```
typedef void(*CompletionCallback)(void* userData);
```

Developers coming from a background other than C or C++03 might find the above declaration hard to parse since the name of the alias, `CompletionCallback`, is embedded in the function pointer type. Replacing **`typedef`** with **`using`** results in a simpler, more consistent formulation of the same alias:

```
using CompletionCallback = void(*)(void* userData);
```

The `CompletionCallback` alias declaration above reads almost completely left-to-right, and the name of the alias is clearly specified after the **`using`** keyword. To make the `CompletionCallback` alias read left-to-right, a trailing return (see Section 1.1."Trailing Return" on page 124) can be used:

```
using CompletionCallback = auto(*)(void* userData) -> void;
```

The alias declaration above can be read as, "`CompletionCallback` is an alias for a pointer to a function taking a **`void*`** parameter named `userData` and returning **`void`**."

Binding arguments to template parameters

An alias template can be used to *bind* one or more template parameters of, say, a commonly used class template, while leaving the other parameters open to variation. Suppose, for example, we have a class, `UserData`, that contains several distinct instances of `std::map`, each having the same key type, `UserId`, but with different payloads:

```
class UserData  // class having excessive code repetition (BAD IDEA)
{
private:
    std::map<UserId, Message>          d_messages;
    std::map<UserId, Photos>           d_photos;
    std::map<UserId, Article>          d_articles;
    std::map<UserId, std::set<UserId>> d_friends;
};
```

The example above, though clear and regular, involves significant repetition, making it more difficult to maintain should we later opt to change data structures. If we were to instead use an **alias template** to bind the `UserId` type to the first type parameter of `std::map`, we could both reduce code repetition and enable the programmer to consistently replace `std::map` ~~to~~ another container, e.g., `std::unordered_map`,[1] by performing the change in only one place:

[1] An `std::unordered_map` is an STL container type that became available on all conforming platforms along with C++11. The functionality is similar except that since it is not required to support ordered traversal or, worst case, `O[log(n)]` lookups and `O[n*log(n)]` insertions, `std::unordered_map` can be implemented as a hash table instead of a balanced tree, yielding significantly faster average access times. See **cpprefb**.