C Programming Language Overview

Author: Eric Laroche

This paper may serve as a short overview of the C programming language.

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C wears well as one's experience with it grows. -- *K&R2* [*The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie] [1]

C *stands* for effectiveness of language, good style, sound design. [1] C typically uses a *compiler* [2]. C is *case-sensitive* [in its keywords and identifiers [3]].

It is recommended to *skip* this text's *footnote texts* on the *first sweep*.

[1] [*The C Programming Language,* Brian W. Kernighan, Dennis M. Ritchie, Prentice Hall, 2nd Ed. 1988, ISBN 0-13-110362-8] [4]
[2] A *compiler* is the tool [program] to translate a [higher-level] [programming] language [to a lower-level language, often into object files [5]].
[3] *Identifier case-sensitiveness* is not guaranteed for the link [6] phase.
[4] For other *C standards* [ISO, ANSI], see *links*.
[5] An *object file* is the output of a *compiler* [2], an *assembler* or a similar tool, usually [machine [language]] code that is *native* to a processor [or alternatively *byte-code* for some *abstract processor architecture* [that is usually *interpreted* on software that is called a *virtual machine*]].
[6] *Linking* is the process of generating a [binary] executable, a shared library, a re-linked object [file] [5] or similar, from *object files* [5], *libraries* [7], start-up object code [and possibly additional resources] [8].
[7] A *library* [in the narrower sense] is a collection of compiled [2] source code files [so called *object files* [5]].
[8] The *link process* is usually *opaque* to a programmer, controlled by the *compiler driver program* [9].
[9] The *compiler driver* [program] steers all phases [10] of compilation, from source code to binary.
[10] The original *compile phases* were *preprocessing, compiling, optimizing, assembling, linking*.

Source code representation

C programming language source code is typically represented in *files* [11]. These *C code source files* typically carry a *file suffix* of `.c` [12] [13] [14]. A C file is called a *module* or *compilation unit* [15].

C source code files do not have [mandatory] header fields [such as a preamble e.g. in the first source code line] [16].

Interfaces are often represented in *header files*, that typically carry a file suffix of `.h`. Such *interfaces* consist of stuff that is used *inter-modular*, i.e. *between* different C source code files [17] [18] [19].

[11] *Development environments* may chose to represent sources [20] in another way, e.g. in some kind of *repository* [21].
[12] Other file suffixes are not usual [22], although compilers often support alternatives by offering *language-specifying compiler options*.
[13] The *C compiler [driver program]* needs *input type information*, to know what to *do* (compile [23], assemble [24] [25] , or link).
[14] *Build systems* (such as `make`) typically expect well-known file suffixes too, to deduct file types without file contents lookup [26].
[15] *Compilation units* are *compiled* [2] *independently* from other compilation units.
[16] There are no `/etc/magic` entries for *C*; the `file` tool deducts C code by heuristics only.
[17] The header file (interface) inclusion mechanism is a general one, i.e. one can include any file (text, code).

[18] These *interface files* can be used to specify *build dependencies* in software build environments. Large systems may compile faster if the interfaces are sufficiently fine granular [27].
[19] *Header files* may represent interfaces of a whole library [7] [facade pattern].
[20] Not just C sources.
[21] *Repositories* may allow a more *fine granular code resolution* (e.g. function-granular) with more *meta information* (such as modification-author and comments) or just be *faster* than with file system based data accesses. *Repositories* may be implemented as a *file system layer abstraction* [28], which makes it easy to use generic editors and build tools.
[22] Unlike with the *suboptimally* [29] *chosen* `.C` for the *C++ programming language*, that led to [more compatible] alternatives such as `.cc`, `.cxx`, `.cpp`, etc.
[23] Compiling possibly *different* languages.
[24] Assembling with or without prior [C-] preprocessing.
[25] The original compile *phases* [10] had assembly sources as an *intermediate form*.
[26] [C] *file contents interpretation* is *not an easy thing* to do [16]; `make` wouldn't be able to deduce a C file's contents.
[27] The focus lies however on providing *optimal interface layer abstractions*, not compilation speed.
[28] Most environments (typically the operation systems) allow the use of *generalized file systems*, typically by means of a *network file system* interface.
[29] The `.C` file suffix [*note the upper case*] is suboptimal with *case-insensitive filesystems*.

Types

The *C programming language* is a *quite* [30] [31] *strongly typed* [32] [33] language. This means that *[data] types, variables and functions must* [34] *be declared* [35] before their use.

[30] For historical reason, the use of undeclared functions is still supported [36].
[31] *Type aliases* (as defined by typedefs) may be used interchangeably; boolean and integer values are treated interchangeably [37].
[32] C is a *typed* language, but not so much a *type-centered* language (such as C++, which implements means to *protect* data accesses).
[33] *Strongly typed* implies *type safe* [38].
[34] Functions *should* be declared before used [30].
[35] *Note* the difference between *declaration* and *definition* [39]. A data *definition* actually reserves space for the data, whereas a data *declaration* [40] introduces *variable name and type* only. A function *definition* provides the function's implementation, whereas a function *declaration* [41] introduces a *name and type* only. *Type definitions* do not per se generate code or data [42], so a distinction to a 'type declaration' (such as with functions and variables) is not needed; however an *incomplete data type* [43] [44], may be considered 'less than a definition'. Since a *definition* is an *implicit declaration* too, the programmer must ensure the explicit declaration is seen at definition location, to ensure *interface consistency*.
[36] The use of undeclared functions generally generates *compiler warnings* though.
[37] There is no *explicit* boolean data type, `int` is used. This may lead to possibly subtle bugs.
[38] *Type safe* means that a lot of problems with types are caught in the compilation stage.
[39] A *definition* is an *implicit declaration*, i.e. it is a *declaration in the broader sense*.
[40] *Data declarations (variable declarations)* are done using the `extern` keyword [41].
[41] *Function declarations* do not require [45] the `extern` keyword as syntactical means, since definition and declaration (implementation and prototype) are distinguished by the former's *code block*.
[42] Unlike with the [more complex] C++ language, whose type (class) definition *may* implicitly generate code and/or data, e.g. a virtual function table, a default constructor, runtime type information [helper data], etc.
[43] *Incomplete data types* are introduced by typedefs on undefined structs [46]. Such a construct introduces a type *name* only, that can be used as pointer (reference) only [44].
[44] *Incomplete data types* are used as an advanced decoupling (modularization) feature [bridge pattern].
[45] Function declarations (prototypes) *can use* a `extern` keyword though.
[46] E.g. `typedef struct T_st T;`

Code and data

C source code mainly consists of *code and data, and type definitions* [mentioned in the previous chapter].

Type definitions appear *early* in a source code file or even separated in a header file [47], since they glue parts of code together and are needed by *both parts*.

Code is bound to *function names*, which are globally or module-wide [48] visible [49].

Data is [at least temporarily [50]] bound to *variables*. Data may be bound to fields of some *other data structure* [which itself must be bound to such fields or a variable [51]].

Lexical stuff is considered *comments* and *spacing* [52]. *Comments* are made of *freely formatted text* inside `/* ... */` [53].

A *simple preprocessor* supports *macros* [54], *file inclusion* [55] and *conditional compiling* [56].

47 *Header files* need to be included sufficiently early [57].
48 *Module* means a single C source code file, so module-wide means file-wide.
49 C does not support *anonymous* functions [lambda expressions].
50 E.g. temporarily bound to a `auto` variable and then returned as value.
51 If this reference *chain* breaks [usually by a software bug], *memory* will *leak* [58].
52 *Spacing* is usually discussed by *coding rules*.
53 Comments cannot be *nested* [59].
54 `#define`, with or without macro parameters, `#undef`.
55 `#include`
56 `#if`, `#endif`, `#elif`, `#else`, `#ifdef`, `#ifndef`.
57 Function or data definitions *must* have seen their declarations too, to ensure *consistency*.
58 C usually does not include *garbage collecting* that could clean up such *memory leaks*.
59 `#if` *preprocessor directives* can be *nested* and can be used to *comment-out code* [60].
60 E.g. inside `#if 0 ... #endif`.

Functions

A C function (procedure, routine, method) has one entry point and possibly several exit points [61]. A function establishes a *code block*, which is delimited by a pair of *braces* (`{ }`) [62].

Any code block can have its own set of *local variables* [63] (which can overwrite (hide) other occurrences of the same name). A function can have argument variables [64] (*parameters*), which are always passed *by-value* [65]; *by-reference* can be implemented by using *pointers* to variables [66].

Function definitions cannot [and need not] be nested [67].

The *user function* that is called at application startup is `main` [68].

61 *Several function exit points* are implemented by *multiple* `return` statements.
62 Sample: `int isqr(int i) {return i * i;}`
63 Such *local variables* can be *automatic* (stack based; exclusive to a stack frame) or *static* (data segment based; typically process-wide shared) [69] [70].
64 C also supports *variable number of arguments* [71] [72].
65 Calls *by-value* enable the programmer to use *any expressions* as *function arguments* [not just variable references [73]].
66 'Calls' *by-name* can be implemented by using *macros*.
67 *Function nesting* would be used to further confine function scope [however, the existing file scope should be narrow enough, especially with small source files], and allow outer-level variable access [which however would weaken data encapsulation].

[68] `int main(void) {...}` or `int main(int argc, char** argv) {...}` or `int main(int argc, char** argv, char** envp) {...}`.
[69] Such a *static variable* is also known as *singleton [pattern]*.
[70] *Static variables* are not per-se *thread-safe*.
[71] As in `int printf(const char*, ...);`, where the first [format-] parameter specifies what follows.
[72] The *variable arguments* however are not *type-checked*.
[73] Not allowing *calls-by-value* probably would imply lots of not-so-elegant [temporary] variables.

Program flow

C is designed to be a *terse programming language* (i.e. able to express *much* on a few lines or pages), so the number of *program flow keywords* is small [74] [75].

Program flow in a function is from top to bottom, executing statement by statement (a statement is terminated by `;` or it is a *compound statement* (a block) in braces), unless one of the following constructs is encountered.

Loop constructs are done with `while` [continuation test at the beginning of the loop], `do` [continuation test at the end of the loop] [76] and `for` [continuation test at the beginning of the loop; additional initializer code and per-loop code [77]]. *Loops* can be left [78] or short-cut by `break` and `continue` (and of course too by `return` and `goto`).

Alternative code flows are done by `if` and `else` [79].

`switch`/`case`/`default` (and `break`) can be used when comparing to *compile-time constants* [80] [81].

`return` is used to *leave a function context*; `goto` allows *arbitrary jumps* inside *function code* [82].

[74] See *links* for a list of the C programming language keywords, with a few comments.
[75] *Keywords* cannot be *reused* as identifiers [they're truly *reserved* words].
[76] `do {...;} while (...);` is about the same as `for (;;) {...; if (!(...)) break;}`.
[77] `for`'s *per-loop code* is usually used for *incrementers*, e.g. in `for (i = 0; i < n; i++) {...;}` [being roughly [83] equivalent to `i = 0; while (i < n) {...; i++;}`].
[78] However, there's no *multilevel-break* [84].
[79] *Specially nested* `if`/`else` lists [*multi-way decisions*] are typically '*linearized*' in C code, e.g. `if (...) {...;} else if (...) {...;} else {...;}` instead of `if (...) {...;} else {if (...) {...;} else {...;}}` [all but terminal else-case do *not* establish nested blocks].
[80] The `switch` statement historically [85] allowed a faster dispatch [86].
[81] An annoyance with the `switch` statement is the *reuse* of the `break` keyword, for which reason one cannot [by `break`] *leave an enclosing loop*.
[82] A `goto` may not be used to jump from one function to another [87].
[83] Apart from behavior if `continue` is used.
[84] Can be done by `goto`.
[85] More recent analyzers/optimizers can quite reasonably handle and take advantage of *expression constantness* in `if`/`else` statements [and elsewhere] too.
[86] E.g. *jump table driven* dispatch.
[87] C is versatile enough to allow *library code* [88] being able to implement out-of-context jumps, e.g. with `longjmp`.
[88] `longjmp` and similar functions are found in [more general] library code [89] *rather than in C source code*, since they e.g. modify *special* [90] CPU registers, which can't be done in *C* [91].
[89] *Libraries* [7] can be of *mixed-language*; the *standard C library* e.g. often contains some [platform-specific] assembly code compilations [92].
[90] In `longjmp` most notably the *stack pointer register* [93], to *adjust stack frames*.

91 But in [platform-specific] *assembly* code.
92 Either to implement things that can't be done in C, such as `longjmp` or `long long` number type multiplication [helper functions], or to provide performance-optimized implementations, e.g. for `memcpy`.
93 In *stack-based* environments; [for very tiny environments [e.g. on small embedded systems]] a C environment can be implemented *without stack*.

Primitive data types

C defines *integer numbers* of different sizes: `char` [94], `short [int]`, `int` [95], `long [int]`, `long long [int]` [96] [97], which can *e.g.* be found to be of sizes 1, 2, 4, 4, 8 bytes [98] [depending on *platform* and *compilation model* [99]]. Those integer types are (with exception of `char` [100]) implicitly `signed`, i.e. *operations* consider the numbers to carry a sign bit [101]). The `unsigned` modifier changes this behavior.

C defines *floating point numbers* of different sizes: `float`, `double`, `long double` [96] [97], which can e.g. be found to be of sizes 4, 8, 16 bytes. Floating point number types do not support `unsigned`.

Implicit (where applicable) and *explicit* conversion rules between these number types are defined.

`void` is used to specify an un-specified reference type [102], an empty function parameter list [103] [104] or a cast on a unused expression [105].

94 `char` is mainly used for [ASCII [106]] strings, i.e. [readable] *text* used in programs.
95 `int` *was* assumed to fit the target CPU's native *register size*.
96 The pattern here is to *re-apply* the `long` attribute.
97 Not defined on every platform.
98 C doesn't define the *exact* integer number sizes. However it defines that `long` is at least as large as `int`, which is at least as large as `short`.
99 A compiler [environment] may allow e.g. different data pointer sizes, to allow larger or more compact applications. In a larger model [107] either the full register size [instead of e.g. half] is used, or two registers are used for addressing [108] or calculating.
100 `char`'s `signed`/`unsigned` status is *implementation specific*. *Explicit casts* [109] should be used where *sign matters*.
101 Usually the most significant bit, in *two's complement* representation.
102 `void*` [110]
103 As e.g. in `int main(void) ....`
104 Leaving the function parameter list *empty*, as in `int main() ...`, as opposed to specifying it `void`, was historically used to leave the parameter list *unspecified* [111].
105 As e.g. in `(void)printf("hello, world\n");`, to indicate one is not interested in a function call's *return value* [but only in its *side-effect(s)*].
106 The alternative *unicode* is supported by a larger, derived data type, `wchar_t` [often a `[unsigned] short` (size 2 bytes)].
107 Such *models* can either be *mixed* [in an application or library [7], or different-model libraries can be used together], in which case often additional data modifier keywords [112] are provided, or such memory models can't be *mixed* [and hence e.g. can only run in *different* processes].
108 *Two register addressing* likely requires *segmented* memory layout.
109 E.g. in `printf("%02x", (int)(unsigned char)c);`, to suppress unwanted sign extension.
110 Which of course cannot be dereferenced.
111 A *unspecified* function parameter list was [historically] used to just specify the function's *return value type*, not its parameter types.
112 Such *proprietary [platform specific] keywords* typically start with _ or __ [113].
113 Which makes them *system-internal* identifiers or keywords [by C definition [114]].
114 To separate at least the *namespace* of the *C environment implementation* from the user namespace.

Composite data types

C allows *data structures* (`struct`), *overlapping* data structures (`union`), enumerated *integer constants* (`enum`) and *type name aliases* [115] (`typedef`). Composite data types are, together with pointers and arrays, called *derived data types*.

[Data] definition modifiers are `auto` [116] (non-static (stack) local variable), `const` (read-only variable or variable's read-only contents [117]) [118], `extern` (data *declaration* instead of *definition*), `register` (hint to the optimizer), `static` (non-stack data or non-globally visible), `volatile` (anti-hint to the optimizer).

115 Type aliases can be defined on primitive and composite types.
116 `auto` is the *default* storage class for *local variables*.
117 Depending on `const`'s position [119].
118 `const` pointers in *function parameters* indicate that contents will not be modified, i.e. *treated read-only* [interface contract]. `const` modifier keyword on *static data* will likely locate that data in a *read-only data segment*.
119 E.g. `char const* p` [120] vs. `char* const p` [vs. `char const* const p`] [121].
120 `const char* p` is a synonym to `char const* p`, i.e. only `const`'s position relative to the indirection specifiers is relevant.
121 There is a possible `const` modifier per *indirection*.

Operators

arithmetic: `+ - * / %` [addition, subtraction, multiplication, division, modulus] [122] [123]
bit: `& | ^ ~ << >>` [and, or, exclusive-or, not, shifts]
logical: `&& || !` [and [124], or [124], not]
relational: `== != < <= > >=` [equal, not equal, ...] [125]
assignment: `= *= /= %= += -= <<= >>= &= |= ^=` [126]
data selection: `. -> []` [data member selection, with prior referencing [127], array element selection]
referencing/dereferencing: `& *`
function call: `()`
explicit conversion: `()`
increment/decrement: `++ --` [128]
size of an expression/type: `sizeof`
ternary operator: `?:`
sequence operator: `,` [129]

The operators have 15 levels of precedences [130] and left- or right-associativities.

The order of sub-expression evaluation in binary operations is not defined; exceptions are the logical operators [124] and the sequence operator.

122 The arithmetic operators are defined for number types, whereas plus and minus are defined for *pointer arithmetic* [131] too.
123 Mathematically associative operators are not treated *computationally associative* for reasons of [intermediate] *overflow* and *rounding*.
124 *Logical and and or* establish *sequence points*, which makes their syntax more useful [132].
125 The relational operators are defined for number types and pointer types [in which case the memory positions [addresses] are compared].
126 The *assignment expressions* have a value too [133] [134].
127 `p->m` can be written as `(*p).m` or `p[0].m`, `->` is more convenient.
128 C defines *pre-* and *post* increment/decrement [to allow even terser expressions].

[129] The *sequence operator* is a *convenience*; to allow several sub-statements where syntactically one statement is expected. It allows *elegant* solutions.
[130] See *links* for a list of the C programming language operators ordered by precedences levels.
[131] `&(p[n])` can be written as `p + n`.
[132] *Useful syntax* by allowing expressions such as `if (p == NULL || *p == '\0') ....`
[133] E.g. `a = b = c;` being `b = c; a = b;` [left associative].
[134] This is part of *C's orthogonality*.

Runtime library

The runtime library required to implement selfcontained programs is tiny. [1] Standard library functions are only called explicitly [135] [136].

Even some *fundamental* functionalities used in programming, such as *input/output* [137] [138] [`stdio.h`: `fopen`, `printf`, ...], *string handling* [139] [`string.h`: `strcpy`, `strcmp`, ...], *conversions* [`stdlib.h`: `strtol`, `strtod`, ...], *dynamic* [140] *memory* [141] [`stdlib.h`: `malloc`, `free`, ...], are *provided in the standard library*, not in the C language [in the narrower sense] itself.

Operating system interfaces are provided syntactically the same way library functions are [142].

[135] This *explicitness* allows *easy* control over C code's performance.
[136] E.g. *structure assignments* may trigger `memcpy`.
[137] I.e. there is no *built-in* input/output; it's just an API [application programming interface].
[138] *Standard input and output* are known as `stdin` and `stdout` [also declared in `stdio.h`]; there's an additional *output channel* known as the standard error output, `stderr` [usually initially mapping to the same as `stdout`].
[139] The *string handling* routines [143] are samples of functions that may be known to a *compiler* as *intrinsics* [inlineable functions] and expanded [directly] into [optimized] code [144].
[140] *Dynamically acquired* [at run-time only; not reserved from application start] and *dynamically sized* [data/buffer size not known at compile-time].
[141] Dynamic memory is bound to *reference (pointer) variables*; dereferencing allows access to the *dynamic storage* [the *heap memory*].
[142] Interfaces such as `open` are e.g. implemented as *stubs* that e.g. map to an interrupt/trap with an appropriate *system call* number.

[143] Together with *memory copying* functions, such as `memcpy`.
[144] E.g. `strlen("hello, world\n")` could be translated into the *compile-time constant* `13` [i.e. function call left away].

Links

Frequently asked C/C++ questions [on lrdev]
: Provides answers to ISO/IEC 9899 availability, useful entry points for C programming language information, C programming language keywords, etc.

C programming language coding guidelines
: C's do's and don't's.

C/C++ operator precedences
: Precedences of the C/C++ programming languages operators.

Eric Laroche, laroche@lrdev.com, Mon Apr 26 2004