Note, however, that neither partial nor **full specialization** of alias templates is supported:

```
template <typename, typename>   // general alias template
using Type9 = char;             // OK

template <typename T>           // attempted partial specialization of above
using Type9<T, int> = char;     // Error, expected = before < token

template <>                     // attempted full specialization of above
using Type10<int, int> = char;  // Error, expected unqualified id before using
```

Used in conjunction with existing **class templates**, alias templates allow programmers to *bind* one or more **template parameters** to a fixed type, while leaving others open:

```
#include <utility>  // std::pair

template <typename T>
using PairOfCharAnd = std::pair<char, T>;
    // alias template that binds char to the first type parameter of std::pair

PairOfCharAnd<int>    pci;  // equivalent to std::pair<char, int> pci;
PairOfCharAnd<double> pcd;  // equivalent to std::pair<char, double> pcd;
```

Finally, note that similar functionality can be achieved in C++03; it suppresses **type deduction** and requires additional **boilerplate code** at both the point of **definition** and the call site:

```
// C++03 (obsolete)
template <typename T>
struct PairOfCharAnd
    // template class holding an alias, Type, to std::pair<char, T>
{
    typedef std::pair<char, T> Type;
        // type alias binding char to the first type parameter of std::pair
};

PairOfCharAnd<int>::Type    pci;  // equivalent to std::pair<char, int> pci;
PairOfCharAnd<double>::Type pcd;  // equivalent to std::pair<char, double> pcd;
```

Use Cases

Simplifying convoluted `typedef` declarations

Complex **`typedef`** declarations involving pointers to functions, **member functions**, or data members require looking in the middle of the declaration to find the alias name. As an example, consider a *callback* type alias intended to be used with asynchronous functions: