Using a Freeze Map in C++

This page describes the C++ code generator and demonstrates how to use a Freeze map in a C++ program.

On this page:

`slice2freeze` Command-Line Options

The Slice-to-Freeze compiler, `slice2freeze`, creates C++ classes for Freeze maps. The compiler offers the following command-line options in addition to the standard options:

`--header-ext` *`EXT`*

Changes the file extension for the generated header files from the default h to the extension specified by *`EXT`*.

`--source-ext` *`EXT`*

Changes the file extension for the generated source files from the default `cpp` to the extension specified by *`EXT`*.

`--add-header` *`HDR[,GUARD]`*

This option adds an include directive for the specified header at the beginning of the generated source file (preceding any other include directives). If *`GUARD`* is specified, the include directive is protected by the specified guard.

For example:

```
--add-header precompiled.h,_PRECOMPILED_H__
```

results in the following directives at the beginning of the generated source file:

C++

```
#ifndef __PRECOMPILED_H__
#define __PRECOMPILED_H__
#include <precompiled.h>
#endif
```

As this example demonstrates, the `--add-header` option is useful mainly for integrating the generated code with a compiler's precompiled header mechanism.

This option can be repeated to create include directives for several files.

`--include-dir` *`DIR`*

Modifies `#include` directives in source files to prepend the path name of each header file with the directory *`DIR`*. The discussion of `slice2cpp` provides more information.

`--dll-export` *`SYMBOL`*

Use *`SYMBOL`* to control DLL exports or imports. See the `slice2cpp` description for details.

`--dict` *`NAME,KEY,VALUE`*`[,sort[,`*`COMPARE`*`]]`

Generate a Freeze map class named *NAME* using *KEY* as key and *VALUE* as value. This option may be specified multiple times to generate several Freeze maps. *NAME* may be a scoped C++ name, such as `Demo::Struct1ObjectMap`. *KEY* and *VALUE* represent Slice types and therefore must use Slice syntax, such as `bool` or `Ice::Identity`. By default, keys are sorted using their binary Ice-encoded representation. Include `sort` to sort with the *COMPARE* functor class. If *COMPARE* is not specified, the default value is `std::less<`*KEY*`>`.

`--dict-index MAP[,`*MEMBER*`] [,case-sensitive|case-insensitive][,sort[,`*COMPARE*`]]`

Add an index to the Freeze map named *MAP*. If *MEMBER* is specified, the map value type must be a structure or a class, and *MEMBER* must be a member of this structure or class. Otherwise, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive. An index adds additional member functions to the generated C++ map:

- `iterator findBy`*MEMBER*`(`*MEMBER_TYPE*`, bool = true);`
- `const_iterator findBy`*MEMBER*`(`*MEMBER_TYPE*`,bool = true) const;`
- `iterator beginFor`*MEMBER*`();`
- `const_iterator begin_For`*MEMBER*`() const;`
- `iterator endFor`*MEMBER*`();`
- `const_iterator endFor`*MEMBER*`() const;`
- `iterator lowerBoundFor`*MEMBER*`(`*MEMBER_TYPE*`);`
- `const_iterator lowerBoundFor`*MEMBER*`(`*MEMBER_TYPE*`) const;`
- `iterator upperBoundFor`*MEMBER*`(`*MEMBER_TYPE*`);`
- `const_iterator upperBoundFor`*MEMBER*`(`*MEMBER_TYPE*`) const;`
- `std::pair<iterator, iterator> equalRangeFor`*MEMBER*`(`*MEMBER_TYPE*`);`
- `std::pair<const_iterator, const_iterator> equalRangeFor`*MEMBER*`(`*MEMBER_TYPE*`) const;`
- `int `*MEMBER*`Count(`*MEMBER_TYPE*`) const;`

When *MEMBER* is not specified, these functions are `findByValue` (const and non-const), `lowerBoundForValue` (const and non-const), `valueCount`, and so on. When *MEMBER* is specified, its first letter is capitalized in the `findBy` function name. *MEMBER_TYPE* corresponds to an in-parameter of the type of *MEMBER* (or the type of the value when *MEMBER* is not specified). For example, if *MEMBER* is a string, *MEMBER_TYPE* is a `const std::string&`.

By default, keys are sorted using their binary Ice-encoded representation. Include `sort` to sort with the *COMPARE* functor class. If *COMPARE* is not specified, the default value is `std::less<`*MEMBER_TYPE*`>`.

`findBy`*MEMBER* returns an iterator to the first element in the Freeze map that matches the given index value. It returns `end()` if there is no match. When the second parameter is true (the default), the returned iterator provides only the elements with an exact match (and then skips to `end()`). Otherwise, the returned iterator sets a starting position and then provides all elements until the end of the map, sorted according to the index comparator.

`lowerBoundFor`*MEMBER* returns an iterator to the first element in the Freeze map whose index value is not less than the given index value. It returns `end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`upperBoundFor`*MEMBER* returns an iterator to the first element in the Freeze map whose index value is greater than the given index value. It returns `end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`beginFor`*MEMBER* returns an iterator to the first element in the map.

`endFor`*MEMBER* returns an iterator to the last element in the map.

`equalRangeFor`*MEMBER* returns a range (pair of iterators) of all the elements whose index value matches the given index value. This function is similar to `findBy`MEMBER (see above).

MEMBER`Count` returns the number of elements in the Freeze map whose index value matches the given index value.

Please note that index-derived iterators do not allow you to set new values in the underlying map.

`--index `*CLASS*`,`*TYPE*`,`*MEMBER*` [,case-sensitive|case-insensitive]`

Generate an index class for a Freeze evictor. *CLASS* is the name of the class to be generated. *TYPE* denotes the type of class to be indexed (objects of different classes are not included in this index). *MEMBER* is the name of the data member in *TYPE* to index. When *MEMBER* has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

Generating a Simple Map for C++

As an example, the following command generates a simple map:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The final argument is the base name for the output files, to which the compiler appends the `.h` and `.cpp` suffixes. As a result, this command produces two C++ source files, `StringIntMap.h` and `StringIntMap.cpp`.

The Freeze Map Class in C++

If you examine the contents of the header file created by the example in the previous section, you will discover that a Freeze map is an instance of the template class `Freeze::Map`:

C++

```
// StringIntMap.h
typedef Freeze::Map<std::string, Ice::Int, ...> StringIntMap;
```

The `Freeze::Map` template class closely resembles the STL container class `std::map`, as shown in the following class definition:

C++

```
namespace Freeze {
template<...> class Map {
public:
    typedef ... value_type;
    typedef ... iterator;
    typedef ... const_iterator;

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    Map(const Freeze::ConnectionPtr& connection,
        const std::string& dbName,
        bool createDb = true,
        const Compare& compare = Compare());

    template<class _InputIterator>
    Map(const Freeze::ConnectionPtr& connection,
        const std::string& dbName,
        bool createDb,
        _InputIterator first, _InputIterator last,
        const Compare& compare = Compare());

    static void recreate(const Freeze::ConnectionPtr& connection,
                         const std::string& dbName,
                         const Compare& compare = Compare());

    bool operator==(const Map& rhs) const;
    bool operator!=(const Map& rhs) const;

    void swap(Map& rhs);

    iterator begin();
    const_iterator begin() const;

    iterator end();
    const_iterator end() const;

    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    iterator insert(iterator /*position*/, const value_type& elem);

    std::pair<iterator, bool> insert(const value_type& elem);
```

```
    template <typename InputIterator>
    void insert(InputIterator first, InputIterator last);

    void put(const value_type& elem);

    template <typename InputIterator>
    void put(InputIterator first, InputIterator last);

    void erase(iterator position);
    size_type erase(const key_type& key);
    void erase(iterator first, iterator last);

    void clear();

    void destroy(); // Non-standard.

    iterator find(const key_type& key);
    const_iterator find(const key_type& key) const;

    size_type count(const key_type& key) const;

    iterator lower_bound(const key_type& key);
    const_iterator lower_bound(const key_type& key) const;
    iterator upper_bound(const key_type& key);
    const_iterator upper_bound(const key_type& key) const;

    std::pair<iterator, iterator>
    equal_range(const key_type& key);

    std::pair<const_iterator, const_iterator>
    equal_range(const key_type& key) const;

    const Ice::CommunicatorPtr&
    communicator() const;

    ...
};
}
```

The semantics of the `Freeze::Map` methods are identical to those of `std::map` unless otherwise noted. In particular, the overloaded `insert` method shown below ignores the `position` argument:

C++

```
iterator insert(iterator /*position*/, const value_type& elem);
```

A Freeze map class supports only those methods shown above; other features of `std::map`, such as allocators and overloaded array operators, are not available.

Non-standard methods that are specific to Freeze maps are discussed below:

- Constructors
  The following overloaded constructors are provided:

C++

```
Map(const Freeze::ConnectionPtr& connection,
    const std::string& dbName,
    bool createDb = true,
    const Compare& compare = Compare());

template<class _InputIterator>
Map(const Freeze::ConnectionPtr& connection,
    const std::string& dbName,
    bool createDb,
    _InputIterator first, _InputIterator last,
    const Compare& compare = Compare());
```

The first constructor accepts a connection, the database name, a flag indicating whether to create the database if it does not exist, and an object used to compare keys. The second constructor accepts all of the parameters of the first, with the addition of iterators from which elements are copied into the map.

Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

- Map copy
  The `recreate` function copies an existing database:

  C++

  ```
  static void recreate(const Freeze::ConnectionPtr& connection,
                       const std::string& dbName,
                       const Compare& compare = Compare())
  ```

  The `dbName` parameter specifies an existing database name. The copy has the name `<`*`dbName`*`>.old-<`*`uuid`*`>`. For example, if the database name is `MyDB`, the copy might be named `MyDB.old-edefd55a-e66a-478d-a77b-f6d53292b873`. (Obviously, a different UUID is used each time you recreate a database).

- `destroy`
  This method deletes the database from its environment and from the Freeze catalog. If a transaction is not currently open, the method creates its own transaction in which to perform this task.

- `communicator`
  This method returns the communicator with which the map's connection is associated.

Using Iterators with Freeze Maps in C++

A Freeze map's iterator works like its counterpart in `std::map`. The iterator class supports one convenient (but nonstandard) method:

C++

```
void set(const mapped_type& value)
```

Using this method, a program can replace the value at the iterator's current position.

Sample Freeze Map Program in C++

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

C++

```
#include <Freeze/Freeze.h>
#include <StringIntMap.h>

int
main(int argc, char* argv[])
{
    // Initialize the Communicator.
    //
    Ice::CommunicatorPtr communicator = Ice::initialize(argc, argv);

    // Create a Freeze database connection.
    //
    Freeze::ConnectionPtr connection = Freeze::createConnection(communicator, "db");

    // Instantiate the map.
    //
    StringIntMap map(connection, "simple");

    // Clear the map.
    //
    map.clear();

    Ice::Int i;
    StringIntMap::iterator p;

    // Populate the map.
    //
    for (i = 0; i < 26; i++) {
        std::string key(1, 'a' + i);
        map.insert(make_pair(key, i));
    }

    // Iterate over the map and change the values.
    //
    for (p = map.begin(); p != map.end(); ++p)
        p.set(p->second + 1);

    // Find and erase the last element.
    //
    p = map.find("z");
    assert(p != map.end());
    map.erase(p);

    // Clean up.
    //
    connection->close();
    communicator->destroy();

    return 0;
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

C++

```
Freeze::ConnectionPtr connection = Freeze::createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files. Note that properties with the prefix `Freeze.DbEnv` can modify a number of environment settings, including the file system directory. For the preceding example, you could change the directory to `FreezeDir` by setting the property `Freeze.DbEnv.db.DbHome` to `FreezeDir`.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, which by default is created if it does not exist:

C++

```
StringIntMap map(connection, "simple");
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

C++

```
map.clear();
```

Next, we populate the map using a single-character string as the key:

C++

```
for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}
```

Iterating over the map will look familiar to `std::map` users. However, to modify a value at the iterator's current position, we use the nonstandard `set` method:

C++

```
for (p = map.begin(); p != map.end(); ++p)
    p.set(p->second + 1);
```

Next, the program obtains an iterator positioned at the element with key `z`, and erases it:

C++

```
p = map.find("z");
assert(p != map.end());
map.erase(p);
```

Finally, the program closes the database connection:

C++

```
connection->close();
```

It is not necessary to explicitly close the database connection, but we demonstrate it here for the sake of completeness.

See Also

- Using the Slice Compilers
- slice2cpp Command-Line Options
- Freeze Evictor Concepts
- Freeze Catalogs
- Freeze Properties