Summary of “Function and Class Templates”

Marc Paterno

8 July 2020

Section 1

My summary

When to write a template

- Templates as machines to generate code; use them to avoid writing repeated things.
  - function overload set generated from a single template
  - family of related classes
- Extensibility without the tighter coupling of inheritance, *e.g.*
  - You can make *your* type behave like a Standard Library container by defining the right set of member functions and member types.
  - This will improve with *concepts* in C++20 (Saks did not address this).
- Why not just a macro?
  - Templates allow the compiler to manipulate types in a way macros can not.
  - Macro expansion involves no type checking; template instantiation does.
  - Templates can be used for type computations, *e.g.* `C::value_type` is the type stored in any Standard Library container template `C`.

Instantiation

- Functions get generated from function templates; classes and structs from class and struct templates
- `std::swap` is not the name of a function; `std::swap<int>` is the name of a function.
- `std::vector` is not the name of a class; `std::vector<double>` is the name of a class.
- `std::vector<T>` and `std::swap<T>` are *template-ids*.
- Two-phase parsing is valuable to understand, especially when reasoning about error messages
  - phase 1: when the template is first read. The compiler does *not* know what the template parameters are — they are just names.
  - phase 2: when the *use* of a template is seen by the compiler. The compiler must have seen the template-id before this. The compiler now knows what the template parameters are, and can (attempt to) instantiate the template.

What can template parameters be?

- **types** (this is most common in code I see, and in code I write)
- **non-types** of a few varieties:
  - integral and enumeration values
  - pointer or reference to a class object
  - pointer or reference to a function (*not* a template-id!)
  - pointer or reference to member function

E.g., `template <typename T, std::size_t N> array`

Writing class templates

- Saks recommends writing member functions of class templates in the class template definition, to reduce typing:

```
template <typename T> struct Thing {
  T func() { return do_something(); }
}
```

- I prefer separating declarations from definitions, which I find makes it easier to see the interface of the class:

```
template <typename T> struct Thing {
  T func();
}

template <typename T> T Thing<T>::func() {return do_something();}
```

- For the discussion: what do you prefer and why?

Efficiency issues

- Each template instantiation is done only once per *compilation unit*
    - `.cc` file, after header inclusion is done; what the compiler (not preprocessor) actually sees.
- Possible “code bloat”: given template instantiation can happen in many compilation units; linker strips out duplicates. Remember linker makes `.so` and executables; so many `.so`s can have the same template instantiation.
- Runtime linker behavior is outside of C++ standard specification. Real ones run into trouble with class templates that have static data initialized in multiple dynamic libraries.

Improvements in recent C++ versions

- `inline` data allows *definition* (not just *declaration*) of `static` data members in class templates.
  - but we should not be encouraging non-`const static` data
  - Much uglier before C++17.
- Function template parameter deduction has been around since C++98:

```
int i = 1, j=0;
std::swap(i, j); // compiler deduces std::swap<int>
```

- Class template parameter deduction new in C++17

```
std::vector v {1.0, 2.5}; // compiler deduces std::vector<double>
std::array a {1, 2};      // compiler deduces std::array<int, 2>
```

Useful names to remember

When you're searching the internet for an answer to a template programming question, it is useful to have the meaning several names clear in mind:

- `std::swap` is the name of a (function) template
- `std::swap<T>` is a *template-id*
- `std::swap<int>` is a *template specialization*
- `template void swap<int>(int& a, int& b)` is an *explicit instantiation definition*
- `extern template void swap<int>(int&a, int& b)` is an *explicit instantiation declaration* (means it is defined elsewhere)
- `template <> void swap<int>(int& a, int&b ) { a = 0; b = 0;}` is an *explicit specialization* (don't really use this!)

Section 2

Discussion