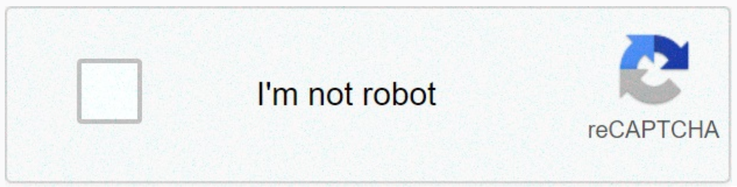

Continue

C curly brackets in string. format

To avoid braces and interpolate a string in the String.format() method, use triple braces {{{ }}}. Here's an example: using the system; EscapeDemo class { static void Main() { string val = 1, 2, 3; output string = String.Format(My values {{{0}}}, val); console. WriteLine(output); } } Output: Similarly, you can also use the string interpolation function c# instead of String.format(). use of the system; EscapeDemo class { static void Main() { row shaft = 1, 2, 3; output string = $ My Values {{{val}}}; Console. WriteLine(output); } Looking for keywords c# ? Try the Ask4Keywords string outsidetext = I'm outside bracket; Row. Format({{I'm in parentheses!}} {0}, non-text); Outputs {I'm in parentheses!:p} Format strings contain replacement fields surrounded by braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you want to include a parentheses character in text, you can avoid it by doubling: {{ and }}. The grammar for the replacement field is: replacement_field ::= { [arg_id] [: format_spec] } arg_id ::= Integer | integer ID ::= number + number ::= id 0...9: := id_start id_continue* id_start ::= a... z | And... Z | _ id_continue ::= id_start | The Less formal terms number of the replacement field can begin with arg_id, which defines the argument whose value should be formatted and inserted into the output instead of the replacement field. The arg_id is followed by format_spec, which is preceded by a colon ':'. They specify the default format for the replacement value. See also the format of the mini-language specification section. If numeric arg_ids in the format string 0, 1, 2, ... consistently, they can all be missed (not just some) and numbers 0, 1, 2, ... will be automatically inserted in this order. Named arguments can be sent to their names or indexes. Some simple examples of format strings: First, you will count {0} // Refers to the first argument Bring Me {} // The original argument from {} to {} // The same as the {0} to {1} Field format_spec contains a specification of how values should be presented, including details such as field width, alignment, padding, decimal precision, etc. Each type of value can define their own mini-language formatting or interpretation of format_spec. Most built-in types support the common mini-formatting language described in the next section. The format_spec field can also include sub-fields replacing in certain positions in it. These sub-fields can only contain an argument ID; format specifications are not allowed. This allows you to dynamically specify the formatting value. See Format section examples for some examples. The format of the specifications is used in the replacement fields contained in the format string to determine how individual values are represented (see Format String Syntax). Each type of formattable can determine how the format specification is interpreted. Most built-in types implement the following options for formatting specifications, although some formatting options are supported only by numeric types. General standard format determinant form: format_spec ::= [[fill]align][sign][#][width][precision][type] fill ::= <a character= other= than= '{',= '}'= \0'=>align ::=< |= => | = | ^ sign ::= + | - | width ::= integer | { arg_id } accuracy::= integer | { arg_id } type ::= int_type | a | A | c | e | E | f | F | g | G | p | s int_type ::= b | B | d | n | oh | x | The X fill symbol can be any character other than {, } or \0. The presence of a fill character is signaled by a character following it, which should be one of the alignment options. If the second format_spec is a valid alignment option, it is assumed that both the fill character and the alignment option are missing. The different alignment options are as follows: Value '<' forces= the= field= to= be= left-aligned= within= the= available= space= (this= is= the= default= for= most= objects).= '=>' Forces the field to be aligned to the right in the available space (this is the default value for numbers). '=' Causes the padding to be placed after the sign (if any), but in front of the numbers. This is used to print fields in the form +000000120. This alignment option is only valid for numeric types. '^' Forces the field to be centered in an accessible space. Note that if the minimum width of the field is not defined, the width of the field will always be the same size as the data to fill it in so that the alignment option does not matter in this case. The character parameter is only valid for number types and can be one of the following: The + parameter indicates that the character should be used for both positive and negative numbers. '-' indicates that the sign should only be used for negative numbers (this is the default behavior). the space indicates that the leading space should be used for positive numbers, and the minus sign should be used for negative numbers. The # parameter causes an alternate form to be used for conversion. An alternative form is defined differently for different types. This option is only valid for integer and floating point types. For integers, when using binary, eight, or hexadecimal output, this option adds a prefix of the corresponding value of 0b (0B), 0, or 0x (0X). Specifies the lowercase prefix or uppercase in the case of a typefir, for example, the 0x prefix is used for type x and 0X for X. For floating-point numbers, an alternative form causes the conversion result to always </'> </> even if the numbers do not follow it. Typically, a decimal place character appears as a result of these transformations only if it is followed by a digit. Additionally, for conversions g and G, the final zeros are not removed from the result. width is a decimal integer that determines the minimum width of a field. If not specified, the width of the field will be determined by the content. You can specify zero fill for numeric types before the width field by zero ('0'). This is equivalent to fill character 0 with alignment type =. Precision is a decimal number that indicates how many digits should be displayed after the decimal point for a floating point value formatted with f and F, or before and after the decimal point for a floating-point value formatted with g or G. For nonnumbers, the field indicates the maximum size of the field - in other words, how many characters will be used from the contents of the field. Precision is not allowed for integers, symbols, logical, and pointeral values. Finally, the type determines how the data should be presented. Available Row Presentation Types: Value Type 's' Format Bar. This is the default type for rows and may be skipped. No one is like s. Available character presentation types: Type Value c Character Format. This is the default type for characters and may be skipped. none of the same as c. Available types of entire presentations: Type Value b Binary Format. Outputs the number at the base of 2. Using the # parameter with this type adds the prefix 0b to the original value. Binary format B. Outputs the number at the base of 2. Using the # parameter with this type adds the prefix 0B to the original value. 'd' Decimal integer. Outputs the number at the base of 10. 'o' Octagon format. Outputs the number at the base of 8. format 'x' Hex. Displays a number in base 16 using lowercase letters for numbers above 9. Using the # parameter with this type adds the 0x prefix to the original value. Format 'X' Hex. Displays the number at the base of 16 using uppercase letters for numbers above 9. Using the # parameter with this type adds the 0X prefix to the original value. 'n' Number. This is the same as d, except that it uses the current locale option to insert the corresponding delimiter characters. no the same as d. An integer of presentation types can also be used with symbols and logical values. Boolean values are formatted by using a text representation, true, or false if the presentation type is empty. Available presentation types for floating point values: Value a six-day floating point format. Prints a number in a base of 16 with a 0x prefix and lowercase letters for numbers above 9. Uses p to refer to the exhibitor. A Just like a, except that it uses uppercase letters for the prefix, the numbers are above 9 and to refer to the exhibitor. 'e' Exhibitor designation. Prints the number in the scientific noting by using the letter e to specify the exhibitor. «E» Mark. Just like e, except that it uses uppercase E as a separator character. 'f' Fixed point. Displays a number as a fixed point number. F Fixed point. Same as f, but converts nan to NAN and inf in INF. g General format. For a given accuracy of p > = 1, it rounds a number to p of meaningful numbers, and then formats the result in a fixed point format or scientific designation, depending on its magnitude. Precision 0 is regarded as the equivalent of 1 accuracy. General format G. Just like g, except switches to E if the number becomes too large. Notions of infinity and NaN are also top. neither is the same as g. Floating-point formatting depends on the locale. Available presentation types for pointers: Type Value 'p' Pointer format. This is the default type for pointers and may be skipped. none of them are the same as p. This section provides examples of format syntax and comparison with printf formatting. In most cases, the syntax is similar to printf formatting, with the addition of {} and with : used instead of %. For example, %03.2f can be translated to {:03.2f}. The new format syntax also supports the new and different

options shown in the following examples. Access to arguments by position: format({0}, {1}, {2}, a, b, c); Result: format a, b, c({}, {}, {}, a, b, c); Result: format a, b, c ({2}, {1}, {0}, a, b, c); Result: format c, b, a ({0}{1}{0}, abra, cad); argument indexes can be repeated // Result: abracadabra Text alignment and width definition: format({:<30}, left= aligned);= result:= left= aligned = format({:=>30}, Aligned to the right); Result: Aligned right ({:^30}, center:*); use '*' as fill character // Result: ***********centered*********Dynamic width: format({: <{}}, left aligned, 30); Result: left aligned Dynamic precision: format({:..{} f}, 3.14, 1); Result: 3.1 Replacing %+f, %-f, and % f and specifying a sign: format({:+f}; {:+f}, 3.14, -3.14); show it always // Result: +3.140000; -3.140000 format({: f}; {: f}, 3.14, -3.14); show a space for positive numbers // Result: 3.140000; -3.140000 format({:-f}; {:-f}, 3.14, -3.14); show only the minus -- same as '{:f}; {:f}' // Result: 3.140000; -3.140000 Replacing %x and %o and converting the value to different bases: format(int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}, 42); Result: int: 42; hex: 2a; oct: 52; bin: 101010 // with 0x or 0 or 0b as prefix: format(int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}, 42); Result: int: 42; hex: 0x2a; oct: 052; bin: 0b101010 Padded hex byte with prefix and always prints both hex characters: format({:#04x}, 0); Result: 0x00 0x00 left= aligned,= 30);= result:= left= aligned= = dynamic= precision:= format({:..{} f},= 3.14,= 1);= result:= 3.1= replacing= %+f,= %-f,= and= %= f= and= specifying= a= sign:= format({:+f};= {:+f},= 3.14,= -3.14);= show= it= always= result:= -3.140000= format({:= f};= {:= f},= 3.14,= -3.14);= show= a= space= for= positive= numbers= result:= = 3.140000;= -3.140000= format({:-f};= {:-f},= 3.14,= -3.14);= show= only= the= minus= --= same= as= '{:f};= {:f}'= result:= 3.140000;= -3.140000= replacing= %x= and= %o= and= converting= the= value= to= different= bases:= format(int:= {0:d};= hex:= {0:x};= oct:= {0:o};= bin:= {0:b},= 42);= result:= int:= 42;= hex:= 2a;= oct:= 52;= bin:= 101010= with= 0x= or= 0= or= 0b= as= prefix:= format(int:= {0:d};= hex:= {0:#x};= oct:= {0:#o};= bin:= {0:#b},= 42);= result:= int:= 42;= hex:= 0x2a;= oct:= 052;= bin:= 0b101010= padded= hex= byte= with= prefix= and= always= prints= both= hex= characters:= format({:#04x},= 0);= result:= 0x00= 0x00=></{}}, left aligned, 30); // Result: left aligned Dynamic precision: format({:..{} f}, 3.14, 1); // Result: 3.1 Replacing %+f, %-f, and % f and specifying a sign: format({:+f}; {:+f}, 3.14, -3.14); // show it always // Result: +3.140000; -3.140000 format({: f}; {: f}, 3.14, -3.14); // show a space for positive numbers // Result: 3.140000; -3.140000 format({:-f}; {:-f}, 3.14, -3.14); // show only the minus -- same as '{:f}; {:f}' // Result: 3.140000; -3.140000 Replacing %x and %o and converting the value to different bases: format(int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}, 42); // Result: int: 42; hex: 2a; oct: 52; bin: 101010 // with 0x or 0 or 0b as prefix: format(int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}, 42); // Result: int: 42; hex: 0x2a; oct: 052; bin: 0b101010 Padded hex byte with prefix and always prints both hex characters: format({:#04x}, 0); // Result: 0x00 0x00 > </30},> </30},>