User Manual

Miracle C Compiler

version 1.5
17 January 1993

The Miracle C Compiler runs on a 386 PC (or better) under MS-DOS, accepting a dialect of the C language and generating object code suitable for Microsoft or compatible linker.

All of traditional (Kernighan & Ritchie) C syntax is implemented, including record (struct/union) and enumerated data types, int, long and floating point data types, user type definition, bit fields in structs, initializers for all data types. Both traditional and new (ANSI) function declaration is supported. There is a comprehensive library of functions.

REGISTRATION

The compiler is shareware, and is supplied on a trial use basis. The compiler package is copyright material and should not be modified or disassembled. Programs written using the compiler may be freely distributed.

If you find it useful you should register. Registration costs 10 pounds in the UK, 25 dollars overseas, and entitles you to receive a printed copy of the documentation and upgrades for one year.

Registration is from the author;

T Szocik
45 Englewood Road
London S.W.12 9PA
United Kingdom.

Miracle C is under active development by the author. By registering you support further development of the compiler. Should you wish to make a direct contribution to a future release (for example by writing a compiler utility, library functions or source code), please write to the developer at the above address.

USING THE COMPILER

The following files are present in a compressed form in the self-extracting archive;

miracle.doc	documentation in MS Word format
miracle.txt	documentation in text format
cc.exe	the C Compiler
ccl.lib	C compiler library
mc.bat	batch file for compiler

and several example programs;

example.c
hanoi.c
cat.c
sieve.c
maze.c *maze*
slr.c *grammar.c*

Floating point support;

The compiler requires an 8087 coprocessor (or better), or emulation program, to perform floating point arithmetic. A public domain emulation program (em87.com) is to be found in the compiler archive.

The compiler uses the following environment variables;

LIB	directory containing library *ccl.lib* (this is only used by a linker)
INCLUDE	directory containing system include files <*.h> (if unset, defaults to \cc\include)
LINKER	name of linker, eg LINKER=BLINK (if unset, defaults to LINK, the Microsoft linker)

Note that a linker is not supplied with the compiler.
Use the Microsoft linker supplied with DOS, or a compatible linker.

CC output is Microsoft/Intel object module format, as documented
in the MS-DOS Encyclopedia. The linker must accept Microsoft object.

Before compiling a program set the environment variables to suitable values;

```
C:\> SET LIB=\CC
C:\> SET INCLUDE=\CC\INCLUDE
C:\> SET LINKER=LINK
```

Compile one of the example files;

```
C:\CC> MC EXAMPLE.C
Microsoft (R) Overlay Linker  Version 3.51
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.

C:\CC>
```

This will compile and link `example.c', creating example.obj, example.exe.

Compiler switches;

-c	compile only (generate .obj, don't link)
-j	signed char constants
-n	noisy compile
-p	disable autoprototyping
-r	set error limit
-s	generate source listing (.src) after preprocessing
-t	allow trigraphs
-w	no warnings
-Dname	define macro
-Uname	undefine macro
-Iname	add include directory

Only one program file may be compiled at one time. To compile and link together several programs, compile them separately with `-c' option, then link them with *ccl.lib*.

If the compiler finds an error (in preprocessing, parsing or other compilation error, eg an undeclared variable) it will print the statement where it occurred and stop if the error limit has been reached. Functions should be declared before being called, although this is not mandatory. If they are not, then auto-prototyping will generate a prototype for a new function returning int. A function definition automatically declares the function if it has not been declared or auto-prototyped earlier.

EXAMPLE PROGRAMS

A few example programs are included to illustrate Miracle C facilities.

example.c	*hanoi.c*
cat.c	*sieve.c*
maze.c	*maze*
slr.c	*grammar.c*

example.c	shows most of CC's features; structs/unions; initializers for structs and arrays; typedefs; function prototyping; functions with local variables, and scoping; data types; signed and unsigned int, char and long variables, floating point; use of some library functions for file handling
cat.c	reverse words in a sentence, eg `I am here' to `here am I'
hanoi.c	non-recursive towers of hanoi (originally from a Programmer's Challenge in Computer Shopper magazine!!) gives the sequence of moves to move all discs from first to third peg via second
sieve.c	Eratosthenes sieve to find prime numbers by eliminating composites developed from a BYTE C benchmark program
maze.c *maze*	maze solver, finds shortest path through maze from A to B eg. C:\> **MAZE<MAZE** uses an algorithm from graph theory to find shortest path
slr.c	simple parser generator for SLR grammar given a grammar file, generates a recognizer program to parse the SLR grammar
grammar	grammar file for slr.c

LANGUAGE FEATURES

This is a summary of the C language as implemented by the Miracle C compiler.

For a tutorial on the C language, read `The C Programming Language' by Kernighan and Ritchie. A comprehensive reference to the language is the book `C: A Reference Manual' by Harbison and Steele, published by Prentice-Hall, which describes both traiditional and the new ANSI C syntax and semantics. Miracle C implements the traditional C language and some of the ANSI extensions.

Lexical Elements

Tokens
The fundamental lexical elements of the language are operators, separators, identifiers, reserved-words and constants.

A C source program consists of tokens separated by whitespace. Whitespace is defined as chars 9-13 and 32. Tokens can be operators, separators, identifiers, reserved-words or constants.

1. Operators
The following operators are legal in C.

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

2. Separators
Whitespace characters 9-13, and space character 32 are separators.
( ) [ ] { } , ; :

3. Identifiers
A sequence of any number of letters/digits/underscores starting with a letter or underscore character. Identifiers are case-sensitive so name, Name and NAME are different, but external symbols (such as function names) are not case-sensitive.

4. Reserved words
auto break case char default do else enum extern for goto if int long register return short signed sizeof static struct switch typedef union unsigned void while

5. Constamts
A C constanr can be an integer, character or string, or floating point constant.

integer
An integer can be represented as a decimal, octal (base 8) or hexadecimal (base 16) constant.

- decimal (digits)
- octal 0(digits), eg 0243
- hex '0x' (or 0X) (hex-digits 0-9 a-f A-F), eg 0x4afb

A numeric constant can be unsigned (suffixed by U, eg 56U), long (eg 128374L) or floating point.
Floating point constants have syntax aaa.bbbEeee where aaa.bbb = mantissa
eee = exponent (signed, optional)

character
A character is 'char' or '\esc-char'
where esc-char= nnn octal literal
xnn hex literal

a alert	b backspace
f form feed	n newline
t tab	r carriage return
v vertical tab	' single quote
? question mark	" double quote

A character constant may contain more than one byte, on which case bytes are packed into a word, for example   int a='xy';

string
A string is "string" where the string has printable characters or \escaped-characters.
Constant strings are null-terminated, eg sizeof("hello")=6

Preprocessor

The preprocessor performs macro substitution, file inclusion and conditional compilation on the source program. Use the `-s' compiler option to see the program text after preprocessing.

The preprocessor allows a line to be continued by ending it with a `\' character. Tokens and strings can be split across lines.

Preprocessor directives are lines starting with `#'. A line with only '#' is a null directive and is ignored by the preprocessor.

Macro substitution
The #define preprocessor directive defines a macro, optional parameter list and associated substitution string. A macro can have no parameter list, eg

```
#define WORD_SIZE 16
```

then every occurrence of WORD_SIZE in the program text is replaced with `16'.
The general form of a macro definition allows zero or more parameters;

```
#define <name>(parameters) <replacement text>

#define afunc() otherfunc()
#define times(x,y) ((x)*(y))
```

every call to afunc() is replaced with a call to otherfunc(); when `times' is used, actual arguments are substituted for macro parameters in the replacement text, eg

```
times(4,times(5,6))
```
gives

```
        ((4)*(((5)*(6))))
```

The number of arguments when the macro is used must match the number of parameters when it is declared.

A macro definition may be split across more than one line by line continuation with \

```
        #define plus(x,y) \
                ((x) + (y))
```

After macro substitution, the line is scanned again so macros created by the expansion can be recognized and expanded.

The following macros are predefined by the preprocessor;

MIRACLE	defined as 1 use for code specific to Miracle C compiler
MSDOS	defined as 1 use for code specific to MS-DOS
__FILE__	current source file
__LINE__	line in current source file
__DATE__	today's date, returned as "Mmm dd yyyy"
__TIME__	time of compilation, returned as "hh:mm:ss"

Redefinition of macros (benign or otherwise) is allowed. No warning or error is given if redefinition occurs.

A macro definition can be cleared using `#undef' directive;

```
        #undef times
```

File inclusion
The `#include' directive allows C source files to be included in the program text. When the end of the included file is reached, compilation continues from the line after the `#include' directive. Include files may be nested; but they should including themselves, directly or indirectly, causes the compiler to crash. Preprocessor include files usually have a .h suffix (header files). There are two forms of #include directive;

```
        #include "filename"
```

includes a file in the current directory,

```
        #include <filename>
```

includes a file from the system include directory, which is given by the INCLUDE environment variable. If INCLUDE is not set, the system include directory defaults to `\cc\include'.

The inclusion-file can be specified by a macro;

```
#define MYINCLUDES "my.h"
#include MYINCLUDES
```

A complete pathname can be given in the first form of include;

```
#include "\cc\graphics\graf.h"
```

Conditional Compilation

The preprocessor allows compilation of sections of code to be conditional on the values of expressions, using the directives `#if' `#ifdef' `#ifndef' `#else' `#endif'. Lines after an unsuccessful conditional compilation directive are discarded until the next conditional compilation directive.

A conditionally compiled section of code starts with #if #ifdef or #ifndef and ends with a matching #endif. It may contain #else or #elif directives, and other (nested) conditionally compiled code sections.

`#if expr'
If the C expression `expr' evaluates to 1, lines are processed until a matching else, elif or endif is found. If it evaluates to 0, following lines of code are discarded until a matching else, elif or endif is found. The expression `expr' must be a constant recognisable to the preprocessor, containing macro, character and number constant values only.

`#ifdef name'  `#ifndef name'
If the macro `name' is defined (ifdef) or undefined (ifndef), lines are processed until a matching else, elif or endif is found. Otherwise, following lines of code are discarded until a matching else, elif or endif is found.

`#else'
The else directive follows an if, ifdef or ifndef, or can follow #elif directives. Lines of code following #else are processed only if preceeding lines were not processed.

`#elif'
This is a switch/case construct for conditional compilation;

```
#if expr1
...
#elif expr2
...
#elif expr3
(etc)
#else
...
#endif
```

`#endif'
terminates a conditionally compiled code section.

Other Features

The #error directive aborts compilation with an error message;

```
#ifdef x_once
#error Illegal include recursion
```

```
#endif
#define x_once
```

detects a program's attempt to include itself; implement "once-only" inclusion of header files similarly.

Stringification converts a token into a string, and is useful when debugging a program;

```
#define fatal(tok1,tok2) printf("bad tokens: %s;%s;",#tok1,#tok2)
```

Concatenation of adjacent strings is supported by Miracle C compiler,
so

```
#define fatal(tok1,tok2) printf("bad tokens: " #tok1 ";" #tok2)
```

is allowed.

Token merging using the ## operator creates a single token from two or more tokens in a macro definition;

```
#define line(i) line ## i
```

then line(1) == line(2)
becomes line1 == line2

Functions

Miracle C supports a both the traditional syntax for function definition, and a syntax similar to ANSI C for function prototyping. Functions can be declared and prototyped before use. The traditional way of introducing functions, eg.

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

is supported. Traditionally, functions weren't declared before use or definition; CC allows functions to be declared and prototyped before use, zero times or exactly once, or a parse error results. Note that 'parse errors' occur when the current symbol is not one of the expected legal symbols, hence a function redeclaration may generate a parse error.

CC's function declaration syntax is similar to ANSI, eg

```
void afunc(int one, char *b);
```

but the ANSI syntax

```
void afunc(int, char *);
```

is not supported (yet). The number and type of parameters, and return value, in a function definition must match exactly that of the function declaration. Miracle CC supports declaration of functions with a variable number of
parameters, eg

```
int printf(char *fmt, ...);
```

but their definition is not supported (yet).

Miracle C has the concept of `function types' which are

```
type1 x .. x typen x varargs -> typer
```

for a function taking n parameters of type (type1, ..., typen) and returning a result of type typer, with a variable number of arguments if varargs set.

A pointer to a function is assigned a function type, which should match the type of a function assigned to it; for example,

```
void (*fptr)();
int printf(char *fmt, ...);

(fptr=&printf,*fptr)("hello");
```

will fail with the error message

```
158: wrong # args in function call

'(fptr=&printf,*fptr)("hello")'
```

Functions may return signed or unsigned int, char or long values, but not struct or union. They may return pointers to any item (including struct), or a user defined type, as long as that type isn't a struct.

Nor can a function return an array, or another function.

Parameters in a function call are pushed on the stack using normal C linkage conventions, ie right-to-left, to allow for functions with a variable number of parameters (eg printf, first argument specifies number and type of subsequent). Parameters should be word entities, ie int char and pointer type.

Functions may be declared as extern, static or register, but these classes are ignored by the compiler. Miracle C marks functions defined in the program text as `internal' and others as `external' and generates object accordingly. Static functions (visible only within a program) are not supported (yet!).

Miracle C needs functions with no parameters to be declared by

```
        int afunc();
```

It does not follow the ANSI convention of declaring such as function by

```
        int afunc(void);
```

nor does it follow the traditional C convention of declaring a function's return type, but omitting a declaration of parameters, using the above syntax.

Parameters in a definition count as local variables in the highest level block in the function body, following the ANSI definition.

Storage classes are not allowed for parameters in a definition. Functions with return type `void' must not return a value.

A function declared to have return type `int' need not specify return type in the function definition, so we allow

```
        int main();
        ....
        main() { ... }
```

Function prototypes may not be nested;

```
  void (*signal(int sig, void (*func)(int sig)))(int sig);
```

is illegal, but can circumvent this by

```
  typedef void sig_handler(int sig);
  sig_handler *signal(int sig,sig_handler *func);
```

Autoprototyping of functions is allowed. If a function call is introduced for a function which has not been declared, a declaration is automatically generated for the function. The automatically generated prototype will use the types of parameters which are given to the function on call, and will assume a function return type of int. If the desired function type is different to that which will be implicitly generated from the function call then an explicit declaration should be made before the function is called. Function autoprototyping may be disabled by a compiler flag.

Types

Miracle C supports void, scalars (pointers, signed and unsigned char int long), enumeration types, floating point types, pointers to any object, struct/union and multi-dimensional array types. It also has `function types' (see the section on functions).

As a small-model 8086 compiler, CC supports 16-bit ints and pointers, 8 bit char values (expanded to 16 bits when passed as parameters on the stack), and 32 bit longs. The type `short' is a synonym for `int'. An object declared by

```
        signed avar;
```

is a signed integer. Floating point numbers are not implemented. Long integers are not completely implemented; for example, adding a long to an int is not allowed, but adding two longs is (so cast the int to long before adding); and long values should not be passed as function arguments.

Miracle C maintains a type value for each expression, computed from the types of its components; a data size is associated with each type value.Traditional C casts are allowed, but type checking of assignments etc is not strict.

Array subscripts, and adding an int to a pointer, is scaled up according to the data size of the type being pointed to, eg if intptr is int * then intptr+1 (or intptr[1]) will point to intptr plus 2 in memory.

Arrays of any type except void can be declared and can be multi-dimensional. Void variables are not allowed.

Enumerations are introduced by an `enum' declaration, such as

```
        enum colour { red, green, blue } mycolour;
        enum colour anothercolour.
```

The tag (here `colour') is optional, and can be used in a later declaration. Enumerations must start at zero, the traditional "red=2" syntax is not supported. An enum variable is int. The enum namespace is separate from others, hence

```
        enum one { one, two } number;
```

is allowed.

Record (Struct/Union)

Structure (record) type is a collection of named components;

```
        struct structag
                {
                int x, y[3];
                char z;
                struct structag *sptr, *tptr;
                }
                sarray[10], *sptr;

        struct structag another, *anotherptrs[5];
```

Structure tag `structag' identifies this structure type for other declarations. The struct has components x,y,z and two pointers to other `structag' structs (so we've declared a tree structure type). Structure declarations cannot be nested (that would be infinitely silly). Structure component names live in a separate namespace for each struct/union type, hence we can introduce variables with the same name. Struct components occupy successive memory locations, and the size of a struct (as given by sizeof) is the sum of the sizes of it's components.

Struct components are referred to using . and -> operators. In the example above we declare sarray an array of 10 `structag' structs and allocate space for it (either static or on the stack). Direct reference by sarray[i].y[0], indirect reference via pointer to struct by sptr->z, sptr->tptr->y[i].

A union is declared using the same syntax as struct, but contains only one of its members at any one time (so the size of a union is the max of the size of it's components, and all items are at offset zero in the union).

Nested structs/unions are allowed. Bit fields in structs aren't implemented.

Struct bitfields are permitted. A single bitfield must not exceed the capacity of a machine word (16 bits). All the normal arithmetic operations and assignment are permitted on bitfields, as are struct bitfield initializers.

Typedef

User defined type synonyms are introduced by the `typedef' statement;

```
typedef int *ptr, (*func)(), afun();
ptr wordptr;
afun main;
func funcptr=main;
```

ptr	`ptr to int'
func	`ptr to function: void->int'
afun	`function: void->int'

Typedef doesn't create a new type, only type synonyms, so type compatibility/comparison works. Typedef for function can include function prototypes;

```
typedef int funci(char *x);
funci funky;
```

A typedef name may be global, or local to a function. The ANSI standard allows typedefs to be redeclared in a block, but CC doesn't (yet).

Type Compatibility

Two expressions are assignment compatible t1=t2 if t1 is an lvalue, and;

- t1 is (u)long, t2 is (u)int/char, extended to 32 bits
- t1 is (u)int/char, t2 is (u)long, truncated to 16 bits
- t1 and t2 are both (u)long, 32 bit assignment
- t1 is (u)char, 8 bit assignment

else 16 bit assignment (of int, pointer etc)

Bitfields may be signed or unsigned integer quantities and have the same type compatibility rules as integers.

Some operators eg || && ^ | & * / % require (u)int operands.
Pointers may be subtracted (t1-t2 yields a pointer of type t1) but not added (addition of pointer is meaningless).

Array types count as ptrs, depth of ptr=dimensionality of array.
Struct/unions types are compatible if they have the same struct tag.

Declarations

Variables, functions and user defined data types are introduced by declaration statements. A declarator gives identifier name and type;

Scalar	int x
Floating point	double x;
Pointer	int *x; char *y[]; (y is pointer to array, same as char **y;)
Array	int (*x)[4]; (x is pointer to array of 4 ints)
Function	int x(), y(int a,char *b), (*x[])(int a);

To parse a declaration, start from name and work out according to precedence rules;

```
        int *a[10]              a is array of 10 ptrs to int
        int (*x[])(int a)  x is array of ptrs to functions: int->int
```

Global variables are visible throughout the program from the point of declaration, unless they are hidden by a local declaration with the same name. Local variables are visible throughout the block in which they are defined, unless they are hidden by a declaration at an inner level with the same name. Globals are extern by default.

Static variables (globals and local statics) have storage allocated at compile time in static data area to hold a possible initializer. Local variables are allocated on the stack at runtime and are local to a block, or are function parameters.

Parameters to a function are considered as being declared at the topmost local level in the function, so a local declaration using a parameter name is an error. Local objects may not be declared more than once.

All floating point types (float, double and long double) are treated internally as 8-byte double quantities. Normal floating point arithmetic is allowed on these quantities. No support for numeric coprocessing is included in this version.

Initializers are allowed for both global and local objects and follow traditional syntax. Global initializers are evaluated at compile-time, a constant is stored in the static data segment. Therefore, expressions in global initializers can only contain things which can be evaluated at compile-time, such as constants and address of objects.

Local initializers are evaluated when a function is called, so expressions may (even) contain function calls.

Multi-dimensional arrays, structs/unions, arrays of structs and initializers for them are supported using traditional C syntax.

A struct tag can be declared before any variables are;

```
        struct S { int a, b; };
        struct S fred;
```

A struct can be declared without tag or variables;

```
        struct { int a,b; };
```

is allowed but pointless.

Also static struct S { int a,b; }; is allowed but `static' is meaningless.

A global variable may be declared more than once, but all declarations must agree on type. A local variable may only be declared once. Global arrays must have the same dimensions when redeclared, but the first dimension need not be specified;

```
char uu[5][4][3]; char uu[][4][3];
```

Only one definition may exist for a variable, so a global may have only one declaration with an initializer. Redeclaration of functions is not supported.

Type definitions, structs/unions and enumerated types are supported; see the section on `types'.

Global variables are allocated in a static data area. Uninitialized globals are set to zero; uninitialized parts of partly initialized globals are zeroed. Initialized globals give PUBDEF records for the linker, uninitialized globals give COMDEF records.

Arrays are allocated at compile-time (static) or run-time (auto), hence static char p1[]="hello" allocated 6 bytes and copies "hello" into it, but pointer initialization is to run-time objects, hence char *p2="hello" allocates 2 bytes for p2 and points it to a static string item.

Statement labels are local to a function body, and the target of goto statements

```
here:
        ...
        goto here;
```

Forward references are allowed if;

- statement label may be used by goto before it is declared

- an identifier should be visible immediately after declaration,
  eg int fred=sizeof(fred)
  but this is not yet implemented

- struct can contain pointer to instance of itself

- function declaration and use before definition

Overloading of identifiers permits an identifier to have different meanings depending on context;

- macro names are defined and used by the preprocessor

- struct tag names have a separate name space

- enum tag names have a separate name space

- typedef, enum and label names are checked before variable names.
  a compile error results if a variable is declared with the same
  name as a typedef or enum name, or goto-label

- struct component names are specific to a struct/union type, so two structs can have components of the same name

External names must be defined at top-level (global variables). Extern declarations inside functions are not supported and are treated as declarations of internal variables. Global variables are defined to the linker and accessible to other programs.

The storage classes auto and register are ignored by the compiler. Static variables local to a function are allocated in the static data area, and should have constant initializers.

Initializers
The compiler allows initialization of scalars, strings, struct/unions and arrays. An initializer sets the initial value of a variable, at compile time for static objects (globals or local statics) or run time for automatic variables. If no initializer exists for a global variable, it's set to zero; if none exists for a local, it's initial value is unpredictable.

The traditional C syntax for initializers is supported;

```
int x=4;

struct { int a; char *s; int b; } icky = { 1, "astr", 2 };

enum { red, blue } colour = blue;
```

Union initializer is for the first component of the union. Structs and multi-dimensional arrays are initialized recursively;

```
int m1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

The `shape' of an initializer (brace structure) must match that of the object being initialized. If there aren't enough initializing items for an array or struct initializer, the rest of the object is zeroed. If there are too many, a compile time error results.

If the outermost dimension of an array is unspecified, it's taken from the initializer;

```
char a[] = { 'a', 'b', 'c', '\0' }, *b="hello";
```

Braces can be dropped in the initializer;

```
int a[2][3]={1,2,3,4,5,6};
```

the number of items in the initializer must not exceed the declared object.

Static variable initializer expressions contain only objects which can be evaluated at compile-time, ie constants and addresses of static objects. The address of a global variable cannot be found if it is declared but not defined;

```
int a, *b=&a;
```

won't compile, but

```
        int a=2, *b=&a;
```

is allowed.

Local variable initializer expressions are evaluated at runtime, when a function is called. They can contain any expression (even function calls). If goto jumps to a label in a block, initializers for that block don't run.

Initializers are permitted for floating point variables, and bitfields in struct (record) types, using the standard C syntax.

Expressions

An expression consists of operators acting on other expressions, or base values (variables or constants). An expression is either an lvalue or an rvalue; an lvalue refers to an object in memory, eg a variable; an rvalue is a non-lvalue.

Lvalues can be `variable' e[k] lvalue.name e->name *e
they are used by & ++ -- assignment-operators

Expressions are formed from operators, in precedence order:

,	sequential evaluation, yields second operand		
= +=	-= *= /= %=		
&= |=	^= >>= <<= assignment		
	assign	(u)long = (u)(int|char)	
		(u)long = (u)long	
		(u)(int|char) = (u)long	
		word/byte lvalue = word/byte rvalue	
?:	conditional exp1 ? exp2 : exp3		
	depending on value of exp1, yields exp2 or exp3		
	type of exp1 must be int		
	types of exp2 and exp3 must match, and must be byte or word		
||	logical or	exp1 || exp2	only evaluate exp2 if exp1 false
&&	logical and	exp1 || exp2	only evaluate exp2 if exp1 true
|	bit or		
^	bit xor		
&	bit and		
	bit or, xor, and take (u)int operands and yield (u)int result		
== !=	compare, signed or unsigned		
< >	signed if at least one operand is signed, both are int or char		
<= >=	unsigned if both operands are unsigned int or char, or pointers		
<< >>	shift, arithmetic for signed, logical for unsigned		
+ -	add sub		
	(u)(char|int), (u)(char|int)		
	ptr, (u)(char|int) scale up by size of object pointed to		
	(u)long, (u)long		
	can subtract two pointers giving an integer		
* / %	mul div rem		
	take (u)int operands, yield (u)int result		
(type)	cast		
	cannot cast to/from structs/voids		

	cannot cast between chr and ptr, but can between int and ptr if casting long->int then lose high word, int->long high word zero	
*	indirection	
&	address of	
-	unary minus	
!	logical not	
~	bit not	
sizeof	sizeof(typename)	size of type
	sizeof("string")	size of constant string + 1
	sizeof(array-expr)	size of array in bytes
	sizeof(expr)	sizeof type of expr
++ --	post/pre decr/incr if pointer, scaled up by size of object pointed to	
->	indirect selection by pointer (struct/union member or bitfield)	
.	direct selection (struct/union member or bitfield)	
f(..)	function call, where f is a function designator f has function type and identifies a function	
(expr)	brackets	
a[k]	array subscript; type of k = (u)(int|char) the result of an array subscript is an lvalue only if a is a pointer, or we have reached the last array dimension for example, if int a[3][2]; then a[1]=4; is meaningless and is flagged as an error but this means that int a[2][3], **x; x=&a[1]; will not compile. The ANSI solution (`modifiable lvalues') is not implemented.	
number	constant	
string	constant	
variable	global, local, lstatic	

Initializer constant expressions are arithmetic constant expressions and address constant expressions (not fully implemented in CC). These can have normal operators & casts, which are evaluated at compile-time.

Statements

All traditional C statements are implemented, except for `continue'.
A statement can be one of the following;

expr	expression; (discarded)
null	null statement (actually a null expression)
label	label: stmt; which can be a goto-label, case-label, or default-label in switch
goto	goto name; where `name' is a goto-label defined somewhere in function
block	{ decl-list; stmt; ... stmt; } A block starts with zero or more declarations, and contains zero or more statements. Blocks may be nested. Names declared within the block hide declarations of the same name outside the block, and are visible throughout the block unless hidden by a declaration at an inner level. Goto a label inside a block jumps past local variable allocation and initialization code; but the locals are de-allocated at end of block anyway, so don't jump into a block!

if	if (expr) stmt
	if (expr) stmt1 else stmt2 NB else belongs to nearest if
	
	if `expr' evaluates to non-zero, do stmt1 (else do stmt2)
	`else' belongs to the nearest `if'

while	while (expr) stmt continue executing `stmt' while `expr' evaluates to non-zero
do	do stmt while(expr) do `stmt' at least once, then while `expr' evaluates to non-zero
for	for(expr1; expr2; expr3) stmt each of exp1,2,3 optional expr1= evaluated once at start of loop expr2= continuation condition, tested before stmt executed expr3= iteration expression, evaluated after statement if expr2 omitted then it defaults to true eg. for(;;) is an infinite loop

switch	switch(expr)
	  {
	  case n1: ...;
	  case n2: ...;
	    ...
	  default: ...;
	  }
	
	case labels must be numbers or char literals;
	case labels should not be duplicated, at most one default exists;
	when a case or default is chosen, execution continues until
	break or end of block

break	terminate smallest enclosing while. do, for, switch
continue	continue smallest enclosing while. do, for
return	return opt-expr return from function with optional value void functions must not return a value

ASSEMBLER PROGRAMMING INTERFACE

Miracle C functions can call, and be called by, assembler routines.

Function call is by pushing arguments onto the stack in right to left order, and call (pushes ip on stack). The called function pushes bp and allocate stack space for local variables. Function return is the reverse; deallocate local variables, pop bp, return; caller pops arguments.

```
+-------+
| arg n |
|   .   |
|   .   |
| arg 1 |
|  ip   |
|  bp   |
| local |
|   .   |
|   .   |
| local |
+-------+
```

At entry to a C function, SP points to the return value of IP on the stack.
To access word arguments;

```
push    bp
mov     bp,sp
mov     AX,[bp+2] ; first argument
mov     BX,[bp+4] ; second argument
...
pop     bp
mov     ax,retvalue ; return value
ret
```

FUNCTION LIBRARY

Startup code

The startup code initializes segment registers, sets argc/argv parameters
to zero, and calls function `main'. If `main' returns to the startup routine,
an exit handler is called which closes files and terminates.

Include header files

The following header files are supplied with the compiler in the 'include' subdirectory.

ctype.h	Prototypes for alphanumeric test and conversion functions.
io.h	Prototypes for elementary file functions.
stdio.h	Prototypes for higher level file handling and input/output functions.
string.h	Prototypes for memory and string handling functions.
system.h	Prototypes for memory and other functions.

Library Functions

The Miracle C library contains functions for string handling, file operations, input/output formatting, memory allocation and other functions.

abs, labs

```
#include <system.h>

int abs(int i)
long labs(long i)
```

Produces the absolute (positive or zero) value of its argument.
abs takes an int argument returning an int, labs takes a long argument and returns a long.

calloc

```
#include <system.h>

void *calloc(int nitems, int itemsize)
```

Allocates a block of memory that ia nitems * itemsize bytes in size from the heap. The initial contents of the block is zeroed.
If not enough memory is available to satisfy the request a null pointer is returned. Since the compiler uses the small memory model, memory requests should be made accordingly.
Allocated memory should be freed explicitly when it is no longer required.

Example

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

void main()
{
   char **buffer;
   int nitems=100;

   buffer = calloc(nitems,sizeof(char *));      /* allocate 50 pointers to char */
   if(buffer==NULL)
      {
         printf("out of memory\n");
         exit(1);
      }
   printf("calloc allocated %d items of %d at : %x\n",nitems,sizeof(char *),buffer);
   free(buffer);
   return;
}
```

close

```
#include <io.h>

int close(int fd)
```

Close the file given by file descriptor handle `fd'. freesing the file descriptor for use by another file.
close does not write an eof character 26.
Returns 0 if successful, otherwise -1 if failed.

Example

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

void main()
{
int fd;
   if((fd = open("tfile",O_RDONLY))<0)
      {
         printf("failed to open tfile\n");
         exit(1);
      }
   printf("close code %d\n",close(fd));
   return;
}
```

create

```
#include <io.h>

int create(char *fname)
```

Create a file with name `fname'.
If successful, returns a file descriptor for the newly created file. Otherwise returns -1 if unsuccessful.

Example

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

void main()
{
    int n;
    n = create ("tfile");
    if (n == -1) {
        printf("Cannot create tfile\n");
    }
    return;
}
```

exit

```
#include <system.h>

void exit(int code)
```

Closes files, flushes all output buffers and terminates with return code `code'.

Example

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

void main()
{
    int n;
    n = create ("tfile");
    if (n == -1) {
        printf("Cannot create tfile\n");
        exit(1);
    }
    exit(0);
}
```

fclose

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

int fclose(FILE *fp)

Close an open file, and flush output buffer for the file. Returns 0 if successful, EOF if not.

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

void main()
{
   FILE *fp;
   if((fp = fopen("tfile","r"))==NULL) {
      printf("failed to open file tfile\n");
      return;
   }
   else {
         fclose(fp);
         printf("closed file tfile\n");
      }
   return;
}
```

feof

#include <stdio.h>

int feof(FILE *fp)

Tests end of file condition for file fp. Returns non-zero if end of file.
After an EOF condition no further reads should be performed.

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

void main()
{
   FILE *fp;
   char buffer[100];

   fp = fopen ("tfile", "r");

         while ( !feof(fp) )
           fgets(buffer, 100, fp);

   return;
}
```

fflush

#include <stdio.h>

int fflush(FILE *fp)

Flush buffer for an output file. If the file is open for writing the output buffer is wriiten to disk. If it is open for reading the buffer is cleared and another read operation is forced to occur.

Returns 0 is operation successful, otherwise EOF if an error occurred.

Example

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

void main()
{
FILE *fp;

   if((fp = fopen("tfile","w"))==NULL)
      exit(1);

   fflush(fp);
}
```

fgetc

#include <stdio.h>

int fgetc(FILE *fp)

Reads and returns a character from a file. Returns next character or EOF.

Example

```
#include <stdio.h>

void main()
{
   FILE *fp;

   fp = fopen ("tfile", "r");

    while ( !feof(fp) )
          putchar(fgetc(fp));

   return;
}
```

fgets

#include <stdio.h>

char *fgets(char *buf, int n, FILE *fp)

Get a string (maximum n characters) from file `fp' to buffer `buf'.
Returns NULL if an error occurred or no characters were read,

otherwise returns the (null-terminated) string.

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

void main()
{
   FILE *fp;
   char buffer[100];

   fp = fopen ("tfile", "r");

        while ( !feof(fp) )
          fgets(buffer, 100, fp);

   return;
}
```

fopen

#include <stdio.h>

FILE *fopen(char *name, char *mode)

Open a file with filename `name' returning a pointer to FILE.
The following modes are allowed;

`r'	open file for reading only
`w'	open file for writing
`a'	open file for append; position at end of file
`r+'	open file for reading and writing
`a+' `w+'	create new file for reading and writing

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

void main()
{
   FILE *fp;
   if((fp = fopen("tfile","r"))==NULL) {
      printf("failed to open file tfile\n");
      return;
   }
   else {
        fclose(fp);
        printf("closed file tfile\n");
      }
   return;
}
```

fprintf

#include <stdio.h>

int fprintf(FILE *fp, char *fmt, ...)

Print formatted values to file. Arguments follow the format string and are interpreted according to the format string.
fprintf writes its characters to the file stream fp.

The format string is a sequence of characters with embedded conversion commands. Characters at are not part of the conversion command are output. Conversion commands are the same as for the printf function.

fprintf returns the number of characters written.

<u>Example</u>

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

void main()
{
   FILE *fp;
   char *msg = "number formats are: ";
   int n = 42;
   fp=fopen("con","w");

   fprintf(fp,"%sx: 0%x d: %d o: %o\n",msg,n,n,n,n);
   fclose(fp);
   return;
}
```

fputc

#include <stdio.h>

int fputc(int c, FILE *fp)

Write a character c to file fp. Returns the character written.

<u>Example</u>

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

void main()
{
   FILE *fp;
   char *buffer = "this text is written to console";
   fp=fopen("con","w");

   while(*buffer)
      fputc(*buffer++, fp);
   fclose(fp);
```

```
    return;
}
```

fputs

#include <stdio.h>

int fputs(char *str, FILE *fp)

Write a string str to file stream fp.
Returns non-negative if successful, EOF if error occurred.

<u>Example</u>

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

void main()
{
    FILE *fp;
    char *buffer = "this text is written to console";
    fp=fopen("con","w");
    fputs(buffer,fp);
    fclose(fp);
    return;
}
```

fread

#include <stdio.h>

int fread(void *buf, int sizelem, int n, FILE *fp);

Read `n' items, each of size `sizelem' from file `fp' into buffer `buf'.
Returns the number of complete elements read.

<u>Example</u>

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

void main()
{
    FILE *fp;
    char *dest;
    int n;

    if((fp = fopen("tfile","r")) == NULL) return;
    dest = calloc(81,1);
    n = fread(dest,1,80,fp);
    printf("read %d bytes\n%s",n,dest);
    return;
}
```

free

```
#include <system.h>

void free(void *ptr)
```

Free memory block pointed to by `ptr'. The memory block described by ptr must have been allocated by calloc, malloc or realloc.

<u>Example</u>

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

void main()
{
  char *p;
  if((p = malloc(100)) == NULL) {
      printf("out of memory\n");
      return; }
  free(p);
  return;
  }
```

fscanf

```
#include <stdio.h>

int fscanf(FILE *fp, char *format,...);
```

Read characters from file `fp' and convert according to format string `format', storing via argument pointers which follow `format'.
See the description of`scanf' for input format specification.

Returns the number of arguments read from input.

<u>Example</u>

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

void main()
{
FILE *fp;
char fst[10], sec[20];
int n;

fp=fopen("con","r");
fscanf(fp, "%s %s %d",fst,sec,&n);
printf("You typed %s %s %d\n",fst,sec,n);
return;
}
```

fwrite

```
#include <stdio.h>

int fwrite(void *buffer,int sizelem, int n,FILE *fp);
```

Write `n' items, each of size `sizelem' to file `fp' from
buffer `buf'.

Returns the number of complete elements written.

getc

```
#include <stdio.h>

int getc(char *fp)
```

Get a single character from file `fp'. Returns the character, or EOF if input error.

<u>Example</u>

```
#include <stdio.h>

void main()
{
    int n;
    FILE *fp;

    fp = fopen ("tfile", "r");

     while ( !feof(fp) )
            putchar(getc(fp));

    return;
}
```

getchar

```
#include <stdio.h>

int getchar()
```

Get a single character from stdin. Returns the character, or EOF if input error.

<u>Example</u>

```
#include <stdio.h>

void main()
{
    int n;
    FILE *fp;

     while ((n=getchar())!=EOF)
            putchar(n);

    return;
}
```

gets

#include <stdio.h>

char *gets(char *buf)

Get string from stdin into buffer `buf'. Reads characters from file until newline or end-of-file. The string written to `buf' is null-terminated.

If a string was read into the buffer it's returned, else returns NULL.

<u>Example</u>

```
#include <stdio.h>

void main()
{
    char buf[80];
    gets(buf); puts(buf);
    return;
}
```

is-ctype

#include <ctype.h>

int isxx(char c)

Test a character to see if it's of a specified type.
If it is, a non-zero value is returned; if not, zero is returned.

isalnum	alphanumeric, letter or digit
isalpha	alpha, letter
isascii	ascii, 0-127
iscntrl	control character
isdigit	digit
isgraph	graphic, printable
islower	lowercase letter
isprint	printable
ispunct	punctuation
isspace	space character
isupper	uppercase letter
isxdigit	hex digit

malloc

```
#include <system.h>

void *malloc(int n)
```

Allocate buffer of n bytes from the heap,
Returns address of buffer, or NULL if no memory available.

<u>Example</u>

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

void main()
{
  char *p;
  if((p = malloc(100)) == NULL) {
      printf("out of memory\n");
      return; }
  free(p);
  return;
  }
```

memchr

```
#include <string.h>

void *memchr(void *buf, int c, int count);
```

Search buffer `buf' for a character `c'. The search stops when a character `c' is found, or after `count' bytes.

memcmp

```
#include <string.h>

int memcmp(void *buf1, void *buf2, int n);
```

Compare data in buffers `buf1' and `buf2', of size `n'.

Returns	= 0	buf1 and buf2 hold identical data
	< 0	first differing byte in buf1 < buf2
	> 0	first differing byte in buf1 > buf2

memcpy, memmove

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

void *memcpy(void *buf1, void *buf2, int count);
void *memmove(void *buf1, void *buf2, int count);

Copy `count' bytes from buffer `buf2' to buffer `buf1'.
Returns `buf1'.

memset

#include <string.h>

void *memset(void *buf, int val, int n);

Sets contents of buffer `buf' of size `n' to value `val'.
Returns buffer `buf'.

open

#include <io.h>

int open(char *name, int mode)

Open a disk file `name' using read/write mode `mode', which can be O_RDONLY, O_WRONLY, O_RDWR
returns file handle for the opened file, or EOF if there was an error opening the file.

The mode may be	0 read-only	O_RDONLY
	1 write-only	O_WRONLY
	2 read-write	O_RDWR

<u>Example</u>

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

void main()
{
   int fd;
   if (EOF == (fd = open("tfile",O_RDWR))) {
      printf ("failed to open tfile");
   }
   return;
}
```

printf

#include <stdio.h>

int printf(char *fmt, ...)

Print formatted values to screen. Arguments follow the format string and are interpreted

according to the format string.
The format string is a sequence of characters with embedded conversion commands.
Characters that are not part of the conversion command are output.

printf returns the number of characters written.

The format string can contain text values, or a format specification for arguments, one per argument, beginning with a `%' character, the type matching the argument's type;

% (conversion-flag) (min-field-width) (precision) (operation)

eg %-20s %12.4d

The conversion flag character can be one of

-	left adjust
+	force sign output
0	pad with 0 instead of space
(space)	produce sign `-' or space

Minimum field width is the minimum number of characters output for the field; if fewer characters are available from the argument, pad characters (0 or space) are inserted.

Precision is the minimum number of characters printed from a number, or the maximum number of characters printed from a string.

Operation specifies the expected argument type and what will be output; the expected output type must match the type of the corresponding argument for output to be meaningful.

`h'	short int
`l'	long int
`c'	character
`d' `i'	integer
`o'	octal integer
`p'	pointer
`x'	hexadecimal
`s'	string
`u'	unsigned

The number of characters written is returned.

<u>Example</u>
```
#include <io.h>
#include <stdio.h>

void main()
{
   char *msg = "number formats are: ";
   int n = 42;

   printf(fp,"%sx: 0%x d: %d o: %o\n",msg,n,n,n,n);
   return;
}
```

putc

#include <stdio.h>

int putc(int c, FILE *fp)

Write character \`c' to file \`fp'. Returns \`c'.

putchar

#include <stdio.h>

int putchar(int c)

Write character \`c' to stdout. Returns \`c'.

<u>Example</u>

```
#include <stdio.h>

void main()
{
   int n;
   FILE *fp;

   fp = fopen ("tfile", "r");

    while ( !feof(fp) )
          putchar(getc(fp));

   return;
}
```

puts

#include <stdio.h>

int puts(char *str)

Writes string \`str' to stdout, then writes a newline character.

<u>Example</u>

```
#include <stdio.h>

void main()
{
   char buf[80];
   gets(buf); puts(buf);
   return;
}
```

read

#include <io.h>

int read(int fd, void *buf, int n)

Read `n' bytes from file given by file descriptor `fd' into buffer `buf'. Direct DOS read from file.

Returns	-1	error
	0	end of file
	n	number of bytes read

<u>Example</u>

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

void main()
{
   char buf[80];
   int fd;
   if (EOF == (fd = open("tfile",O_RDWR))) {
      printf ("failed to open tfile");
   }
   read(fd,buf,80); puts(buf);
   return;
}
```

scanf

#include <stdio.h>

int scanf(char *format,...);

Read arguments from stdin. Parse it according to specification in `format' string, and assign input to arguments following the format string.

The arguments following `format' must be pointers to objects where the input is to be stored, and must match the specification in the format string.

Returns the number of arguments assigned to. If no arguments were assigned, an EOF is returned.

The format string may contain;

- space characters; skip whitespace characters in input.

- any other characters (except %) which should match input
  (match a `%' character by specifying `%%')

- input conversion specification

% (size) conversion-char

If the conversion specification is %*(size) c-char then the converted input is not stored and no argument is used.

If `size' is specified then at most `size' characters are converted. Otherwise conversion stops when an invalid input character is read.

The conversion character specifies the input object type and must match the type of argument pointer;

`h'	short int
`l'	long int
`c'	character
`d' `i'	integer
`x'	hexadecimal
`s'	string

<u>Example</u>

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

void main()
{
   unsigned n;
   printf("type a number: ");
   scanf("%i",&n);
   printf("number %d is %x hex",n,n);
   return;
}
```

sprintf

```
#include <stdio.h>

int sprintf(char *buf, char *fmt, ...)
```

Print formatted values to memory. Arguments follow the format string and are interpreted according to the format string.

sprintf returns the number of characters written.
See printf for description of format string.

sscanf

```
#include <stdio.h>

int sscanf(char *buf, char *fmt, ...)
```

Parse string in buffer `buf' according to specification in `format' string, and assign input to arguments following the format string.

The arguments following `format' must be pointers to objects where the input is to be stored, and must match the specification in the format string.

Returns the number of arguments assigned to. If no arguments were assigned, an EOF is returned.

strcat

#include <string.h>

char *strcat(char *buf1, char *buf2)

Catenate null-terminated string in `buf2' to the string in buffer `buf1'. Returns `buf1'.

strchr

#include <string.h>

char *strchr(char *str, int c)

Search string `str' for character `c', return first occurrence.

strcmp

#include <string.h>

int strcmp(char *str1, char *str2)

Compare null-terminated strings `str1' and `str2'. If they are identical then return 0. If the first differing character in `str1' is greater than that in `str2' then return a positive value, else return a negative value.

strcpy

#include <string.h>

char *strcpy(char *buf1, char *buf2)

Copy null-terminated string in `buf2' to buffer `buf1'.

strdup

#include <string.h>

char *strdup(char *str)

Create copy of null-terminated string `str' in newly allocated buffer, and return the buffer.

strlen

#include <string.h>

int strlen(char *str)

Return length of null-terminated string `str'.

strlwr

#include <string.h>

char *strlwr(char *str)

Convert string `str' to lowercase and return it.

strncat

#include <string.h>

char *strncat(char *buf1, char *buf2, int n)

Catenate null-terminated string in `buf2' to null-terminated string in `buf1'. At most `n' bytes are catenated.
Returns `buf1'.

strncmp

#include <string.h>

int strncmp(char *buf1, char *buf2, int n)

Compare null-terminated strings `str1' and `str2'. The strings are compared for at most `n' characters. If they are identical then return 0. If the first differing character in `str1' is greater than that in `str2' then return a positive value, else return a negative value.

strncpy

#include <string.h>

char *strncpy(char *buf1, char *buf2, int n)

Copy null-terminated string in buffer `buf2' to buffer `buf1'. Exactly `n' characters are copied. If `buf2' contains fewer than `n' characters, destination `buf1' is padded with nulls.

strnicmp

#include <string.h>

int strnicmp(char *buf1, char *buf2, int n)

Compares `n' characters of strings `buf1' and `buf2', ignoring case.

Returns	0	strings equal, ignoring case
	< 0	first differing character `buf1' < `buf2'
	> 0	first differing character `buf1' > `buf2'

strpbrk

#include <string.h>

char *strpbrk(char *str1, char *str2)

Search string `str1' for a character occurring in string `str2', return first occurrence.

strrchr

#include <string.h>

char *strrchr(char *buf, int c)

Search null-terminated string in buffer `buf' for character `c', returning pointer to last occurrence of `c' in `buf'.

strrev

#include <string.h>

char *strrev(char *str)

Reverses the contents of string `str'. Returns `str'.

strspn

#include <string.h>

int strspn(char *str1, char *str2)

Span string `str1' skipping any characters in string `str2'.

strupr

```
#include <string.h>

char *strupr(char *str)
```

Convert string `str' to uppercase, and return uppercased string.

tolower, toupper

```
#include <string.h>

int tolower(int c)
int toupper(int c)
```

Convert character `c' to lower/upper case.

ungetc

```
#include <stdio.h>

int ungetc(int c, FILE *fp)
```

Push character `c' back to input file `fp'. Only one character may be ungetc'd. Returns `c' if succeeded, else EOF.

<u>Example</u>

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

void main()
{
   FILE *fp;
   char c;

   fp = fopen("tfile","r");
   while((c = fgetc(fp)) != EOF)
      if(isspace(c))
        break;
      else
        putchar(c);

  ungetc(c,fp);
  fclose(fp);
  return;
}
```

vsprintf

#include <stdio.h>

int vsprintf(char *buf, char *fmt, char *args)

Prints the arguments pointed to by stack segment pointer `args' to output buffer `buf'.
See printf for description of format string,
Returns the number of characters written.

write

#include <stdio.h>

int write(int fd, void *buffer, int size);

Write memory buffer of length `size' to file `fd'. This is a direct DOS write to file.
Returns the number of characters written or -1 if error.

<u>Example</u>

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

void main()
{
   int n, fd;
   char *buf = "test data to be written";

   if((fd = open("tfile",O_WRONLY)) == -1)
      {
         puts("failed to open file");
         return;
      }
   n = write(fd,buf,strlen(buf));
   printf("%u bytes written\n",n);
   close(fd);
   return;
}
```