### C# Language Structure

To write good C# programs, you need to have a good understanding of the structure of C#. This chapter examines the language or lexical structure of C# programs.

The order in which we are going to tackle this topic is as follows:

* + C# programs
  + Grammar
  + Line terminators
  + Comments
  + White space
  + Tokens
  + Keywords
  + Directives

##### C# Programs

All C# programs are made up of one or more *source files*. These source files, also known as compi- lation units, can be standalone text files or files contained within an IDE (Integrated Development Environment) such as Visual Studio.

These compilation units contain an ordered sequence of Unicode characters (a round-about way of saying text) and for maximum portability, all file source files should be encoded using UTF-8 encoding. By using a simple text editor (like Notepad) or a specific development environment for C#, you will be sure that you are using the right format.

A compilation unit consists of:

* Zero or more using directives
* Zero or more global attributes
* Zero or more namespace member declarations

An attribute is an object that represents data you want to associate with an element in your program, while an element to which you attach an attribute is called the target of that attribute.

Each of these has a specific purpose:

* **Using directives.** Using directives allow for the use of namespaces (which are used to logically arrange classes, structs, interfaces, enums and delegates) and types defined in other names- paces. These affect the global attributes and namespace member declarations of a compilation unit. A Using directive from one compilation unit has no effect on other compilation units.
* **Global attributes.** These allow the specification of attributes for the whole project. Assemblies and modules both act as physical containers for types (or as a code placeholder; we’ll look at these in greater detail later). An assembly can consist of several separate modules or for simpler projects, just the one.
* **Namespace member declarations.** These contribute members to a single declaration space called the global namespace.

When a C# program is compiled, all the compilation units are processed together, and this means that there is a dependency among them — if a program consists of more than one compilation unit, the com- piler will need access to all the compilation units to be able to successfully compile the source code.

When a C# program is compiled, it goes through three steps:

* **Transformation.** This process converts the file into Unicode characters (from whatever character type and encoding scheme is used for the compilation units).
* **Lexical analysis.** This process translates the Unicode characters into a stream of tokens.
* **Syntactic analysis.** This is when the stream of tokens is transformed into Microsoft Intermediate Language (MSIL) before being converted to executable code.

There are several kinds of tokens in C#:

* Identifiers
* Keywords
* Literals
* Operators
* Punctuators

Whitespace and comments are not tokens.

A conforming compiler should be able to take in Unicode compilation units or source files encoded using UTF-8 and transform that into a sequence of Unicode characters. It is also possible that some com- pilers will take in compilation units using different encoding schemes (such as UTF-16 or UTF-32), but this should not be relied upon.

##### Grammars

The C# programming language uses two different kinds of grammar.

* + **Lexical grammar.** This defines how Unicode characters are combined to form:
    - Line terminators
    - Whitespace
    - Comments
    - Tokens
    - Preprocessing directives
  + **Syntactic grammar.** This defines how the valid tokens resulting from following the lexical gram- mar rules are combined to create C# programs.

Grammar Ambiguities

With any programming language, there is always scope for ambiguity. For example, take the following code statement:

F(X<Y, Z>(5));

This simple statement can be interpreted in two ways:

* 1. A call to F with two arguments: X<Y and Z>(5)
  2. A call to F with one argument that is a call to a generic method X that has two type arguments (an argument where each argument is simply a type) and a single regular argument

Fortunately, there are rules that the compiler follows to remove ambiguity. In the preceding example (where we have a sequence of tokens that end in a type argument list), the compiler takes note of the token that immediately follows the closing >. If it is one of the following:

* + (
  + )
  + ]
  + :
  + ;

❑ ,

* .
* ?

❑ ==

* !=

The type argument list is taken to form part of the simple name, member access, or pointer member access preceding it, and all other options are discarded.

If the next token isn’t one listed above, the type argument list will not form part of the simple name, member access, or pointer member access preceding it.

*The preceding rule does not apply to parsing a type argument list in a namespace or type names.*

Going back to our original, rather ambiguous example:

F(X<Y, Z>(5));

Following the rules laid out above, this will be interpreted as a call to F with one argument that is a call to a generic method X that has two type arguments and a single regular argument.

A couple of examples of a statement that would be interpreted as a call to F with two arguments would be as follows:

F(X<Y, Z>5);

F(X<Y, Z>>5);

Let’s examine another statement:

X = F<Y> + Z;

This statement will, from the perspective of the operators used, be interpreted as:

* Less than operator <
* Greater than operator >
* Unary-plus operator +

Another way to write the preceding statement would be:

X = (F < Y) > (+Z);

Lexical Analysis

Every source file of a C# program has to adhere to the following lexical grammar pattern:

input:

input-sectionopt

input-section:

input-section-part

input-section input-section-part

input-section-part:

input-elementsopt new-line pp-directive

input-elements: input-element

input-elements input-element

input-element: whitespace comment token

Five basic elements come together to form the lexical structure of a C# compilation unit. These are:

* + Line terminators
  + Whitespace
  + Comments
  + Tokens
  + Preprocessing directives

Of all these, only tokens are important to the syntactic grammar of any C# program (except when the > token is combined with another token to make a single operator).When a compiler carries out lexical processing on a C# compilation unit, it is condensing the file into a series of tokens that then become the input for later syntactic processing. The line terminators, whitespace, and comments separating tokens are purely lexical and have no impact at all on the syntax of a C# program. Equally, preprocessing direc- tives are used only to skip portions of the code in the source file and are again not important when it comes to syntax.

Whenever there are several possible lexical grammar outputs from processing a source file, the lexical processor always picks the longest valid lexical element. For example, if the compiler encounters the following character sequence:

//

It processes and interprets it as the beginning of a single line of comment rather than two instances of the / token (which wouldn’t be a single-line comment). Similarly, when the following is encountered:

!=

It is interpreted as a comparison operator. With this in mind, it is easy to see how a simple typographical mistake in the source code can result in the end program behaving in a very unusual way. More likely, though, there will be an error.

Line Terminators

A line terminator is used to divide sequences of characters in a C# source file into separate lines. There are a number of different possible line terminators:

* Carriage return: U+000D
* Line feed: U+000A
* A carriage return followed by a line feed: U+000D U+000A
* Next line: U+2085
* Line separator: U+2028
* Paragraph separator: U+2029

To maintain a high level of compatibility with the various source code editing tools available that add end-of-file markers and to allow source files to be looked at as a valid sequence of terminated lines, a couple of transformations are applied to every C# source file:

* If the final character in a C# source file is a Control-Z character (U+001A), this is deleted.
* A carriage return (U+000D) is added to the end of a C# source file if that file is not empty and if the last character is not a carriage return (U+000D), line feed (U+000A), next line (U+2085), line separator (U+2028), or a paragraph separator (U+2029).

Comments

Two types of comments are supported in C# source files:

* Delimited comments
* Single-line comments

The following sections provide a more detailed look at the two kinds of comments.

***Delimited Comments***

A delimited comment always begins with the /\* characters and always ends with the \*/ characters. Delimited comments can also occupy a portion of a line:

/\* Hello World test program

\*/ class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

A single line:

/\* Hello World test program \*/ class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

Or multiple lines:

/\*

Hello World test program

\*/

class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

Delimited comments can appear anywhere in the code, as long as they occupy a separate line. For exam- ple, the following are all valid:

/\*

Hello World test program

\*/

class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

and:

class Test

{

/\*

Hello World test program

\*/

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

and:

/\*

Hello World test program

\*/

class Test

{

static void Main() {

/\*

String outputted to screen

\*/

System.Console.WriteLine(“Hello, World!”);

}

}

However, this kind of comment layout is invalid:

class Test

{

static void Main() { System.Console. /\*

Hello World test program

\*/

WriteLine(“Hello, World!”);

}

}

***Single-Line Comments***

Single-line comments are, as their name suggests, comments on a single line. They begin with the //

characters and extend to the end of the line:

class Test

{

static void Main() {

System.Console.WriteLine(“Hello, World!”); // displays “Hello, World!”

}

}

You can use as many single-line comments as you require:

// displays “Hello, World!” class Test

{

static void Main() {

System.Console.WriteLine(“Hello, World!”); // displays “Hello, World!”

}

}

Do not place single-line comments in the middle of statements. The following comments are invalid:

// displays “Hello, World!” class Test

{

static void Main() {

System.Console.WriteLine// displays “Hello, World!” (“Hello, World!”);

}

}

And:

class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”); }

}

***Nesting Comments***

You cannot and should not nest comments. For code clarity, you should *not* do the following:

// /\* Improper nesting of comments \*/

/\* // Improper nesting of comments \*/

*Nesting comments won’t cause any error to be displayed; it is just bad form and makes the code difficult to read.*

Whitespace

A white space is any character with the Unicode class Zs. This includes the space character as well as the horizontal tab character, the vertical tab character, and the form feed character.

Tokens

There are five kinds of token:

* + Identifiers
  + Keywords
  + Literals
  + Operators
  + Punctuation

*Whitespace and comments aren’t considered to be tokens, only separators for tokens.*

***Unicode Escape Sequences***

Unicode escape sequences are used to represent Unicode characters. One Unicode sequence represents a single Unicode character.

Unicode escape sequences are composed of the \U or \u characters followed by a hexadecimal number:

\Uxxxx or \uxxxx.

\U0066 is equivalent to the character f. However, \U00660066 would be f0066, not ff. To get ff, the following escape sequences would be required:

\U0066\U0066

The following code shows Unicode escape sequences in action:

class Test

{

static void Main() { System.Console.WriteLine(“\u0048\u0065\u006C\u006c\u006f, World!”);

}

}

The preceding code is equivalent to:

class Test

{

static void Main() { System.Console.WriteLine(“Hello, World!”);

}

}

Any Unicode escape sequences encountered in the following will be processed:

* Identifiers
* Regular string literals
* Character literals

Unicode escape sequences won’t be processed if encountered anywhere else.

***Identifiers***

Rules for identifiers are exactly the same as those recommended by the Unicode Standard Annex 15 ([http://www.unicode.org/reports/tr15/),](http://www.unicode.org/reports/tr15/)) except that:

* An underscore is allowable as an initial character, as is the tradition in C programming.
* Unicode escape sequences are allowable in identifiers.
* The @ character is allowable as a prefix to allow keywords to be used as identifiers. This can be extremely useful when you are using C# to interface with other programming languages. When the @ prefix is used to prefix an identifier, the identifier is called a *verbatim identifier*. While it is valid to use the @ prefix for identifiers that are not keywords, the practice is discouraged because of style.

Here’s a look at the syntax of identifiers:

identifier:

available-identifier

@ identifier-or-keyword

available-identifier:

An identifier-or-keyword that is not a keyword identifier-or-keyword:

identifier-start-character identifier-part-charactersopt

identifier-start-character: letter-character

\_ (the underscore character U+005F)

identifier-part-characters: identifier-part-character

identifier-part-characters identifier-part-character

identifier-part-character: letter-character decimal-digit-character connecting-character combining-character formatting-character

letter-character:

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

A unicode-escape-sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:

A Unicode character of classes Mn or Mc

A unicode-escape-sequence representing a character of classes Mn or Mc

decimal-digit-character:

A Unicode character of the class Nd

A unicode-escape-sequence representing a character of the class Nd

connecting-character:

A Unicode character of the class Pc

A unicode-escape-sequence representing a character of the class Pc

formatting-character:

A Unicode character of the class Cf

A unicode-escape-sequence representing a character of the class Cf

Here are a few valid identifiers:

* + identifier1
  + \_identifier
  + @private

Two identifiers are considered identical if they are the same after the following transformations have been applied (in the order listed):

* + The @ prefix is removed from verbatim identifiers.
  + Each Unicode escape sequence is transformed into Unicode characters.
  + All formatting characters are removed.

*Identifiers that make use of two consecutive underscore characters (\_) are reserved for future use.*

***Keywords***

A keyword is similar to an identifier, except that it is reserved. Keywords cannot be used as identifiers, except when prefixed with @.

Here is a list of C# keywords:

abstract as

base bool break byte case catch char checked class const continue decimal default delegate do double else enum event explicit extern false finally fixed float for foreach goto

if implicit in

int interface internal is

lock long

namespace new

null object operator out override params private protected public readonly ref return sbyte sealed short sizeof

stackalloc static string struct switch this

throw true try typeof uint ulong

unchecked unsafe ushort using virtual void volatile while

***Literals***

The job of a literal is simple — it is used to represent a value in the source code. There are a number of different literals.

**Boolean Literals**

There are two Boolean literals:

* + true
  + false

The type of a Boolean literal is bool.

**Integer Literals**

Integer literals are used to write values for the following types:

* + int
  + uint
  + long
  + ulong

Integer literals can take on two forms:

* + Decimal value
  + Hexadecimal value

You can determine the type of an integer literal as follows:

* + If the integer literal has no suffix, it is of the type:

int uint long ulong

* + If the integer literal is suffixed with U or u, it is of the type:

uint ulong

* + If the integer literal is suffixed with L or l, it is of the type:

long ulong

* + If the integer literal is suffixed with UL, uL, Ul, LU, lU, or Lu, it is of the type:

ulong

*If the value of any integer literal falls outside the range of the* ulong *type, a compiler error will be generated.*

**Real Literals**

Real literals are used to write values for the following types:

* float
* double
* decimal

The three types use different suffixes:

* F or f for float
* D or d for double
* M or m for decimal

If no suffix is specified, the default type is double.

**Character Literals**

A character literal is used to represent a single character in quotes, as in ‘x’. The following table takes a look at the escape characters in C#:

|  |  |  |
| --- | --- | --- |
| **Escape sequence** | **Character name** | **Unicode** |
| \’ | Single quote | 0x0027 |
| \” | Double quote | 0x0022 |
| \\ | Backslash | 0x005C |
| \a | Alert | 0x0007 |
| \b | Backspace | 0x0008 |
| \f | Form feed | 0x000C |
| \n | New line | 0x000A |
| \o | Null | 0x0000 |
| \r | Carriage return | 0x000D |
| \t | Horizontal tab | 0x0009 |
| \v | Vertical tab | 0x000B |

Here’s a look at the syntax of character literals:

character-literal:

‘ character ‘ character:

single-character simple-escape-sequence

hexadecimal-escape-sequence unicode-escape-sequence

single-character:

Any character except ‘ (U+0027), \ (U+005C), and new-line-character

simple-escape-sequence: one of

\’ \” \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:

\x hex-digit hex-digitopt hex-digitopt hex-digitopt

A character that follows a backslash character (\) must be one of the characters listed in the preceding table; otherwise, a compile-time error will occur.

**String Literals**

There is support for two types of string literals in C#:

* + Regular string literals
  + Verbatim string literals

A *regular string literal* is a string that consists of zero or more characters enclosed in double quotes. Regular string literals can include both simple escape sequences and hexadecimal and Unicode escape sequences.

string = “Hello, World!”;

A *verbatim string literal* is composed of the @ character followed by a double quote, then zero or more characters, and finally a closing double quote.

string = @”Hello, World!”;

The difference between a regular string literal and a verbatim string literal is that, in the latter, the char- acters that appear between the double-quote delimiters are interpreted verbatim, and string literals can span multiple lines in the source code.

*Note that the only exception to this verbatim processing are the* \” *characters, which represent a double quote.*

string = @”Hello

,

World!”;

***Null Literal***

There isn’t much to the null literal — it is simply a null type.

Operators and Punctuators

C# has a number of operators and punctuators.

* Operators are used in expressions to describe operations involving one, two, or more operands.
* Punctuators are used for grouping and separating.
  + {
  + }
  + [
  + ]
  + (
  + )
  + .

❑ ,

* + :
  + ;

❑ +

* + -
  + \*
  + /
  + %
  + &
  + |
  + ^
  + !
  + ~

❑ =

* + <
  + >
  + ?
  + ??
  + ::

❑ ++

* + - --
    - &&
    - ||
    - ->

❑ ==

* + - !=
    - <=
    - >=

❑ +=

* + - -=
    - \*=
    - /=
    - %=
    - &=
    - |=
    - ^=
    - <<
    - <<=
    - > > (right shift, composed of two tokens, > and >)
    - > >= (right shift assignment, comprised of two tokens, > and >=)

Preprocessing Directives

Preprocessing directives add a great deal of functionality to your C# coding. With them, you can:

* Conditionally skip sections of source files
* Report errors
* Report warning conditions
* Delineate sections of code

*The word “preprocessing” harks back to C and C++ and is used for consistency with these languages, as there is no preprocessing step with C#.*

In C# the following preprocessing directives are available:

* #define and #undef — Used to define and undefine conditional compilation symbols
* #if, #elif, #else, and #endif — Used to skip sections of code
* #line — Used to control line numbers of errors and warnings
* #error and #warning — Used to issue errors and warnings
* #region and #endregion — Used to mark sections of code
* #pragma — Used to provide contextual information to the compiler

Preprocessing directives are not C# tokens and do not form part of the syntactic grammar of C#.

Each preprocessing directive must be on a new line in the source code. Additionally, each must always begin with # followed by the preprocessing directive name.

*Note that you can have whitespace before the* # *character and also between the # and the directive name, although this isn’t recommended, as it can make the code harder to read.*

Any line of source code that contains the #define, #undef, #if, #elif, #else, #endif, or #line directive can end with a single-line comment. Delimited comments are not allowed on lines that contain preprocessing directives.

Preprocessing directives can have a huge impact on the end result of compiling C# source code. For example, compiling the following:

#define A #undef B #define C #undef B

class D

{

#if A

void E() {}

#else

void F() {} #endif

#if B

void G() {}

#else

void H() {} #endif

#if C

void I() {}

#else

void J() {}

#endif

#if D

void K() {}

#else

void L() {}

#endif

}

Is equivalent to the following:

class D

{

void E() {}

void H() {}

void I() {}

void L() {}

}

***Conditional Compilation Symbols***

The conditional compilation functionality is provided by #if, #elif, #else, and #endif directives, and they are controlled using preprocessing expressions and conditional compilation symbols.

A conditional compilation symbol has two possible states:

* + Defined
  + Undefined

Initially, the symbol is set to undefined unless it has been explicitly defined otherwise. When a #defined directive is encountered, it remains as such until #undef is processed or the end of the source file is reached.

***Preprocessing Expressions***

Preprocessing expressions can occur in #if and #elif directives. The following operators are allowed in preprocessing expressions:

* + !

❑ ==

* + !=
  + &&
  + ||

Parentheses can be used to group operators.

Evaluation of a preprocessing expression always yields a Boolean value.

***Declaration Directives***

Declaration directives are used to define or undefine conditional compilation symbols.

The processing of a #define directive causes the conditional compilation symbol to become defined, starting with the source line that immediately follows the directive.

The processing of a #undef directive will cause the conditional compilation symbol to become unde- fined, starting with the source line that immediately follows the directive.

A #define can redefine a conditional compilation symbol that is already defined, without the need for an #undef directive for that symbol.

***Conditional Compilation Directives***

A conditional compilation directive can be used conditionally to include or exclude portions of a C# source file.

When you use a conditional compilation directive, no more than one section of code is processed. The rules for processing are as follows:

* #if and #elif directives are evaluated in order until one results in true. If an expression is true, that section of code is selected.
* If all directives yield *false*, an #else directive, if present, is selected.
* In the event that all directives yield false and no #else is present, no selection is made.

*Skipped code is not subject to lexical analysis.*

***Diagnostic Directives***

Diagnostic directives are used explicitly to generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

Both

#warning Check code!

and

#error Code trouble here

produce a compile-time error and serve as a reminder that code needs altering.

***Region Control Directives***

Region control directives are used explicitly to mark regions of source code. No semantic meaning is attached to any region of code. These regions are for programmers or for use by automated tools.

Region control directives are used as follows:

#region

...

#endregion

This is equivalent to the following:

#if true

...

#endif

***Line Directives***

Line directives are used to alter the line numbers and source file names reported by the compiler in out- put such as warnings and errors.

When no #line directives are present in the source code, the compiler will report the correct line num- bers and source file names in any output given.

***Pragma Directives***

#pragma is a preprocessing directive used to specify contextual information to a compiler. Examples of when a pragma directive might be used include:

* + Enabling/disabling specific warnings
  + Specifying information that will be used by a debugger

##### Summary

In this chapter you examined the lexical structure of C#, paying close attention to C# programs, gram- mar, line terminators, comments, whitespace, tokens, keywords, and directives. Paying close attention to the lexical grammar of C# can save you a lot of time in fewer bugs and reduced debugging time.

In Chapter 5, you look at a variety of C# concepts.