C++17 Meta Programming Best Practices Cheat Sheet

Daniel Nikpayuk

February 8, 2021

1. **Cache Minimization**: `constexpr` < `alias templates` < `struct templates`

   C++17 meta functions can be implemented in three different styles: constexpr functions, alias templates as functions, and struct templates as functions. A given meta function can be implemented in any of these three ways, but the compiler does additional preprocessing depending on the style chosen, and can in turn create orders of magnitude difference in performance. In general constexpr functions are least expensive; then alias templates; then struct templates.

   In the case you are working with variadic parameter packs (`Vs...`), the following paradigms help mitigate the accumulation of compile time costs:

   - Continuation passing.
   - Alias selection (dispatching).
   - Alias recursion.

2. **Universe Minimization**: `U_type_T` : `typename` → `auto`

   Template parameters come in two flavors: `typename` and `auto`. As objects of each belong to different *universes*, meta programmers are forced to duplicate every algorithm if they want to apply them to both. The existence of two distinct object *kinds* also increases the complexity of meta function signatures.

   To mitigate the costs associated with these issues, it's better to reencode `typename` objects as `auto` objects by using the `U_type_T` functor. This way we only have to interact with `auto` objects within our signatures. In the case that the type of an encoded object is needed down the line, the encoded object can always be decoded using the `T_type_U` : `auto` → `typename` inverse.

3. **Nesting Call Minimization**:

   In the previous cache ordering `struct templates` were ranked highest as they cache the most. When it comes to variadic packs (`Vs...`) they are the most expensive *bottleneck* in terms of compile time performance. Once these costs are mitigated though, the next biggest bottleneck has to do with `alias templates`.

   Individually `alias templates` cache less than struct templates, but every time a parameter pack is passed from one alias template to another, the whole pack is hard copied (by value). If the parameter pack is a long list, it gets expensive to copy every time, so it's best to minimize the number of *nesting calls*.

   There is a small handful of grammatical constructs C++17 supports when working with variadic packs. Such grammar tends to be orders of magnitude faster than simulating those effects directly ourselves:

   - `sizeof...(Vs)`
   - `op(Vs)...` as well as `op⟨Vs⟩...`
   - `array[] = { Vs... }`
   - `(Vs op ...), (... op Vs), (Vs op ... op init), (init op ... op Vs)`
   - Non variadic pack recursion

4. **Nesting Depth Mitigation**:

   The third meta programming bottleneck that requires thoughtful mitigation is that of *nesting depths*. Nesting depths are set by each compiler, and although they can be extended through *compiler options*, there are many reasons to instead design our meta functions to directly mitigate this issue themselves.

   The primary strategy for mitigating nesting depth limits within meta functions is to code them as **single depth loops** when possible. If you have two nesting calls within a single loop iteration, you've effectively cut your nesting depth limit in half for that particular function.

   Derived from this value system (constraint) are the following paradigms:

   - fast tracking (dropping/applying more than one variadic object at a time)
   - zero padding (applied in conjunction with fast tracking; eg. backfilled join)
   - trampolining (split a variadic pack up into parts, each within the nesting limits of the function being applied)
   - double trampolining (general recursion: Build a stack of partially applied functions, then fold the stack)
   - array type sorting (preprocess the list: Sort it into type arrays, keeping record of the original list positions)

5. **psfpae**: Preventing substitution failure prevents an error

   C++17 handles this through the `constexpr-if` construct, but it can also be simulated as an alias template through *lazy dispatching* by means of a **colist** grammatical construct.