slice2cpp Command-Line Options

On this page:

`slice2cpp` Command-Line Options

The Slice-to-C++ compiler, `slice2cpp`, 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*.

You can also change the header file extension with a global metadata directive:

Slice

```
[["cpp:header-ext:hpp"]]

// ...
```

Only one such directive can appear in each source file. If you specify a header extension on both the command line and with a metadata directive, the metadata directive takes precedence. This ensures that included Slice files that were compiled separately get the correct header extension (provided that the included Slice files contain a corresponding metadata directive). For example:

Slice

```
// File example.ice
#include <Ice/BuiltinSequences.ice>

// ...
```

Compiling this file with

```
$ slice2cpp --header-ext=hpp -I/opt/Ice/include example.ice
```

generates `example.hpp`, but the `#include` directive in that file is for `Ice/BuiltinSequences.h` (not `Ice/BuiltinSequences.hpp`) because `BuiltinSequences.ice` contains the metadata directive `[["cpp:header-ext:h"]]`.

You normally will not need to use this metadata directive. The directive is necessary only if:

- You `#include` a Slice file in one of your own Slice files.
- The included Slice file is part of a library you link against.
- The library ships with the included Slice file's header.
- The library header uses a different header extension than your own code.

For example, if the library uses `.hpp` as the header extension, but your own code uses `.h`, the library's Slice file should contain a `[["cpp:header?ext:hpp"]]` directive. (If the directive is missing, you can add it to the library's Slice file.)

`--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
```

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

As suggested by the preceding example, this option is useful mainly to integrate the generated code with a compiler's precompiled header mechanism.

`--include-dir` *DIR*

Modifies `#include` directives in source files to prepend the path name of each header file with the directory *DIR*.

`--impl`

Generate sample implementation files. This option will not overwrite an existing file.

`--depend`

Prints makefile dependency information to standard output. No code is generated when this option is specified. The output generally needs to be filtered before it can be included in a makefile; the Ice build system uses the script `config/makedepend.py` for this purpose.

`--dll-export` *SYMBOL*

Use *SYMBOL* to control DLL exports or imports. This option allows you to selectively export or import global symbols in the generated code. As an example, compiling a Slice definition with:

```
$ slice2cpp --dll-export ENABLE_DLL x.ice
```

results in the following additional code being generated into `x.h`:

C++

```
#ifndef ENABLE_DLL
#   ifdef ENABLE_DLL_EXPORTS
#       define ENABLE_DLL ICE_DECLSPEC_EXPORT
#   else
#       define ENABLE_DLL ICE_DECLSPEC_IMPORT
#   endif
#endif
```

`ICE_DECLSPEC_EXPORT` and `ICE_DECLSPEC_IMPORT` are platform-specific macros. For example, for Windows, they are defined as `declspec (dllexport)` and `declspec(dllimport)`, respectively; for Solaris using `CC` version 5.5 or later, `ICE_DECLSPEC_EXPORT` is defined as `global`, and `ICE_DECLSPEC_IMPORT` is empty.

Similar definitions exist for other platforms. For platforms that do not have any concept of explicit export or import of shared library symbols, both macros are empty.

The symbol name you specify on the command line (`ENABLE_DLL` in this example) is used by the generated code to export or import any symbols that must be visible to code outside the generated compilation unit. The net effect is that, if you want to create a DLL that includes `x.cpp`, but also want to use the generated types in compilation units outside the DLL, you can arrange for the relevant symbols to be exported by compiling `x.cpp` with `-DENABLE_DLL_EXPORTS`.

`--checksum`

Generate checksums for Slice definitions.

`--stream`

Generate streaming helper functions for Slice types.

Include Directives

he `#include` directives generated by the Slice-to-C++ compiler can be a source of confusion if the semantics governing their generation are not well-understood. The generation of `#include` directives is influenced by the command-line options `-I` and `--include-dir`; these options are discussed in more detail below. The `--output-dir` option directs the translator to place all generated files in a particular directory, but has no impact on the contents of the generated code.

Given that the `#include` directives in header files and source files are generated using different semantics, we describe them in separate sections.

Header Files

In most cases, the compiler generates the appropriate `#include` directives by default. As an example, suppose file `A.ice` includes `B.ice` using the following statement:

Slice

```
// A.ice
#include <B.ice>
```

Assuming both files are in the current working directory, we run the compiler as shown below:

```
$ slice2cpp -I. A.ice
```

The generated file `A.h` contains this `#include` directive:

C++

```
// A.h
#include <B.h>
```

If the proper include paths are specified to the C++ compiler, everything should compile correctly.

Similarly, consider the common case where `A.ice` includes `B.ice` from a subdirectory:

Slice

```
// A.ice
#include <inc/B.ice>
```

Assuming both files are in the `inc` subdirectory, we run the compiler as shown below:

```
$ slice2cpp -I. inc/A.ice
```

The default output of the compiler produces this `#include` directive in `A.h`:

C++

```
// A.h
#include <inc/B.h>
```

Again, it is the user's responsibility to ensure that the C++ compiler is configured to find `inc/B.h` during compilation.

Now let us consider a more complex example, in which we do not want the `#include` directive in the header file to match that of the Slice file. This can be necessary when the organizational structure of the Slice files does not match the application's C++ code. In such a case, the user may need to relocate the generated files from the directory in which they were created, and the `#include` directives must be aligned with the new structure.

For example, let us assume that `B.ice` is located in the subdirectory `slice/inc`:

Slice

```
// A.ice
#include <slice/inc/B.ice>
```

However, we do not want the `slice` subdirectory to appear in the `#include` directive generated in the header file, therefore we specify the additional compiler option `-Islice`:

```
$ slice2cpp -I. -Islice slice/inc/A.ice
```

The generated code demonstrates the impact of this extra option:

C++

```
// A.h
#include <inc/B.h>
```

As you can see, the `#include` directives generated in header files are affected by the include paths that you specify when running the compiler. Specifically, the include paths are used to abbreviate the path name in generated `#include` directives.

When translating an `#include` directive from a Slice file to a header file, the compiler compares each of the include paths against the path of the included file. If an include path matches the leading portion of the included file, the compiler removes that leading portion when generating the `#include` directive in the header file. If more than one include path matches, the compiler selects the one that results in the shortest path for the included file.

For example, suppose we had used the following options when compiling `A.ice`:

```
$ slice2cpp -I. -Islice -Islice/inc slice/inc/A.ice
```

In this case, the compiler compares all of the include paths against the included file `slice/inc/B.ice` and generates the following directive:

C++

```
// A.h
#include <B.h>
```

The option `-Islice/inc` produces the shortest result, therefore the default path for the included file (`slice/inc/B.h`) is replaced with `B.h`.

In general, the `-I` option plays two roles: it enables the preprocessor to locate included Slice files, and it provides you with a certain amount of control over the generated `#include` directives. In the last example above, the preprocessor locates `slice/inc/B.ice` using the include path specified by the `-I.` option. The remaining `-I` options do not help the preprocessor locate included files; they are simply hints to the compiler.

Finally, we recommend using caution when specifying include paths. If the preprocessor is able to locate an included file via multiple include paths, it always uses the first include path that successfully locates the file. If you intend to modify the generated `#include` directives by specifying extra `-I` options, you must ensure that your include path hints match the include path selected by the preprocessor to locate the included file. As a general rule, you should avoid specifying include paths that enable the preprocessor to locate a file in multiple ways.

Source Files

By default, the compiler generates `#include` directives in source files using only the base name of the included file. This behavior is usually appropriate when the source file and header file reside in the same directory.

For example, suppose `A.ice` includes `B.ice` from a subdirectory, as shown in the following snippet of `A.ice`:

Slice

```
// A.ice
#include <inc/B.ice>
```

We generate the source file using this command:

```
$ slice2cpp -I. inc/A.ice
```

Upon examination, we see that the source file contains the following `#include` directive:

C++

```
// A.cpp
#include <B.h>
```

However, suppose that we wish to enforce a particular standard for generated `#include` directives so that they are compatible with our C++ compiler's existing include path settings. In this case, we use the `--include-dir` option to modify the generated code. For example, consider the compiler command shown below:

```
$ slice2cpp --include-dir src -I. inc/A.ice
```

The source file now contains the following `#include` directive:

C++

```
// A.cpp
#include <src/B.h>
```

Any leading path in the included file is discarded as usual, and the value of the `--include-dir` option is prepended.

See Also