The C Programming Language

Hello World

```
#include <stdio.h>

main()
{
    printf("hello world\n");
}
```

The program must be suffixed with `.c` E.g. `helloworld.c`

To compile and run the program

```
gcc helloworld.c
./a.out
```

Every program must have a `main` function somewhere.

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

tells the program to include the standard input/output lirary.

Control Flow

The control-flow of a language specify the order in which computations are performed.

An expression such as x = 0 or i++ or printf(...) becomes a statement when it is followed by a semicolon. E.g.

```
x = 5;
```

Braces `{` and `}` are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement.

Variables

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

In C, all variables must be declared before they are used.

Type	Description
int	integer
float	floating point
char	character - a single byte
short	short integer
long	long integer
double	double-precision floating point

The `%` operator cannot be applied to a float or double.

Variable Definition vs Declaration

`Definition` refers to the place where the variable is created or assigned storage; `Declaration` refers to places where the nature of the variable is stated but no storage is allocated.

Initialization

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int x = 1;
cha delim = '\';
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptionally before the program begins execution. For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

In effect, initialization of automatic variables are just shorthand for assignment statements.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas. For example, to initialize an array days with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
}
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

There is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the preceding values as well.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char pattern = "ould";
```

is a shorthand for the longer but equivalent

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

In this case, the array size is five (four characters plus the terminating '\0').

Precedence and Conversion

If an arithmatic equation has one floating point operand and one integer operand, the integer will be converted to a float before the operation is done.

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a `narrower'' operand into a` wider'' one without losing information, such as converting an integer into floating point in an expression.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions.

- If either operand is long double, convert the other to long double.
- Otherwise, if either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise, convert char and short to int.
- Then, if either operand is long, convert the other to long.

A `!=` has higher precedence than a `=`

The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Operator	Associativity
(expr) [index] -> .	Left ==> Right
! ~ ++ -- (type) sizeof Unary operator: + - * &	Right <== Left
* / %	Left ==> Right
+ -	Left ==> Right
<< >>	Left ==> right
< <= > >=	Left ==> Right
== !=	Left ==> Right
Binary operator: &	Left ==> Right
Binary operator: ^	Left ==> Right
Binary operator:	| Left ==> Right
&&	Left ==> Right
OR OPERATOR	Left ==> Right
expr ? true_expr : false_expr	Right <== Left
+= -= *= /= <<= &= ^=	= %= >>= = | Right <== Left
,	Left ==> Righ

NOTE: writing code that depends on order of evaluation is a bad programming practice in any language.

Forcing Conversion with `cast`

Explicit type conversions can be forced (``coerced'') in any expression, with a unary operator called a cast. In the construction

```
(type name) expression

sqrt((double) n)            #converts the value of n to double
 before passing it
                            #to sqrt
```

External Variables

The scope of all variables declared in a function are local to that function.

As an alternative, you can define variables that are external to all functions. I.e. variables that can be accessed by name by any function.

External variables are defined outside of any function, and are thus potentionally available to many functions.

`external variables` can be used to communicate data between functions.

They also stay around even after functions that have used them have returned their values. I.e. they remain in existence until the program closes.

An `external variable` must be **defined**, exactly once, outside of any function; this sets aside storage for it. The variable must also be **declared** in each function that wants to access it; this states the type of the variable. The declaration may be an explicit extern statement or may be implicit from context.

e.g.

```
#include <stdio.h>

//define external variable
int a;

void funcCheck(void) {
    extern int a;
    int b;

    a = 5;
    b = 50;
}

main() {
    extern int a;
    int b;

    a = 0;
    b = 1;

    funcCheck();
    printf ("a: %d\nb: %d\n", a, b);
}
```

In certain circumstances, the `extern` declaration can be omitted. If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an `extern` declaration in the function. So all ove the uses of `extern` above are redundant. Yeah, thanks.

In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all `extern` declarations.

External and static variables are initialized to zero by default.

Static Variables (and Functions)

The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled.

Static storage is specified by prefixing the normal declaration with the word `static`.

If you declare variables statically, their names will not conflict with the same names in other files of the same program.

The external static declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared static, however, its name is invisible outside of the file in which it is declared.

The static declaration can also be applied to internal variables. Internal static variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal static variables provide private, permanent storage within a single function.

Register Variables

A register declaration advises the compiler that the variable in question will be heavily used.

The idea is that register variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

```
register int  x;
register char c;
```

and so on. The register declaration can only be applied to automatic variables and to the formal parameters of a function. In this later case, it looks like

```
f(register unsigned m, register long n) {
    register int i;
}
```

Excess register declarations are harmless, however, since the word register is ignored for excess or disallowed declarations.

The `PRINTF` Statement

The following statement will cause `fahr` to print in the first `%d`, and `celsius` in the second `%d`.

```
printf ("%d %d", fahr, celsius);
```

To adjust the screen space assigned to each variable, use:

```
printf("%3d %6d\n", fahr, celsius);
```

This gives the first var 3 character spaces, and the second var 6 character spaces, and right-aligns the values.

Working with Floats

```
printf("%3.0f %6.1f\n", fahr, celsius);
```

`%3.0f` says that a floating point number is to be printed at least 3 chars wide, with no decimal point and no fraction digits.

`%6.1f` says that the second var should be printed 6 chars wide, right aligned, with 1 digit after the decimal point.

Printf Values

Value	Description
%d	print as decimal integer
%6d	print as decimal integer, at least 6 characters wide
%f	print as floating point
%6f	print as floating point, at least 6 characters wide
%.2f	print as floating point, 2 characters after decimal point
%6.2f	print as floating point, at least 6 wide and 2 after decimal point
%o	octal
%x	hexadecimal
%c	character
%s	string
%%	itself

Special Characters

```
\n      #newline character
\t      #tab character
\b      #backspace
\'      #single quote
\"      #escaped "
\\      #escaped \
\a      #alert (beep)
\r      #carriage return
\f      #formfeed
\0      #null character
\v      #vertical tab
\?      #question mark
```

The Difference Between `i++` and `++i`

The increment and decrement operators can only be applied to variables.

The expression `++i` increments `i` before its value is used, while `i++` increments n after its value has been used. E.g.

```
i = j = 5;
x = ++i;
y = j++;
```

So here, `x` would equal 6, but y would equal 5.

The `IF` Statement

This is a regular `if` statement

```
if (a > b) {
    y = a;
} else {
    y = b;
}
```

It can also be written with the ternary operator `?:`. as either of the following:

```
y = (x > w) ? x : w;
(x > w) ? (z = x) : (z = w);
```

The conditional expression is indeed an expression, and it can be used wherever any other expression can be.

`else if` can also be used.

```
if (a > b) {
    x = a;
} else if ( b > a) {
    x = b;
}
```

The `SWITCH` Statement

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place.

Cases and the default clause can occur in any order.

```
switch (c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    x++;
    break;

    case ' ':
    case '\n':
    case '\t':
    x--;
    break;

    default:
    y++;
    break;
}
```

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch. A break statement can also be used to force an immediate exit from while, for, and do loops.

As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary.

The `WHILE` Loop

```
while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%3.0f %6.1f\n", fahr, celsius);
      fahr = fahr + step;
}
```

A `while` loop keeps looping until the condition is met.

The `DO WHILE` Loop

The while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

```
do
    statement
while (expression);
```

The `FOR` Loop

```
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Best to use `for` when the initialization and increment are single statments and are logically related.

The for statement

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

statement is equivalent to

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

except for the behaviour of continue.

The `,` operator – A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a for statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. E.g.

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--) {}
```

The commas that separate function arguments, variables in declarations, etc., are not comma operators, and do not guarantee left to right evaluation. E.g.

```
int x, y, z;        #NOT comma operators!
```

`BREAK` and `CONTINUE`

The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or switch to be exited immediately.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

Symbolic Constants

It's good to give important variables a meaningful name.

```
#define NAME value
```

E.g.

```
#define LOWER 0
```

The constant must not be in quotes, or be part of another name. No semi-colon `;` is needed at the end of the `#define` line.

Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a `\` at the end of each line to be continued.

The scope of a name defined with #define is from its point of definition to the end of the source file being compiled.

A definition may use previous definitions.

Any name may be defined with any replacement text. For example

```
#define forever for (;;) /* infinite loop */
```

defines a new word, forever, for an infinite loop.

It is also possible to define macros with arguments, so the replacement text can be different for different calls of the macro. As an example, define a macro called max:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Although it looks like a function call, a use of max expands into in-line code. Each occurrence of a formal parameter (here A or B) will be replaced by the corresponding actual argument.

Thus the line

```
x = max(p+q, r+s);
```

will be replaced by the line

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

If, however, a parameter name is preceded by a # in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument. e.g.

```
#define  dprint(expr)   printf(#expr " = %g\n", expr)
```

The preprocessor operator ## provides a way to concatenate actual arguments during macro expansion. If a parameter in the replacement text is adjacent to a ##, the parameter is replaced by the actual argument, the ## and surrounding white space are removed, and the result is re- scanned.

Const

Objects may also be declared `const`, which prevents them from being changed.

```
const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

Character I/O

Text input or output is treated as streams of characters. A **text stream** is a sequence of characters divided into lines. Each consists of 0 or more chars followed by a newline char.

```
getchar     #reads the next input character and returns that a
s its value
putchar     #prints a character each time it is called

d = getchar();
putchar(d);
```

read or write one char at a time.

Character Constants: A character written between single quotes e.g. `'A'` represents the corresponding integer value in ASCII. The escape sequences used in string constants are also legal in character constants, e.g. `'\n'`. Remember, `\n` is a single character (10 in ASCII).

The `EOF` Integer

`EOF` is an integer defined in `<stdio.h>`. It is a symbolic constant.

Counting Lines

The standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline.

Headers

`<ctype.h>` defines a family of functions that provide tests and conversions that are independent of character set.

`<string.h>` – `strlen` and other string functions are declared in this header.

The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of these sizes, along with other properties of the machine and compiler.

`<math.h>` – Math functions

Compiling Across Mulptiple

There must be only one definition of an external variable among all the files that make up the source program; other files may contain extern declarations to access it. (There may also be extern declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an extern declaration.

Initialization of an external variable goes only with the definition.

Suppose that the three functions are stored in three files called main.c, getline.c, and strindex.c. Then the command declarations and statements 64 cc main.c getline.c strindex.c compiles the three files, placing the resulting object code in files main.o, getline.o, and strindex.o, then loads them all into an executable file called a.out. If there is an error, say in main.c, the file can be recompiled by itself and the result loaded with the previous object files, with the command

Functions

Functions can be defined in any order, in one source file or several, although no file can be split between files. (Not sure.)

Functions must be declared before main.

The function definition (aka *function prototype*) is as follows:

```
return-type function-name(parameter declarations, if any)
{
    declarations
    statements
}
```

If the function returns something, it must be of type `return-type`.

The names used by a function for its parameters are local to the function, and are not visible to any other function: other routines can use the same names without conflict. This is also true of any variables used inside of the function.

I.e. - the scope of all variables declared in a function are local to that function.

The value that a function computes is returned to `main` by the `return` statement. Any expression may follow `return`:

```
return expression;
```

A function need not return a value; a return statement with no expression causes control, but no useful value, to be returned to the caller. The calling function can also ignore a value returned by a function.

`main` can have a return statement in it, because it is a function like any other function. It would return a value to the environment in which the program was executed.

Typically, a `return` value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions.

Undeclared parameters in a function are taken as an `int`.

When the name of an `array` is used as an argument, the value passed to the function is the location or address of the beginning of the `array` - there is no copying of `array` elements. By subscripting this value, the function can access and alter any argument of the `array`.

If the function takes arguments, declare them; if it takes no arguments, use void.

Functions themselves are always external, because C does not allow functions to be defined inside other functions.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled.

Function Recursion

C functions may be used recursively; that is, a function may call itself either directly or indirectly.

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set.

Arrays

Arrays always start at [0] in C.

```
int myArray[50];
```

creates an array of integers with 50 places.

Character Arrays

when a string constant like

```
"hello\n"
```

appears in a C program, it is stored as an array of characters containing the characters in the string and terminated with a `'\0'` to mark the end.

Signed vs Unsigned Chars or Ints

The qualifier `signed` or `unsigned` may be applied to char or any integer. unsigned numbers are always positive or zero. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two's complement machine.)

The `/0` Character in C

The character constant '\0' represents the character with value zero, the null character. '\0' is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

Character Functions

`lower` – the function `lower` maps a single character to lower case for the ASCII character set. If the character is not an upper case letter, lower returns it unchanged.

`tolower` is a portable replacement for the function `lower` shown above. Needs header `<ctype.h>`.

Assignment Operators

```
i = i + 5;
```

can also be written as

```
i += 5;
```

All these can be used with `=`

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

E.g

```
x *= y + 1
```

means

```
x = x * (y + 1)
```

Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`.

Operator	Description
`&`	bitwise AND
`	` | bitwise OR
`^`	bitwise exclusive OR
`<<`	left shift
`>>`	right shift
`~`	one's complement (unary)

The bitwise AND operator `&` is often used to mask off some set of bits, e.g.

```
n = n & 0177;
```

sets to zero all but the low-order 7 bits of n.

The bitwise OR operator `|` is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in x the bits that are set to one in `SET_ON`.

The bitwise exclusive OR operator `^` sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operators `&` and `|` from the logical operators `&&` and `||`, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then `x & y` is zero while `x && y` is one.

The shift operators `<<` and `>>` perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus `x << 2` shifts

46 47

the value of x by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity always fits the vacated bits with zero. Right shifting a signed quantity will fill with bit signs (arithmetic shift) on some machines and with 0-bits (logical shift) on others.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example x = x & ~077 sets the last six bits of x to zero. Note that x & ~077 is independent of word length, and is thus preferable to, for example, x & 0177700, which assumes that x is a 16-bit quantity. The portable form involves no extra cost, since ~077 is a constant expression that can be evaluated at compile time.

Strings

A string constant, or string literal, is a sequence of zero or more characters surrounded by double quotes. The quotes are not part of the string, but serve only to delimit it. E.g.

```
"I am a string"
""                    #empty string
```

The same escape sequences used in character constants apply in strings; \" represents the double-quote character.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes.

NOTE: 'x' is not the same as "x". One is the integer value of a character, and the other is a string that exists inside an array.

String Functions

The standard library function `strlen(s)` returns the length of its character string argument `s`, excluding the terminal `'\0'`.

`strlen` and other string functions are declared in the standard header `<string.h>`

`atoi` converts a string of digits into its numeric equivalent.

`strcat(s,t)` concatenates the string `t` to the end of string `s`. `strcat` returns no value; the standard library version returns a pointer to the resulting string.

`strindex(s,t)` that returns the position or index in the string s where the string t begins, or -1 if s does not contain t.

The standard library provides a function `strstr` that is similar to strindex, except that it returns a pointer instead of an index.

The Enumeration Constant

There is one other kind of constant, the enumeration constant. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value.

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t', NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you.

`GOTO` and Labels

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the goto. The scope of a label is the entire function.

It can be useful if you're trying to break out of a heavily nested loop, and `break` simply won't help.

Code involving a goto can always be written without one, though perhaps at the price of some repeated tests or an extra variable.

With a few exceptions like those cited here, code that relies on goto statements is generally harder to understand and to maintain than code without gotos. Although we are not dogmatic about the matter, it does seem that goto statements should be used rarely, if at all.

The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation. The two most frequently used features are #include, to include the contents of a file during compilation, and #define, to replace a token by an arbitrary sequence of characters.

File Inclusion

Any source line of the form

```
#include "filename" or
#include <filename>
```

is replaced by the contents of the file filename. If the filename is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in `<` and `>`, searching follows an implementation-defined rule to find the file. An included file may itself contain `#include` lines.

`#include` is the preferred way to tie the declarations together for a large program. Naturally, when an included file is changed, all files that depend on it must be recompiled.

Conditional Preprocessing

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The #if line evaluates a constant integer expression (which may not include sizeof, casts, or enum constants). If the expression is non-zero, subsequent lines until an #endif or #elif or #else are included. (The preprocessor statement #elif is like else-if.) The expression defined(name) in a #if is 1 if the name has been defined, and 0 otherwise.

For example, to make sure that the contents of a file hdr.h are included only once, the contents of the file are surrounded with a conditional like this:

```
#if !defined(HDR)
#define HDR /* contents of hdr.h go here */
#endif
```

A similar style can be used to avoid including files multiple times.

This sequence tests the name SYSTEM to decide which version of a header to include:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   #define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
#else
   #define HDR "default.h"
#endif
#include HDR
```

The #ifdef and #ifndef lines are specialized forms that test whether a name is defined. The first example of #if above could have been written

```
#ifndef HDR
    #define HDR
    /* contents of hdr.h go here */
#endif
```