I Got You, FAM

d1039r2: Flexible Array Members for C++23

ThePhD, Nicole Mazzuca, Arvid Gerstmann

EWG (Incubator), Köln 2019

The Goal

```
#include <cstddef>
#include <fam>

struct id_list {
	std::size_t len;
	std::string names[];

	id_list(std::fam_size fs) : len(fs.size()) {}
};

int main(int, char* args[]) {
	std::size_t num = compute_size(args);
	id_list list(std::fam_size(num));
	/* one allocation: use! */
	return 0;
}
```

Why FAMs?

Efficiency and Ease-of-Use

Primary Reason: Efficiency

- C++ is leaving room for a lower-level language, C
  - No way to type-safely allocate a (header) structure + its data

- Double-allocating is a non-starter in critical applications
  - Impossible for C++ to model many in-line structures without hacks or much error-prone manual handling

Primary Reason: Type Safety

- Tail-allocated data structures are violently unsafe in C
  - Malloc-based
  - Allocate and pray the byte size is correct
  - Properly manage every access to it and more

- Library-only data structure of `tail_allocated<Head, T>` is error-prone
  - ```
    struct bad {
         tail_allocated<packet_header, std::byte> data;
         /* … */
         tail_allocated<other_header, std::byte> data2;
         // no compiler error !!
    };
    ```
    ```

Secondary Reason: Existing Practice

- The C standard has them
  - Linux uses them effectively
  - Embedded and HSA-Briggs modules
- LLVM has tail-allocated data structures
  - hidden under library layers and vigorous code review to prevent misuse

Overrides for Memory Efficiency

- Many arrays track their own element count
  - do not need the implementation to store it for them alongside any other implementation-specific information

```cpp
#include <fam>
#include <cstddef>

struct id_list {
	std::size_t len;
	int64_t ids[];

	id_list(std::fam_size fs) : len(fs.size()) {}
};

namespace std {
	template <>
	struct fam_traits<::id_list> {
		constexpr static ::std::size_t size (const ::id_list& il) noexcept {
			return il.len;
		}
	};
}
```

std::fam_traits<T>

- 1 member: `size(const T&)` to return the size of the Flexible Array Type
  - required
- If not specialized by user: compiler uses implementation-specific storage scheme
  - Otherwise: object itself is passed to `size(const T&)` member and user can do whatever

`std::is_fam(_v)<T>`, `std::fam_element(_t)<T>`

- Some extra type traits
  - Check if something is a flexible array member
  - `std::fam_element` is not SFINAE friendly: it either has the proper member that gives the element type, or errors

Challenges

Safety + Efficiency in the C++ world

Simple part: C compatibility

- This feature goes nowhere if incompatible with C
  - But C compatibility is easy to design for
- Define size retrieval for all types where `std::is_trivial_v<element_type>` is true…
  - To only have to return the element count equal to ***or greater than***
  - Most implementations (libc, libstdc++) will only save the byte count, not element count (and it can over-allocate!)
  - For all types in C: `std::is_trivial_v<T>` is true!
  - Do not break C ABI: a plus!

Mildly Difficult: Non-Trivial types

- C++ lifetime revolves around constructor / destructor
  - Default-allocation of some expensive flexible array members prohibitive since it will do so on default construction

- Destructors are easy to implement
  - just filled out with individually destroying the elements of the FAM

Impossible Difficulty: Ease of Use + Performance

- Constructor automatically initializes all array elements of the FAT.
  - Raw memory initialization and then manual emplacement of individual elements: back to where we started
  - Is there a constructor we can specify to not have to rewrite everything?

- Should not solve for just FAMs?
  - Lots of types have problems with constructors member constructor syntax
  - Need full expressivity of statements to fill in members of array
  - Solve for everyone: FAMs benefit?

Thank You!