C Language Reference

Highlighted sections are those that might catch out an unwary Matlab programmer.

|  |  |  |
| --- | --- | --- |
| **Comments** | Single line comments:  // This is a comment  Multi-line comments:  /\*  Comment  \*/ | |
| **Organisation of a C program** | Header files:   * .h file extension. * Primary purpose: to alert the compiler to the existence of functions defined elsewhere (in other files).     C files:   * .c file extension. * Contains code. * C files are compiled separately and combined in a final stage called linking. | |
| **Preprocessor include statement** | The pre-processor runs before the main compiler. These statements instruct the pre-processor to include the contents of the named file at the point of the include statement. They are typically listed at the top of each C file before any other code.  User-defined headers (search path includes the current project)  #include “motors.h”  System headers (search path includes operating system or vendor directories)  #include <stdio.h> | |
| **Data types** | Boolean (true or false)  bool // must #include <stdbool.h> first  Floating point (single and double precision)  float  double  Integers (signed and unsigned)  int // native integer size; commonly 16 or 32 bits  short int // commonly 16 bits  unsigned int  unsigned short int  /\* after #include <stdint.h> \*/  int32\_t // the final “\_t” means “type”  uint32\_t  int16\_t  uint16\_t // and so on  Characters  char // also used as an 8 bit integer type  Strings  char \* | |
| **Declaring variables** | Variables must be declared before they can be used. The declaration tells the compiler the data type of the variable.  int counter;  float scaling;  char c; | |
| **Variable assignment** | Notice that all statements end in a semicolon.  Basic assignment operator:  counter = 0;  Increment / decrement operators:  counter++;  counter--;  Assignment with arithmetic:  counter = counter + 2;  counter += 2; // equivalent to the above | |
| **Numbers** | Decimal numbers:  42  Hexadecimal numbers:  0x42  Octal numbers: (note that the leading zero means a different number system!)  042  Binary numbers: (not standard C but supported by our compiler)  0b00101100 | |
| **Logical operators** | Equal to: ==  Not equal to: !=  Less than: < | Less than or equal to: <=  Greater than: >  Greater than or equal to: >= |
| **Logical operators** | AND &&  OR ||  NOT ! | |
| **Arithmetic operators** | Add, subtract, multiply, divide  + - \* /  Note that integer division rounds down. To get decimal results, at least one operand must be a float or double.  float a = 1/3; // returns zero since ‘1’ and ‘3’ are integers  float b = 1.0 / 3; // returns 0.33333 | |
| **Bitwise operators** | AND &  OR |  NOT ~ | XOR ^  Shift left <<  Shift right >> |
| **If statement** | The condition is always in parentheses.  If statement:  if (counter == 10) {  printf(“Done\n”);  }  If-else structure:  if (counter == 10) {  printf(“Done\n”);  } else {  printf(“Not done\n”);  }  If there’s only a single statement the braces can be omitted:  if (counter == 10)  printf(“Done\n”);  Omitting the braces means only the first statement after the if is included:  if (counter == 10)  printf(“Done\n”);  counter = 0; // NOT PART OF THE IF STATEMENT  A value of zero is considered false. Any non-zero value is considered true. | |
| **While loop (precondition)** | The condition is tested before the loop runs.  while (counter < 10) {  // . . .  } | |
| **Do-while loop (postcondition)** | The condition is tested after the loop has run once.  do {  // . . .  } while (counter < 10); | |
| **For loop** | Syntax:  for (initialiser; condition; update) {  // ...  }  Example:  int counter;  for (counter = 0; counter < 100; counter++) {  // ...  }  Sequence of evaluation:   1. Run the initialiser 2. Check the condition. If the condition fails, end the loop. 3. Run the body of the loop, i.e. the code inside the braces { } 4. Run the update code 5. Return to step 2 | |
| **Break and continue** | Break statement:  for (i = 0; i < 10; i++) {  if (i == 5) {  break; // immediately exit the loop  }  }   * In case of loops inside loops, break affects only the innermost loop.   Continue statement:  for (i = 0; i < 10; i++) {  if (i == 5) {  continue;  // jump back to top and try the next value of "i"  }  }   * In case of loops inside loops, continue affects only the innermost loop. | |
| **Function definitions**  Implements a particular function. | Syntax:  return\_type function\_name(type1 arg1, type2 arg2, ...)  {  // ...  return value;  }  Example 1 (no arguments, no return value)  void f()  {  // ...  }  Example 2 (one argument, no return value)  void f(int a)  {  // ...  }  Example 3 (no arguments, integer return value)  int f()  {  return 4;  }  Example 4 (two arguments, double precision floating point return value)  double f(double a, double b)  {  return 2\*a\*b;  } | |
| **Function prototypes / function declarations** | Declares that a function with this name and these arguments exists. The compiler needs the function prototype for compile-time checking of the data types. Usually placed in header files with the “.h” file extension.  Notice that the prototype is always followed by a semicolon whereas the definition is not.  Syntax:  return\_type function\_name(type1 arg1, type2 arg2, ...);  Examples:  void f();  void f(int a);  int f(); | |
| **Pointers** | Declaring a pointer:  uint32\_t \*p, \*q;  char \*s;  Taking the address of a variable:  p = &value;  Setting the address that a pointer points to:  p = 0x40001010;  // No \* means we are accessing the pointer itself.  Dereferencing a pointer, or accessing the memory that a pointer points to:  val = \*p; // sets val to the memory pointed to be “p”  \*p = 2; // sets the memory pointed to be “p” to be 2.  // An \* means accessing the memory that is being pointed to.  The NULL pointer:   * C specifies that the address zero is illegal. Address zero cannot be used. * Zero (in the context of pointers) is given the name NULL. * Often pointers are initialised to NULL to indicate that they do not point to anything.   int \*ptr = NULL; | |
| **Arrays** | Declaring arrays:  int array [10];  Indexing arrays:   * Starts from zero. * No bounds checking! The compiler will not stop you from indexing off the end of an array.   array[0] = 0; // sets first item to zero  array[9] = 100; // sets last item to 100  array[10] = 9000; // will overwrite another variable and cause chaos  Assigning arrays   * Arrays are not first class objects in C! * You cannot assign to them like variables.   int a [10];  int b [10];  a = b; // WRONG! Does not copy 10 items from b to a!  // Correct approach: write a for loop that copies elements one by one. | |
| **Link between arrays and pointers** | Array variables are pointers.  int values [10];  Here, ‘values’ is an int\* (pointer) that points to the first element in the array.  A function expecting a pointer can be given an array without an address-of operator:  // Given a function prototype:  int sum(int \*array, int number\_of\_elements);  // Call it like so:  int values [10];  sum(values, 10); // notice: no & operator needed | |
| **Strings** | * C does not have a proper string type. * Instead, it has arrays of chars. * C strings are NULL-terminated, meaning that a zero byte indicates the end of the string.   Consider:  char \*message = "Hello world";  The compiler:   1. Arranges for the string “Hello world” followed by a null (zero) byte to be placed in read-only memory. 2. Creates a variable ‘message’ of type char \*. 3. Initialises ‘message’ to point to the first character in the string (‘H’).   Strings are character pointers so assignment statements do not copy strings! Instead, an assignment to the pointer will change where it points. | |
| **Structures** | Structures are variables with named sub-fields.  Defining a structure:  struct motor\_setting {  bool forward;  int speed;  };  // Defines a data type but not a variable.  Declaring a structure variable:  struct motor\_setting setting1;  // variable declaration. Allocates memory.  Accessing structure fields  setting1.forward = true;  setting1.speed += 2;  Pointers to structures   * Pointers to structures often arise for efficiency. It is faster to pass a pointer instead of copying all the fields inside a structure.   void increment\_speed(struct motor\_setting \*s) {  s->speed += 2;  // alternative syntax: (\*s).speed += 2;  // s.speed would be an error here!  }  // Call it like this:  increment\_speed(&setting1); | |
| **Preprocessor macro definitions** | Define a macro without giving its expansion (useful for #ifdef conditions, see below)  #define USE\_SERIAL\_COMMS  Define a macro with replacement text. Causes the pre-processor to replace every instance of the macro with the replacement text.  #define MAX\_SPEED 100  Undo a macro definition:  #undef USE\_SERIAL\_COMMS | |
| **Preprocessor conditions** | If a macro is defined:  #ifdef USE\_SERIAL\_COMMS  // ...  #endif  If a macro is not defined:  #ifndef USE\_SERIAL\_COMMS  // ...  #endif  Else block: (optional)  #ifdef USE\_SERIAL\_COMMS  // ...  #else  // ...  #endif | |
| **Header file guard** | Ensures that a given header file is included only once so that the compiler does not see duplicated definitions.  Method 1: pre-processor definitions  #ifndef MOTORS\_H\_  #define MOTORS\_H\_  // Different people use different conventions of underscores before/after the filename.  #endif  Method 2: #pragma definitions on modern compilers  #pragma once | |
| **Const qualifier** | Const indicates a constant value.  const float pi = 3.14159;  // a floating point value that cannot be modified  void send\_string(const char \*msg);  // This function promises not to modify the memory pointed to by ‘msg’. | |
| **Static qualifier** | Static has multiple uses.  Outside of functions static means the object (variable or function) is localised to the current C file. By default, global variables can be accessed by other C files in the same project. Static prevents this.  // At the top level of a file, outside of a function:  static int fault\_count = 0; // Other files cannot see this variable  static void check\_inputs(); // Other files cannot call this function  Inside of functions static means the variable is statically allocated and hence its value is preserved across multiple calls to the function. This is the same as Matlab persistent variable.  void receive() {  static int count\_messages;  // All static variables start at zero  count\_messages++;  // ...  } | |
| **Volatile** | Volatile indicates that reads or writes may have side-effects and cannot be eliminated by the optimiser.  Two main uses:   1. As an interface to memory-mapped hardware where reading or writing triggers a response from the hardware. 2. For sharing variables between threads / tasks, where another section of code may concurrent modify the underlying variable.   Example:  int \*p = /\* address of something \*/;  \*p = 10; // This line is redundant. The compiler may remove it.  \*p = 20;  volatile int \*p = /\* address of something \*/;  \*p = 10; // Because the pointer is volatile, the compiler is forced to keep this line.  \*p = 20; | |
| **Extern** | Extern indicates that a global variable is defined in a different C file.  // file1.c  // top level, outside any function  int fault\_count = 0;  // file2.c  // top level, outside any function  extern int fault\_count; | |
| **Typecasting** | Typecasting means converting from one data type to another.  Place the new data type in parenthesis in front of the value to be cast. Example:  int numerator = 1;  int denominator = 3;  float ratio = (float)numerator / denominator;  // without the typecast, would perform integer division that rounds to zero | |