C++ Mapping for Dictionaries

On this page:

Default Dictionary Mapping in C++

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

C++

```
typedef std::map<Ice::Long, Employee> EmployeeMap;
```

Again, there are no surprises here: a Slice dictionary simply maps to a standard `std::map`. As a result, you can use the dictionary like any other `map`, for example:

C++

```
EmployeeMap em;
Employee e;

e.number = 42;
e.firstName = "Stan";
e.lastName = "Lippman";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

Custom Dictionary Mapping in C++

You can override the default mapping of Slice dictionaries to C++ maps with a metadata directive, for example:

Slice

```
[["cpp:include:unordered_map"]]

["cpp:type:std::unordered_map<Ice::Long, Employee>"] dictionary<long, Employee> EmployeeMap;
```

With this metadata directive, the dictionary now maps to a C++ `std::unordered_map`:

C++

```
#include <unordered_map>

typedef std::unordered_map<Ice::Long, Employee> EmployeeMap;

}
```

The `cpp:type` metadata directive can be applied to a sequence or dictionary definition; anything following the `cpp:type:` prefix is taken to be the name of the type. For example, we could use `["cpp:type:::std::unordered_map<int, std::string>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++

```
typedef ::std::unordered_map<int, std::string> IntStringDict;
```

To avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <unordered_map>
```

to the generated header file.

Instead of `std::unordered_map`, you can specify a type of your own as the dictionary type, for example:

Slice

```
[["cpp:include:CustomMap.h"]]

["cpp:type:MyCustomMap<Ice::Long, Employee>"] dictionary<long, Employee> EmployeeMap;
```

With these metadata directives, the compiler will use a C++ type `MyCustomMap` as the dictionary type, and add an include directive for the header file `CustomMap.h` to the generated code.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.
- The class must provide nested types named `key_type`, `mapped_type` and `value_type`.
- The class must provide `iterator` and `const_iterator` types and provides `begin` and `end` member functions with the usual semantics; these iterators must be comparable for equality and inequality.
- The class must provide a `clear` function.
- The class must provide an `insert` function that takes an `iterator` (as location hint) plus a `value_type` parameter, and returns an `iterator` to the new entry or to the existing entry with the given key.

Less formally, this means you can use any class or template class that looks like a standard `map` or `unordered_map` as your custom dictionary type.

In addition to modifying the type of a dictionary itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:unordered_map"]]

module HR {

    struct Employee {
       long   number;
       string firstName;
       string lastName;
    };
    dictionary<long, Employee> EmployeeMap;

    interface Office {
        ["cpp:type:std::unordered_map<Ice::Long, Employee>"] EmployeeMap getAllEmployees();
    };
};
```

With this definition, `getAllEmployees` returns an `unordered_map`, while other unqualified parameters of type `EmployeeMap` would use the default mapping (to a `std::map`).

See Also

- Dictionaries
- C++ Mapping for Enumerations
- C++ Mapping for Structures
- C++ Mapping for Sequences