

O'REILLY®

2nd Edition
Covers C11 standard



Free Sampler

C in a Nutshell

THE DEFINITIVE REFERENCE

Peter Prinz & Tony Crawford

C in a Nutshell, Second Edition

by Peter Prinz and Tony Crawford

Copyright © 2016 Peter Prinz and Tony Crawford. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Rachel Roumeliotis and
Katie Schooling

Production Editor: Kristen Brown

Copyeditor: Gillian McGarvey

Proofreader: Jasmine Kwityn

Indexer: Angela Howard

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

December 2005: First Edition
December 2015: Second Edition

Revision History for the Second Edition

2015-12-07: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491904756> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *C in a Nutshell*, Second Edition, the cover image of a cow, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-90475-6

[M]

Table of Contents

Preface.....	ix
--------------	----

Part I. Language

1. Language Basics.....	3
Characteristics of C	3
The Structure of C Programs	4
Source Files	6
Comments	7
Character Sets	8
Identifiers	14
How the C Compiler Works	19
2. Types.....	23
Typology	23
Integer Types	24
Floating-Point Types	30
Complex Floating-Point Types	32
Enumerated Types	33
The Type void	34
The Alignment of Objects in Memory	36
3. Literals.....	39
Integer Constants	39
Floating-Point Constants	40
Character Constants	42
String Literals	45
4. Type Conversions.....	49
Conversion of Arithmetic Types	50
Conversion of Nonarithmetic Types	58

5. Expressions and Operators.....	67
How Expressions Are Evaluated	68
Operators in Detail	73
Constant Expressions	97
6. Statements.....	99
Expression Statements	99
Block Statements	100
Loops	101
Selection Statements	105
Unconditional Jumps	108
7. Functions.....	113
Function Definitions	113
Function Declarations	120
How Functions Are Executed	122
Pointers as Arguments and Return Values	122
Inline Functions	123
Non-Returning Functions	125
Recursive Functions	126
Variable Numbers of Arguments	127
8. Arrays.....	129
Defining Arrays	129
Accessing Array Elements	131
Initializing Arrays	132
Strings	134
Multidimensional Arrays	136
Arrays as Arguments of Functions	138
9. Pointers.....	141
Declaring Pointers	141
Operations with Pointers	144
Pointers and Type Qualifiers	148
Pointers to Arrays and Arrays of Pointers	152
Pointers to Functions	156
10. Structures, Unions, and Bit-Fields.....	159
Structures	159
Unions	169
Anonymous Structures and Unions	171
Bit-Fields	172

11. Declarations.....	175
Object and Function Declarations	176
Type Names	184
typedef Declarations	185
_Static_assert Declarations	186
Linkage of Identifiers	187
Storage Duration of Objects	189
Initialization	190
12. Dynamic Memory Management.....	193
Allocating Memory Dynamically	194
Characteristics of Allocated Memory	195
Resizing and Releasing Memory	196
An All-Purpose Binary Tree	198
Characteristics	198
Implementation	199
13. Input and Output.....	209
Streams	209
Files	211
Opening and Closing Files	213
Reading and Writing	216
Random File Access	235
14. Multithreading.....	239
Threads	240
Accessing Shared Data	244
Communication Between Threads: Condition Variables	251
Thread-Local Objects and Thread-Specific Storage	256
15. Preprocessing Directives.....	261
Inserting the Contents of Header Files	262
Defining and Using Macros	264
Type-generic Macros	272
Conditional Compiling	272
Defining Line Numbers	274
Generating Error Messages	275
The #pragma Directive	275
The _Pragma Operator	276
Predefined Macros	277

Part II. Standard Library

16. The Standard Headers.....	283
Using the Standard Headers	284
Functions with Bounds-Checking	287
Contents of the Standard Headers	289
17. Functions at a Glance.....	321
Input and Output	321
Mathematical Functions	323
Character Classification and Conversion	330
String Processing	332
Multibyte Characters	333
Converting Between Numbers and Strings	335
Searching and Sorting	336
Memory Block Handling	336
Dynamic Memory Management	337
Date and Time	337
Process Control	339
Internationalization	340
Nonlocal Jumps	341
Multithreading (C11)	341
Debugging	345
Error Messages	346
18. Standard Library Functions.....	349

Part III. Basic Tools

19. Compiling with GCC.....	669
The GNU Compiler Collection	669
Obtaining and Installing GCC	670
Compiling C Programs with GCC	671
C Dialects	681
Compiler Warnings	683
Optimization	684
Debugging	688
Profiling	688
Option and Environment Variable Summary	689

20. Using make to Build C Programs.	695
Targets, Prerequisites, and Commands	695
The Makefile	696
Rules	696
Comments	703
Variables	704
Phony Targets	711
Other Target Attributes	712
Macros	714
Functions	715
Directives	719
Running make	722
21. Debugging C Programs with GDB.	731
Installing GDB	732
A Sample Debugging Session	732
Starting GDB	736
Using GDB Commands	741
Analyzing Core Files in GDB	763
22. Using an IDE with C.	767
IDEs for C	767
The Eclipse IDE for C/C++	768
Developing a C Program with Eclipse	770
Debugging a C Program in Eclipse	773
Further Information on Eclipse	776
Index.	777



Language Basics

This chapter describes the basic characteristics and elements of the C programming language.

Characteristics of C

C is a general-purpose, procedural programming language. Dennis Ritchie first devised C in the 1970s at AT&T Bell Laboratories in Murray Hill, New Jersey, for the purpose of implementing the Unix operating system and utilities with the greatest possible degree of independence from specific hardware platforms. The key characteristics of the C language are the qualities that made it suitable for that purpose:

- Source code portability
- The ability to operate “close to the machine”
- Efficiency

As a result, the developers of Unix were able to write most of the operating system in C, leaving only a minimum of system-specific hardware manipulation to be coded in assembler.

C’s ancestors are the typeless programming languages BCPL (the Basic Combined Programming Language), developed by Martin Richards; and B, a descendant of BCPL, developed by Ken Thompson. A new feature of C was its variety of data types: characters, numeric types, arrays, structures, and so on. Brian Kernighan and Dennis Ritchie published an official description of the C programming language in 1978. As the first de facto standard, their description is commonly referred to sim-

ply as *K&R*.¹ C owes its high degree of portability to a compact core language that contains few hardware-dependent elements. For example, the C language proper has no file access or dynamic memory management statements. In fact, there aren't even any statements for console input and output. Instead, the extensive C standard library provides the functions for all of these purposes.

This language design makes the C compiler relatively compact and easy to port to new systems. Furthermore, once the compiler is running on a new system, you can compile most of the functions in the standard library with no further modification, because they are in turn written in portable C. As a result, C compilers are available for practically every computer system.

Because C was expressly designed for system programming, it is hardly surprising that one of its major uses today is in programming embedded systems. At the same time, however, many developers use C as a portable, structured high-level language to write programs such as powerful word processor, database, and graphics applications.

The Structure of C Programs

The procedural building blocks of a C program are *functions*, which can invoke one another. Every function in a well-designed program serves a specific purpose. The functions contain *statements* for the program to execute sequentially, and statements can also be grouped to form *block statements*, or *blocks*. As the programmer, you can use the ready-made functions in the standard library, or write your own when no standard function fulfills your intended purpose. In addition to the C standard library, there are many specialized libraries available, such as libraries of graphics functions. However, by using such nonstandard libraries, you limit the portability of your program to those systems to which the libraries themselves have been ported.

Every C program must define at least one function of its own, with the special name `main()`, which is the first function invoked when the program starts. The `main()` function is the program's top level of control, and can call other functions as sub-routines.

Example 1-1 shows the structure of a simple, complete C program. We will discuss the details of declarations, function calls, output streams, and more elsewhere in this book. For now, we are simply concerned with the general structure of the C source code. The program in **Example 1-1** defines two functions, `main()` and `circularArea()`. The `main()` function calls `circularArea()` to obtain the area of a circle with a given radius, and then calls the standard library function `printf()` to output the results in formatted strings on the console.

¹ The second edition, revised to reflect the first ANSI C standard, is available as *The C Programming Language*, 2nd ed., by Brian W. Kernighan and Dennis M. Ritchie (Englewood Cliffs, NJ: Prentice Hall, 1988).

Example 1-1. A simple C program

```
// circle.c: Calculate and print the areas of circles

#include <stdio.h>           // Preprocessor directive

double circularArea( double r ); // Function declaration (prototype form)

int main()                  // Definition of main() begins
{
    double radius = 1.0, area = 0.0;

    printf( "    Areas of Circles\n\n" );
    printf( "    Radius          Area\n"
           "    -----\n" );

    area = circularArea( radius );
    printf( "%10.1f    %10.2f\n", radius, area );

    radius = 5.0;
    area = circularArea( radius );
    printf( "%10.1f    %10.2f\n", radius, area );

    return 0;
}

// The function circularArea() calculates the area of a circle
// Parameter:    The radius of the circle
// Return value: The area of the circle

double circularArea( double r )    // Definition of circularArea() begins
{
    const double pi = 3.1415926536; // Pi is a constant
    return pi * r * r;
}
```

Output:

Areas of Circles	
Radius	Area
1.0	3.14
5.0	78.54

Note that the compiler requires a prior *declaration* of each function called. The prototype of `circularArea()` in the third line of **Example 1-1** provides the information needed to compile a statement that calls this function. The prototypes of standard library functions are found in standard header files. Because the header file `stdio.h` contains the prototype of the `printf()` function, the *preprocessor directive* `#include <stdio.h>` declares the function indirectly by directing the compiler's

preprocessor to insert the contents of that file. (See also “How the C Compiler Works” on page 19.)

You may arrange the functions defined in a program in any order. In [Example 1-1](#), we could just as well have placed the function `circularArea()` before the function `main()`. If we had, then the prototype declaration of `circularArea()` would be superfluous, because the definition of the function is also a declaration.

Function definitions cannot be nested inside one another: you can define a local variable within a function block, but not a local function.

Source Files

The function definitions, global declarations, and preprocessing directives make up the source code of a C program. For small programs, the source code is written in a single source file. Larger C programs consist of several source files. Because the function definitions generally depend on preprocessor directives and global declarations, source files usually have the following internal structure:

1. Preprocessor directives
2. Global declarations
3. Function definitions

C supports modular programming by allowing you to organize a program in as many source and header files as desired, and to edit and compile them separately. Each source file generally contains functions that are logically related, such as the program’s user interface functions. It is customary to label C source files with the filename suffix `.c`.

Examples [1-2](#) and [1-3](#) show the same program as [Example 1-1](#), but divided into two source files.

Example 1-2. The first source file, containing the `main()` function

```
// circle.c: Prints the areas of circles.
// Uses circulararea.c for the math

#include <stdio.h>
double circularArea( double r );

int main()
{
    /* ... As in Example 1-1... */
}
```

Example 1-3. The second source file, containing the `circularArea()` function

```
// circulararea.c: Calculates the areas of circles.
// Called by main() in circle.c
```

```
double circularArea( double r )
{
    /* ... As in Example 1-1... */
}
```

When a program consists of several source files, you need to declare the same functions and global variables, and define the same macros and constants, in many of the files. These declarations and definitions thus form a sort of file header that is more or less constant throughout a program. For the sake of simplicity and consistency, you can write this information just once in a separate *header file*, and then reference the header file using an `#include` directive in each source code file. Header files are customarily identified by the filename suffix *.h*. A header file explicitly included in a C source file may in turn include other files.

Each C source file, together with all the header files included in it, makes up a *translation unit*. The compiler processes the contents of the translation unit sequentially, parsing the source code into tokens, its smallest semantic units, such as variable names and operators. See “[Tokens](#)” on [page 21](#) for more detail.

Any number of whitespace characters can occur between two successive tokens, allowing you a great deal of freedom in formatting the source code. There are no rules for line breaks or indenting, and you may use spaces, tabs, and blank lines liberally to create “human-readable” source code. The preprocessor directives are slightly less flexible: a preprocessor directive must always appear on a line by itself, and no characters except spaces or tabs may precede the hash mark (`#`) that begins the line.

There are many different conventions and “house styles” for source code formatting. Most of them include the following common rules:

- Start a new line for each new declaration and statement.
- Use indentation to reflect the nested structure of block statements.

Comments

You should use comments generously in the source code to document your C programs. There are two ways to insert a comment in C: *block comments* begin with `/*` and end with `*/`, and *line comments* begin with `//` and end with the next newline character.

You can use the `/*` and `*/` delimiters to begin and end comments within a line, and to enclose comments of several lines. For example, in the following function proto-

type, the ellipsis (...) signifies that the `open()` function has a third, optional parameter. The comment explains the usage of the optional parameter:

```
int open( const char *name, int mode, ... /* int permissions */ );
```

You can use `//` to insert comments that fill an entire line, or to write source code in a two-column format, with program code on the left and comments on the right:

```
const double pi = 3.1415926536;    // pi is constant
```

These line comments were officially added to the C language by the C99 standard, but most compilers already supported them even before C99. They are sometimes called “C++-style” comments, although they originated in C’s forerunner, BCPL.

Inside the quotation marks that delimit a character constant or a string literal, the characters `/*` and `//` do not start a comment. For example, the following statement contains no comments:

```
printf( "Comments in C begin with /* or //.\\n" );
```

The only thing that the preprocessor looks for in examining the characters in a comment is the end of the comment; thus it is not possible to nest block comments. However, you can insert `/*` and `*/` to comment out part of a program that contains line comments:

```
/* Temporarily removing two lines:
const double pi = 3.1415926536;    // pi is constant
area = pi * r * r                // Calculate the area
Temporarily removed up to here */
```

If you want to comment out part of a program that contains block comments, you can use a conditional preprocessor directive (described in [Chapter 15](#)):

```
#if 0
const double pi = 3.1415926536;    /* pi is constant */
area = pi * r * r                /* Calculate the area */
#endif
```

The preprocessor replaces each comment with a space. The character sequence `min/*max*/Value` thus becomes the two tokens `min Value`.

Character Sets

C makes a distinction between the environment in which the compiler translates the source files of a program (the *translation environment*) and the environment in which the compiled program is executed (the *execution environment*). Accordingly, C defines two character sets: the *source character set* is the set of characters that may be used in C source code, and the *execution character set* is the set of characters that can be interpreted by the running program. In many C implementations, the two character sets are identical. If they are not, then the compiler converts the characters in character constants and string literals in the source code into the corresponding elements of the execution character set.

Each of the two character sets includes both a *basic character set* and *extended characters*. The C language does not specify the extended characters, which are usually dependent on the local language. The extended characters together with the basic character set make up the *extended character set*.

The basic source and execution character sets both contain the following types of characters:

The letters of the Latin alphabet

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

The decimal digits

0 1 2 3 4 5 6 7 8 9

The following 29 graphic characters

! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~

The five whitespace characters

Space, horizontal tab, vertical tab, newline, and form feed

The basic execution character set also includes four nonprintable characters: the *null* character (which acts as the termination mark in a character string), *alert*, *backspace*, and *carriage return*. To represent these characters in character and string literals, type the corresponding *escape sequences* beginning with a backslash: `\0` for the null character, `\a` for alert, `\b` for backspace, and `\r` for carriage return. See [Chapter 3](#) for more details.

The actual numeric values of characters—the character codes—may vary from one C implementation to another. The language itself imposes only these conditions:

- Each character in the basic character set must be representable in one byte.
- The null character is a byte in which all bits are 0.
- The value of each decimal digit after 0 is greater by one than that of the preceding digit.

Wide Characters and Multibyte Characters

C was originally developed in an English-speaking environment where the dominant character set was the 7-bit ASCII code. Since then, the 8-bit byte has become the most common unit of character encoding, but software for international use generally has to be able to represent more different characters than can be coded in one byte. Furthermore, a variety of multibyte character encoding schemes have long been in use internationally to represent non-Latin alphabets and the nonalphabetic Chinese, Japanese, and Korean writing systems. In 1994, with the adoption of “Normative Addendum 1,” ISO C standardized two ways of representing larger character sets:

- *Wide characters*, in which the same bit width is used for every character in a character set
- *Multibyte characters*, in which a given character can be represented by one or several bytes, and the character value of a given byte sequence can depend on its context in a string or stream



Although C now provides abstract mechanisms to manipulate and convert the different kinds of encoding schemes, the language itself doesn't define or specify any encoding scheme, or any character set except the basic source and execution character sets described in the previous section. In other words, it is left up to individual implementations to specify how to encode wide characters, and what multibyte encoding schemes to support.

Wide characters

Since the 1994 addendum, C has provided not only the type `char` but also `wchar_t`, the *wide character* type. This type, defined in the header file `stddef.h`, is large enough to represent any element of the given implementation's extended character sets.

Although the C standard does not require support for Unicode character sets, many implementations use the Unicode transformation formats UTF-16 and UTF-32 (see <http://www.unicode.org/>) for wide characters. The Unicode standard is largely identical with the ISO/IEC 10646 standard, and is a superset of many previously existing character sets, including the 7-bit ASCII code. When the Unicode standard is implemented, the type `wchar_t` is at least 16 or 32 bits wide, and a value of type `wchar_t` represents one Unicode character. For example, the following definition initializes the variable `wc` with the Greek letter α :

```
wchar_t wc = '\x3b1';
```

The escape sequence beginning with `\x` indicates a character code in hexadecimal notation to be stored in the variable—in this case, the code for a lowercase alpha.

For better Unicode support, C11 introduced the additional wide-character types `char16_t` and `char32_t`, which are defined as unsigned integer types in the header file `uchar.h`. Characters of the type `char16_t` are encoded in UTF-16 in C implementations that define the macro `__STDC_UTF_16__`. Similarly, in implementations that define the macro `__STDC_UTF_32__`, characters of the type `char32_t` are encoded in UTF-32.

Multibyte characters

In multibyte character sets, each character is coded as a sequence of one or more bytes. Both the source and execution character sets may contain multibyte characters. If they do, then each character in the basic character set occupies only one byte,

and no multibyte character except the null character may contain any byte in which all bits are 0. Multibyte characters can be used in character constants, string literals, identifiers, comments, and header filenames. Many multibyte character sets are designed to support a certain language, such as the Japanese Industrial Standard character set (JIS). The multibyte UTF-8 character set, defined by the Unicode Consortium, is capable of representing all Unicode characters. UTF-8 uses from one to four bytes to represent a character.

The key difference between multibyte characters and wide characters (that is, characters of the type `wchar_t`, `char16_t`, or `char32_t`) is that wide characters are all the same size, and multibyte characters are represented by varying numbers of bytes. This representation makes multibyte strings more complicated to process than strings of wide characters. For example, even though the character *A* can be represented in a single byte, finding it in a multibyte string requires more than a simple byte-by-byte comparison, because the same byte value in certain locations could be part of a different character. Multibyte characters are well suited for saving text in files, however (see [Chapter 13](#)). Furthermore, the encoding of multibyte characters is independent of the system architecture, while encoding of wide characters is dependent on the given system's byte order: that is, the bytes of a wide character may be in *big-endian* or *little-endian* order, depending on the system.

Conversion

C provides standard functions to obtain the `wchar_t` value of any multibyte character, and to convert any wide character to its multibyte representation. For example, if the C compiler uses the Unicode standards UTF-16 and UTF-8, then the following call to the function `wctomb()` (read: “wide character to multibyte”) obtains the multibyte representation of the character *α*:

```
wchar_t wc = L'\x3B1';    // Greek lowercase alpha, α
char mbStr[10] = "";
int nBytes = 0;
nBytes = wctomb( mbStr, wc );
if( nBytes < 0 )
    puts("Not a valid multibyte character in your locale.");
```

After a successful function call, the array `mbStr` contains the multibyte character, which in this example is the sequence `"\xCE\xB1"`. The `wctomb()` function's return value, assigned here to the variable `nBytes`, is the number of bytes required to represent the multibyte character—namely, 2.

The standard library also provides conversion functions for `char16_t` and `char32_t`, the new wide-character types introduced in C11, such as the function `c16rtomb()`, which returns the multibyte character that corresponds to a given wide character of the type `char16_t` (see [“Multibyte Characters” on page 333](#)).

Universal Character Names

C also supports universal character names as a way to use the extended character set regardless of the implementation's encoding. You can specify any extended character by its *universal character name*, which is its Unicode value in the form:

```
\uXXXX
```

or:

```
\UXXXXXXXX
```

where `XXXX` or `XXXXXXXX` is a Unicode code point in hexadecimal notation. Use the lowercase `u` prefix followed by four hexadecimal digits, or the uppercase `U` followed by exactly eight hex digits. If the first four hexadecimal digits are zero, then the same universal character name can be written either as `\uXXXX` or as `\U0000XXXX`.

Universal character names are permissible in identifiers, character constants, and string literals. However, they must not be used to represent characters in the basic character set.

When you specify a character by its universal character name, the compiler stores it in the character set used by the implementation. For example, if the execution character set in a localized program is ISO 8859-7 (8-bit Greek), then the following definition initializes the variable `alpha` with the code `\xE1`:

```
char alpha = '\u03B1';
```

However, if the execution character set is UTF-16, then you need to define the variable as a wide character:

```
wchar_t alpha = '\u03B1'; // or char16_t alpha = u'\u03B1';
```

In this case, the character code value assigned to `alpha` is hexadecimal `3B1`, the same as the universal character name.



Not all compilers support universal character names.

Digraphs and Trigraphs

C provides alternative representations for a number of punctuation marks that are not available on all keyboards. Six of these are the *digraphs*, or two-character tokens, which represent the characters shown in [Table 1-1](#).

Table 1-1. Digraphs

Digraph	Equivalent
<:	[
:>]
<%	{
%>	}
%:	#
%:%:	##

These sequences are not interpreted as digraphs if they occur within character constants or string literals. In all other positions, they behave exactly like the single-character tokens they represent. For example, the following code fragments are perfectly equivalent, and produce the same output. With digraphs:

```
int arr<::> = <% 10, 20, 30 %>;
printf( "The second array element is <%d>.\n", arr<:1:> );
```

Without digraphs:

```
int arr[] = { 10, 20, 30 };
printf( "The second array element is <%d>.\n", arr[1] );
```

Output:

```
The second array element is <20>.
```

C also provides *trigraphs*, three-character representations, all of them beginning with two question marks. The third character determines which punctuation mark a trigraph represents, as shown in Table 1-2.

Table 1-2. Trigraphs

Trigraph	Equivalent
??([
??)]
??<	{
??>	}
??=	#
??/	\
??!	
??'	^
??-	~

Trigraphs allow you to write any C program using only the characters defined in ISO/IEC 646, the 1991 standard corresponding to 7-bit ASCII. The compiler's preprocessor replaces the trigraphs with their single-character equivalents in the first phase of compilation. This means that the trigraphs, unlike digraphs, are translated into their single-character equivalents no matter where they occur, even in character constants, string literals, comments, and preprocessing directives. For example, the preprocessor interprets the following statement's second and third question marks as the beginning of a trigraph:

```
printf("Cancel???(y/n) ");
```

Thus, the line produces the following unintended preprocessor output:

```
printf("Cancel?[y/n] ");
```

If you need to use one of these three-character sequences and do not want it to be interpreted as a trigraph, you can write the question marks as escape sequences:

```
printf("Cancel\\?\\?(y/n) ");
```

If the character following any two question marks is not one of those shown in [Table 1-2](#), then the sequence is not a trigraph, and remains unchanged.



As another substitute for punctuation characters in addition to the digraphs and trigraphs, the header file *iso646.h* contains macros that define alternative representations of C's logical operators and bitwise operators, such as `and` for `&&` and `xor` for `^`. For details, see [Chapter 16](#).

Identifiers

The term *identifier* refers to the names of variables, functions, macros, structures, and other objects defined in a C program. Identifiers can contain the following characters:

- The letters in the basic character set, a–z and A–Z (identifiers are case-sensitive)
- The underscore character, `_`
- The decimal digits 0–9, although the first character of an identifier must not be a digit
- Universal character names that represent the letters and digits of other languages

The permissible universal characters are defined in Annex D of the C standard, and correspond to the characters defined in the ISO/IEC TR 10176 standard, minus the basic character set.

Multibyte characters may also be permissible in identifiers. However, it is up to the given C implementation to determine exactly which multibyte characters are permitted and what universal character names they correspond to.

The following 44 keywords are *reserved* in C, each having a specific meaning to the compiler, and must not be used as identifiers:

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

The following examples are valid identifiers:

```
x dollar Break error_handler scale64
```

The following are not valid identifiers:

```
1st_rank switch y/n x-ray
```

If the compiler supports universal character names, then α is also an example of a valid identifier, and you can define a variable by that name:

```
double  $\alpha$  = 0.5;
```

Your source code editor might save the character α in the source file as the universal character `\u03B1`.

When choosing identifiers in your programs, remember that many identifiers are already used by the C standard library. These include the names of standard library functions, which you cannot use for functions you define or for global variables. See [Chapter 16](#) for details.

The C compiler provides the predefined identifier `__func__` (note that there are four underscore characters), which you can use in any function to access a string constant containing the name of the function. This is useful for logging or for debugging output; for example:

```
#include <stdio.h>
int test_func( char *s )
{
    if( s == NULL ) {
        fprintf( stderr,
                "%s: received null pointer argument\n", __func__ );
        return -1;
    }
}
```

```

    }
    /* ... */
}

```

In this example, passing a null pointer to the function `test_func()` generates the following error message:

```
test_func: received null pointer argument
```

There is no limit on the length of identifiers. However, most compilers consider only a limited number of characters in identifiers to be significant. In other words, a compiler might fail to distinguish between two identifiers that start with a long identical sequence of characters. To conform to the C standard, a compiler must treat at least the first 31 characters as significant in the names of functions and global variables (that is, identifiers with external linkage), and at least the first 63 characters in all other identifiers.

Identifier Name Spaces

All identifiers fall into exactly one of the following four categories, which constitute separate *name spaces*:

- Label names
- Tags, which identify structure, union, and enumeration types
- Names of structure or union members (each structure or union constitutes a separate name space for its members)
- All other identifiers, which are called *ordinary identifiers*

Identifiers that belong to different name spaces may be the same without causing conflicts. In other words, you can use the same name to refer to different objects, if they are of different kinds. For example, the compiler is capable of distinguishing between a variable and a label with the same name. Similarly, you can give the same name to a structure type, an element in the structure, and a variable, as the following example shows:

```

struct pin { char pin[16]; /* ... */ };
_Bool check_pin( struct pin *pin )
{
    int len = strlen( pin->pin );
    /* ... */
}

```

The first line of the example defines a structure type identified by the tag `pin`, containing a character array named `pin` as one of its members. In the second line, the function parameter `pin` is a pointer to a structure of the type just defined. The expression `pin->pin` in the fourth line designates the member of the structure that the function's parameter points to. The context in which an identifier appears always determines its name space with no ambiguity. Nonetheless, it is generally a good

idea to make all identifiers in a program distinct, in order to spare human readers unnecessary confusion.

Identifier Scope

The *scope* of an identifier refers to that part of the translation unit in which the identifier is meaningful. Or to put it another way, the identifier's scope is that part of the program that can “see” that identifier. The type of scope is always determined by the location at which you declare the identifier (except for labels, which always have function scope). Four kinds of scope are possible:

File scope

If you declare an identifier outside all blocks and parameter lists, then it has file scope. You can then use the identifier anywhere after the declaration and up to the end of the translation unit.

Block scope

Except for labels, identifiers declared within a block have block scope. You can use such an identifier only from its declaration to the end of the smallest block containing that declaration. The smallest containing block is often, but not necessarily, the body of a function definition. Starting with C99, declarations do not have to be placed before all statements in a function block. The parameter names in the head of a function definition also have block scope, and are valid within the corresponding function block.

Function prototype scope

The parameter names in a function prototype have function prototype scope. Because these parameter names are not significant outside the prototype itself, they are meaningful only as comments, and can also be omitted. See [Chapter 7](#) for further information.

Function scope

The scope of a label is always the function block in which the label occurs, even if it is placed within nested blocks. In other words, you can use a `goto` statement to jump to a label from any point within the same function that contains the label. (Jumping into nested blocks is not a good idea, though; see [Chapter 6](#) for details.)

The scope of an identifier generally begins *after* its declaration. However, the type names—or tags—of structure, union, and enumeration types and the names of enumeration constants are an exception to this rule: their scope begins immediately after their appearance *in* the declaration so that they can be referenced again in the declaration itself. (Structures and unions are discussed in detail in [Chapter 10](#); enumeration types are described in [Chapter 2](#).) For example, in the following declaration of a structure type, the last member of the structure, `next`, is a pointer to the very structure type that is being declared:

```

struct Node { /* ... */
    struct Node *next; };           // Define a structure type
void printNode( const struct Node *ptrNode); // Declare a function

int printList( const struct Node *first ) // Begin a function
{                                         // definition
    struct Node *ptr = first;

    while( ptr != NULL ) {
        printNode( ptr );
        ptr = ptr->next;
    }
}

```

In this code snippet, the identifiers `Node`, `next`, `printNode`, and `printList` all have file scope. The parameter `ptrNode` has function prototype scope, and the variables `first` and `ptr` have block scope.

It is possible to use an identifier again in a new declaration nested within its existing scope, even if the new identifier does not have a different name space. If you do so, then the new declaration must have block or function prototype scope, and the block or function prototype must be a true subset of the outer scope. In such cases, the new declaration of the same identifier *hides* the outer declaration so that the variable or function declared in the outer block is not *visible* in the inner scope. For example, the following declarations are permissible:

```

double x;           // Declare a variable x with file scope
long calc( double x ); // Declare a new x with function prototype
                      // scope

int main()
{
    long x = calc( 2.5 ); // Declare a long variable x with block scope

    if( x < 0 )           // Here, x refers to the long variable
    { float x = 0.0F;      // Declare a new variable x with block scope
      /*...*/
    }
    x *= 2;               // Here, x refers to the long variable again
    /*...*/
}

```

In this example, the `long` variable `x` declared in the `main()` function hides the global variable `x` with type `double`. Thus, there is no direct way to access the `double` variable `x` from within `main()`. Furthermore, in the conditional block that depends on the `if` statement, `x` refers to the newly declared `float` variable, which in turn hides the `long` variable `x`.

How the C Compiler Works

Once you have written a source file using a text editor, you can invoke a C compiler to translate it into machine code. The compiler operates on a *translation unit* consisting of a source file and all the header files referenced by `#include` directives. If the compiler finds no errors in the translation unit, it generates an *object file* containing the corresponding machine code. Object files are usually identified by the filename suffix `.o` or `.obj`. In addition, the compiler may also generate an assembler listing (see [Chapter 19](#)).

Object files are also called *modules*. A library, such as the C standard library, contains compiled, rapidly accessible modules of the standard functions.

The compiler translates each translation unit of a C program—that is, each source file with any header files it includes—into a separate object file. The compiler then invokes the *linker*, which combines the object files and any library functions used in an *executable file*. [Figure 1-1](#) illustrates the process of compiling and linking a program from several source files and libraries. The executable file also contains any information that the target operating system needs in order to load and start it.

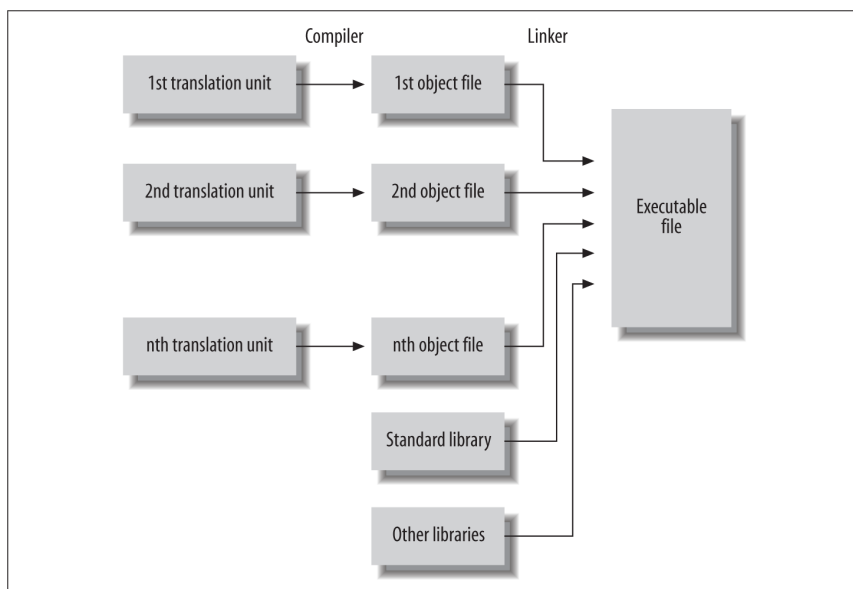


Figure 1-1. From source code to executable file

The C Compiler's Translation Phases

The compiling process takes place in eight logical steps. A given compiler may combine several of these steps as long as the results are not affected. The steps are:

1. Characters are read from the source file and converted, if necessary, into the characters of the source character set. The end-of-line indicators in the source file, if different from the newline character, are replaced. Likewise, any trigraph sequences are replaced with the single characters they represent. (Digraphs, however, are left alone; they are not converted into their single-character equivalents.)
2. Wherever a backslash is followed immediately by a newline character, the preprocessor deletes both. Because a line-end character ends a preprocessor directive, this processing step lets you place a backslash at the end of a line in order to continue a directive, such as a macro definition, on the next line.



Every source file, if not completely empty, must end with a newline character.

3. The source file is broken down into preprocessor tokens (see “[Tokens](#)” on page 21) and sequences of whitespace characters. Each comment is treated as one space.
4. The preprocessor directives are carried out and macro calls are expanded.



Steps 1 through 4 are also applied to any files inserted by `#include` directives. Once the compiler has carried out the preprocessor directives, it removes them from its working copy of the source code.

5. The characters and escape sequences in character constants and string literals are converted into the corresponding characters in the execution character set.
6. Adjacent string literals are concatenated into a single string.
7. The actual compiling takes place: the compiler analyzes the sequence of tokens and generates the corresponding machine code.
8. The linker resolves references to external objects and functions, and generates the executable file. If a module refers to external objects or functions that are not defined in any of the translation units, the linker takes them from the standard library or another specified library. External objects and functions must not be defined more than once in a program.

For most compilers, either the preprocessor is a separate program, or the compiler provides options to perform only the preprocessing (steps 1 through 4 in the preceding list). This setup allows you to verify that your preprocessor directives have the intended effects. For a more practically oriented look at the compiling process, see [Chapter 19](#).

Tokens

A token is either a keyword, an identifier, a constant, a string literal, or a symbol. Symbols in C consist of one or more punctuation characters, and function as operators or digraphs, or have syntactic importance, like the semicolon that terminates a simple statement or the braces { } that enclose a block statement. For example, the following C statement consists of five tokens:

```
printf("Hello, world.\n");
```

The individual tokens are:

```
printf  
(  
"Hello, world.\n"  
)  
;
```

The tokens interpreted by the preprocessor are parsed in the third translation phase. These are only slightly different from the tokens that the compiler interprets in the seventh phase of translation:

- Within an `#include` directive, the preprocessor recognizes the additional tokens `<filename>` and `"filename"`.
- During the preprocessing phase, character constants and string literals have not yet been converted from the source character set to the execution character set.
- Unlike the compiler proper, the preprocessor makes no distinction between integer constants and floating-point constants.

In parsing the source file into tokens, the compiler (or preprocessor) always applies the following principle: each successive non-whitespace character must be appended to the token being read, unless appending it would make a valid token invalid. This rule resolves any ambiguity in the following expression, for example:

```
a+++b
```

Because the first + cannot be part of an identifier or keyword starting with `a`, it begins a new token. The second + appended to the first forms a valid token—the increment operator—but a third + does not. Hence the expression must be parsed as:

```
a ++ + b
```

See [Chapter 19](#) for more information on compiling C programs.