```
class UserData  // class with well-factored implementation (GOOD IDEA)
{
private:
    template <typename V>                  // using a template alias to bind
    using Mapping = std::map<UserId, V>;   // UserId as the key type

    Mapping<Message>          d_messages;
    Mapping<Photos>           d_photos;
    Mapping<Article>          d_articles;
    Mapping<std::set<UserId>> d_friends;
};
```

Providing a shorthand notation for type traits

Alias templates can provide a shorthand notation for **type traits**, avoiding boilerplate code in the usage site. As an example, consider a simple type trait that adds a pointer to a given type (similar to `std::add_pointer`):

```
template <typename T>
struct AddPointer
{
    typedef T* Type;
};
```

To use the trait above, the `AddPointer` class template must be instantiated, and its nested `Type` alias must be accessed. Furthermore, in the **generic** context, it has to be prepended with the **typename** keyword::

```
template <typename T>void f()
{
    T t;
    typename AddPointer<T>::Type p = t;
}
```

The syntactical overhead of `AddPointer` can be removed by creating an alias template for its nested type alias, such as `AddPointer_t`:

```
template <typename T>
using AddPointer_t = typename AddPointer<T>::Type;
```

Using `AddPointer_t` instead of `AddPointer` results in shorter code devoid of boilerplate:

```
void g()
{
    int i;
    AddPointer_t<int> p = &i;
}
```