A Short C Primer

Charles E. Campbell, Jr., PhD.

October 12, 2010

Abstract

This document is a short primer to the C language.

Contents

1 Introduction **1**

2 Data **2**
2.1 Simple Variables . . . 2
2.2 Arrays . . . 2
2.3 Data Structures . . . 3
2.4 Pointers . . . 3
2.5 Complex Types . . . 4
2.6 Shorthand for Complex Types . . . 4
2.7 Scope . . . 5
2.8 Duration . . . 5
2.9 Prototypes . . . 6

3 Programming Constructs **6**
3.1 Functions . . . 6
3.2 Operators . . . 7
3.3 Expressions and Statements . . . 8
3.4 Choices . . . 9
3.5 Loops . . . 10
3.6 Pointers and Arrays . . . 11
3.7 Heap Memory . . . 12
3.8 Input and Output . . . 14

4 The Pre-Processor **15**

1 Introduction

This is a very short introduction to the C language. To really learn the language, I suggest that you purchase a book on C (or C++) such as

- *The C Programming Language, 2nd Edition* by Kernighan and Ritchie, ISBN 0-13-110362-8
- *The C++ Programming Language, 2nd Edition* by Stroustrup, ISBN 0-201-53992-6

This paper will introduce you to some C programming constructs, and will not cover C++ extensions. Many of those extensions, such as the concept of class and overloaded operators, are very useful, but this tutorial is intended to be short. Most "C" compilers these days are both C and C++ compilers, including the ones in GW's Robotics Laboratory.

This tutorial does not cover `union`s and bitfields.

Programs may be considered to have two co-equal parts: data and algorithms. Thus, this tutorial is split into two main sections on these topics.

All C programs begin executing in the `main` function. A famous example program appears in 1.1:

Example 1.1 *Hello World Program*

```
#include <stdio.h>

int main()
{
 printf("Hello, world\n");
}
```

Example 1.1 illustrates several interesting things:

1. The preprocessor: the `#include <stdio.h>` causes the compiler to read in the compiler's `stdio.h` file. This file sets up standard input/output routines, macros, etc.
2. All C programs must have a `main` function in them. The "Hello, world" program's main has no arguments. More generally, the call to `main` includes two arguments:

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

   The `argc` is a count of arguments on the command line. The `argv`, to be covered later, is a list of command line strings.
3. Strings are delimited by `"`, and may include special sequences such as `\n` (which is translated into a newline).
4. Standard input and output is done by a library of functions which follow standard C syntax for functions (in this case, `printf`).

2 Data

All C names, ie. variables, function names, data structure names, and keywords, must begin with an alphabetic character or an underscore (_). Subsequent characters may be alphanumeric or an underscore, and up to the first 31 are significant. Note: *case is important*!

2.1 Simple Variables

Simple variables in C are declared with the type shown in Table 1.

Name Type	Description
`void`	nothing
`char`	a byte (character)
`int`	integer
`float`	floating point number
`double`	a longer and thus more accurate number

Table 1: Simple Types

In addition, one may modify the types in Table 2.

`unsigned`	`signed`		Typical char modifiers
`unsigned`	`short`	`long`	Typical int modifiers
`const`	`volatile`		Special purpose modifiers

Table 2: Type Modifiers

Unsigned integers are zero or positive and typically can represent twice the largest positive value of a signed integer. Integers are signed by default (ie. `signed int` is the same as `int`).

Short integers typically are two bytes and long integers are typically four bytes long. Thus, long integers can represent much larger values than can short ones. An `int`'s length varies from compiler to compiler; it is usually either 2 or 4 bytes long. Often one can change the compiler's default length of an `int`, too.

Example 2.1 *Types*

```
int    i;
float  f;
double d;
short unsigned int sui;
```

2.2 Arrays

One may declare arrays of variables (or of data structures, or of more complex types) – see Ex 2.2.

Example 2.2 *Arrays*
```
int    i[5];
float  f[10];
double d[3];
```

2.3 Data Structures

The C language supports the concept of data structures with `struct`s. The square braces [] in 2.3 indicate that the *structure-name* is optional.

Example 2.3 *Data Structures*
```
struct [structure-name] {
  int   i;
  float f[3];
  };
```

The data structure of Ex. 2.3 set up a type but did not actually set up any variables of that type. One may do that as shown in 2.4:

Example 2.4 *Data Structures*
```
struct [structure-name] {
  int   i;
  float f[3];
  } ds1;
```

Once the data structure type has been set up, it may be reused more succinctly as shown in Ex. 2.5.

Example 2.5 *Data Structures*
```
struct MyData_str {
  int   i;
  float f[3];
  };
...
struct MyData_str mydatastr;
```

2.4 Pointers

Pointers are helpful in dealing with functions that need to modify variables in whatever called them, setting up dynamic linked lists, in dynamic memory allocation, etc. Unfortunately, they seem to be confusing to novice C programmers. Setting them up is simple, however:

Example 2.6 *Types*
```
int    *i;
float  *f;
double *d;
short unsigned int *sui;
```

Note that `int *i` does *not* allocate an integer; it sets up a pointer to an integer. Please see Section for more on how to use these.

One may declare pointers to structures, too, as shown in Ex. 2.7.

Example 2.7 *Pointers to Data Structures*
```
struct MyData_str *ptr_mydatastr;
```

2.5 Complex Types

One of C's strengths and weaknesses is its ability to construct complex data types. It is a strength in that it provides considerable flexibility to the programmer. It is a weakness in that, as with any flexibility, some programmers will build inscrutable messes.

Example 2.8 *Complex Types*

`int x;`	*is an integer*
`int *x;`	*is a pointer to an integer*
`int **x;`	*is a pointer to a pointer to an integer*
`int ***x;`	*is a pointer to a pointer to a pointer to an integer*
`int x[5];`	*is an array of 5 integers*
`int *x[5];`	*is an array of 5 pointers to integers*
`int *(x[5]);`	*same as above*
`int (*x)[5];`	*is a pointer to an array of 5 integers*
`int (*f)(void);`	*is a pointer to a function taking no arguments, returning an integer*
`struct RobotLink rlink;`	*is a RobotLink structure*
`struct RobotLink *prlink;`	*is a pointer to a RobotLink structure*
`struct RobotLink *prlink[5];`	*is an array of 5 pointers to RobotLink structures*
`struct RobotLink *(prlink[5]);`	*same as above*
`struct RobotLink (*prlink)[5];`	*is a pointer to an array of 5 RobotLink structures*

As an example of "inscrutable" types:

Example 2.9 *A few inscrutable types*

`int (*(*f)())()`	*ptr to a function returning a ptr to a function returning an integer*
`int (*(*(*f)())())[5];`	*ptr to a function returning a ptr to a function returning a ptr to an array of 5 integers*

2.6 Shorthand for Complex Types

In order to avoid "inscrutable" types, and often just as a shorthand for data structures, the `typedef` comes in handy.

Example 2.10 *Typedefs*

`typedef int *Pint;` – `Pint` *is now "shorthand" for pointer-to-integer*
`Pint px;` – `px` *is now a variable of type pointer-to-integer*
`typedef structure RobotLink_str RobotLink;` – `RobotLink` *is now a new type*
`RobotLink rl1;` – `rl1` *is now an instantiation of a* `struct RobotLink_str`.

2.7 Scope

Variables declared inside functions are visible only inside that function. Variables declared outside functions may be declared normally (as done in preceding sections) and are potentially visible to functions outside the file which declared them (ie. global scope). Variables declared `static` outside of functions are only visible to those functions in that same file (ie. file scope). See Table 3.

File 1	File 2	Coments
`int x;`	`extern int x;`	`x` declared in file1, visible to file2
`static int x;`		`x` declared in file1, not visible to file2

Table 3: Scope

2.8 Duration

Variables declared inside functions are normally *dynamic* in duration. This means that they exist only while the function is executing, and in fact, if the function calls itself, there will be a separate copy of those variables for each instance of the executing function. Such variables are often implemented by the use of a stack.

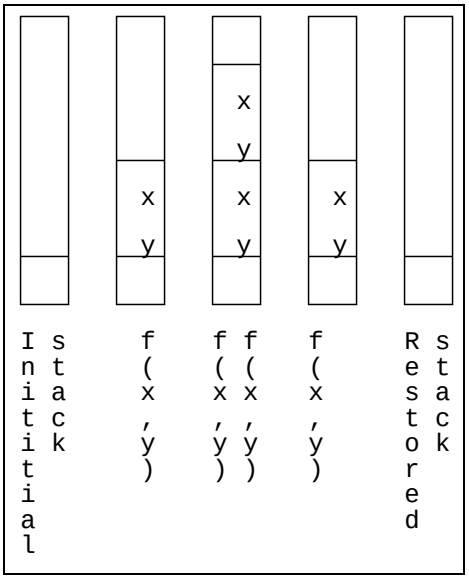


Figure 1: Dynamic Memory Stack

In Fig 1, the stack initially starts with "something". The function `f(x,y)` is then called. The dynamic variables `x,y` are pushed on the stack. The function `f(x,y)` then calls itself, and so another copy of the function's variables is pushed onto the stack. When the second copy of `f(x,y)` returns, the first instance of `f` still has its two variables to use. Finally, when the first instance of `f` returns, no variables of `f` are left on the stack.

Variables declared outside of functions (with either file or global scope) are not dynamic, they are permanent. To get a function's variables to become permanent, the keyword `static` is re-used:

merely precede their normal declaration by that keyword.

Example 2.11 *Static Duration, Function Scope Variables*

```
int f(int x)       begin function f
{
static float f;    this variable is permanent, but only one copy
float e;           this variable exists only when f is executing
}
```

2.9 Prototypes

ANSI C (also known as Standard C) introduced *prototypes* to the C language. These are simply declarations of functions and their arguments. They greatly facilitate automatic error checking; ie. that you're using your functions correctly.

Example 2.12 *Prototypes*

```
int main(int,char *);
int myfunc(int,double,struct ABC *);
void RobotMover(double *);
```

Prototypes tell the compiler what your functions should return. Thus, if your function returns an integer, but you try to stuff it into a pointer variable, the compiler will flag that problem for you.

3 Programming Constructs

The other "half" of a program is its algorithms. These are implemented using various keywords and programming constructs. In C (not C++), all variables must be declared at the top of functions. Executable statements follow the variable declarations.

3.1 Functions

Functions follow the general format:

Example 3.1 *Function Format*

```
type func-name ( var-list )
{
variable declarations
executable statements
}
```

You may have been mystified by the `void` type given in Section 2.1; frequently the "type" of a function will be `void` – ie. it doesn't return anything. However, if you do want a function to return something, C provides the `return` keyword:

Example 3.2 *How functions return data*

```
return expr;
```

An `expr` is a general C expression (see Section 3.3).
In Ex 3.3, a complete, albeit small, function which computes the secant of x is given.

Example 3.3 *A simple function*

```
/* sec.c: the secant of x */    illustrates a comment
#include <math.h>               math functions need this standard header file
double sec(double x)
{
 return 1./cos(x);              compute the secant of x using cos()
}
```

3.2 Operators

Like most programming languages, C provides a number of operators. It has a few that you may not have seen before, too. Table 3.2 gives a list of all C's operators (C++ even has some more!), a very brief description, associativity (ie. which side of the operator is evaluated first), and syntax. Operators with differing precedence are separated by horizontal lines. The highest precedence operators appear at the top of the table; the lowest precedence operator (the comma), appears at the bottom. You may use parentheses `()` to change precedence, too.

Operators and Precedence			
Operator	*Description*	*Associativity*	*Syntax*
`()`	function call	←	*expr (expr_list)*
`[]`	subscripting	←	*pointer [expr]*
`->`	member selection	←	*pointer* `->` *member*
`.`	member selection	←	*struct . member*
`!`	logical not	←	*! expr*
`~`	bitwise not	←	*~ expr*
`++`	post increment	←	*lvalue ++*
`++`	pre increment	←	*++ lvalue*
`--`	post decrement	←	*lvalue* `--`
`--`	pre decrement	←	*--lvalue*
`-`	unary minus	←	`--` *expr*
`*`	dereference	←	** expr*
`&`	address of	←	`&` *lvalue*
`(type)`	address of	←	`&` *lvalue*
`sizeof`	address of	←	`&` *lvalue*
`*`	multiply	←	*expr * expr*
`/`	divide	←	*expr / expr*
`%`	modulus	←	*expr % expr*
`+`	add	←	*expr + expr*
`-`	subtract	←	*expr - expr*
`<<`	bitwise shift left	←	*expr* `<<` *expr*
`>>`	bitwise shift left	←	*expr* `>>` *expr*
`<`	logical less than	←	*expr* `<` *expr*
`<=`	logical less than	←	*expr* `<=` *expr*
`>`	logical less than	←	*expr* `>` *expr*
`>=`	logical less than	←	*expr* `>=` *expr*
`==`	logical equal to	←	*expr* `==` *expr*
`!=`	logical equal to	←	*expr* `!=` *expr*
`&`	bitwise and	←	*expr* `&` *expr*
`^`	bitwise exclusive or	←	*expr* `^` *expr*
`|`	bitwise exclusive or	←	*expr* `|` *expr*
`&&`	logical and	←	*expr* `&&` *expr*
`||`	logical or	←	*expr* `||` *expr*
`?:`	if else	→	*expr? expr : expr*
`=`	set variable	→	*lvalue* `=` *expr*
`+=`	add to variable	→	*lvalue* `+=` *expr*
`-=`	subtract from variable	→	*lvalue* `-=` *expr*
`*=`	multiply into variable	→	*lvalue* `*=` *expr*
`/=`	divide into variable	→	*lvalue* `/=` *expr*
`%=`	modulus into variable	→	*lvalue* `%=` *expr*
`&=`	bitwise and into variable	→	*lvalue* `&=` *expr*
`^=`	^ bitwise xor into variable	→	*lvalue* `^=` *expr*
`|=`	— bitwise or into variable	→	*lvalue* `|=` *expr*
`<<=`	<< bitwise or into variable	→	*lvalue* `<<=` *expr*
`>>=`	>> bitwise or into variable	→	*lvalue* `>>=` *expr*
`,`	expression separator	←	*expr , expr*

3.3 Expressions and Statements

One may combine operators with variables and "lvalues" (*lvalues* are typically variable names that may go on the left-hand-side of an "`=`" operator) to form *expressions*. Ex 3.4 illustrates some simple

expressions.

Example 3.4 *Simple expressions*

```
x+2
x*(y+3)
5*sin(x)
5 + (x=y)
```

Expressions may be converted into statements by terminating them with a semi-colon as done in Ex. Simple-Statement.

Example 3.5 *Simple statements*

```
 int x,y,z;                                  a declarative statement
 x=3;;                                       a simple assignment statement
 x= 5*(cos(x) + sin(y))*pow(sqrt(3),5);      another statement
```

One may form *compound-statements* by enclosing a number of statements in curly braces as done in Ex. 3.6. Note that functions are basically function declarations followed by a compound statement.

Example 3.6 *A compound statement*

```
 {
   int x,y,z;                                  a declarative statement
   x=3;;                                       a simple statement
   x= 5*(cos(x) + sin(y))*pow(sqrt(3),5);      another statement
   }
```

3.4 Choices

C provides two main ways of accomplishing choice selection.

Example 3.7 *If-Then-Else Expressions*

```
if     (expr) stmt
else if(expr) stmt
else          stmt
```

The `else if(expr) stmt` and `else stmt` clauses are optional, and the `stmt` (statement) may be a compound-statement.

The second primary way of accomplishing choice selection is to use the switch-case construct as in Ex. 3.8.

Example 3.8 *Switch – Case*

```
switch(expr) {
case constant-expression: stmt-list
case constant-expression: stmt-list
...
default:
```

The `constant-expression` is a number (1, 2., etc) or a character ('a', 'X', etc). You can't put variables or expressions there, sorry! C will drop through cases (unlike Pascal, for example). If you want, as one normally does, to execute the statements associated with a case and not the subsequent ones, use `break` as your last statement in each case's `stmt-list`.

Example 3.9 *More Switch – Case*

```
switch(*s) {
case 'a':
  aflag= !aflag;
  break;
case 'b':
  bflag= !bflag;
  break;
default:
  fprintf(stderr,"***error*** *s<%s> is unsupported as yet\n",*s);
  exit(1);
  }
```

Ex. 3.9 shows a simple example of a pointer to a character. The `*s` expression in the `switch()` dereferences (ie. looks up) the character pointed to by `s`. Then, if it was the character `a`, the `aflag` is toggled. If it was the character `b`, the `bflag` is toggled. Any other character will trigger the dreaded error message and terminate the program.

3.5 Loops

C provides three ways to do loops. Note that loops' expressions may be comma separated (ie. `x=0, y=1, z=2` is one expression). C treats 0 as meaning *false*, any other value means *true*. The three loops' syntax are given in Table 4.

Syntax	Description
`for(expr1;expr2;expr3) stmt;`	initialize with `expr1`; test with `expr2`, execute statement, and update with `expr3`
`for(expr1;expr2;expr3) {stmt-list}`	initialize with `expr1`; test with `expr2`, execute statements, and update with `expr3`
`while(expr) stmt;`	while `expr` is true, do the `stmt`
`while(expr) {stmt-list};`	while `expr` is true, do the `{stmt-list}`
`do stmt ; while(expr2);`	do the statement, then continue if `expr2` is true
`do {stmt-list} while(expr2);`	do the list of statements, continuing if `expr2` is true

Table 4: Loops

The `for` loop's first expression is executed, and then the test (`expr2`) is made. So long as that expression evaluates to true, the `stmt`/`stmt-list` is executed. The update `expr3` is then made, and the test-stmt-update cycle so continues. One may `break` out of any of the looping constructs, and one may also `continue` to bypass the rest of the `stmt-list` in any of them.

The `while` loop's test `expr` is done prior to executing its associated `stmt` or `stmt-list`. The `do`–`while`'s test is evaluated after its associated `stmt` or `stmt-list`.

Example 3.10 *An example for loop*

```
for(i= 0; i < 4; ++i) printf("i=%d\n",i);
```

i=0 – *results* –
i=1
i=2
i=3

Example 3.11 *An example while loop*

```
i= 0;
while(i < 4) {
  printf("i=%d\n",i);
  ++i;
  }
```

i=0; – *results* –
i=1
i=2
i=3

Example 3.12 *An example do-while loop*

```
i= 0;
do {
  printf("i=%d\n",i);
  ++i;
  } while(i < 4);
```

i=0; – *results* –
i=1
i=2
i=3

In Ex. 3.10 – Ex. 3.12, the same output results. The `for` loop has its initialization, test, and update (`++i` means to increment i by one) in its construct. The `while` and `do–while` loops have their initialization prior to the loops themselves, and their updates (the `++i`) occurs inside their `stmt-lists`. Note that a `continue` inside those latter two loops would not have caused the update code to be executed.

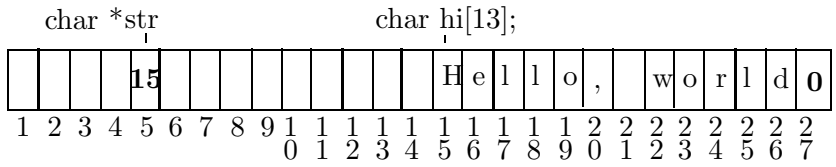
3.6 Pointers and Arrays

C treats pointers and arrays in a similar fashion but they are not identical. A pointer is typically four bytes long (sometimes two for certain memory models), and basically just contains an integer. This "integer", however, is used to "look up" other location(s) in memory. In Fig 2, both a pointer and an array have been declared using Ex. 3.13.

Example 3.13 *A simple pointer and array*

```
char   hi[13]="Hello, world";   an array
char *str= hi;                  a pointer
```

Figure 2: Pointers and Arrays



The `hi` is an array. The word `hi` is itself a label that the compiler knows about and uses, but the label itself does not take up any space in the program. However, the pointer `str` does take up space, as it holds an address (in this case, 15). The array begins at address 15. To get the character "H", you can use `hi[0]` or `*str`. To get the character "e", you can use `hi[1]` or `*(str+1)`. In fact, C even lets you pretend that the pointer is an array, and so you can also access "H" and "e" with `str[0]` and `str[1]`, respectively.

Note that C strings are terminated by a null byte (a 0). Address 27 holds the null byte for the "Hello, world" string. In the example, I reserved 13 bytes for the `hi` array, but C will do the counting in Ex 3.14.

C only lets you do array or structure initialization to static (in duration) variables. All, including dynamic ones, can be initialized to simple values.

Example 3.14 *A simple pointer and array*
```
char   hi[]="Hello, world";
```
– an array

3.7 Heap Memory

So far, you've been introduced to static and dynamic duration variables (and file and function scope ones). Static duration variables reside somewhere in memory, and dynamic ones reside on the stack. Yet a third option exists: variables "on the heap." These variables are allocated by the program while it is running using `malloc()`, `calloc()`, `realloc()`, and free'd using `free()`.

You will need to look up a book on the details of these functions. An example of a doubly-linked list appears in Ex 3.15.

Example 3.15 *Doubly linked lists*
```
typedef struct ABC_str ABC;
struct ABC_str {
```

```
  int a;
  int b;
  int c;
  ABC *nxt;
  ABC *prv;
  };
ABC *abc_head= NULL;
ABC *abc_tail= NULL;
...
char *abc;
...
abc= (ABC *) malloc(sizeof(ABC));
if(abc_tail) abc_tail->nxt= abc;
else         abc_head     = abc;
abc->prv= abc_tail;
abc->nxt= NULL;
abc_tail= abc;
...
for(abc= abc_head; abc; abc= abc->nxt) {
  printf("abc: a=%d b=%d c=%d\n",abc->a,abc->b,abc->c);
  }
```

An `ABC` data structure which contains three members and two links is defined. Pointers to the head (beginning) and tail (end) of the `ABC` list (`abc_head` and `abc_tail`). Initially, there are no members, and so the head and tail pointers are `NULL`. As needed, new `ABC` structures are allocated "from the heap" and links are set up. Thus, if `abc` points to a linked `ABC` structure, the preceding structure on the list is `abc->prv` and the subsequent one is `abc->nxt`. A simple `for` loop is shown which goes through the entire `ABC` linked list (from head to tail), printing out each one's three elements (`a`, `b`, and `c`).

Linked lists are heavily used by programs that are designed to handle varying amounts of information. For example, when writing a linear equation solver, instead of allocating, say, a 1000×1000 array (`big_array[1000][1000]`) and hoping that no-one will need a larger array (and making all the 3×3 types take up inordinate amounts of memory), the programmer will allocate memory dynamically using `calloc` as done in Ex 3.16.

Example 3.16 *Allocating an array*

```
int      ia;
double **array;
array= (double **) calloc((size_t) nrows,sizeof(double *));
for(ia= 0; ia < ncols; ++ia) {
  array[ia]= (double *) calloc((size_t) ncols,sizeof(double));
  }
```

In Ex 3.16, `array` is a pointer to a pointer to double. The first executable statement allocate `nrows` pointers-to-double. The `for` loop then allocates a separate vector of `ncols doubles` for each `array[ia]`. Note that `nrows` and `ncols` could be defined by input to the program, and so the programmer here has sized his memory use to what the user needs, and is imposing no possibly harmful constraints or inefficiencies.

In both the doubly-linked list example and the `calloc` example, flexibility comes at the price of extra work for the programmer. C'est la vie!

3.8 Input and Output

C does not provide any input or output primitives, rather the entire burden of input and output lies upon the standard libraries provided with C. ANSI C contains a standardized set of such input and output routines:

`getc, fgetc, fgets, putc, puts, fputs, printf, scanf`,

etc. One also needs to open and close files using the functions `fopen, fclose`.

Type	*Terminal Input*	*Terminal Output*
Character	`int getchar(void)`	`int putchar(int)`
String	`char *gets(char *)`	`int puts(char *)`
	File Input	*File Output*
Character	`int fgetc(FILE *)`	`int fputc(int,FILE *)`
String	`char *fgets(char *,int,FILE *)`	`int fputs(char *,FILE *)`

The code in Ex 3.17 illustrates how to put and get a character to and from the user's terminal and keyboard or to and from a file. In addition, it illustrates how to open a file for reading or writing and to close it.

Example 3.17 *Putting and Getting a character*

```
char c;
FILE *fp;
putchar('a');                  puts a single character 'a' out to the terminal
c= getchar();                  gets a character and stores it into variable c
fp= fopen("myfilename","r");   opens a file <myfilename> for reading
c= fgetc(fp);                  gets a character from the file <myfilename>
fclose(fp)                     close the file
fp= fopen("myfilename","w");   opens a file <myfilename> for reading
fputc('b',fp);                 puts a character 'b' in the file <myfilename>
fclose(fp)                     close the file
```

One may also work with `scanf` and `print`; the latter function has been used several times in preceding sections. C manuals will cover these two functions in depth; for now, I will cover a few of the format codes.

Example 3.18 *Reading and writing from the terminal*

Writing	Reading
`float f=1.;` `double d=2.;`	
`print("f=%f\n",f);` → f=1.000000	`scanf("%f",&f);`
`print("d=%lf\n",d);` → d=2.000000	`scanf("%lf",&d);`

In Ex 3.18, there are two variables declared, one float (ie. 4 bytes) and one double(ie. 8 bytes). The `%f` form deals with the shorter `float` type of variable, and the `%lf` deals with the *l=/*onger `double`. In the `printf` side, note that the `f` and `d` are printed to six decimal places. You could optionally control this with a `%W.Df` style of output command (ex. `%7.2f`). The `W` is the width of the entire field to be printed, and the `D` part is the number of digits to the right of the decimal point.

In Ex 3.18, the `scanf` side is reading two numbers from the terminal (ie. user). Note that the variables are preceded by an ampersand (`&`). That's because we're not really interested in what scanf itself is returning (the number of items matched, by the way), but we're interested in a side effect – modifying the values of `f, d`. Remember that C allows recursion (functions can call themselves), and it does so by pushing variables onto a dynamic stack. Thus, functions (and `scanf` in particular) don't have access to the variables passed to it, they have access to the *values* passed to it.

The ampersand (`&`) is the operator which takes the address of a variable. Thus, the `scanf` functions are passed a copy of the address of the variables `f` and `d`. Internally, `scanf` then writes to the locations pointed to by the variables passed to it. Forgetting the `&` is one of the most common mistakes made by beginning C programmers. What's worse is that the `scanf` function takes an arbitrary number of variables of arbitrary types, so the prototype for `scanf` doesn't help the compiler figure out that sort of problem.

Format Code	Meaning
`%c`	handles char
`%uc`	handles unsigned char
`%s`	handles char * (strings)
`%d`	handles int
`%ld`	handles long
`%f`	handles float
`%lf`	handles double
`%p`	handles pointers
`%u`	handles unsigned
`%lu`	handles unsigned long
`%x`	handles unsigned (hexadecimal)
`%lx`	handles long (hexadecimal)

Table 5: Format Codes

In Table 5, the format codes are briefly given and explained.

4 The Pre-Processor

The pre-processor is sort of a language inside a language! It supports including other files, conditional compilation, setting up constants, and defining macros. All pre-processor commands begin with a `#` which must appear in the first column.

You can use `#define LABEL ***` where `***` is any text you'd like to have. Whenever the pre-processor sees `LABEL` (whatever label you choose), it will substitute the `***` text for it. See

Ex 4.1.

Example 4.1 *A simple* `#define`
`#define BUFSIZE 256` – *wherever* `BUFSIZE` *appears,* 256 *will be substituted for it*

You can make more complicated *macros* with the same pre-processor command – see Ex 4.2.

Example 4.2 *A* `#define` *macro*
`#define abs(x) (((x) < 0)? -(x) : (x))` – *a macro*

Using Ex 4.2, wherever `abs(...)` is encountered, it will be expanded to

```
(((...) < 0)? -(...) : (...))
```

Macros appear to be functions, but they are expanded in-line, not called. You have to watch for unintended side effects – guess what happens when you try `abs(++x)`!

Including other files is simple; see Ex 4.3.

Example 4.3 *Including files*

```
#include <systemfile.h>   system header files
#include "yourfile.h"     your header files
```

The first form (with angle brackets) is used to include system files such as `stdio.h`, `math.h`, `stdlib.h`, etc. The second form is useful for including header files that you yourself have defined.

Finally, the C pre-processor has several commands to handle conditional compilation. Perhaps the easiest is the `#ifdef` types as shown in Ex 4.4. It is used to include and/or omit code based on the `#define` status of labels.

Example 4.4 *Ifdef-Else-Endif*

```
#ifdef A_LABEL          if A_LABEL has been #define'd, then...
...code segment 1...
#else                   otherwise (this part is optional)
...code segment 2...
#endif
```

The second conditional compilation form supports expressions; see Ex 4.5.

Example 4.5 *If-Else-Endif*

```
 #if expr               expression may include operators
 ...code segment 1...
 #elif expr             as many #elifs as you like, including none
 ...code segment 2...
. #elif expr
 ...code segment 3...
 #else                  otherwise (this part is optional, too)
 ...code segment 4...
 #endif
```

There's one additional "function" supported by the `#if` expressions: `#defined(...)`, which evaluates to 1 if the `...` label is defined by a `#define` previously or 0 otherwise. Expressions can include operators such as `|| && | & < > <= >= ! ()`.