Introduction to C++ Complementary Library

CCL

Luxury for Developers

Covers CCL version 0.1 Beta
1st Edition
Written by Daniel T. McGinnis
Published on December 9th, 2022

Table of Contents

- Chapter 1: Getting Started
  - Library Organization
  - Stability Promises
  - Consuming CCL
    - Using CCL without CMake (in Visual Studio, Xcode, Code::Blocks, etc.)
    - Using CCL in a CMake Project
  - Configuration Constants
    - CMP_CONFIG_LTO_REQUESTED
    - CMP_CONFIG_LTO_ENABLED
    - CMP_CONFIG_HEADER_ONLY
    - CMP_CONFIG_DEFAULT_BUFFER_CAPACITY
    - CMP_CONFIG_DEFAULT_STDOUT_BUFFER_CAPACITY
    - CMP_CONFIG_DEFAULT_STDERR_BUFFER_CAPACITY
    - CMP_CONFIG_DEFAULT_STDIN_BUFFER_CAPACITY
    - CMP_CONFIG_IO_PACKAGE_EXCLUDED
- Chapter 2: Unicode
  - Reviewing ASCII
  - Say Hello to Unicode
  - Unicode Terminology
  - Unicode in the Standard Library
  - Unicode in CCL
    - Reading Unicode Strings from Files
    - Writing Unicode Strings to Files
    - Printing Unicode Strings to Standard Output
    - Reading Unicode Strings from Standard Input
    - Iterating Over Unicode Strings
    - Converting Unicode Strings
  - Endianness
  - Handling Endianness in CCL
  - A Note About Source File Encoding
- Chapter 3: I/O
  - Introduction
    - Transfer Resources
    - Transfer Streams
    - How They Fit Together
  - Files
  - Data Input Streams
  - Data Output Streams
  - String I/O Resources
  - Text Input Streams
  - Text Output Streams
  - Wrapping Up

Chapter 1: Getting Started

Welcome to CCL. CCL is a native C++ library for developing high-performance cross-platform applications. This chapter will get you started with CCL and discuss its essential terminology and practices. While this book doesn't explore every nook and cranny of CCL, reading this chapter alone will give you a firm grasp of the fundamentals of CCL. This chapter gives an overview of the library, and later chapters explore how to apply CCL to specific kinds of problems, like Unicode and file I/O.

Chapter Table of Contents

- Library Organization
- Stability Promises
- Consuming CCL
  - Using CCL without CMake (in Visual Studio, Xcode, Code::Blocks, etc.)
  - Using CCL in a CMake Project
- Configuration Constants
  - CMP_CONFIG_LTO_REQUESTED
  - CMP_CONFIG_LTO_ENABLED
  - CMP_CONFIG_HEADER_ONLY
  - CMP_CONFIG_DEFAULT_BUFFER_CAPACITY
  - CMP_CONFIG_DEFAULT_STDOUT_BUFFER_CAPACITY
  - CMP_CONFIG_DEFAULT_STDERR_BUFFER_CAPACITY
  - CMP_CONFIG_DEFAULT_STDIN_BUFFER_CAPACITY
  - CMP_CONFIG_IO_PACKAGE_EXCLUDED

Library Organization

CCL is a **library**. It consists of several **packages**. Different software projects use different terminology to distinguish between their various parts. Boost, for example, is a set of C++ libraries, and each library is useful for a particular kind of task. For example, Boost.Container or Boost.Asio. In this case, the "parts" of Boost are called "libraries". Qt is a framework and its parts are called "modules". For example, Qt Core, Qt Network, etc. CCL is a library, and its parts are called "packages", and each package is useful for a particular kind of task. These packages are listed below.

- CCL Core
- CCL IO
- CCL Unicode

Each package is discussed in detail in the following chapters. For now, know that CCL Core is the only package that every other package depends on, CCL IO handles tasks related to I/O (like reading and writing files), and CCL Unicode handles tasks related to Unicode (like iterating Unicode strings or converting between different encoding forms).

It is important to note that some packages can be implemented purely in portable C++, whereas others require the operating system's native API to get the job done. The supported operating systems are Windows, macOS and GNU/Linux. The term "supported operating systems" here means operating systems that CCL is specifically designed for, compiled on, and tested on before every release. OS-dependent packages can be disabled with a compile-time flag if your particular operating system is not supported or if the functionality in the package is not needed. Currently, the only OS-dependent package is CCL IO.

Stability Promises

CCL does not guarantee backwards compatibility. This means that if you have an existing application that uses CCL and you want to upgrade to a newer version of CCL when it comes out, then you may have to change your source code but you will have to at least recompile your application. This makes upgrading to newer versions of CCL less practical but it makes it easier to improve CCL as time goes forward. Source-breaking changes will always be highlighted in the release notes, along with suggestions on how to do the upgrade.

Consuming CCL

When you use CCL, you can use all packages or exclude OS-dependent packages, but in either case, CCL is used as one library. You cannot use CCL packages individually like you can with Boost libraries.

CCL can be used header-only or compiled as a static or dynamic library. The advantage of compiling CCL and linking it with your code is that you cut down on compile times because you don't recompile CCL code every time you compile your own code. The advantage of using CCL header-only is that you get the guarantee that you literally only compile and ship the code that is needed by your application and nothing else, so if you use one class from CCL, your binary will only include the code for that one class and whatever else that one class depends on, but you won't necessarily pay for the cost of the entire library.

The following sections explain how to consume CCL based on what approach you take to build your code. We look at how to integrate CCL into a CMake-based project and how to use CCL in popular IDEs like Visual Studio or Xcode.

Using CCL without CMake (in Visual Studio, Xcode, Code::Blocks, etc.)

CMake is really a great build tool, but there are a lot of software projects out there that are tied to a particular IDE, and there are several reasons why this is common, so this section goes over how to consume CCL in IDEs without using CMake.

After downloading CCL, copy the CCL directory to your location of choosing and then go to your IDE's settings dialog and add the following directories to your include paths:

- <CCL root>/api/include
- <CCL root>/api/src

Then you have to choose whether to use CCL header-only or compile it as a static or dynamic library.

To use it header-only, make sure that the preprocessor constant **`CMP_CONFIG_HEADER_ONLY`** is defined as **`true`** before including any CCL header files in your code. This can be done more easily by using your IDE's settings dialog to create a global preprocessor constant. Then just include CCL headers in your code and build your project the way you normally would in the IDE you're using.

To use CCL as a static or dynamic library, don't define **`CMP_CONFIG_HEADER_ONLY`** at all, or define it as **`false`**, and then actually compile CCL with CMake and link the compiled binary with your project using your IDE's settings dialog. Then just include the headers and build your project normally through your IDE.

The reason you have to add the **src** directory as an include path is because every CCL header knows whether it's being used header-only, and if it is then it pulls in the corresponding source file. This setup allows the interface (which is in the header files) to be physically separate from the implementation (which is in the source files) and still allow users of the library to choose whether to compile it or use it header-only.

To compile CCL, simply build it like any other CMake library:

```
cd ccl
mkdir build
cd build
cmake ..
cmake --build . --config Release
ctest
```

This will produce a dynamic library. If you wish to use CCL as a static library then add the **-DCMP_BUILD_STATIC=true** flag to the CMake configuration command. Like this:

```
cmake .. -DCMP_BUILD_STATIC=true
```

There are several other configuration constants besides **CMP_CONFIG_HEADER_ONLY** that you can define to customize CCL, which we will discuss later in this chapter. These are simply preprocessor constants that begin with **CMP_CONFIG**. Make sure that whatever configuration constants you define in your IDE's settings dialog are spelled the same in the CMake configuration command. For example, if you define **CMP_CONFIG_IO_PACKAGE_EXCLUDED** to be **true** in your IDE's settings dialog, and wish to compile CCL and link it with your project, then make sure to include **-DCMP_CONFIG_IO_PACKAGE_EXCLUDED=true** in the configuration step of CMake. Ultimately that would look like this:

```
cmake .. -DCMP_CONFIG_IO_PACKAGE_EXCLUDED=true
```

This of course also works with static builds, which for completeness is illustrated below:

```
cmake .. -DCMP_BUILD_STATIC=true -DCMP_CONFIG_IO_PACKAGE_EXCLUDED=true
```

Of course, feel free to specify a generator with the **-G** flag, if you wish. CCL is known to work with the latest versions of Visual C++, GCC and Clang.

The unit tests are executed before every version of CCL is released, but it is a good idea to run them after you build the library just to make sure that everything is working as expected. This is done with the **ctest** command. After that, CCL is built and ready to be linked with your project.

Using CCL in a CMake Project

This sections goes over how to consume CCL within a CMake project.

CMake is very flexible in that it lets you define the directory structure of your project. Let's look at an example application that comes with CCL. The root directory of CCL has a subdirectory called **examples** where you can find the **unicode_stdio_example** directory, which contains a simple CMake-based application that uses CCL to print Unicode text to standard output and read Unicode text from standard input. The CMakeLists.txt file for that application is shown below:

```
# Copyright (C) 2022 Daniel T. McGinnis
# SPDX-License-Identifier: BSL-1.0

cmake_minimum_required(VERSION 3.17)
project(ccl CXX)

add_subdirectory(../../api cmp_build)

add_executable(
    unicode_stdio_example
    src/main.cpp
)
set_target_properties(
    unicode_stdio_example PROPERTIES
        LINKER_LANGUAGE CXX
        CXX_STANDARD 20
        CXX_STANDARD_REQUIRED TRUE
        CXX_EXTENSIONS OFF
)
target_link_libraries(
    unicode_stdio_example
    cmp
)
```

Let's go over this code piece by piece. The instructions here will assume you are starting a brand new CMake project, and that you are copying the code above and pasting it into your own CMakeLists.txt file, and then tweaking it for your own needs. Of course, if you do copy the code above, be sure to remove the copyright notice as that would imply that I am the owner of your CMakeLists.txt file, which of course is not correct. Also, the second comment at the top specifies that this file is licensed under the Boost Software License 1.0, feel free to remove that comment as well.

```
cmake_minimum_required(VERSION 3.17)
```

This line of code simply states that the minimum version of CMake required to build this application is version 3.17. Requiring version 3.17 is not a law, your own CMake project may require a different minimum version.

```
project(ccl CXX)
```

This line of code says that **`ccl`** is a C++ project. Change **`ccl`** to the name of your own CMake project.

```
add_subdirectory(../../api cmp_build)
```

This line of code pulls in the **`cmp`** library target defined in the CMakeLists.txt file of the **api** directory, which your application's executable target will later link with. Change this line of code to match where you put CCL. For example, if you put CCL in the same directory as your CMakeLists.txt file, then this line should read **`add_subdirectory(ccl/api cmp_build)`**.

```
add_executable(
    unicode_stdio_example
    src/main.cpp
)
```

This piece of code adds an executable target called **`unicode_stdio_example`**. Change this to the name you desire to use for your own application's executable target. Additionally, we are listing all the source files of this target, which is just the **`main.cpp`** file in the **`src`** directory. Again, change this to your heart's content. Add all the other source files your project uses to this list, if any. I do recommend that you put your source files in the **`src`** directory but that is optional as well.

```
set_target_properties(
    unicode_stdio_example PROPERTIES
        LINKER_LANGUAGE CXX
        CXX_STANDARD 20
        CXX_STANDARD_REQUIRED TRUE
        CXX_EXTENSIONS OFF
)
```

This piece of code confines the **`unicode_stdio_example`** target to standard C++20. Change **`unicode_stdio_example`** to the name of your actual executable target, specified in the **`add_executable`** command shown previously. Feel free to change this piece of code if you want to use compiler extensions, you don't necessarily have to conform to the C++ standard if you don't want to but it is important to understand that CCL **requires** that you use C++20 or later. C++17 and earlier standards are not supported.

```
target_link_libraries(
    unicode_stdio_example
    cmp
)
```

This piece of code links the **`unicode_stdio_example`** executable target with the **`cmp`** library target. Change **`unicode_stdio_example`** to your own target's name. This will make CCL available to your application. Now just include CCL headers in your code, build it with CMake, and you're set. Here's the contents of the **`main.cpp`** file for this example:

```
// Copyright (C) 2022 Daniel T. McGinnis
// SPDX-License-Identifier: BSL-1.0

#include <iostream>
#include <cmp/io/uio.hpp>

int
main ()
{
    std::cout << "How easy does CCL make it to do real Unicode in C++?\n";
    cmp::uout << u8"I dunno, try it ¯\\_(ツ)_/¯\n";
    cmp::uout << u"Huh, let me see. Say something in Unicode: ";
    std::u8string input;
    cmp::uin >> input;
    cmp::uout << U"You said: " << input << '\n';
    cmp::uout << "Wow. Just wow.\n";
```

```
    return 0;
} // function ----------------------------------------------------------------
```

Configuration Constants

CCL can be customized, or **configured**, with preprocessor constants called **configuration constants**. The full list, along with their default values, is shown in Table 1-1.

Name	Default Value
CMP_CONFIG_LTO_REQUESTED	true
CMP_CONFIG_LTO_ENABLED	false
CMP_CONFIG_HEADER_ONLY	false
CMP_CONFIG_DEFAULT_BUFFER_CAPACITY	1024
CMP_CONFIG_DEFAULT_STDOUT_BUFFER_CAPACITY	CMP_CONFIG_DEFAULT_BUFFER_CAPACITY
CMP_CONFIG_DEFAULT_STDERR_BUFFER_CAPACITY	CMP_CONFIG_DEFAULT_BUFFER_CAPACITY
CMP_CONFIG_DEFAULT_STDIN_BUFFER_CAPACITY	CMP_CONFIG_DEFAULT_BUFFER_CAPACITY
CMP_CONFIG_IO_PACKAGE_EXCLUDED	false

Table 1-1: CCL's configuration constants.

Let's take a closer look at each of these.

CMP_CONFIG_LTO_REQUESTED

This constant determines whether link-time optimization (also called interprocedural optimization, or IPO) is desired. If IPO is requested and supported then it will be enabled. Otherwise, it will be disabled.

CMP_CONFIG_LTO_ENABLED

This constant determines whether link-time optimization (also called interprocedural optimization, or IPO) is enabled. You do not set this constant yourself. Instead, you set the **CMP_CONFIG_LTO_REQUESTED** constant, and then CMake will define **CMP_CONFIG_LTO_ENABLED** for you. If IPO is requested and supported, this constant will be set to **true**. Otherwise, it will be set to **false**. These LTO constants do nothing if you use CCL header-only.

CMP_CONFIG_HEADER_ONLY

This constant determines whether the library will be compiled or used header-only. The entire library is set up to be used in either way. In either case, make sure that CCL's **api/include** and **api/src** directories are both in your compiler's include paths. Then, if this constant is set to false, compile the library, include the headers in your code, and link with the compiled binary (statically or dynamically, both are supported). Or, if this constant is set to true, just include the headers in your code.

CMP_CONFIG_DEFAULT_BUFFER_CAPACITY

This constant determines the capacity of the I/O buffer of transfer resources in general.

CMP_CONFIG_DEFAULT_STDOUT_BUFFER_CAPACITY

This constant determines the capacity of the I/O buffer used when writing to standard output.

CMP_CONFIG_DEFAULT_STDERR_BUFFER_CAPACITY

This constant determines the capacity of the I/O buffer used when writing to standard error.

CMP_CONFIG_DEFAULT_STDIN_BUFFER_CAPACITY

This constant determines the capacity of the I/O buffer used when reading from standard input.

CMP_CONFIG_IO_PACKAGE_EXCLUDED

This constant determines whether the CCL IO package should be excluded. A reason for doing this is if you are using a platform that this package doesn't support. Another reason is if you simply don't need the functionality provided by this package.

Chapter 2: Unicode

Processing text in C++ can be difficult. C++ has extensive support for ASCII, allowing you to make programs that speak English, but if you want to make your software ready for use by people all around the world then you need support for Unicode. CCL provides some support for Unicode so you can stay in C++ and interoperate well with the world at large. However, to use CCL's Unicode facilities well you must first understand Unicode. This chapter introduces you to Unicode and how to use it effectively in C++ with CCL. CCL does not wrap existing Unicode libraries, it is written from scratch, giving you all the performance and flexibility enabled by C++20 in an elegant, modern API.

Chapter Table of Contents

- Reviewing ASCII
- Say Hello to Unicode
- Unicode Terminology
- Unicode in the Standard Library
- Unicode in CCL
  - Reading Unicode Strings from Files
  - Writing Unicode Strings to Files
  - Printing Unicode Strings to Standard Output
  - Reading Unicode Strings from Standard Input
  - Iterating Over Unicode Strings
  - Converting Unicode Strings
- Endianness
- Handling Endianness in CCL
- A Note About Source File Encoding

Reviewing ASCII

A **character set** is a system that makes it possible to represent characters in a computer. Character sets are usually implemented as a set of mappings between numbers and characters, where each number represents a unique character. The simplest of these is ASCII (American Standard Code for Information Interchange). The way ASCII works is it defines 128 different integers, where each one is associated with a unique character. So for example, the decimal number 65 represents an upper-case 'A'. ASCII encodes all letters of the English alphabet, all decimal digits, the basic punctuation marks, as well as some other common characters. Because a character encoded in ASCII requires only 7 bits of space, it easily fits in a single octet.

A common misunderstanding is that every byte is exactly 8 bits long. While 8 bits is certainly the most common size of a byte by far, some systems do have bytes of other sizes. The term **octet** refers to a number of exactly 8 bits. To simplify this discussion, this chapter will use the term "byte" from here on to refer to an 8-bit number. So we can say things like "each ASCII character takes up one byte".

The limitation of ASCII is that it doesn't represent all the characters used throughout the world, it only represents the characters that are common in English, hence the "American" in American Standard Code for Information Interchange. One solution to this problem would be to create a new character set where we simply list more number-to-character mappings, allowing more characters to be represented. This would increase the size of each character. You might say that it would be easy to just make each character use up all 8 bits so that you can represent 256 different characters and each character would still just take up one byte. That sounds interesting but 256 different numbers still doesn't let you encode all characters used in the world. You may suggest 16-bit characters. Certainly 65,536 different numbers is enough to encode a ton of characters, right? Well, it turns out that 16 bits is still not enough to encode all characters used in the world. Plus, it seems like a lot of work to specify all those characters! Luckily for us, the hard work has already been done. Enter Unicode.

Say Hello to Unicode

Unicode is a character set that assigns a unique number to pretty much every character used in the world. Every ASCII character is included in the Unicode character set, but Unicode adds thousands more characters to cater not only to speakers of English, but also to speakers of Spanish, French, German, Russian, Chinese, Japanese, Korean, and every other language in use today. Unicode also encodes historic characters that are no longer commonly used.

Unicode currently encodes more than a hundred thousand characters. You may assume then that every character in Unicode takes up more than 16 bits because 16 bits can only encode up to 65,536 different characters. That's not exactly how it works. Unicode has three different ways of expressing characters: UTF-8, UTF-16 and UTF-32. In UTF-32, every character is expressed as a single 32-bit number. In UTF-16, every character is expressed as one or two 16-bit numbers. In UTF-8, every character is expressed as one, two, three or four 8-bit numbers.

The advantage of UTF-32 is that every character really is a single value, allowing true random access (not as useful as it might seem, more on this later). The advantage of UTF-8 is that it is backwards-compatible with ASCII, that is, one byte containing an ASCII character is a valid Unicode character in UTF-8, without any processing needed. So for example, ASCII associates the number 65 with the character 'A'. In UTF-32, that character would take up 32 bits but it would still just be the value 65. In UTF-8, the character would take up 8 bits and the value would also be 65, the most significant bit would just be zero. Example:

```
'A' = 65 (in both ASCII and Unicode)
65 in ASCII:                                    1000001 (7 bits)
65 in UTF-8:                                   01000001 (8 bits)
65 in UTF-16:                         00000000 01000001 (16 bits)
65 in UTF-32: 00000000 00000000 00000000 01000001 (32 bits)
```

Figure 2-1: How ASCII and Unicode represent the character 'A'.

Let's look at another example. The character '∞' (the symbol for infinity) maps to the number 8,734. Because this number doesn't fit in 7 bits, UTF-8 expresses it in more than one byte, but because it still fits comfortably in 16 bits and 32 bits, UTF-16 represents it as a single 16-bit number and UTF-32 represents it as a single 32-bit number. This is what that looks like:

```
'∞' = 8,734 in Unicode
8,734 in UTF-8:                  11100010 10001000 10011110
8,734 in UTF-16:                        00100010 00011110
8,734 in UTF-32: 00000000 00000000 00100010 00011110
```

Figure 2-2: How Unicode represents the character '∞'.

Notice how in this case, the UTF-8 representation takes up more space (3 bytes) than the UTF-16 representation (2 bytes). Contrast this with how UTF-8 is more compact when representing the character 'A' (1 byte) than UTF-16 (2 bytes). UTF-8 isn't always more compact than UTF-16. UTF-8 is more compact in text files that have lots of ASCII characters, and UTF-16 is more compact in text files that have lots of characters from East-Asian languages.

Unicode Terminology

Unicode defines some terms that are important to understand in order to use Unicode effectively, which we will look at now. In Unicode, a **code point** is the number assigned to a character. For example, 65 is the number assigned to the character 'A', so 65 is a code point. 8,734 is the number assigned to the character '∞', so 8,734 is a code point. The code point 8,734 is expressed differently in UTF-8 than it is in UTF-16, but that doesn't change the fact that the number 8,734 is a single code point.

A **character** is a (possibly invisible) unit of text. A **grapheme cluster** is a visible unit of text as defined by a written language. The difference between a character and a grapheme cluster is that a grapheme cluster necessarily represents a graphic symbol that speakers of a language understand as indivisible, whereas a character may also represent other kinds of information necessary to represent text in a computer, for example line breaks, accents that are applied to letters, and format control codes. This distinction is important, so let's look at an example to clarify. 'ä' is a lower-case 'a' with a diaeresis applied. This is a single graphic symbol that represents an atomic unit of written text, just like the letter 'a' without a diaeresis on top. However, 'ä' can be encoded as one or two characters. One possible representation is the code point 228 ('ä'), another possible representation is the two code points 97 ('a') and 168 ("¨"). Unicode says that the diaeresis mark on its own is a character but not a grapheme cluster because people don't think of a single diaeresis mark as a full-blown unit of text, it is always placed on top of a letter like 'a'. Suddenly the term "character" doesn't sound as useful as it used to. So if you have a UTF-32 string object in C++, which is an array of characters, then a particular element in that string may be a single grapheme cluster as understood by speakers of a particular language, or it may represent a portion of a grapheme cluster. This is why the random access capability you get with UTF-32 isn't all that useful, since a single character in Unicode may not be what a user of a computer program would think of as an atomic unit of text. Since no Unicode encoding form provides random access to grapheme clusters, the variable length nature of UTF-8 is not a drawback because you still have to traverse a string object linearly if you want to manipulate grapheme clusters.

An **encoding form** is a method of expressing code points in memory. Unicode defines three encoding forms: UTF-8, UTF-16 and UTF-32. A **code unit** is the smallest value an encoding form works with, and it can either represent a code point or a part of a code point. The UTF-8 encoding form works with 8-bit code units, the UTF-16 encoding form works with 16-bit code units, and the UTF-32 encoding form works with 32-bit code units. UTF-8 can take up to four 8-bit code units to express one code point, UTF-16 can take up to two 16-bit code units to express one code point, and UTF-32 always uses one 32-bit code unit to express one code point.

Unicode in the Standard Library

The C++ standard library defines classes that represent Unicode strings. **`std::u8string`** represents a string encoded in UTF-8, **`std::u16string`** represents a string encoded in UTF-16, and **`std::u32string`** represents a string encoded in UTF-32. These classes store code units, that is, each element of a **`std::u8string`** is a **`char8_t`**, each element of a **`std::u16string`** is a **`char16_t`**, and each element of a **`std::u32string`** is a **`char32_t`**.

You can initialize strings of these types with string literals. The C++ language allows you to express string literals encoded in Unicode. You can initialize a Unicode string object with a Unicode string literal that contains fancy Unicode content, like this:

```
std::u8string utf8_string{u8"¯\\_(ツ)_/¯"};
std::u16string utf16_string{u"¯\\_(ツ)_/¯"};
std::u32string utf32_string{U"¯\\_(ツ)_/¯"};
```

That's about as far as the standard library goes. You can't read or write Unicode text files, you can't output Unicode strings to standard output or read them from standard input, there aren't any Unicode-aware algorithms, etc. This is where CCL comes in.

Unicode in CCL

CCL doesn't implement the entire Unicode standard but it does provide a lot of useful Unicode functionality. With CCL, you can read and write Unicode text files. You can print Unicode strings to standard output. You can read Unicode strings from standard input. You can iterate over the code points of Unicode strings of any encoding form. You can convert strings from any Unicode encoding form to any other Unicode encoding form. CCL also handles both little endian and big endian order.

Reading Unicode Strings from Files

With CCL, you can read Unicode-encoded text files into Unicode-encoded string objects. CCL does not define its own Unicode string types, it simply manipulates standard string types, aka **`std::u8string`**, **`std::u16string`** and **`std::u32string`**. However, CCL does define its own types for performing I/O, it doesn't integrate with the C++ IO streams in the standard library.

When reading Unicode text, you can mix and match all you want, so for example, you can read a line of text from a file encoded in UTF-8 and store that string as UTF-32 in a **`std::u32string`**. The chapter on I/O will go over how I/O is done in CCL but the basic gist of it is that to read from a file you must instantiate the **`cmp::file`** class with the path to the file you want to open, and then you read from that file through a stream. When you instantiate the stream, you specify the file object as well as the

encoding that the file is in (that is, whether it is UTF-8, UTF-16 or UTF-32). When reading a Unicode text file, you have to know what encoding the file is in. That's just how Unicode works, you have to know the encoding beforehand. Here's some code to demonstrate how all of this works:

```
#include <cmp/io/file.hpp>
#include <cmp/io/text_input_stream.hpp>

...

cmp::file f{"utf8_file.txt", cmp::read_only, cmp::if_not_there::fail};

if (!f.is_open()) {
    std::cout << "utf8_file.txt could not be opened for reading.\n";
    return -1;
}

cmp::text_input_stream stream{f, cmp::utf8};
std::u32string utf32_line;
stream.read_line(utf32_line);
// Now do something with utf32_line.
```

Notice how **`cmp::utf8`** is specified when the stream is constructed. This tells the stream what encoding the file is in. The other options are **`cmp::utf16`** and **`cmp::utf32`** for reading UTF-16 and UTF-32 text respectively. It is important to note that the code above shows how to read UTF-8 text, which is not affected by endianness. To read UTF-16 text or UTF-32 text, you must check for a BOM (byte order mark), which is discussed later in this chapter.

Writing Unicode Strings to Files

You can also write Unicode-encoded string objects out to a file. You can mix and match all you want here too, so for example, you can write a string object encoded in UTF-16 out to a file as UTF-8. Again, you instantiate **`cmp::file`** with the path to the file you want to open and you write to that file through a stream. Here's what that looks like:

```
#include <cmp/io/file.hpp>
#include <cmp/io/text_output_stream.hpp>

...

cmp::file f{"utf8_file.txt", cmp::write_only, cmp::if_not_there::fail};

if (!f.is_open()) {
    std::cout << "utf8_file.txt could not be opened for writing.\n";
    return -1;
}

cmp::text_output_stream stream{f, cmp::utf8};
std::u16string utf16_string{get_a_string_somehow()};
stream << utf16_string;
// Now utf16_string has been written out as UTF-8.
```

It is important to note that the code above shows how to write a string without first writing a BOM (byte order mark). Writing a BOM is not strictly required but it makes it easier for other programs to interpret the contents of a Unicode text file, and this is discussed later in this chapter.

Printing Unicode Strings to Standard Output

You can write any Unicode-encoded string object to standard output and standard error with the **`cmp::uout`** and **`cmp::uerr`** objects. Here's what that looks like:

```
#include <cmp/io/uio.hpp>

...

std::u16string utf16_string{u"UTF-16 string here ¯\\_(ツ)_/¯.\n"};
cmp::uout << utf16_string << u8"UTF-8 literal here ¯\\_(ツ)_/¯.\n";
```

The only requirement is that your terminal or shell have a font that supports the Unicode characters you try to print out. If it does, you'll get gorgeous Unicode text. If it doesn't, you'll get weird-looking garbage characters.

Reading Unicode Strings from Standard Input

You can read Unicode-encoded strings from standard input with the **cmp::uin** object. Here's an example of how to read 3 lines of Unicode text from standard input:

```
#include <cmp/io/uio.hpp>

...

std::u8string first_string;
std::u16string second_string;
std::u32string third_string;

cmp::uin >> first_string >> second_string >> third_string;

// 3 Unicode strings were read from standard input.
```

Beware of the fact that, on Windows, reading Unicode text from standard input with **cmp::uin** will fail if standard input is redirected. This is due to a known bug in Windows, and until Microsoft fixes it, there is nothing CCL (or any other software) can do about it. On all other operating systems, **cmp::uin** works normally, regardless of whether standard input has been redirected. If you need to be able to read from standard input on Windows regardless of whether it has been redirected then you cannot read Unicode text, you must fall back to non-Unicode mechanisms like **std::cin** or **scanf** or the **ReadFile** Windows API function.

Iterating Over Unicode Strings

You can iterate over a Unicode string object by code point, regardless of the encoding form of that string. You do this by constructing a **cmp::by_code_point** with the string you want to traverse, and then you just iterate over that object and you'll get a full code point at each iteration of the loop. Here's what that looks like:

```
#include <cmp/unicode/by_code_point.hpp>

...

std::u8string utf8_string{get_a_string_somehow()};
for (char32_t current_code_point : cmp::by_code_point{utf8_string}) {
    process_a_code_point_somehow(current_code_point);
}
// Now all code points stored in utf8_string have been processed.
```

Converting Unicode Strings

You can convert strings from one encoding form to another. To do that, call the functions **cmp::to_u8string**, **cmp::to_u16string** and **cmp::to_u32string** and pass in a standard Unicode string object (i.e. a **std::u8string**, a **std::u16string** or a **std::u32string**). Here's an example:

```
#include <cmp/unicode/algorithms.hpp>

...

std::u16string utf16_string{get_a_string_somehow()};
std::u32string utf32_string{cmp::to_u32string(utf16_string)};
process_a_string_somehow(utf32_string);
// utf16_string has been converted to UTF-32
// and the converted string has been processed.
```

Endianness

One challenge to supporting Unicode is endianness. Endianness decides whether the bytes of a multi-byte value are ordered from most significant to least significant, or least significant to most significant, and it applies to every multi-byte value your CPU manipulates, for example **int**s and **float**s. There are two types of endianness: big endian and little endian. In big endian systems, the bytes are ordered from most significant first (in earlier addresses) to least significant last (in later addresses). In little endian systems, the bytes are ordered from least significant first (in earlier addresses) to most significant last (in later addresses).

Endianness is a property of the CPU architecture. For example, x86 CPUs arrange multi-byte values in little endian order, whereas SPARC V8 CPUs arrange multi-byte values in big endian order. The default endianness of TCP/IP is big endian, meaning when two computers communicate with each other, data sent across the network is formatted in big endian order, and if one of the computers uses a little endian CPU then that computer will have to reverse the bytes every time it sends and receives data so that it can correctly interpret and process the data.

Let's look at an example. Imagine a 16-bit number with the decimal value 512. In C++ this could be a **std::uint16_t**. The two bytes that make up that 16-bit number would show up like this in big endian order: [earliest address] 00000010 00000000

[latest address]. In little endian order, the bytes would show up like this: [earliest address] 00000000 00000010 [latest address]. In both cases, the mathematical value 512 is represented, just in different ways. Notice how only the order of the **bytes** changes, not the order of the bits in each byte.

Because code units in UTF-16 and UTF-32 are comprised of more than one byte, it is important to know the order of those bytes. If you want to create a text file in UTF-16 or UTF-32, and you want any Unicode-compliant program to be able to read that file correctly, then you can either just default to big endian order or you can specify a **byte order mark**, or **BOM** for short. The BOM is just the code point 0xFEFF, and it must be at the very beginning of the file. If a BOM is present, it must come before everything else in the file. When a program reads that code point it will know what endianness is being used in the text file.

UTF-16 BOMs are two bytes long, and UTF-32 BOMs are four bytes long. In UTF-16, the first byte of a BOM that signals big endian is 0xFE and the second byte is 0xFF. Still in UTF-16, the first byte of a BOM that signals little endian is 0xFF and the second byte is 0xFE. UTF-32 uses the same technique but two of the four bytes have the value zero because the code point 0xFEFF fits in just 2 bytes.

If no BOM is present then the Unicode standard says that, in the absence of a higher-level protocol, the default endianness is big endian. This means that Unicode does let you read a file according to little endian order if you know that the file is in little endian order, even when no BOM is present. This can happen if, for example, a video game always writes save files in little endian order, because that file is subject to a higher-level protocol (in this case the video game) and it isn't expected to be opened and read in a regular text editor. For general interchangeable text files, the rule is to use big endian order if no BOM is present.

Handling Endianness in CCL

CCL does provide a way to handle endianness. In CCL, when you want to read a file, you instantiate the **`cmp::file`** class specifying the path to the file you want to open. Then you instantiate a stream and pass in the file object you created earlier, as well as the encoding that the file is in. Then you read all data from the file through the stream object, but before reading anything, you call the **`read_bom`** function on the stream, which will read the BOM (if present) and adjust the endianness of the stream according to the BOM read from the file. Regardless of whether there really was a BOM in the file, the **`read_bom`** function will return the code point that was read, which you can compare with the constant **`cmp::bom`**. If the returned code point compares equal to **`cmp::bom`** then a BOM was read and the endianness of the stream was set to the endianness of the file, meaning all future read operations will produce valid and accurate data. However, if the returned code point does not compare equal to **`cmp::bom`** then a regular character was read, and is part of the text content of the file. You can do whatever you want with this character. If your intention was to read the entire file into a string object, then just append that character to a string object and then call the **`append_all`** function on the stream and give it the string object. That will make the specified string object contain the entire contents of the file. Here's some code to illustrate this:

```
#include <cmp/io/file.hpp>
#include <cmp/io/text_input_stream.hpp>

...

cmp::file f{"utf16_file.txt", cmp::read_only, cmp::if_not_there::fail};

if (!f.is_open()) {
    std::cout << "utf16_file.txt could not be opened for reading.\n";
    return -1;
}

cmp::text_input_stream stream{f, cmp::utf16};
std::u8string file_contents;
char32_t possible_bom{stream.read_bom()};
if (possible_bom != cmp::bom) {
    cmp::append_code_point(file_contents, possible_bom);
}
stream.append_all(file_contents);
// Now do something with file_contents. file_contents has the file's accurate
// contents regardless of whether there was a BOM. If no BOM was present then
// the endianness specified in the stream's constructor was used. The stream's
// constructor defaults to big endian if you don't specify the endianness, thus
// conforming to the Unicode standard with minimal effort.
```

To write a Unicode string to a file, instantiate the **`cmp::file`** class specifying the path to the file you want to open, and write the string to the file with a stream. To write a BOM to the file, just call the **`write_bom`** function on the stream. Here's some code to illustrate this:

```
#include <cmp/io/file.hpp>
#include <cmp/io/text_output_stream.hpp>

...
```

```
cmp::file f{"utf16_file.txt", cmp::write_only, cmp::if_not_there::fail};

if (!f.is_open()) {
    std::cout << "utf16_file.txt could not be opened for writing.\n";
    return -1;
}

cmp::text_output_stream stream{f, cmp::utf16};
std::u8string content{get_a_string_somehow()};
stream.write_bom();
stream << content;
// Now a BOM and a string have been written to the file as UTF-16.
```

std::endian has three members: **std::endian::little** (which represents little endian), **std::endian::big** (which represents big endian), and **std::endian::native** (which represents the endianness of the platform you're compiling for). **std::endian::native** may be equivalent to **std::endian::little**, **std::endian::big** or neither. When **std::endian::native** is neither **std::endian::little** nor **std::endian::big**, that means the platform you're compiling for has **mixed** endianness. It is important to note that CCL assumes that a platform is either fully little endian or fully big endian, and may not work on platforms of mixed endianness.

A Note About Source File Encoding

GCC and Clang assume that source files are encoded in UTF-8, whereas Visual C++ assumes UTF-16. However, you can tell Visual C++ that your source files are encoded in UTF-8 by giving it the **/utf-8** flag. By using **/utf-8** on Visual C++ and encoding your source files in UTF-8, you get the guarantee that the Unicode text you put in string literals, rendered beautifully in your code editor or IDE, will appear picture-perfect when your compiled program prints it out. To use UTF-8, use a Unicode-capable code editor or IDE, examples of which include Visual Studio, Visual Studio Code, CLion, Code::Blocks, and others. Most of these editors show the encoding of the file in the bottom-right corner of the window. If you see "UTF-8" down there then your file is encoded correctly and you can enjoy the advanced Unicode editing capabilities of your editor, and you can go nuts on typing out accents, math symbols, greek letters, and so on, in your string literals.

Chapter 3: I/O

CCL has its own infrastructure for doing I/O (input and output). In this chapter we look at how to read and write text and binary data. Currently, CCL only supports doing I/O on files and Unicode strings, as well as standard I/O, but a future release will add support for doing I/O on network sockets.

Chapter Table of Contents

- Introduction
  - Transfer Resources
  - Transfer Streams
  - How They Fit Together
- Files
- Data Input Streams
- Data Output Streams
- String I/O Resources
- Text Input Streams
- Text Output Streams
- Wrapping Up

Introduction

CCL separates the entities on which I/O takes place from the interpretation of the data that flows through them. In CCL, I/O takes place on transfer resources. A transfer resource knows how to read or write bytes, but has no concept of what those bytes mean. It is the job of transfer streams to interpret those bytes.

Transfer Resources

An **input resource** is a source of input, and an **output resource** is a destination for output. Input resources and output resources are collectively called **transfer resources**. An object that is both an input resource and an output resource is called an **I/O resource**. The **`cmp::file`** class represents a file and it is an I/O resource because it supports both reading and writing.

Creating your own transfer resources is a more advanced topic and the details of doing this are not documented yet because CCL's internal API is still unstable. A future version of CCL will have a more mature internal API and a future version of this book will include a thorough explanation on how to use it to create your own transfer resources.

Transfer Streams

An **input stream** is an object that reads data from an input resource, and an **output stream** is an object that writes data to an output resource. Input streams and output streams are collectively called **transfer streams**. An object that is both an input stream and an output stream is called an **I/O stream**. CCL provides two kinds of streams: those that carry binary data and those that carry text data. The full list of transfer stream types in CCL is shown in Table 3-1.

Name	Purpose
`cmp::data_input_stream`	Reads binary data from an input resource.
`cmp::data_output_stream`	Writes binary data to an output resource.
`cmp::data_io_stream`	Reads and writes binary data from/to an I/O resource.
`cmp::text_input_stream`	Reads Unicode text from an input resource.
`cmp::text_output_stream`	Writes Unicode text to an output resource.
`cmp::text_io_stream`	Reads and writes Unicode text from/to an I/O resource.

Table 3-1: CCL's transfer stream types.

How They Fit Together

To perform I/O, you first instantiate the transfer resource, then you instantiate the transfer stream with the transfer resource, and lastly you use the transfer stream to do the I/O. As an example, let's look at how to read an integer from a file. To do that, you instantiate the **`cmp::file`** class with the path to the file you want to read from. When dealing with a file, you should then check if the file really is open, which may not be the case if, for example, the file you specified doesn't exist. Then, with the file object open, you instantiate the input stream with the file object. Finally, you read from the file through the stream object. Let's now look at how to do this in code:

```
#include <cmp/io/file.hpp>
#include <cmp/io/data_input_stream.hpp>

...

cmp::file f{"binary_file.dat", cmp::read_only, cmp::if_not_there::fail};

if (!f.is_open()) {
    std::cout << "binary_file.dat could not be opened for reading.\n";
    return -1;
}

cmp::data_input_stream stream{f, std::endian::native};
int number;
stream >> number;

std::cout << "The file contained the number " << number << '\n';
```

The transfer resources are completely decoupled from the transfer streams. CCL comes with 4 transfer resource types:

- **`cmp::file`**

  Represents a file.

- **`cmp::u8string_io_resource`**

  Represents a UTF-8 string I/O resource.

- **`cmp::u16string_io_resource`**

  Represents a UTF-16 string I/O resource.

- **`cmp::u32string_io_resource`**

  Represents a UTF-32 string I/O resource.

The string I/O resource types allow us to do I/O on Unicode strings, similar in spirit to **`std::stringstream`**, and they will be discussed later, but now let's take a closer look at the options we have with files, and subsequently we will look at transfer streams in detail.

Files

The **`cmp::file`** class has a constructor with the following parameters:

- const std::filesystem::path& file_path

  The path to the file you want to open.

- cmp::access_mode mode

  The access mode to use, indicating whether you want to open the file for reading, writing or both.

- cmp::if_not_there if_file_not_there

  What to do if the requested file does not exist: fail to open the file or create it.

- std::size_t buffer_capacity = cmp::io_buffer::default_buffer_capacity

  The capacity of the I/O buffer. Specify 0 for unbuffered operations.

For example, to open a file just for writing, and to create it if it doesn't already exist, write this:

```
cmp::file f{"path/to/file.txt", cmp::write_only, cmp::if_not_there::create};
```

To open a file for reading and writing, and to fail if it doesn't already exist, and to not buffer any I/O operations, write this:

```
cmp::file f{"path/to/file.dat", cmp::read_and_write, cmp::if_not_there::fail, 0};
```

Data Input Streams

The **`cmp::data_input_stream`** class has a constructor with the following parameters:

- wrapped_resource_type& resource

  The input resource to read from.

- std::endian endianness = std::endian::big

  The endianness the input data is expected to be in.

For example, to read from a file object called **`f`**, whose contents is in little endian order, write this:

```
cmp::data_input_stream stream{f, std::endian::little};
```

Then, to read from the file, use the overloaded **`>>`** operator, like this:

```
int integer;
stream >> integer;
```

Data input streams support reading data in all fundamental types.

Data Output Streams

The **`cmp::data_output_stream`** class has a constructor with the following parameters:

- wrapped_resource_type& resource

  The output resource to write to.

- std::endian endianness = std::endian::big

    The endianness the output data is going to be in.

For example, to write to a file object called **f**, whose contents will be in little endian order, write this:

```
cmp::data_output_stream stream{f, std::endian::little};
```

Then, to write to the file, use the overloaded **<<** operator, like this:

```
double number{get_some_number()};
stream << number;
```

Data output streams support writing data in all fundamental types.

String I/O Resources

String I/O resources behave similarly to files, except you read from, and write to, a Unicode string object, instead of a file. For example, to read from a **std::u8string**, you would write something like this:

```
const std::u8string& s{grab_some_string()};
cmp::u8string_io_resource resource{s};
cmp::text_input_stream stream{resource, cmp::utf8, std::endian::native};
std::u8string content;
stream.read_all(content);
cmp::uout << content << '\n';
```

This example is a little silly because we are simply copying a string into a string I/O resource and then just reading it in its entirety with the call to **read_all**, and then printing it. The power of string I/O resources becomes clear when you need to read a string methodically, one piece at a time.

To write to a string, use the output counterpart, **cmp::text_output_stream**. Like this:

```
cmp::u8string_io_resource resource;
cmp::text_output_stream stream{resource, cmp::utf8, std::endian::native, cmp::flush_strategy::automatic};
stream << u8"abc\nxyz";
cmp::uout << resource.grab_content() << '\n';
```

Here we create a string I/O resource, write to it through a stream, and then grab a reference to the string we have built up. Nothing too fancy. The important thing to note here is that you're manipulating a string I/O resource with text streams, so let's take a look at those now.

Text Input Streams

The **cmp::text_input_stream** class has a constructor with the following parameters:

- wrapped_resource_type& resource

    The input resource to read from.

- encoding_form source_encoding_form

    The Unicode encoding form the input text is expected to be in.

- std::endian endianness = std::endian::big

    The endianness the input text is expected to be in.

For example, to read from a file object called **f**, whose contents is in UTF-8 and little endian order, write this:

```
cmp::text_input_stream stream{f, cmp::utf8, std::endian::little};
```

Then, to read from the file object, use the overloaded **>>** operator, like this:

```
int integer;
stream >> integer;
```

In this example, we are reading an integer, which means we are going to parse an integer from the text. If a valid integer cannot be parsed, we'll get an exception.

Text input streams support reading data in all fundamental types.

Text Output Streams

The **cmp::text_output_stream** class has a constructor with the following parameters:

- wrapped_resource_type& resource

  The output resource to write to.

- encoding_form target_encoding_form

  The Unicode encoding form the output text is going to be in.

- std::endian endianness = std::endian::big

  The endianness the output data is going to be in.

- flush_strategy fs

  The flush strategy to use. That is, whether to flush at every newline.

For example, to write to a file object called **f**, whose contents will be in UTF-8 and little endian order, and to flush every time a newline is written out, write this:

```
cmp::text_output_stream stream{f, cmp::utf8, std::endian::little, cmp::flush_strategy::automatic};
```

Then, to write to the file, use the overloaded **<<** operator, like this:

```
double number{get_some_number()};
stream << number;
```

In this example, we are writing a floating-point number, which means we are going to write out the text representation of that number.

Text output streams support writing data in all fundamental types.

Wrapping Up

CCL is very capable when it comes to I/O. You can read and write both binary data and Unicode text. You can manipulate files (data and text) and strings (text only). The general approach is to create your transfer resource first, then create your transfer stream with the transfer resource, and finally perform all of your I/O with the stream.