This last chapter is a summary of the more advanced capabilities in C. The preceding chapters cover what you need for the majority of programming tasks. Whether you need the stuff in this chapter depends on the kinds of applications you are developing.

In this chapter you will learn:

- What facilities there are to support international character sets and working with several national languages
- What Unicode is and how the encodings are represented
- What locales are and how they help with international data representations
- Which C data types store Unicode characters
- The integer data types you can use to ensure your code is portable
- How you can work with complex numbers
- What threads are, how you create them, and what joining them does
- What a mutex is and how you use it

Working with International Character Sets

Unicode is the standard character representation for encoding characters for most of the world’s languages. Unicode also defines codes for a large number of special character sets, such as punctuation characters, mathematical symbols, and many others. Unicode is fundamental to writing applications that will be used internationally. Such a program must present its user interface and output in the language and conventions of the environment in which it is to be used.

Understanding Unicode

Unicode characters are represented by code values from 0 to 0x10ffff. This range of code values provides for representing more than a million characters, which is more than enough to accommodate the character sets for all the languages in the world. The codes are divided into 17 code planes, each of which contains 65,536 code values. Code plane 0 contains code values from 0 to 0xffff, code plane 1 contains codes from 0x10000 to 0x1ffff, code plane 2 contains codes from 0x20000 to 0x2ffff, and so on, with code plane 17 containing codes from 0x100000 to 0x10ffff. Character codes for most of the national languages are contained within code plane 0, which contains code values from 0 to 0xffff, so strings in the majority of languages can be represented as a sequence of single 16-bit Unicode codes.
One aspect of Unicode that can be confusing at first sight is that it provides more than one character encoding method. The most commonly used encodings are referred to as Universal Character Set Transformation Format (UTF)-8 and UTF-16, either of which can represent all the characters in the Unicode set. The difference between UTF-8 and UTF-16 is only in how a given character code value is presented; the numerical code value for any given character is the same in either representation. Here’s how these encodings represent characters:

- **UTF-8** is a character encoding that represents a character as a variable length sequence of between 1 and 4 bytes. I’ll explain how these are distinguished later in this chapter. The ASCII character set appears in UTF-8 as single byte codes that have the same code values as ASCII (ASCII code values are shown in Appendix B). Most web pages use UTF-8 to encode the text they contain. Code plane 0 is accommodated by 1-byte and 2-byte codes in UTF-8.

- **UTF-16** represents Unicode characters as one or two 16-bit values. UTF-16 includes UTF-8. Because a single 16-bit value accommodates all of code plane 0, UTF-16 covers most situations in programming for a multilingual context.

You have three integer types that store Unicode characters. Type `wchar_t` has been available in the C standard library for some time, but it has been augmented by types `char16_t` and `char32_t`. The problem with `wchar_t` is that its size is implementation defined, and this uncertainty and variability reduces its usability when code needs to be ported between different systems. Types `char16_t` and `char32_t` have fixed sizes of 2 and 4 bytes, respectively, which removes this uncertainty. I’ll describe all three data types so you know about the complete set, but first I’ll introduce you to locales.

## Setting the Locale

A `locale` identifies a country or territory. Selecting a locale for a program selects a national language, and consequently a character set, and determines how formatted data will be presented. You set the locale for an application by calling the `setlocale()` function that is declared in `locale.h`. The function has the prototype:

```c
char *setlocale(int category, const char *locale);
```

The locale is specified by the string you pass as the second argument. This is usually a name from the ISO 3166 standard for identifying countries. This standard includes codes of two letters, three letters, and three digits, but the two-letter codes are used most often. For example, the codes "US", "USA", and "840" all identify the United States, the codes "GB", "GBR", and "826" all identify the United Kingdom, and the codes "FR", "FRA", and "250" all identify France. The string "C" represents the minimum environment for compiling C code, and an empty string, "", specifies the native environment. The locale that is set can determine the representation of many things, including the character set, numerical values, and monetary symbols. The first argument to `setlocale()` enables you to control which value representations are affected by the call. You can use any of the following as the first argument to select what should be affected by the `locale` string value:

- **LC_ALL** causes everything to be set.
- **LC_COLLATE** affects how the `strcoll()` and `strxfrm()` functions in `string.h` behave.
- **LC_CTYPE** affects the character classification functions in `ctype.h` and the multi-byte and wide character functions in `wchar.h` and `wctype.h`.
- **LC_MONETARY** affects the representation of monetary values and the currency symbols used.
- **LC_NUMERIC** affects the decimal point symbol and the thousands separator in the representation of numeric data.
- **LC_TIME** affects how `strftime()` and `wcsftime()` in `time.h` behave.