ENCE260 C summary

Electrical and Computer Engineering
University of Canterbury

September 15, 2011

[*This is an abridged summary of some of the more common C language features and standard library functions. The ISO C standard is the authoritative document—ISO/IEC 9899:1999 (E).*]

1 Data types

The predefined C data types (in order of precedence for type conversions) are:

`long double`	Extended precision float
`double`	Double precision float
`float`	Single precision floating point
`unsigned long long`	Unsigned doubly long integer
`long long`	Doubly long integer
`unsigned long`	Unsigned long integer
`long`	Long integer
`unsigned int`	Unsigned integer
`int`	Integer
`unsigned short`	Unsigned short integer
`short`	Short integer
`unsigned char`	Unsigned character
`char`	Character

- The size of each data type is implementation dependent. The number of bytes required to store a data type can be determined with the `sizeof` operator (by definition `sizeof (char) = 1`). For example, on an ATmega8 `sizeof (int) = 2` whereas on on a 32-bit PC `sizeof (int) = 4`.

1.1 Fixed size data types

The system header file `stdint.h` defines a number of data types of fixed size, for example:

`int8_t`	8 bit signed integer
`uint8_t`	8 bit unsigned integer
`int16_t`	16 bit signed integer
`uint16_t`	16 bit unsigned integer
`int32_t`	32 bit signed integer
`uint32_t`	32 bit unsigned integer

1.10 Integer data type ranges

- Unsigned integer data types of N bits can represent integers within the range $[0, 2^N - 1]$.
- Signed integer data types of N bits can represent both positive and negative integers in the range $[-2^{N-1}, 2^{N-1} - 1]$ using two's complement where the most significant bit is set if the number is negative.

For example, consider $N = 3$ bits. This can represent $2^3 = 8$ different integers:

Bit pattern	Unsigned value	Signed value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

10 Constants

Integer constant A number of type `int`. There are three bases:

Decimal Contains only the digits 0–9 and does not start with a 0, for example, 42.

Octal (base 8) Contains only the digits 0–7 and starts with a 0, for example, 042.

Hexadecimal (base 16) Contains only the digits 0–9, letters a–f (or A–F) and prefixed by 0x (or 0X). For example, `0xcafe101`.

If the constant is followed by a u (or U) suffix it is considered to have type `unsigned int`, for example, `42u`. If the constant is followed by a l (or L) suffix it is considered to have type `long`, for example, `42l`. If it has both u and l suffixes it is considered type `unsigned long`. Long long is indicated by an `ll` (or `LL`) suffix.

Double constant A number containing the digits 0–9 and either a decimal point and/or the character `e` (or `E`) to denote a number in scientific notation. For example, `1.0`, `4.2e-1`, `1e3`. These have type `double`.

Float constant Similar to double constants but with an f suffix, for example, `1.0f` and `1e3f`. These have type `float`.

Character constant A character, or escape sequence prefixed with a \, within single quotes. For example, `'a'`, `'\n'`, `'\0'`. These have type `int`.

String constant Zero or more characters within double quotes. For example, `"Don't panic!"`. These are null terminated arrays of type `char`.

11 Variables

Local variables can only be used in the block below where they are declared. They have no default value.

Persistent local variables are local variables prefixed with the `static` qualifier. Unlike local variables their contents are retained.

File variables are defined outside of a function and can be used in any function below their definition. They have a default value of zero. They need to be prefixed with the `static` qualifer otherwise they have global scope.

- All variables must start with a letter and can contain any combination of upper and lower case letters, digits, and the underscore character. Variable names are case sensitive. For example,

```
int thing1;
double shoe_size;
float DalekCount;
```

- It is a convention that variable names are not all uppercase since this is used for constants defined with `#define` and `enum`.

- When a variable is used in an expression (except on the left hand side of an assignment) it refers to its value stored in memory.

100 Arrays

- Arrays are a collection of homogeneous variables. Array elements are accessed by an integer starting from 0. For example, `int things[42]` requests the compiler to allocate memory to hold 42 integers of type `int`.

- Unlike variables, the name of an array refers to its starting address, not its value. So when an array is an argument to a function a copy of the array address is passed to the function as a pointer, not a copy of the array. Similarly, arrays cannot be copied using the assignment operator. Instead each element must be copied using a loop, for example,

```
int a[4] = {2, 3, 5, 7};
int b[4];
int i;

for (i = 0; i < 4; i++)
    b[i] = a[i];
```

  or using the system library function `memcpy`:

```
memcpy (b, a, sizeof (a));
```

101 Pointers

- Pointers are variables that can hold the memory address of another variable, array, or pointer.

```
int variable;
int array[10];
int *pointer;

pointer = &variable;
pointer = array;
pointer = &array[4];
```

  Note the subtle difference between obtaining the address of a variable, an array, and an array element. In the case of an array the & operator is not required.

- Pointers are mostly used for function parameters that receive array addresses and for linking data structures to create lists and trees.

110 Memory qualifiers

const indicates that memory contents are not modified after initialisation.

volatile indicates that memory contents may spontaneously change, for example, a hardware register.

111 Operators

Operators perform operations on operands (variables, constants, expressions).

111.1 Arithmetic operators

-	Negation
*	Multiplication
/	Division (truncates for integer operands)
%	Modulo division (remainder), integer only
-	Subtraction
+	Addition

111.10 Bitwise operators

~	Bitwise not (one's complement)
&	Bitwise and
|	Bitwise or (inclusive-or)
^	Bitwise xor (exclusive-or)
<<	Bitwise left shift
>>	Bitwise right shift

111.11 Logical operators

!	Logical *not* operator
&&	Logical *and* operator
||	Logical *or* operator.

These operators return a value of 1 if true or 0 if false. The && operator will short circuit the rest of the expression if the first operand is zero and the || operator will short circuit the rest of the expression if the first operand is non-zero.

111.100 Relational operators

==	Equal
!=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

These operators return a value of 1 if true or 0 if false.

111.101 Assignment operators

=	Assignment
+=	Add with assignment
-=	Subtract with assignment
*=	Multiply with assignment
/=	Divide with assignment
%=	Modulo with assignment
&=	Bitwise-and with assignment
|=	Bitwise-or with assignment
^=	Bitwise-exclusive-or with assignment
<<=	Left shift with assignment
>>=	Right shift with assignment
||=	Logical-or with assignment
&&=	Logical-and with assignment

Here `a += 5` is equivalent to `a = a + 5`.

111.110 Miscellaneous operators

&	Address of
*	Dereference
.	Member of
(type)	Cast to type
->	Dereferenced member of
? :	Conditional

```
    y = (x >= 0) ? x : 0;
```

is equivalent to

```
    if (x >= 0)
        y = x;
    else
        y = 0;
```

111.111 Operator precedence

The precedence of the C binary operators from highest to lowest is:

```
*   /   %
+   -
<<  >>
<   <=  >   >=
==  !=
&
^
|
&&
||
```

For example, 3 + 5 * 7 is evaluated as 3 + (5 * 7). If in doubt, always use parentheses.

- The associativity of binary operators is left to right, so a - b + c is equivalent to (a - b) + c.
- The associativity of unary, ternary, and assignment operators is right to left, so -*p is equivalent to -(*p).

1000 Control structures

Control structures alter the evaluation of program statements.

1000.1 Statements and blocks

A *statement* is an *expression* followed by a semicolon. Statements can be grouped into *blocks* using curly braces {}. Local variables can be declared at the start of any block for use within the block only.

1000.10 While loops

While loops are used to repeat statements while an expression is non-zero (true). They have the form:

```
while (expression)
   statement_or_block
```

1000.11 Do-while loops

Do-while loops are used to repeat statements until an expression is zero (false). They have the form:

```
do statement_or_block
while (expression);
```

1000.100 For loops

For loops are used to repeat statements, usually by incrementing a counter. They have the form:

```
for (expr1; expr2; expr3)
    statement_or_block
```

Generally expr1 is an initialisation expression, expr2 is a relational expression to determine when the loop terminates, and expr3 is an increment expression.

1000.101 If-else

The if-else construct is used to express decisions by testing if an expression is non-zero (true). Formally, the if-else statement looks like:

```
if (expression)
    true_statement_or_block
else
    false_statement_or_block
```

1000.110 Else-if

Multiway decisions can be performed using the else-if construct. This has the form:

```
if (expression1)
    statement_or_block
else if (expression2)
    statement_or_block
else if (expression3)
    statement_or_block
else
    statement_or_block
```

1000.111 Switch

Switches can simplify long multiway decisions. They have the form:

```
switch (expression)
{
case value1:
    statement_or_block
    break;
case value2:
    statement_or_block
    break;
case value3:
    statement_or_block
    break;
default:
    statement_or_block
    break;
}
```

Note that the break statements are required to prevent program flow from *falling through* to the next case. It is a good idea to always have a default case.

1001 Functions

C functions have the following form:

```
return_type function_name (parameter_list)
{
    variable_declarations

    statements

    return return_value;
}
```

where

`return_type`
: This specifies the type of the value returned from the function, for example, `int` specifies that an integer value is returned. If no value is to be returned, `void` is used as `return_type`.

`function_name`
: This is the name of the function being declared. Two functions cannot have the same name within a program.

`(parameter_list)`
: This is a comma separated list declaring the *parameters* (or *formal arguments*) that are passed to the function. Each parameter declaration has the form `parameter_type parameter_name` where `parameter_type` specifies the data type that the corresponding argument is converted to when passed.

`variable_declarations`
: These are statements that declare local variables for use in the function only. Note that these variables are only temporary variables and only exist while the function is being executed. Also note that all the variable declarations must become before the statements in the body of the function.

`statements`
: These statements compute the return result of the function.

`return return_value;`
: This statement uses the `return` keyword to specify that the function finishes, or *returns* to the *caller*, where `return_value` is the value returned by the function. Usually functions have a single `return` statement at the end, but it is possible to use these statements anywhere in the body of the function.

1001.1 Pass by value

- When a function call is made all the arguments are passed by value to the parameters of the function. This means that the parameters are given a copy of the arguments and thus if a parameter is modified it has no effect on the argument.

- Arrays are a little different. The contents of the array is not copied but the address of the first element is copied. Thus the parameter needs to be defined as a pointer to receive the address.

1001.10 The main function

- Every C program has to have one function called `main`. This is the first function that runs. Its prototype is

  ```
  int main (int argc, char **
     argv);
  ```

  The return value is 0 for success and any non-zero value for failure. `argc` specifies the number of command line arguments and `argv` is a pointer to an array of strings. `argv[0]` is a string specifying the name of the program, `argv[1]` is a string specifying the first command line argument, `argv[2]` is a string specifying the second command line argument, etc.

- For simple embedded applications without command line arguments an alternate prototype is used for `main`:

  ```
  void main (void);
  ```

  Here `void` indicates there are no function parameters and no return value.

1010 Typedefs

Typedefs allow user defined data types to be created. For example,

```
typedef unsigned int uint;
```

1011 Structures

Structures group data and create a new compound data type. For example,

```
struct point
{
     int x;
     int y;
};

struct circle
{
     struct point centre;
     double radius;
};
```

1100 Unions

Unions allow different variables to share the same memory location. For example,

```
union thing
{
    int x[2];
    double y;
};
```

1101 Enumerations

Enumerations allow named integers to be created, for example,

```
enum card {ACE = 1, TWO, THREE,
   FOUR, FIVE, SIX, SEVEN, EIGHT,
   NINE, TEN, JACK, QUEEN, KING};
```

1110 Library functions

The following functions are some of the common functions in the standard C library.

1110.1 Character testing, `ctype.h`

`int isalpha (int c);`
Returns non-zero if c is alphabetic.

`int isdigit (int c);`
Returns non-zero if c is a digit.

`int isalnum (int c);`
Returns non-zero if c is a alphanumerical.

`char tolower (int c);`
Returns lower-case equivalent of c.

`char toupper (int c);`
Returns upper-case equivalent of c.

1110.10 String operations, `string.h`

`int strlen (char *str);`
Returns length of string `str`.

`char *strcat (char *dst, char *src);`
Appends string `src` to string in array `dst`.

`int strcmp (char *str1, char *str2);`
Compares two strings `str1` and `str2` and returns zero if they are the same.

`char *strcpy (char *dst, char *src);`
Copies string `src` to the array `dst`.

1110.11 Standard input/output, `stdio.h`

`int printf (char *fmt, ...);`
Perform formatted output conversion to standard output `stdout`, using the format string `fmt`.

`int fprintf (FILE *stream, char *fmt, ...);`
Perform formatted output conversion to stream `stream`, using the format string `fmt`.

`int sprintf (char *str, char *fmt, ...);`
Perform formatted output conversion to character array `str`, using the format string `fmt`.

These functions return the number of characters printed. The following format specifiers are recognised:

%%	Output % character
\n	Output newline character
%c	Output character argument
%s	Output character string argument
%d	Convert integer argument to decimal
%u	Convert unsigned integer argument to decimal
%x	Convert unsigned integer argument to hexadecimal
%f	Convert double argument to decimal
%e	Convert double argument to decimal (scientific)

`int scanf (char *fmt, ...);`
Perform formatted input conversion from standard input `stdin`, using the format string `fmt`.

`int fscanf (FILE *stream, char *fmt, ...);`
Perform formatted input conversion from stream `stream`, using the format string `fmt`.

`int sscanf (char *str, char *fmt, ...);`
Perform formatted input conversion from character array `str`, using the format string `fmt`.

These functions return the number of input items assigned. Zero is returned if there was input available but no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a `%d` conversion. The value `EOF` is returned if an input failure occurs before any conversion such as an end-of-file occurs. The following format specifiers are recognised:

`%c`	Input character argument
`%s`	Input string argument (up to first space)
`%d`	Convert decimal number to an int
`%u`	Convert unsigned decimal number to an int
`%x`	Convert hexadecimal number to an int
`%e`, `%f`	Convert decimal number to float
`%le`, `%lf`	Convert decimal number to double

`FILE *fopen (char *filename, char *mode);`
Open input file `filename` as a text stream, using mode `"r"` for reading, `"w"` for writing, or `"a"` appending, `"r+"` for reading/writing, or `"a+"` for appending/writing (among others). If the file cannot be opened, zero is returned (the `NULL` pointer).

`int fclose (FILE *stream);`
Close text stream `stream`.

`int fgetc (FILE *stream)` Input a single character from `stream`. If the end of file is reached, the value `EOF` is returned.

```
char *fgets (char *dst, int size,
                FILE *stream);
```
Input the next line from `stream` (*including the newline character*) and store into array `dst` with a trailing `'\0'` character. At most `size-1` characters are to be read. If the end of file is reached, zero is returned (the `NULL` pointer).

`int fputc (int c, FILE *stream)` Output a single character `c` to `stream`.

`int fputs (char *str, FILE *stream)` Output a string `str` to `stream`.

1110.100 Utility functions, `stdlib.h`

`int rand (void);`
Returns a pseudo-random integer in the range 0 to `RAND_MAX`, which is at least 32767.

`void srand (unsigned int seed);`
Seed the pseudo-random integer with `seed`. The initial seed is `1`.

`int abs (int j);`
Returns the absolute value of the integer argument `j`.

`int exit (int status);`
Terminate the program immediately returning `status` as the exit value.

1110.101 Floating point math, `math.h`

Trigonometric functions (argument in radians)

```
double cos (double x);
double sin (double x);
double tan (double x);
```

Inverse trigonometric functions

```
double acos (double x);
double asin (double x);
double atan (double x);
double atan2 (double y, double x);
```

Hyperbolic functions

```
double cosh (double x);
double sinh (double x);
double tanh (double x);
```

Exponential and logarithmic functions

```
double exp (double x);
double log (double x);
double log10 (double x);
```

Power functions

```
double pow (double x, double pow);
double sqrt (double x);
```

Nearest value, absolute value, modulus functions

```
double ceil (double x);
double floor (double x);
double fabs (double x);
double fmod (double x, double y);
```

1111 ASCII character table

0	NUL	32		64	@	96	‘
1	^A	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	^D	36	$	68	D	100	d
5	^E	37	%	69	E	101	e
6	^F	38	&	70	F	102	f
7	BELL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	z
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	^N	46	.	78	N	110	n
15	^O	47	/	79	O	111	o
16	^P	48	0	80	P	112	p
17	XON	49	1	81	Q	113	q
18	^R	50	2	82	R	114	r
19	XOFF	51	3	83	S	115	s
20	^T	52	4	84	T	116	t
21	^U	53	5	85	U	117	u
22	^V	54	6	86	V	118	v
23	^W	55	7	87	W	119	w
24	^X	56	8	88	X	120	x
25	^Y	57	9	89	Y	121	y
26	^Z	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

10000 Character constant escape sequences

`'\0'`	Null, indicates end of a string
`'\a'`	Bell (alert), outputs a beep
`'\b'`	Backspace (BS), used for over-printing
`'\h'`	Horizontal tab (HT), used for column alignment
`'\n'`	Linefeed (newline) (LF), moves down a line
`'\f'`	Formfeed (FF), moves to start of next page
`'\r'`	Carriage return (CR), moves to start of line
`'\v'`	Vertical tab (VT), used for row alignment
`'\\'`	Backslash
`'\''`	Single quote
`'\"'`	Double quote
`'\?'`	Question mark

10001 Comments

- Anything between /* and */ or after // on a line is treated as a comment and is ignored.
- Comments cannot be nested.

10010 Pre-processor

- Pre-processor directives start with the # symbol. The most common use of the pre-processor is to expand macros defined using,

  ```
  #define MACRO expression
  ```

  The expression should be enclosed in parentheses to avoid unexpected operator precedence.

- Header files can be included with the `#include` directive. There are two forms:

  ```
  #include <system_header.h>
  #include "user_header.h"
  ```

- Lines can be ignored using:

  ```
  #if 0
      statements
  #endif
  ```

- Header guards (ensuring the header file is idempotent) can be constructed using:

  ```
  #ifndef HEADERFILENAME_H
  #define HEADERFILENAME_H
      statements
  #endif
  ```