BaseClass\* in addition to buffer?

# Use case 2: State Pattern

- Usage:

```
GenericHierarchyFactory<BaseState, State1, State2,  
                        State3, State4, State5> stateFactory;
```

```
BaseState* state = &stateFactory.construct<State1>();  
stateFactory.destruct();
```

```
state = &stateFactory.construct<State2>(1, 2, 3.4f);  
stateFactory.destruct();
```

# Use case 3: Type-Erased Function Object

- Type erasure comes at a cost:
  - Indirect call (virtual function) – usually minor
  - Code size (templated called type)
- In some cases not necessary
  - Although will usually still need the virtual call



# Use case 3: Type-Erased Function Object

- Problem definition:
  - Input: Maximum size and alignment
  - Should wrap any callable type
  - If Callable is too big, fail at compile time

# Use case 3: Type-Erased Function Object

- A simplified type-erased function implementation, **heap based**:

```
template <typename FunctionSignature> class Function;
```

```
template <typename ReturnType, typename... Args>  
class Function<ReturnType(Args...)>
```

```
{
```

```
    struct CalleeInterface
```

```
    {
```

```
        virtual ~CalleeInterface() {}
```

```
        virtual ReturnType call(Args&&... args) = 0;
```

```
    };
```

```
    template <class CalleeType>
```

```
    struct Impl : public CalleeInterface
```

```
    {
```

```
        Impl(CalleeType callee) : m_callee(callee) {}
```

```
        virtual ReturnType call(Args&&... args) override
```

```
        {
```

```
            return m_callee(std::forward<Args>(args)...);
```

```
        }
```

```
        CalleeType m_callee;
```

```
    };
```

```
    ...
```

# Use case 3: Type-Erased Function Object

- A simplified type-erased function implementation, **heap based**:

```
std::unique_ptr<CalleeInterface> m_impl;

public:
    Function() = default;

    template <typename CalleeType>
    Function(CalleeType callee) :
        m_impl(std::make_unique<Impl<CalleeType>>(callee)) {}

    template <typename CalleeType>
    Function& operator=(CalleeType callee)
    {
        m_impl = std::make_unique<Impl<CalleeType>>(callee);
        return *this;
    }

    ReturnType operator()(Args... args) const
    {
        assert(m_impl);
        return m_impl->call(std::forward<Args>(args)...);
    }
};
```

# Use case 3: Type-Erased Function Object

- That was really super simplified
  - Don't use in production code
- Now for the storage-based version

# Use case 3: Type-Erased Function Object

- For **storage-based**, we need a storage for unlimited types:

```
template <std::size_t Size, std::size_t Alignment>
class AnyStorage
{
    std::aligned_storage_t<Size, Alignment> m_storage;
    using DestructorFunction = void(*)(void* objectPtr);
    DestructorFunction m_destructorFunction = nullptr;

public:
    template <typename T, typename... Args>
    T& construct(Args&&... args)
    {
        static_assert(sizeof(T) <= sizeof(decltype(m_storage)),
            "Type is too big for buffer");
        static_assert(alignof(T) <= alignof(decltype(m_storage)),
            "Type is misaligned for buffer");
        destruct();
        auto obj = ::new (&m_storage)
            T(std::forward<Args>(args)...);
        m_destructorFunction = [](void* ptr) {
            reinterpret_cast<T*>(ptr)->~T(); };
        return *obj;
    }
    ...
};
```

# Use case 3: Type-Erased Function Object

- For **storage-based**, we need a storage for unlimited types:

```
template <std::size_t Size, std::size_t Alignment>
class AnyStorage
{
    ...
    void destruct()
    {
        if (m_destructorFunction)
        {
            m_destructorFunction(&m_storage);
            m_destructorFunction = nullptr;
        }
    }

    ~AnyStorage() { destruct();}

    AnyStorage() = default;

    AnyStorage(const AnyStorage&) = delete;
    AnyStorage(AnyStorage&&) = delete;
    AnyStorage operator=(const AnyStorage&) = delete;
    AnyStorage operator=(AnyStorage&&) = delete;
};
```

# Use case 3: Type-Erased Function Object

- And the InplaceFunction class:

```
template <typename FunctionSignature,  
         std::size_t Size, std::size_t Alignment>  
class InplaceFunction;
```

```
template <typename ReturnType, typename... Args,  
         std::size_t Size, std::size_t Alignment>  
class InplaceFunction<ReturnType(Args...), Size, Alignment>  
{
```

```
    AnyStorage<Size, Alignment> m_storage;  
    CalleeInterface* m_impl = nullptr;
```

```
    ...
```

# Use case 3: Type-Erased Function Object

- And the InplaceFunction class:

```
...

public:
    InplaceFunction() = default;

    template <typename CalleeType>
    InplaceFunction(CalleeType callee) :
        m_impl(&m_storage.construct<Impl<CalleeType>>(callee)) {}

    template <typename CalleeType>
    InplaceFunction& operator=(CalleeType callee)
    {
        m_impl = &m_storage.construct<Impl<CalleeType>>(callee);
        return *this;
    }

    ReturnType operator()(Args... args)
    {
        assert(m_impl);
        return m_impl->call(std::forward<Args>(args)...);
    }
};
```



# Use case 3: Type-Erased Function Object

- An already invented wheel:
  - <https://github.com/WG21-SG14/SG14>
- `std::aligned_storage` might be bigger than you expect
- Again, super simplified

# Use case 4: Multi-Client Event

- Problem definition:
  - Service class can raise an event
  - Multiple Client classes should be able to register to the event
  - Service does not know the clients or how many are there
- Two variants:
  - Asynchronous – clients will handle the event in their own flow
  - Synchronous – clients will handle the event in the service flow
    - Callbacks

# Use case 4A: Multi-Client Async Event

- “Async” – client will check event on its own cycle.
- No callbacks
- Arbitrary number of ‘observers’
- No multithread synchronization

# Use case 4A: Multi-Client Async Event

What we want to achieve – independent clients:

```
Event e;  
e.trigger(); // No observers registered
```

```
AsyncObserver o1(e);  
AsyncObserver o2(e);
```

```
// No triggers since observers construction  
assert(o1.wasEventTriggered() == false);  
assert(o2.wasEventTriggered() == false);
```

```
e.trigger();  
e.trigger();
```

```
assert(o1.wasEventTriggered() == true);  
o1.resetEvent();
```

```
// Event was reset for this observer  
assert(o1.wasEventTriggered() == false);
```

```
// For this observer the event is not reset  
assert(o2.wasEventTriggered() == true);
```

# Use case 4A: Multi-Client Async Event

Can also work with data – application code sample:

```
class Calculator
{
    ConfigurationManager::Observer m_configChanged;

public:
    Calculator(const ConfigurationManager& manager)
    {
        manager.registerForConfigurationChange(m_configChanged);
    }

    void update()
    {
        if (m_configChanged.wasEventTriggered())
        {
            updateParameters(m_configChanged.getLastEventData());
            m_configChanged.resetEvent();
        }
    }
};
```

# Use case 4A: Multi-Client Async Event

Implementation – no need to hold a list of all clients:

- Each observer holds a reference to the event.

```
class BaseEvent
{
public:
    unsigned int getCounter() const { return m_counter; }

protected:
    void countEvent() { ++m_counter; }

private:
    unsigned int m_counter = 0;
};
```

# Use case 4A: Multi-Client Async Event

```
template <typename EventType = void>
class Event : public BaseEvent
{
public:
    void trigger(const EventType& data)
    {
        countEvent();
        m_lastData = data;
    }
    const EventType& getLastData() const { return m_lastData; }

private:
    EventType m_lastData;
};
```

```
// No data
template <> class Event<void> : public BaseEvent
{
public:
    void trigger() { countEvent(); }
};
```

# Use case 4A: Multi-Client Async Event

Observer implementation:

```
template <typename EventType = void>
class AsyncObserver
{
    const Event<EventType>* m_event = nullptr;
    unsigned int m_lastObservedCounter;

public:
    AsyncObserver() = default;
    AsyncObserver(const Event<EventType>& event)
    {
        observe(event);
    }

    void observe(const Event<EventType>& event)
    {
        m_event = &event;
        resetEvent();
    }

    void resetEvent()
    {
        if (m_event) { m_lastObservedCounter = m_event->getCounter(); }
    }
}
```



# Use case 4A: Multi-Client Async Event

Observer implementation (cont.):

```
bool wasEventTriggered() const
{
    if (!m_event) return false;
    return m_event->getCounter() != m_lastObservedCounter;
}

template <class ReturnT = const EventType>
typename std::enable_if_t<
    !std::is_same_v<EventType, void>, ReturnT>
getLastEventData() const
{
    assert(m_event);
    return m_event->getLastData();
}
};
```

# Use case 4B: Multi-Callback Event

- Heap-based solution:
  - Service class holds a `vector<callback>`
- Non-heap based solution: Again, a storage issue
- A special case of 'container of unknown size':
  - At compile time size is unknown locally, but known globally
- Solution: A 'distributed' list, where each node is allocated in the storage of a client

# Use case 4B: Multi-Callback Event

We want to have something like this (Service interface):

```
class Service
{
public:
    using EventCallbackType = ...;
    void registerCallback(EventCallbackType& item);
    void sendData(const DataItem& data);
};
```

# Use case 4B: Multi-Callback Event

We want to have something like this (Client implementation):

```
class Client
{
    Service::EventCallbackType myCallbackNode;
    DataItem m_lastData;
    int m_receiveCounter = 0;

public:
    Client(Service& service)
    {
        myCallbackNode.m_callback = [this](const DataItem& data)
        {
            ++m_receiveCounter;
            m_lastData = data;
        };
        service.registerCallback(myCallbackNode);
    }
};
```

# Use case 4B: Multi-Callback Event

Node implementation:

```
template <typename CallbackType>  
class CallbacksList;
```

```
template <typename CallbackType>  
class CallbackNode  
{  
private:  
    using List = CallbacksList<CallbackType>;  
    using Node = CallbackNode <CallbackType>;  
    friend List;
```

```
List* m_list = nullptr;  
Node* m_next = nullptr;  
Node* m_prev = nullptr;
```

...

# Use case 4B: Multi-Callback Event

Node implementation:

```
public:
    CallbackType m_callback;

    ~CallbackNode() { unlink(); }
    bool isLinked() { return m_list != nullptr; }

    void link(List& list)
    {
        unlink();
        m_list = &list;
        m_list->addToEnd(*this);
    }
    void unlink()
    {
        if (!isLinked()) { return; }
        m_list->remove(*this);
        m_list = m_next = m_prev = nullptr;
    }
    template <typename DataType>
    void onEventRaised(const DataType& data)
    {
        m_callback(data);
    }
};
```

# Use case 4B: Multi-Callback Event

List implementation:

```
template <typename CallbackType>
class CallbacksList
{
public:
    using Node = CallbackNode <CallbackType>;

    template <typename DataType>
    void trigger(const DataType& data)
    {
        auto node = m_head;
        while (node)
        {
            node->onEventRaised(data);
            node = node->m_next;
        }
    }
    ...
}
```

# Use case 4B: Multi-Callback Event

List implementation:

```
template <typename CallbackType>
class CallbacksList
{
    ...
private:
    Node* m_head = nullptr;
    Node* m_tail = nullptr;

    // Called within Node::link()
    void addToEnd(Node& node) { ... }

    // Called within Node::unlink()
    void remove(Node& node) { ... }
    friend Node;
};
```



# Use case 4B: Multi-Callback Event

Service implementation:

```
class Service
{
public:
    using CallbackType = InplaceFunction<void(const DataItem&), 8, 8>;
    using EventCallbackList = CallbacksList<CallbackType>;
    using EventCallbackType = EventCallbackList::Node;

    void registerCallback(EventCallbackType& item)
    {
        item.link(m_dataList);
    }

    void sendData(const DataItem& data)
    {
        m_dataList.trigger(data);
    }

private:
    EventCallbackList m_dataList;
};
```

# Summary

- To avoid dynamic memory allocations:
  - Use static instead of automatic storage
  - Use `std::aligned_storage`, except where actual size is important
  - Use compile time size calculations if possible types are known at compile time
  - To get type erasure, specify maximum size and alignment and check at compile time
    - For a function wrapper, use an existing solution
  - Even if container size is unknown at compile time locally, it might be known globally, so can use other code for storage
    - Distributed list of callbacks