C

Lots of it, and more!

Table of Contents

Introduction

This presentation assumes that you have some familiarity with programming. It also assumes you're running on Linux. It aims to provide the details needed to understand how to write C, from the simplest programs to relatively complex ones. The limited time available necessitates skimming over much of the material presented here.

C is a compiled, low level programming language. It is old (circa 1972), but the de-facto language used to implement interpreted languages such as Python, Bash etc; and for the majority of programs requiring low-level, fast code.

Being a compiled language, it is an order of magnitude faster than interpreted languages, because interpreted languages are interpreted by a run-time engine (written in C!). Whereas C's interpretation is done at compile time and the result is executed by the CPU, eliminating run-time interpretation.

Oh, and C is ubiquitously portable, running on just about every processor. Which minimises your need to understand multiple CPU architectures when writing portable programs.

C evolves, slowly and with some reluctance for change given its age and extensive usage. Releases aim to maintain solid backward compatibility, and which version of C you choose to use depends on your expected platforms. C89/90 is old but reliably portable. C99 has added a modest number of nice features. C11 is the latest standard and is not yet fully implemented. The default for gnu C is C11, but you can choose which C version you want gcc to conform to.

C is to an extent ill-defined even with the attempted conformance of various standards applied to it. The temptation to extend its basic facilities has been too much for C vendors, making it all too easy to write a C program on one platform and then find it doesn't compile or run ok on another. And the utilities used to compile, link, run and debug your program may vary too. So, before you start programming, try to have an idea of what system(s) you'll want your program to run on, and code appropriately. Your knowledge of C needs to be very good to write a fully portable program of any significant size. And expect to do lots of testing and revision as you go along.

Unusually, but usefully, C is essentially a symbiosis of two languages: pure “**C**”; and the “**C preprocessor**” **cpp**. Preprocessing is performed automatically.

A simple C example

Here's “hello world” in C. First, create file “`hello.c`” containing code:

```
#include <stdio.h>
int main () {
  printf ("Hello world!\n");
}
```

Then build `hello.c` to produce an executable `hello` (**cc** is the name of the C compiler):

```
cc -o hello hello.c
```

Then run it:

```
./hello
```

That outputs to the window where you typed `./hello`:

```
Hello world!
```

File “`hello.c`” **includes** C library file **`stdio.h`**, which contains the C declarations of various standard input/output functions (such as **`printf`**) and associated bits and pieces. Such “header files” as `stdio.h` are located in some system-dependent standard place such as under **`/use/include/`**.

The **C preprocessor** gets first dibs on the C code and processes lines whose first non-space character is **`#`**. To it, **`#include`** means: “include the contents of the specified file here”. The preprocessor does literally copy the contents of `stdio.h` into the code at that point.

The **`int main ()`** declares a function called **`main`**, which is necessary in all programs because `main` is the “starting function” for the execution of the whole program.

The function has a “body part” containing the code to be executed when `main` is called, and that is: **`{printf("Hello world!\n");}`**

The **`\n`** added to the string is the new-line character – `printf` does not automatically issue a new line.

The **`cc`** command invokes the **compiler** and **linker**. The **`-o hello`** option causes it to generate an executable file called **`hello`**, and the supplied argument **`hello.c`** gives it the name of the file to compile. Invisibly, that file is compiled to assembler (the kind of assembler is specific to the kind of processor), which is automatically assembled to a temporary **`hello.o`** “**object file**”. Then the **`hello.o`**, and **`stdio.o`** (located in some system-dependent standard place but typically within the C standard library **`libc.a`** or **`libc.so`**) are linked to form the **`hello`** **executable**.

Language composition

The textual language is defined at the lowest level in terms of the **lexis/lexicon/lexical** structure. The C language consists of:

keywords	`int  for  while  void  return  break`
symbols (operators and delimiters)	`=  >=  (  *  >=  +=`
names (aka identifiers)	`a  Fred  radius  b9  small_parts`
literals (numbers, characters, strings)	`44  5.6  'a'  "abc"`
separators (space, tab, newline)	
comments	`/* Some comment. */` (can be spread over multiple lines) or `// Some comment.` (ends at the end of the line)

An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Keywords

The 44 C11 (i.e. 2011) language keywords.

auto	A now-redundant storage class.	
break	Break out of a loop.	`break;`
case	Used in labeling a statement that is one of a **switch** statement's multi-way branches for the **switch** value.	`case 65:  c = 'A';` `case 'A':  c = 'A';`
char	An integer type that is typically a byte (8-bits), representing one character.	`char a_char = 'w';` Sets `a_char` to 119 (i.e. Ascii w)
const	Type qualifier for a constant.	`const a_const = 44;` You can't change `a_const` afterwards.
continue	Jump to the end of a loop, triggering the next iteration.	`continue;`
default	Labels a statement within a **switch** statement, that is jumped to if none of the **case** labels match the **switch** statement's value.	`default:  c = ' ';`
do	Starts a **do** loop.	`i = 1;` `do` `  printf ("i: %d\n", i++);` `while (i <= 4);`
double	Type qualifier indicating double-precision (64-bit) floating point.	`double f_dbl = 1.0 / 3;`
else	The **else** part of an if statement.	`if (n > 4)` `  gt_4 = 1;` `else` `  gt_4 = 0;`
enum	Used to define an enumeration type.	`enum names{Bob, Sally};` Equivalent to: `int Bob = 0, Sally = 1;` Or choose your own numbers: `enum names{Bob=12, Sally=20};`
extern	Indicates something is externally defined.	`extern int ext_int;` The `ext_int` is defined in an external file.
float	Type qualifier indicating single-precision (32-bit) floating point.	`float f_sgl = 1.0 / 3;`

for	Starts a **for** loop.	`for (i = 1; i <= 4; i++)` `  print("i: %d\n", i);`
goto	Jump to a label	`goto fred;` `...` `fred: x = 12;`
if	Starts an **if** statement.	`if (n > 4)  printf("n gt 4\n");`
inline	Request that the specified function be inlined to save the overhead of a function call.	`inline int freq_func() { ... }`
int	An integer type (usually 32-bits)	`int i = 2147483647;`
long	Type qualifier indicating a longer integer type - usually twice as many bits. However, on Linux sizeof(int) == sizeof(long int) and you need **long long int** to get a 64-bit integer.	`long long int ll_int = 9223372036854775807;` `// 2 ** 63 - 1`
register	Storage class encouraging variable to be held in a CPU register.	`register int oft_used;`
restrict	Optimization aid indicating that a specific pointer is the only one to access some specific structure.	`void* memcpy (void*       restrict s1,` `              const void* restrict s2,` `              size_t               n);`
return	Exits from a function and optionally returns a value.	`return;` `return 42;`
short	Type qualifier indicating a shorter integer type - usually half as many (16) bits.	`short int sh_int = 32767;`
signed	Type qualifier indicating the integer type is signed (the default).	
sizeof	An expression operator (actually not a function) providing the size in bytes of a type or variable.	`nbytes = sizeof(int);  // nbytes is 4.`
static	Storage class making a variable global to the file.	`static int in_file;`
struct	Defines a structure.	`struct person {char *name; int age;};` `struct person Bob = {"Bob", 42};` `print("Bob's age: %d\n", Bob.age);`
switch	Starts a **switch** statement.	`switch (i) {` `  case 1:  printf("i is 1\n");  break;` `  default:  printf("i not 1\n");` `}`
typedef	Defines a new type derived from some other type.	`typedef unsigned short int ushint;` `ushint x = 65535;`

union	Same as struct except all members are overlaid.	`union a_union {int f; int f_too};` `union a_union same_adr;` `same_adr.f = 42;  // same_adr.f_too is also now 42.`
unsigned	Type qualifier indicating the integer type is unsigned.	`unsigned short ush_int = 65535;`
void	Explicitly indicates no value.	`void a_fn(void);` `// (Function with no return value, no parameters)` `void *p;` `// Makes p a no-type pointer. Each usage must` `// then explicitly state its type:` `p = (int *) &x;  // (Where x is an int)` `*(int *)p += 2;  // Adds 2 to x.`
volatile	Tells compiler that variable may change unexpectedly. Only reliable use is for memory-mapped hardware.	`volatile short *vol_p = (short *) 0x1000000;` `// (0x1000000 is mapped to a 16-bit hardware` `// register.)`
while	Starts a **while** loop.	`while (i <= 4)  printf("i: %d\n", i);`
_Alignas		
_Alignof		
_Atomic		
_Bool		
_Complex		
_Generic		
_Imaginary		
_Noreturn		
_Static_assert		
_Thread_local		

Notes:

Operators

The following is taken from http://en.cppreference.com/w/c/language/operator_precedence :-

The following table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(`*`type`*`){`*`list`*`}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*`type`*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of[note 1]	
	`_Alignof`	Alignment requirement(C11)	
3	`* / %`	Multiplication, division, and remainder	Left-to-right
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13[note 2]	`?:`	Ternary conditional[note 3]	Right-to-Left
14	`=`	Simple assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Comma	Left-to-right

1. ↑ The operand of `sizeof` can't be a type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
2. ↑ Fictional precedence level, see Notes below

3. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to **?:** is ignored.

When parsing an expression, an operator which is listed on some row will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it. For example, the expression *p++ is parsed as *(p++), and not as (*p)++.

Operators that are in the same cell (there may be several rows of operators listed in a cell) are evaluated with the same precedence, in the given direction. For example, the expression a=b=c is parsed as a=(b=c), and not as (a=b)=c because of right-to-left associativity.

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b	a[b] *a &a a->b a.b	a(...) a, b (type) a ? : sizeof _Alignof (since C11)

Language concepts and structures

Iteration Stmt	There are 3 kinds of loop statements, allowing for repeated execution of statements.	`for (`*`init`*`;` *`cond`*`;` *`step`*`)` *`stmt`* `// General` **`for`** `loop` `for (;;)` *`stmt`* `// Infinite loop` `while (`*`cond`*`)` *`stmt`* `//` **`while`** `loop` `do` *`stmt`* `while (`*`cond`*`);` `//` **`do`** `loop`
Jump Stmt	There are 4 kinds of jump statements. These provide various ways to exit loops and functions, or goto a specific labeled statement (mostly deprecated).	`break;` `continue;` `return` *`expr`*`;` `return;` `goto` *`label`*`;`
Selection Stmt	Here's where you select between various code paths.	`if (`*`icond`*`)` *`stmt`* `if (`*`icond`*`)` *`stmt`* `else` *`stmt`* `switch (`*`expr`*`)` *`stmt`*
Labeled Stmt	You label a statement if you want to be able to jump to that statement.	*`identifier`*`:` *`stmt`* `// So you can` **`goto`** `that stmt.` `case` *`const-expr`*`:` *`stmt`* `//` **`case`** `stmt, used in a` **`switch`**`.` `default: stmt` `//` **`default case`** `stmt.`
Compound Stmt	When you want to have consecutive statements.	`{`*`stmts`*`}`
Expression Stmt	As in mathematical expressions, but with esssential differences to facilitate the programming world.	`x = y + z * 3;` `my_value += 44;`
Statement	Any of the above Stmts. A *stmt* always has a terminating semicolon.	
Function	Similar to mathematical functions, but with esssential differences to facilitate the programming world. You can reference a function before it is defined by **declaring** it first.	`void fmul (double a, double b);  // Just declare fmul` `int main (int argc, char **argv) {` `  printf ("Number of arguments: %d\n", argc-1);` `  (void)fmul (1, 2);` `  return 0;` `}` `void fmul (double a, double b) {  // Define function fmul` `  printf ("%f * %f is %f\n", a, b, a * b);` `}`
Type definition	Defines a new type derived from some other type.	`typedef unsigned short int ushint;` `ushint x = 65535;`
Variable declaration	A variable must be declared before it is used. This specifies the variable's type. Its value is undefined until a value is assigned to it.	`int x;` `int y = 4 + x;  // Not a good idea if x is undefined!`
Record structure	Holds multiple values. Note: you can pass a struct to a function, and return a struct.	`// Define struct person:` `struct person {char *name; int age;};` `// Define Bob of type struct person:` `struct person Bob = {"Bob", 42};` `print("Bob's age: %d\n", Bob.age); // Struct member access`

		`struct my_bits {unsigned int three_bits : 3;` `  unsigned int two_bits : 2;}` `// (A 3-bit and a 2-bit field. You can only store 0-7 in` `// three_bits, and 0-3 in two_bits. But, overall, it fits` `// n 5 bits of space, so is very compact.)`
Record union	Holds one of a selection of values.	`union a_union {int f; int f_too};` `union a_union same_adr;` `same_adr.f = 42;  // same_adr.f_too is also now 42.`
Array definition	Holds a list of values of the same type. Note: the name of an array is a pointer to its first element.	*`type name`*`[`*`length`*`];  // General 1D form (uninitialised).` `int ary[] = {1, 2, 3};  // Create and init array.` *`type name`*`[`*`length1`*`][`*`length2`*`];  // 2D form (uninitialised).` `double ary2D[2][3] = {{1.0, 1.1, 1.2}, {1.3, 1.4, 1.5}};`
Array indexing	Indexing starts at 0. There are no bounds checks.	`v = ary[1];  // Set v to the 2nd element of ary, i.e. 2` `v2 = ary2D[0][2];` `// (sets v2 to the 0,1 element of ary2D, i.e. to 1.2)`
String indexing	Analogous to one-dimensional array indexing.	`char my_char = "abc"[1];  // Define and set my_char to 'b'.` `char *my_str = "abc";  my_char = my_str[1];` `// (Sets my_char to 'b').`
Pointers	Access by reference. A pointer is typically a 64-bit address.	`int my_int = 43;` `int *pmy_int = &my_int;  // pmy_int address of my_int.` `*pmy_int = 44;  // my_int is now 44.` `pmy_int = NULL;  // pmy_int is now a null pointer` `// (A null pointer points to address 0).` `struct person *pperson = &Bob;` `// (Sets pperson to pointer to the Bob structure.)` `int age = pperson->age;`
…		

Note: The **goto** statement can easily lead to obfuscated code and so is generally deprecated except when used to jump to a block of statements at the end of a function, e.g.:

```
#include <stdio.h>

int ProcessFile(char *filename) {
  int result = 1;

  FILE *fd = fopen(filename, "r");
  if (!fd)  goto cleanup;  // fd is NULL, so file failed to open.

  ... Code to process the file contents goes here – each detected processing error issues: goto cleanup ...


cleanup:
  if (fd)  fclose(fd);  // fd is not NULL, so close the opened file.
  return result;
}
```

C Libraries

There are many C libraries. Some are used frequently and it is essential to be familiar with those. It is also generally useful to know of the other libraries, and to research what external libraries may be available for any particular development project.

The GNU C Library provides a rich set of libraries. Some of the libraries, such as **stdio**, **stdlib**, **string** are essential for just about any program. See here for information on the standard libraries.

To make use of the variables and functions defined in a library file, you have to include it in your program. You include the header (.h) file. When you perform the link step, you need to then ensure that the body (.o or .so file) for each library file is linked in when building the resultant executable program.

It's typical to have the includes grouped togther at the top of each file that needs them. E.g.:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

C Preprocessor directives

The C preprocessor is a standard precursor to compilation. Its directives are executed prior to compilation so don't have any runtime impact. The preprocessor analyses the file, finds preprocessor directives, and immediately executes them. It does not evaluate the C code, but it does evaluate preprocessor expressions.

`#include`	Used to include code from other files. The code is literally copied from the referenced file.	`#include <`*`lib_file`*`>  // Include code from library file` `#include "`*`my_file`*`"  // Include code from your file.`
`#if`	Analogous to the C **`if`** statement.	`#define DEBUG 1` `#if DEBUG` *... <code to enable debug> ...* `#endif`
`#else`	Analogous to the C **`else`** statement.	`#define DEBUG 1` `#if DEBUG` *... <code to enable debug> ...* `#else` *... <non-debug specific code> ...* `#endif`
`#elif`	Allows multiple else parts (no C analogy).	`#define DEBUG 1` `#if DEBUG` *... <code to enable debug> ...* `#elif DEBUG == 2` *... <code to enable extensive debug> ...* `#else` *... <non-debug specific code> ...* `#endif`
`#endif`	Ends **`#if`** statements.	
`#ifdef`	An “if defined” if statement.	`#ifdef DEBUG` *... <code to enable debug> ...* `#endif`
`#ifndef`	An “if not defined” if statement.	`#ifndef DEBUG` *... <code to disable debug> ...* `#endif`
`#define` *(Object-like)*	**`#define`** *`identifier tokens`* Used to create “macros”. Defines a preprocessor identifier. Preprocessor identifiers are completely separate from C identifiers. The tokens supplied must be valid C tokens, but need not be valid C syntax, as far as the preprocessor is concerned. Bracketing variable names in *tokens*, plus the whole substitution, is not required but is **strongly** recommended to avoid any unexpected C interpretation of the preprocessed	`#define X  (1 | 2)` `printf("X: %d\n", X);  // Prints X: 3`

	end result.	
#define *(Function-like)*	**#define** *identifier*() *tokens* There must be no spaces before the parentheses. As for object-like, but used in a function-like manner.	`#define MIN(A,B) (((A) <= (B)) ? (A) : (B))` `v = MIN(100, v);`
#undef	Undefines a preprocessor identifier if it is defined.	`#undef DEBUG  // Undefine DEBUG if it is defined.`
#pragma	Platform specific.	`#pragma GCC warning "look out!"` `// Issues a warning messsage.`
#error	Issues an errro message.	`#error "That's wrong!!"  // Issues an error message.`
#line	#line directives are generated by programs that generate C code from some higher-level language. They generate #line directives that give the originating file and line number. Errors reported would then identify the originating file and line.	`#line 7 “higher.hlc”` *...<Generated C code from line 7 of file higher.hlc>...*
#	“Stringize”. Used within a function-line **#define** directive to turn a parameter into a string.	`#define stringify(v) #v` `printf(“v, stringified: %s\n”, stringify(v))` `// That prints: v, stringified: v`
##	Performs “token pasting”. This merges two tokens into one whilst macro-expanding.	`#define cmd_prefix(v) cmd_##v` `int cmd_prefix(fred) = 5;` `printf(“Just set cmd_fred to: %d\n”, cmd_fred);`
defined	The only preprocessor keyword. Useful if, in one preprocessor directive, you want to check if two preprocessor identifiers are defined.	`#if defined(BIG_INTS) & defined(SMALL_INTS)`

Note: You can pass in a preprocessor define or value to a compilation command by using the -D flag, e.g.:
gcc -o sketchy -DDEBUG sketchy.c
That causes the preprocessor to see DEBUG as defined.
Note: It is the general convention to use all uppercase for preprocessor identifiers.

Predefined preprocessor macros

__FILE__	Path string of current input file.	`"/usr/local/include/string.h"`
__LINE__	Current line number.	`44`
__DATE__	11-character date string	`"Jan  1 2017"`
__TIME__	Time string.	`"15:30:01".`
__STDC__	1 usually! meaning the compiler is ISO standard C.	`1`
__STDC_VERSION__	The year and month of the C standard's version number.	`201112`
__STDC_HOSTED__	1 if complete C standard library is available.	`1`
__cplusplus	Defined if C++ compiler is in use, giving the language standard's year and month.	`201703`
__OBJC__	Defined if is Objective-C.	

__ASSEMBLER__	Defined when processing assembler language.	

Strings and string formatting

A string is a sequence of characters (normally, one-byte ASCII characters). It is represented in C by enclosing the string of characters in double quotes, e.g.:

```
char *my_string = "I'm a string\n";
```

Note that the type of a string is: `char *`

Note: You can change characters of a string, but you cannot change characters of a string literal. E.g.:

```
my_string[2] = 'M';
```

may or may not fail in practice, but should not be done because it's trying to change a string literal, which the C compiler/linker may choose to store in read-only memory.

You can insert a small selection of non-printable characters by use of backslash escapes, e.g. the backslash n in the above example inserts the newline character.

Adjacent string literals will be automatically joined together for you as though they were one long string. E.g.:

```
"abc"
"def"
```

is the same as:

```
"abcdef"
```

Strings are null terminated, i.e. terminated by a zero character '\0' (i.e. a byte of zero). C doesn't store the length of a string, so to determine the length of some string, count its characters from the start of the string and stop when you encounter a '\0' character.

It is very useful, particularly when printing values, to have good string formatting capabilities. I.e. to have powerful, easy-to-use features for parameterising strings and for then formatting the parameterised values in typically conventional ways, e.g. financial values in a spreadsheet.

String formatting consists of embedding **format specifiers** within a string, where the format specifier allows you to parameterise part of that string. E.g. the following string has some fixed text, and some variable text (*myname* and *myage).*

```
"My name is myname, and I am myage years old."
```

The values to be substituted are passed to the formatting function (printf) along with the string to be formatted.

Since programming languages don't provide niceties such as italics, there has to be some convention whereby a format specifier can be embedded in a string and recognised as a format specifier. At the same time, it is useful to enhance the format specifier to provide useful mechanisms for formatting ("converting”) the inserted variable values.

The printf style formatting

The general format specifier is a multi-character sequence:

`%FW.PLT` where *F*, *W*, *P*, *L* and *T* are parts of the format specifier.

`%`		Introduces the **start** of the format specifier.
F	Optional	Conversion flags. Characters from the set: `# 0 - +` and *space*.
W	Optional	Minimum field width, or `*` to indicate the field width is supplied as one of the values. E.g. `10`
.P	Optional	Precision, or `.*` to indicate the precision is supplied as one of the values. E.g. `2`
L	Optional	Length modifier, one of: `h l L`
T		Simultaneously marks the **end** of the format specifier and provides the conversion specifier, one of: `d i o u x X e E f g G n s p %`

See http://en.cppreference.com/w/c/io/fprintf for a description of the conversion specifiers. (And including the various printf-style functions.)

Python's basic usage of the printf format is quite similar to that of C:

C	`printf("STR%10.2fING", 012.345);`	`STR     12.35ING`
Python	`print("STR%10.2fING" % 012.345)`	`STR     12.35ING`

Almost identical, except that Python uses the `%` operator which eliminates the need for an explicit formatting function such as printf.

Examples:

`printf("a:%d b:%.2f c:0x%X\n", 44, 5.123, 127);`	`a:44 b:5.12 c:0x7f`

C file input/output

Discussion here relates to Gnu C on Linux/UNIX, but other C's, and Windows and other operating systems, are typically quite similar.

There are two ways to do i/o in C: the low-level **File descriptor**; and the high-level **Stream**.

File descriptors

See https://www.gnu.org/software/libc/manual/html_node/#toc-Low_002dLevel-Input_002fOutput

This is a primitive, low-level i/o interface that may sometimes be appropriate to use.

A File descriptor is just a simple integer. The file descriptor integer is in fact an index into an array of descriptors (structs), where a struct contains useful fields relating to a "File descriptor", such as current file position.

There are three file descriptors defined in `<unistd.h>` for input, output and error i/o: **`STDIN_FILENO`**, **`STDOUT_FILENO`** and **`STDERR_FILENO`**. These have values 0, 1 and 2 respectively.

Files are "system-wide", and may be accessed simultaneously by multiple processes. There is necessarily a single system-wide table holding information for each currently-accessed (i.e. "open") file. Another field of the file descriptor struct will be for referencing into that system-wide table.

Example program to read from a file whose path is supplied as an argument, and to write the contents of the file to stdout. Such a program would need to be enhanced with more complete error handling etc. if it is to be used with confidence in all situations. Writing code to correctly handle i/o opeerations at the primitive file descriptor level is not straightforward. I haven't included error reporting here because you'd generally always use stream i/o for that and I don't want to mix descriptors and streams in this example!

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

int main (int argc, char **argv) {
  char buf[1024];
  ssize_t bytes_read;
  ssize_t bytes_written;
  int fd;

  fd = open (argv[1], O_RDONLY);
  if (fd < 0)  return EXIT_FAILURE;

  // Loop reading 1024 bytes and writing the bytes until no more (which indicates either EOF or a read error).
  for (;;) {
    bytes_read = read (fd, buf, 1024);
    if (bytes_read < 0)  return EXIT_FAILURE;
    if (bytes_read == 0)  break;  // EOF

    // Write all the bytes read. If writing is interrupted then keep trying.
    do {
      bytes_written = write (STDOUT_FILENO, buf, bytes_read);
      if (bytes_written < 0)  return EXIT_FAILURE;  // (A more "complete" implementation would check for EINTR interrupt
and continue here.)
      bytes_read -= bytes_written;
    } while (bytes_read > 0);
  }

  close (fd);
  return 0;
}
```

Streams

See https://www.gnu.org/software/libc/manual/html_node/I_002fO-on-Streams.html#I_002fO-on-Streams

Streams provide a high-level interface that is generally used in preference to file descriptors.

The "handle" to a stream is held as a pointer to a FILE type, where FILE is a struct containing useful fields relating to the stream.

There are three streams defined in `<stdio.h>` for input, output and error i/o: **`stdin`**, **`stdout`** and **`stderr`**. These are of type **`FILE *`**.

Streams are implemented on top of file descriptors.

Example program to read from a file whose path is supplied as an argument, and to write the contents of the file to stdout. It is straightforward to code this, but some care does need to be taken in getting a line of input and storing it prior to output, to ensure no “buffer overflow” is possible.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

static size_t buf_size = 0;            // The current size of the line buffer (it could increase if we encounter longer
lines).
static char*  pbuf_start = NULL;       // Pointer to the start of the current line buffer (it could change).

int main (int argc, char **argv) {
  FILE *fs;

  fs = fopen (argv[1], "r");
  if (!fs) {
    printf ("ERROR %d: Failed to open file: \"%s\". %s.\n", errno, argv[1], strerror(errno));
    return EXIT_FAILURE;
  }

  // Loop getting and printing lines until getline returns -1 (which indicates either EOF or a read error).
  while (getline (&pbuf_start, &buf_size, fs) != -1) {  // Gnu C getline safely handles lines of any length.
    printf("%s", pbuf_start);
  }

  if (!feof(fs)) {  // getline returned -1 but it's not EOF, so getline must have detected an error:
    printf ("ERROR %d: Failure during read of file: \"%s\". %s.\n", errno, argv[1], strerror(errno));
    return EXIT_FAILURE;
  }

  fclose (fs);
  return 0;
}
```

Building programs

The simple case

The simplest C program is just one file with includes of various standard libraries, which can be simply built and executed. E.g.:

Create a file `hello.c` using a text editor containing the four lines:

```
#include <stdio.h>
int main () {
  printf ("Hello World!\n");
  return 0;
}
```

At the terminal, type:

```
cc -o hello ./hello.c
./hello
```

The first line builds an executable `hello` and the second line runs it. The output is:

```
Hello World!
```

The general case

A larger and more sophisticated program is best split into multiple files, where the content of a file is code related to some distinct aspect of the overall program.
For each aspect it is conventional to have two files: a header (e.g. `logging.h`) and a body (e.g. `logging.c`). The header contains definitions of variables and functions that need to be accessible by code external to the body. The main file, containing the `main` function, is perhaps best organised to not need a header file.

As there will be multiple files that need to be compiled, it is typical to use the `make` program to manage (re-)compiles and (re-)links of files, to produce the end executable(s).

The overall development process is then:

- Create `make` build instructions in a file called `Makefile`
- Edit `*.h` header files and `*.c` body files.
- Run the `make` program to perform a build.
- Run and test the built program(s).

Repeat steps as necesssary to fix problems and improve the program(s).

Use make to automate builds

Use **make** to automate rebuilding your program(s). This requires a **makefile** describing the build steps. Using **make** saves much time over manually re-typing build commands.

There is a nice description of **makefiles** on wikipedia.

And a comprehensive description in the GNU make manual.

Build steps in detail

Starting with file.c containing C code you've created, these are the processing steps to turn it into an executable program :-

Preprocess

The C preprocessor parses file.c, looking for and obeying preprocessor directives. Each directive transforms part of the written code. The end result is a preprocessed file.

```
file.c -> preprocessed_file.c
```

Note: You can use the `-E` option to see the preprocessed result, which will be output to stdout, or to a file specified by the `-o` option.
Note: ONLY the preprocessing phase is performed if you specify `-E`.

Compile

The C compiler analyses the C code and generates processor-specific assembler for it, creating an assembler file.

```
preprocessed_file.c -> compiled_file.s
```

Note: You can use the -S option to see the generated assembler, which will be output to `file.s` (AND to any file specified by the `-o` option.)
Note: ONLY the preprocessing and compilation phases are performed if you sepecify `-S`.

Assemble

The assembler for the operating system processes the assembler code to an object file of byte-code machine instructions and data.

```
compiled_file.s -> assembled_file.o (“object” files)
```

Note: You can use the `readelf` utility to dump the generated object file in a readable format.

Link

The linker processes all object files into one executable file. This mostly involves fixing up addresses for references between object files.

```
assembled_file.o* -> executable_file
```

The linker inserts "startup" code specific to the C language that will set up the C environment prior to calling `main`.

The end result is a "loadable" executable file. I.e. a file that requires minimal further processing for actual execution (obeying the byte-coded instructions contained in the executable file).

Static versus Dynamic linking
Linking can be performed statically or dynamically.

A static link pulls all referenced code into the generated executable_file.

A dynamic link doesn't pull in the library code - it is just referenced from the executable. The references are to “shared objects”, that is files that have been assembled in a way that makes them shareable at execution time. Those files are created by the assembler and given the `.so` extension.

In general, dynamic linking is preferable to static linking, and dynamic linking is the default.

Note: You can use the `readelf` utility to dump the generated executable file in a readable format.

Run

The operating system employs a loader program to read an executable_file and establish it in memory. It does that by memory-mapping the pages (a page is 4096 bytes) of the executable. The executable's contents were arranged by the linker so that the contents start at 4096-byte disk byte boundaries.

The loaded program is then executed, under control of the operating system which maintains separation between multiple running programs. Since there are usually many more running programs than CPUs available, the programs are time-sliced across the CPUs. Pages are pulled into memory as and when required for execution.

Debug

The GNU debugger **gdb** provides comprehensive debug facilities.

Processing command line arguments

Processing command line arguments ("options") is only done once per program, but generally every program written needs it. Accordingly, there is a library `unistd.h` available to assist. See here for: info on the getopt function used to do the processing of successive arguments; and link to an exampe of how to use it. The getopt function assumes that named arguments are of the form:

> *-letter value*

E.g.:

> -b my_arg

Whether a value is needed or not depends on how you process the letter.

As an alternative, Gnu C recommends using getopt_long. This function is defined in the getopt library, not in unistd.h. It facilitates processing "long-named options", i.e. arguments of the form:

> *--argument*

where argument can be a name or: *name=value*

Abbreviations of the name are allowed provided thay are unique. Some --argument forms may allow a short form synonym, e.g. --help and -h would generally work the same way.

Both getopt forms allow non-named arguments, which are simply arguments not preceded by any hyphens. These should be placed after all hyphenated arguments.

References

GNU C

GCC, the GNU Compiler Collection
The GNU C Reference Manual
The C Preprocessor
The GNU C Library
Extensions to the C Language Family
GNU make (not just for C)
GNU debugger **gdb**

Other useful C references

cppreference.com
Learn C Programming In Simple Steps
C Tutorial (tutorialspoint)
C Tutorial (Cprogramming.com)
Interactive C Tutorial (learn-c.org)
Learn C Programming
The C Book
Wikibooks: A Little C Primer (lots of examples)
Wikibooks: C Programming
Wikipedia C
Library Cheat Sheets
C Reference Card
Learn X in Y minutes Where X=c

Raspberry Pi GPIO C code

RPi_GPIO_Code_Samples#C

Interesting articles on building programs

Building programs - the detail
Shared libraries

Appendix A - Example code

List a Directory (from gnu.org)

Defining, declaring, calling and using functions.

```
#include <stdio.h>

// Define function somefunc1:
char *somefunc1 (int x) {
  printf ("somefunc1 %d\n", x);
  return "ok";
}


// Declare function somefunc2:
char *somefunc2 (int);

// Declare function somefunc3.
// NOTE that the braces around somefunc3 have no effect.
char *(somefunc3)(int);


// Call the functions:
int main () {
  somefunc1 (5);
  somefunc2 (7);
  somefunc2 (9);
  return 0;
}


// Define function somefunc2:
char *somefunc2 (int x) {
  printf ("somefunc2 %d\n", x);
  return "ok";
}


// Define function somefunc3:
char *somefunc3 (int x) {
  printf ("somefunc3 %d\n", x);
  return "ok";
}
```

Function pointers

```
#include <stdio.h>


// Define function somefunc1:
char *somefunc1 (int x) {
  printf ("somefunc1 %d\n", x);
  return "ok";
}


// Declare function somefunc2:
char *somefunc2 (int);

// Declare function somefunc3.
// NOTE that the braces around somefunc3 have no effect.
char *(somefunc3)(int);


// Define pointer psomefunc.
// This pointer can point to any function that takes an int parameter and returns a char *.
// What makes this a pointer to a function is the ...(*...)(...) syntax.
// The pointer-to-a-function definition is VERY similar to a function declaration but with the
// named part being enclosed in (*...)
char *(*psomefunc)(int);


// Call the functions:
int main () {
  somefunc1 (3);
  somefunc2 (5);
  somefunc3 (7);

  psomefunc = &somefunc1;  // Sets psomefunc to address of somefunc1.
  (*psomefunc)(9);  // Call the function that psomefunc points to.

  psomefunc = somefunc1;  // C allows you to omit the &, and does the sensible thing.
  (*psomefunc)(9);  // Call the function that psomefunc points to.

  *psomefunc(9);  // The braces around the function pointer aren't needed.

  psomefunc(9);  // Also, the dereference isn't needed either - C does the sensible thing.

  return 0;
}
```

```
// Define function somefunc2:
char *somefunc2 (int x) {
  printf ("somefunc2 %d\n", x);
  return "ok";
}


// Define function somefunc3:
char *somefunc3 (int x) {
  printf ("somefunc3 %d\n", x);
  return "ok";
}
```

Function typedef pointers

```
#include <stdio.h>


// Define function somefunc1:
char *somefunc1 (int x) {
  printf ("somefunc1 %d\n", x);
  return "ok";
}


// Declare function somefunc2:
char *somefunc2 (int);

// Declare function somefunc3.
// NOTE that the braces around somefunc3 have no effect.
char *(somefunc3)(int);


// Define a type called PSOMEFUNC which is a pointer to any function that
// takes an int parameter and returns a char *.
typedef char *(*PSOMEFUNC)(int);

// Define pointer psomefunc.
// This pointer can point to any function that takes an int parameter and returns a char *.
// What makes this a pointer to a function is the ...(*...)(...) syntax.
// The pointer-to-a-function definition is VERY similar to a function declaration but with the
// named part being enclosed in (*...)
// (If you're using typedef, then you need to look at the typedef definition.)
PSOMEFUNC psomefunc;


// Call the functions:
int main () {
  somefunc1 (3);
  somefunc2 (5);
  somefunc3 (7);

  psomefunc = &somefunc1;  // Sets psomefunc to address of somefunc1.
  (*psomefunc)(9);  // Call the function that psomefunc points to.

  psomefunc = somefunc1;  // C allows you to omit the &, and does the sensible thing.
  (*psomefunc)(9);  // Call the function that psomefunc points to.

  *psomefunc(9);  // The braces around the function pointer aren't needed.

  psomefunc(9);  // Also, the dereference isn't needed either - C does the sensible thing.
```

```
  return 0;
}


// Define function somefunc2:
char *somefunc2 (int x) {
  printf ("somefunc2 %d\n", x);
  return "ok";
}


// Define function somefunc3:
char *somefunc3 (int x) {
  printf ("somefunc3 %d\n", x);
  return "ok";
}
```

Variable arguments list

```
// Program to demonstrate a simplistic printf.
// Provides a neat interface.

#include <stdio.h>
#include <stdarg.h>


// The allowed type indicators passable to function MyPrint.
enum types {Tint, Tstr};


// Expects two parameters: a "type" indicator; and a value of that type.
int MyPrint (int type, ...) {
  va_list  args;
  int      retval;
  int      i;
  char     *s;

  va_start (args, type);

  switch (type) {
  case Tint:
    i = va_arg (args, int);
    retval = printf ("%d\n", i);
    break;
  case Tstr:
    s = va_arg (args, char *);
    retval = printf ("%s\n", s);
    break;
  default:
    printf ("ERROR: Unknown type: %d\n", type);
    retval = -1;
  }

  va_end (args);
  return retval;
}


#define Try(stmt)  if ((result = (stmt)) < 0)  {return (result);}

int main() {
  int result;

  Try (MyPrint (Tint, 45));
  Try (MyPrint (Tstr, "Hi"));
  Try (MyPrint (99, 4.6));
```

```
  Try (MyPrint (Tint, 46));

  return 0;
}
```