THE EXPERT'S VOICE® IN C

C Quick Syntax Reference

Mikael Olsson

Apress®

Mikael Olsson

C Quick Syntax Reference

~~Mikael Olsson~~

ISBN 978-1-4302-6499-6 e-ISBN 978-1-4302-6500-9
DOI 10.1007/978-1-4302-6500-9

C Quick Syntax Reference

Managing Director: Welmoed Spahr

Lead Editor: Steve Anglin

Technical Reviewer: Michael Thomas

Coordinating Editor: Mark Powers

Copy Editor: Karen Jameson

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotiona use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary material referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`.

About the Author

Mikael Olsson is a professional web entrepreneur, programmer, and author. He works for an R&D company in Finland where he specializes in software development. In his spare time he writes books and creates websites that summarize various fields of interest. The books he writes are focused on teaching their subject in the most efficient way possible, by explainin only what is relevant and practical without any unnecessary repetition or theory. The portal to his online businesses and other websites is `Siforia.com`.

About the Technical Reviewer

Michael Thomas has worked in software development for over 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has over 10 years of experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and health care providers.

Introduction

The C programming language is a general-purpose, middle-level language originally developed by Dennis M. Ritchie at Bell Labs. It was created over the period 1969 through 1973 for the developmer of the UNIX operating system, which had previously been written in assembly language. The name C was chosen because many of its features derived from an earlier language called B. Whereas the B language is no longer in common use, C became and still remains one of the most popular and influential programming languages in use today.

Although C is a general-purpose language it is most often used for systems programming. This includes software that controls the computer hardware directly, such as drivers, operating systems, and software for embedded microprocessors. C can also be used for writing applications, which run c top of system software. However, it has largely been superseded in that domain by higher-level languages, such as C++, Objective-C, C#, Swift, and Java. The features of these and many other languages are heavily influenced by C, as can be seen in some of their names.

The development of C was a major milestone in computer science as it was the first widely successful middle-level language for system development. The foremost reasons for its success were that the language was concise, fast, and powerful. It offered comparable speed to assembly with far improved usability. The high-level constructs of the language allowed programmers to focus mainly on the software's design, while its low-level capabilities still provided direct access to the hardware when needed, as assembly had done. Furthermore, the language is relatively simple to understand wi few keywords and what many consider to be an elegant syntax.

Another major reason for the success of C was its portability. Unlike assembly the C language is platform independent. A standards-compliant C program can therefore be compiled for a wide variety of computer systems with few changes to its source code. Moreover, the C compiler was small and easy to port to different CPU architectures, which together with the language's popularity has made C compilers available on most computer systems.

C versions

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as K&R C. This description was succeeded in 1989 when the American National Standar Institute (ANSI) provided a comprehensive definition of C known as ANSI C or C89. In the followin year the same specification was adopted as an international standard by the International Organizatio for Standardization and became known as ISO C90 or just C90. C has since undergone three more revisions by ISO (successively adopted by ANSI) with further language extensions, including C95, C99, and most recently C11, which is the latest ANSI standard for the C programming language.

Contents

Contents at a Glance

© Mikael Olsson 2015
Mikael Olsson, C Quick Syntax Reference, DOI 10.1007/978-1-4302-6500-9_1

1. Hello World

Mikael Olsson[1]

(1) Aland, Finland

To begin programming in C you need a text editor and a C compiler. You can get both at the same time by installing an Integrated Development Environment (IDE) that includes support for C. A good choice is Microsoft's Visual Studio Community Edition, which is a free version of Visual Studio that is available from Microsoft's website.[1] This IDE has built-in support for the C89 standard and also includes many features of C99 as of the 2013 version.

Some other popular cross-platform IDEs include Eclipse CDT, Code::Blocks, and CodeLite. Alternatively, you can develop using a simple text editor – such as Notepad – although this is less convenient than using an IDE. If you choose to do so, just create an empty document with a .c file extension and open it in the editor of your choice. By convention, the .c extension is used for files tha contain source code for C programs.

Creating a Project

After installing Visual Studio, go ahead and launch the program. You then need to create a project, which will manage the C source files and other resources. Go to File ➤ New ➤ Project to display the New Project window. From there select the Visual C++ template type in the left frame. Then select the Win32 Console Application template in the right frame. At the bottom of the window you can configure the name and location of the project. When you are finished, click the OK button, and another dialog box will appear titled Win32 Application Wizard. Click next, and a couple of application settings will be displayed. Leave the application type as Console application and check th Empty project checkbox. Then click Finish to let the wizard create your empty project.

Adding a Source File

You have now created a C/C++ project. In the Solution Explorer panel (View ➤ Solution Explorer) you can see that the project consists of three empty folders: Header Files, Resource Files, and Source Files. Right-click on the Source Files folder and select Add ➤ New Item. From the Add New Item dialog box choose the C++ File (.cpp) template. Give this source file the name "myapp.c." The .c file extension will make the file compile in C instead of C++. Click the Add button, and the empty C file will be added to your project and opened for you.

Hello World

The first thing to add to the source file is the `main` function. This is the entry point of the program, an the code inside of the curly brackets is what will be executed when the program runs. The brackets, along with their content, is referred to as a code block, or just a block.

```
int main(void) {}
```

Your first application will simply output the text "Hello World" to the screen. Before this can be done the stdio.h header needs to be included. This header provides input and output functionality for the program, and is one of the standard libraries that come with all C/C++ compilers. What the `#include` directive does is to effectively replace the line with everything in the specified header befor the file is compiled.

```
#include <stdio.h>
int main(void) {}
```

With stdio.h included you gain access to several new functions, including the `printf` function tha is used for printing text – in this case to a console window. To call this function you type its name followed by a set of parentheses that includes the text string that will be displayed. The string is delimited by double quotes, and the whole statement is followed by a semicolon. The semicolon is used in C to mark the end of a code statement.

```
    #include <stdio.h>
    int main(void) {
printf("Hello World");

return 0;

    }
```

The `main` function here ends with a `return` statement, which returns a status code as the program exits. This can be useful if the intent is for your program to be executed by another program. The status code can then signal to the caller the success or failure of your program to complete its function. By convention, the return code zero is used to indicate that a program or function has executed successfully.

The C89 standard requires the `return` statement to be present, but following C90 the statement became optional. As of C90 the compiler will automatically include the `return` statement if it is omitted. For brevity the statement will be left out from future code examples.

IntelliSense

When writing code in Visual Studio a window called IntelliSense will pop up wherever there are multiple predetermined alternatives from which to choose. This window can be also brought up manually at any time by pressing Ctrl+Space to provide quick access to any code entities you are abl to use within your program. This is a very powerful feature that you should learn to make good use o

---

Footnotes

1 http://www.visualstudio.com.

© Mikael Olsson 2015
Mikael Olsson, C Quick Syntax Reference, DOI 10.1007/978-1-4302-6500-9_2

2. Compile and Run

Mikael Olsson[1]

(1) Aland, Finland

Before a program can be run, the source code has to be translated into an executable format by a compiler. This step transforms the human-readable source code into binary machine code, which is a sequence of instructions that can be executed by a computer.

Visual Studio Compilation

Continuing from the last chapter, the Hello World program is now complete and ready to be compile and run. You can do this by going to the Build menu and clicking on Debug ➤ Start Without Debugging (Ctrl+F5). Visual Studio then compiles and runs the application that displays the text in a console window.

If you select Start Debugging (F5) from the Debug menu instead, the console window displaying Hello World will close as soon as the `main` function is finished. To prevent this, you can add a call to the `getchar` function at the end of `main`. This function, included with the stdio.h header, will read a character from the keyboard and thereby prevent the program from exiting until the return key is pressed.

```
    #include <stdio.h>
    int main(void) {
  printf("Hello World");

  getchar();

    }
```

Console Compilation

As an alternative to using an IDE you can also compile source files from a terminal window as long a you have a C compiler. For example, on a Linux machine you can use the GNU C compiler, which is available on virtually all Unix systems, including Linux and the BSD family, as part of the GNU Compiler Collection (GCC). This compiler can also be installed on Windows by downloading MinGW[1] or on Mac as part of the Xcode development environment.[2]

To use the GNU compiler you type its name "gcc" in a terminal window and give it the input and output filenames as arguments. It then produces an executable file, which when run gives the same result as one compiled under Windows in Visual Studio.

```
gcc myapp.c -o myapp.exe
./myapp.exe
```

```
    Hello
           World
```

Comments

Comments are used to insert notes into the source code. They have no effect on the end program an are meant only to enhance the readability of the code, both for you and for other developers. The C89 standard featured only one comment notation, a multiline comment delimited by `/*` and `*/`.

```
    /* multi-line
   comment */
```

The C99 standard added the single-line comment, which starts with `//` and extends to the end of the line. This comment was standardized since it was a convenient feature found in many other programming languages, such as C++. Many C compilers also started to support the single-line comment long before the C99 standard was formalized.

```
    // single-line comment
```

Keep in mind that whitespace characters – such as comments, spaces, and tabs – are generally ignored by the compiler. This allows you a lot of freedom in how to format your code.

Footnotes

1 http://www.mingw.org.

2 https://developer.apple.com/xcode/.

© Mikael Olsson 2015
Mikael Olsson, C Quick Syntax Reference, DOI 10.1007/978-1-4302-6500-9_3

3. Variables

Mikael Olsson[1]

(1) Aland, Finland

Variables are used for storing data during program execution.

Data Types

Depending on what data you need to store, there are several different kinds of data types. The simple
types in C consist of four integer types: three floating-point types as well as the `char` type.

Data Type	**Size (byte)**	**Description**
`char`	1	Integer or character
`short  int  long  long long`	2 4 4 or 8 8	Integer
`float  double  long double`	4 8 8 or 16	Floating-point number

In C, the exact size of the data types is not fixed. The sizes shown in the previous table are those commonly found on 32-bit and 64-bit systems. The C standard only specifies the minimum range tha is guaranteed to be supported. The minimum size for `char` is 8 bits, for `short` and `int` it is 16 bits, for `long` it is 32 bits and `long long` must contain at least 64 bits. Most modern compilers make `int` 32 bit which nearly universally means 4 bytes. Each integer type in the table must also be at least as large a the one preceding it. The same applies to the floating-point types where each one must provide at lea as much precision as the preceding one.

Declaring Variables

Before a variable can be used, it first has to be declared. To declare a variable you start with the data type you want the variable to hold followed by an identifier, which is the name of the variable.

```
int myInt;
```

The identifier can consist of letters, numbers, and underscores, but it cannot start with a number. also cannot contain spaces or special characters and must not be a reserved keyword.

```
int _myInt32; /* allowed */
int 32Int;    /* incorrect (starts with number) */
int my Int;   /* incorrect (contains space) */
int Int@32;   /* incorrect (contains special character) */
int int;      /* incorrect (reserved keyword) */
```

Note that C is a case sensitive programming language, so uppercase and lowercase letters have different meanings.

Assigning Variables

To assign a value to a declared variable the equal sign is used, which is known as the assignment operator (=). This is called assigning or initializing the variable.

```
myInt = 50;
```

The declaration and assignment can be combined into a single statement. When a variable is assigned a value it then becomes *defined*.

```
int myInt = 50;
```

If you need to create more than one variable of the same type there is a shorthand way of doing it using the comma operator (`,`).

```
int x = 1, y = 2, z;
```

Once a variable has been defined (declared and assigned), you can use it by simply referencing th variable's name: for example, to copy the value to another variable.

```
int a = x;
```

Printing Variables

In addition to strings the `printf` function can be used to print values and variables to the standard output stream. This is done by embedding format specifiers into the string where the value is to be printed. Each specifier must be matched by a corresponding argument to `printf` of the correct type, a seen in the following example.

```
    #include <stdio.h>
    int main() {
int x = 5;

printf("x is %d and 2+3 is %d", x, 2+3);

    }
```

The `%d` specifier displays an integer of the `char`, `short` or `int` type. Other commonly used format specifiers are seen in the following table.

Specifier	**Output**
`%d` or `%i`	`char`, `short`, or `int`
`%c`	character
`%s`	string of characters
`%f`	`float` or `double`
`%Lf`	`long double`
`%ld`	`long int`
`%lld`	`long long int`
`%u`	unsigned `char`, `short` or `int`
`%lu`	unsigned `long int`
`%llu`	unsigned `long long int`
%p	pointer address

For more information on `printf` and other standard library functions, you can visit the C library reference on cplusplus.com.[1]

Integer Types

There are four native integer (whole number) types you can use depending on how large a number yo need the variable to hold. Typical ranges on 32-bit and 64-bit systems are given below.

```
char  myChar  = 0; /* -128   to +127 */
short myShort = 0; /* -32768 to +32767 */
int   myInt   = 0; /* -2^31  to +2^31-1 */
long  myLong  = 0; /* -2^31  to +2^31-1 */
```

C99 added support for the `long long` data type, which is guaranteed to be at least 64 bits in size.

```
long long myLL = 0; /* -2^63 to +2^63-1 */
```

To determine the exact size of a data type you can use the `sizeof` operator. This operator returns the number of bytes that a type occupies in the system you are compiling for. The type returned is `size_t`, which is an alias for an integer type. The specifier `%zu` was introduced in C99 as a portable wa to format this type with `printf`. Visual Studio does not support this specifier and uses `%Iu` instead.

```
    #include <stdio.h>
    int main(void) {
  size_t s = sizeof(int);

  printf("%zu", s); /* "4" (C99) */

  printf("%Iu", s); /* "4" (Visual Studio) */

    }
```

In addition to standard decimal notation, integers can also be assigned by using octal or hexadecimal notation. The following values all represent the same number, which in decimal notatio is 50.

```
int myDec = 50    /* decimal notation */
int myOct = 062;  /* octal notation (0) */
int myHex = 0x32; /* hexadecimal notation (0x) */
```

Signed and Unsigned

By default, all integer types in C are signed and may therefore contain both positive and negative values. This can be explicitly declared using the `signed` keyword.

```
signed char  myChar;   /* -128   to +127 */
signed short myShort;  /* -32768 to +32767 */
signed int   myInt;    /* -2^31  to +2^31-1 */
signed long  myLong;   /* -2^31  to +2^31-1 */
signed long long myLL; /* -2^63  to +2^63-1 */
```

If only positive values need to be stored the integer types can be declared as `unsigned` to double their upper range.

```
unsigned char  uChar;   /* 0 to 255 */
unsigned short uShort;  /* 0 to 65535 */
unsigned int   uInt;    /* 0 to 2^32-1 */
unsigned long  uLong;   /* 0 to 2^32-1 */
unsigned long long uLL; /* 0 to 2^64-1 */
```

~~When an unsigned value is printed the specifier `%u` is used for the unsigned `char`, `short`, and `int`~~ types. The unsigned `long` type is specified with `%lu` and unsigned `long long` with `%llu`.

```
unsigned int uInt = 0;
printf("%u", uInt); /* "0" */
```

The `signed` and `unsigned` keywords may be used as types on their own, in which case the `int` type is assumed by the compiler.

```
unsigned uInt; /* unsigned int */
signed sInt;   /* signed int */
```

In the same way, the `short` and `long` data types are abbreviations of `short int` and `long int`.

```
short myShort; /* short int */
long myLong;   /* long int */
```

Sized Integers

As mentioned before, the actual sizes of the integer types are implementation dependent. For more precise specification of size the C99 standard introduced a number of exact-width integer types. They can be enabled by including the stdint.h standard header.

```
#include <stdint.h>
/* Signed exact-width integers */
int8_t  iSmall;  /* 8 bits */
int16_t iMedium; /* 16 bits */
int32_t iLarge;  /* 32 bits */
int64_t iHuge;   /* 64 bits */
```

Unsigned versions of these types are available as well. Like the signed versions, these exact-widt integer types are guaranteed to have the same number of bits across all implementations.

```
/* Unsigned exact-width integers */
uint8_t  uSmall;  /* 8 bits */
uint16_t uMedium; /* 16 bits */
uint32_t uLarge;  /* 32 bits */
uint64_t uHuge;   /* 64 bits */
```

It is recommended to use sized integers when available, to more easily keep track of the range of your integer variables and to enhance the portability of your programs. Compilers that comply with standards prior to C99 may provide sized integers with different type names. Visual Studio, for example, has built-in support for the following signed exact-width integers.

```
/* Visual Studio signed exact-width integers */
__int8  iSmall;  /* 8 bits */
__int16 iMedium; /* 16 bits */
__int32 iLarge;  /* 32 bits */
__int64 iHuge;   /* 64 bits */
```

Floating-Point Types

The floating-point types can store real numbers with different levels of precision.

```
float myFloat;    /* ~7 digits */
double myDouble;  /* ~15 digits */
long double myLD; /* typically same as double */
```

The precision shown above refers to the total number of digits. A `float` can accurately represent ~~about 7 digits, whereas a `double` can handle around 15 of them.~~

```
float myFloat = 12345.678;
printf("%f", myFloat); /* "12345.677734" */
```

When printing a floating-point number you can limit the decimal places to, for instance, two in th following way.

```
printf("%.2f", myFloat); /* "12345.68" */
```

Floating-point numbers can be expressed using decimal, exponential, or hexadecimal notation. Exponential (scientific) notation is used by adding E or e followed by the decimal exponent, while th hexadecimal floating-point notation uses P or p to specify the binary exponent. Support for the hexadecimal notation was not standardized until C99.

```
double fDec = 1.23;
double fExp = 3e2;   /* 3*10^2 = 300 */
double fHex = 0xAp2; /* 10*2^2 = 40 */
```

Literal Suffixes

An integer literal (constant) is normally treated as an `int` by the compiler, or a larger type if needed t fit the value. Suffixes can be added to the literal to change this evaluation. With integers the suffix ca be a combination of U and L, for unsigned and `long` respectively. C99 also added the LL suffix for th `long long` type. The order and casing of these letters do not matter.

```
int i = 10;
long l = 10L;
unsigned long ul = 10UL;
```

A floating-point literal is treated as a `double`. The `F` or `f` suffix can be used to specify that a literal is of the `float` type instead. Likewise, the `L` or `l` suffix specifies the `long double` type.

```
float f = 1.23F;
double d = 1.23;
long double ld = 1.23L;
```

The compiler implicitly converts literals to whichever type is necessary, so this type distinction for literals is usually not necessary. If the F suffix is left out when assigning to a `float` variable the compiler may give a warning since the conversion from `double` to `float` involves a loss of precision.

Char Type

The `char` type is commonly used to represent ASCII characters. Such character constants are enclose in single quotes and can be stored in a variable of `char` type.

```
char c = 'x'; /* assigns 120 (ASCII for x) */
```

When the `char` is printed with the `%c` format specifier the ASCII character is displayed.

```
printf("%c", c); /* "x" */
```

Use the `%d` specifier to instead display the numerical value.

```
printf("%d", c); /* "120" */
```

Bool Type

C99 introduced a `_Bool` type to increase compatibility with C++. Variables of this type can store a Boolean value, which is a value that can only be either 1 (true) or 0 (false).

```
        _Bool b = 0; /* false value */
```

~~The type `_Bool` is usually accessed via its alias name `bool` defined by the standard header~~ stdbool.h. This header also defines the macros `true` and `false` as aliases for 1 and 0.

```
        #include <stdbool.h>
        bool b = true; /* true value */
```

Variable Scope

The scope of a variable refers to the region of code within which it is possible to use that variable. Variables in C may be declared both globally and locally. A global variable is declared outside of an code blocks and is accessible from anywhere after it has been declared. A local variable, on the othe hand, is declared inside of a function and will only be accessible within that function after it has bee declared. The lifetime of a local variable is also limited. A global variable will remain allocated for the duration of the program, while a local variable will be destroyed when its function has finished executing.

```
        int globalVar; /* global variable */
        int main(void) {
     int localVar; /* local variable */

        }
```

The default values for these variables are also different. Global variables are automatically initialized to zero by the compiler, whereas local variables are not initialized at all. Uninitialized loc variables will therefore contain whatever garbage is already present in that memory location.

```
        int globalVar; /* initialized to 0 */
        int main(void) {
     int localVar; /* uninitialized */

        }
```

Using uninitialized variables is a common programming mistake that can produce unexpected results. It is therefore a good idea to always give your local variables an initial value when they are declared.

```
        int main(void) {
     int localVar = 0; /* initialized to 0 */

        }
```

In C89, local variables must be declared before any other statements within their scope. The late C99 standard changed this to allow variables to be declared anywhere within a function’s scope, whic can be more intuitive.

```
        int main(void) {
     int var1;

     /* Other statements */

     int var2; /* C99 only */

        }
```
