PHYS-4007/5007: Computational Physics
Course Lecture Notes
Appendix E

Dr. Donald G. Luttermoser
East Tennessee State University

Version 7.0

Abstract

These class notes are designed for use of the instructor and students of the course PHYS-4007/5007: Computational Physics I taught by Dr. Donald Luttermoser at East Tennessee State University.

Appendix E. Scientific Computing Using C

A. Tutorial Introduction to C

1. Here is everyone's first C program: It print the words "hello, world!" on the screen.

```
/*--------------------------------------------------
  hello.c
  j.g.c. 31/1/93
  from K&R
--------------------------------------------------*/
#include <stdio.h>
int main(void)
{
  printf("hello, world!\n");
  return 0;
}
```

a) A "C" program = functions (executable code) + variables (data).

b) Every program must have a function called main, execution starts there. main may call other functions. These functions can be taken from one of the following sources:

i) As written by the programmer in the same file as main.

ii) As written by the programmer in separate file(s).

iii) Called from predefined libraries.

c) C has no concept of a Program as in Fortran, just functions.

d) The C view is that main is called by the operating system; the operating system may pass arguments to main, as well as receive return values (*e.g.*, return 0;); however, above, we choose to make main take no arguments – (void).

e) /* ... */ is a comment and is ignored by the compiler; the C standard says that comments cannot be nested. Also, comments **must** be terminated explicitly, *i.e.*, newline does not terminate them — this is a common source of compiler errors, and, for the unwary, can be very difficult to trace. For your own sanity the header comment should include:

i) Name of the program — to correspond to the filename.

ii) Authors name — even if copied!

iii) Date.

iv) Brief indication of function and purpose of the program, *e.g.*, assignment number, or project, and what it does.

f) Program layout is very important; I suggest two spaces indentation for each block; I suggest that you avoid tabs - certainly, avoid having the program pressed up against the right margin, with all the white space on the left hand side.

g) The #include <stdio.h> is a directive to the C preprocessor to include the contents of a file called stdio.h; stdio.h contrains declarations of *STanDard Input-Output* functions; include means that the full contents of the file stdio.h are inserted where the #include directive appears; only then is the program passed to the C-compiler proper. #include is C's way of importing.

h) In the definition of a function, () encloses the argument list; in this case main expects no arguments; (void) explicitly denotes *has no arguments*.

i) { ... } enclose the statements in a function.

j) printf is a library function for printing output on the users screen — in this case the string of characters between quotes; \n is C notation for *newline*; printf will not insert a newline by itself.

k) \n represents a single character — neccessary, since typing the <return> key in the middle of a string is not really practical! Other control characters are: \t = tab, \a = alert bell, \" = double quote, \r = carriage return, \\ = backslash itself!

l) return 0; in C, programs can return values to the operating system; in UNIX, '0' signifies that the program terminated normally; other values indicate error conditions. This is useful if you intend putting your program in a UNIX shell script; likewise DOS '.bat' files.

B. Variables and Arithmetic.

1. Convert degrees Fahrenheit to degrees Celsius:

$$T_C = \frac{5}{9}\left(T_F - 32\right).$$

Then, print a list as follows:

0	-17
20	-6
40	4
60	15
300	148

```
/*-----------------------------------------------
  ftoc.c  - Fahr. to Celsius table
  j.g.c.
  3/10/89
  Copied from K&R p.9
-----------------------------------------------*/
#include <stdio.h>
int main(void)
```

```
{
  int fahr, celsius;
  int lower, upper, step;

  lower = 0; /* lower limit of table - fahr.*/
  upper = 300; /*upper limit*/
  step = 20;   /*step size*/

  fahr = lower;
  while(fahr <= upper){
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

2. Dissection:

a) All variables must be declared. Usually at the beginning of their function.

b) Built-in types in C:

int:	integer, can be 16-bits, or 32-bits
float:	floating point
char:	single text character, really it is a small integer taking on values in [0...255].
short:	short integer; possibly 8 bits, maybe 16.
long :	long integer - 32 bits.
double:	double precision floating point — more significant digits, larger exponent.

c) Assignment statement: upper = 300;

d) While loop:

```
while(fahr <= upper){
/* executable statements — body in here */
}
```

Operation of while:

i) Condition is tested.

ii) If condition is true — body is executed.

iii) Go back to 1.

iv) However, if condition is false — execution continues at the statement following the loop.

3. Note the textual layout style used: as mentioned earlier, my suggestion is to indent each block by 2 characters — the Kernighan & Ritchie (K&R) book on C (*The C Programming Language* — considered by many to be the C bible) use the next tab, but with that you very quickly get to the right-hand side of the page; on the other hand, we have to be able see what is part of a block and what isn't.

4. For me it is essential that the closing brace lines up with `while`.

5. Integer arithmetic causes truncation: hence $5/9 \rightarrow 0$.

6. `printf`:

a) Has multiple arguments; the first is always a string constant — then, this may contain codes (*e.g.*, '`%d`') to say how subsequent arguments are to be printed.

b) `%d`: print the second argument as a decimal integer.

c) `printf` is not part of the C language — but it's in the standard library, which is available with all C compilers.

d) Field widths can be specified, *e.g.*, `%6d`.

e) Uses right justification, unless specified otherwise.

7. In ftoc.c, we have a problem with integer arithmetic: not really satisfactory — truncation — *e.g.*, $5/9 = 0$! Therefore, we need a floating point version of ftoc.c:

```
/*-------------------------------------------------
  ftoc1.c  - Fahr. to Celsius table, floating point version
  j.g.c.
  3/10/89
  Copied from K&R p.12
-------------------------------------------------*/
#include <stdio.h>
int main(void)
{
  float fahr, celsius;
  int lower, upper, step;

  lower = 0; /* lower limit of table - fahr.*/
  upper = 300; /*upper limit*/
  step = 20;   /*step size*/

  fahr = lower;
  while(fahr <= upper){
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%3.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

8. Dissection:

a) Mixing of operands is allowed in C, *e.g.*, fahr = lower; while(fahr <= upper); but this does not mean that there is no type checking; in the examples given, the *ints* are converted to *floats* before the operation is done. In general, you should be very careful with this feature. Actually, it might be better to make type conversion explicit, this can be done with a type *cast*, thus:

```
fahr = (float)lower;
```

b) f conversion specification in printf:

i) %3.0f: 3 characters wide, no decimal point, no fraction digits.

ii) %6.1f: 6 characters wide, decimal point, plus 1 fractional digit.

c) Other conversions:

i) %d: print as decimal integer — no width specified.

ii) %6d: decimal integer at least 6 chars wide.

iii) %f: floating point — no width spec.

iv) %6f: float — 6 chars wide.

v) %.2f: 2 chars after decimal point, width unconstrained.

vi) %x: hexadecimal.

vii) %o: octal.

viii) %s: char string.

ix) %%: for % itself!

C. The for Statement

1. Another version of ftoc.c: ftoc2.c:

```
/*-------------------------------------------------
  ftoc2.c  - Fahr. to Celsius table, demo of for statement.
  j.g.c.
  3/10/89
  Copied from K&R p.13
-------------------------------------------------*/
#include <stdio.h>
int main(void)
{
  int fahr;
  for(fahr=0;fahr<=300;fahr=fahr+20){
```

```
    printf("%d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));

  }
  return 0;
}
```

2. Major differences:

a) In printf(), celsius is replaced by a complex expression that evaluates to celsius; this should be nothing new to those who have used Fortran; the general rule is:

> A variable of some type can be replaced by an arbitrarily complicated expression that evaluates to a value of that type.

b) for statement and loop: There are three statements contained within the '(.)' in a for statement:

i) fahr = 0: initialization.

ii) fahr <= 300: loop continuation control; evaluate the condition — if true execute the body — otherwise jump out.

iii) fahr = fahr + 20: increment; do this AFTER the first and subsequent loops — BEFORE evaluation of control condition.

iv) The for(...) body may be a single statement or a block { ... } .

D. Symbolic Constants and the Preprocessor.

1. The general form of a symbolic definition is:

#define <symbolic-name> <replacement-text>.

2. The C preprocessor replaces all occurrences of <symbolic-name> with <replacement-text> — just like an editor global-replace command.

3. Final version — ftoc3.c:

```
/*-----------------------------------------------
  ftoc3.c  - Fahr. to Celsius table, demo of Symbolic
             constants
  j.g.c.
  3/10/89
  Copied from K&R p.15
-----------------------------------------------*/
#include <stdio.h>

#define LOWER 0 /*lower limit of table*/
#define UPPER 300
#define STEP 20

int main(void)
{
  int fahr;
  float celsius;

  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP){
    celsius = (5.0/9.0) * (fahr-32);
    printf("%3d %6.1f\n", fahr, celsius);
  }
  return 0;
}
```

4. Dissection:

a) The preprocessor is an extremely important part of C. When you *compile* a C program, two things happen: First, the preprocessor runs through the file, second, actual compilation. Therefore, in the example given above, all occurrences of the string "LOWER" get replaced with "0"; the compiler never sees the "LOWER"; and, LOWER is not a variable.

b) Adopt a convention to use upper-case for symbolic constants; be very careful what else you use upper case for.

c) There is no ';' after the '#define' statement,

d) Nor does a #define statement contain an '='.

E. Character Input and Output.

1. There is a standard input-output library which deals primarily with streams of characters, or *text-streams*; cf. UNIX files.

a) A **text stream** = a sequence of characters divided into *lines*.

b) A **line** = zero or more characters followed by a newline character.

2. The standard input-output library MUST make each input-output stream conform to this model — no matter what is in the physical file.

3. c = getchar() /* read next input char from the standard input stream, i.e., usually the keyboard */

4. putchar(c) /* put/print char 'c' on the screen — standard output stream */

5. There are equivalent functions for files.

6. Buffered and Echoed Input.

a) On most computers, input from the keyboard is *echoed* and *buffered*.

b) **Echoed input.** When you type a character on the keyboard, the computer input-output system immediately

echoes it to the screen / terminal; *immediately* means BEFORE it is presented to the reading program (see buffered below). (Incidentally, this means that the terminal, itself, does not display the typed character — just what is echoed from the host computer.)

c) **Buffered input.** While the user is typing, the computer stores all the typed characters in a buffer (array) and presents the array (line) of characters to the reading program ONLY AFTER A CARRIAGE RETURN (Enter) has been typed.

7. Terminal input-output and I-O Redirection

a) C (and UNIX) has a fairly unified view of text input-output. Reading from the keyboard is like reading from a file device called stdin. However, there are special stdin routines that hide that fact, *e.g.*, getchar() below is exactly equivalent to getc(stdin). Likewise, putchar(c) is equivalent to putc(c, stdout), where stdout is the *standard-output* device, *i.e.*, the screen.

b) Incidentally, putc(c, stdprn) is a handy way of writing to a PC printer, as is putc(c, stdaux) — to the auxiliary (communications) port, where stdprn is the *standard printer* device, and stdaux is the *standard auxiliary* device. **BUT THESE ARE NOT STANDARD C / UNIX.**

c) So, when we talk about *file* I-O below, we include terminal I/O. All the programs below can be tested using the terminal. (Note: <Ctrl> z is EOF (end-of-file) for a keyboard).

d) If you want to test the programs using files, you can use input-output redirection.

i) First, you must get the compiler to compile to an .exe file, say cio.exe.

ii) Second, create a text file, say test.dat. Then, enter at the Unix prompt

cio < test.dat

which tells cio to *read* from test.dat $\Longrightarrow$ the < redirects the program to read from the file instead of the keyboard.

iii) If you want to send the output to a file, say testout.dat, use

cio < test.dat > testout.dat.

8. File Copying. Make a program with the following algorithm:

read a char
while(char is not an eof char)
output the char just read
read a char

```
/*-------------------------------------------------
  cio1.c - copy input to output, version 1.
  j.g.c.
  3/10/89
  copied from K&R p.16.
-------------------------------------------------*/
#include <stdio.h>

int main(void)
{
  int c;

  c = getchar();
  while(c != EOF){
```

```
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

Dissection:

a) "!=" denotes *not equal to*.

b) int c instead of char c! $\Longrightarrow$ getchar() must be able to indicate errors, *e.g.*, when there is no more input — end-of-file (EOF); EOF is defined in stdio.h using # define; normally, it requires more bits than a plain character.

9. Another version, using a common C *cliche*:

```
/*-------------------------------------------------
  cio2.c - copy input to output, version 2.
           demo of C file reading cliche/idiom
  j.g.c.
  3/10/89
  copied from K&R p.17.
-------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
  int c;

  while((c=getchar()) != EOF)
    putchar(c);
  return 0;
}
```

Dissection:

a) while((c=getchar()) != EOF): This is quite a common idiom/cliche in C; it is made possible because assignment statements in C have a value; the value is the last value assigned. Experienced C programmers will all understand what it does. However, do NOT take this as general permission to write dense, cryptic code.

b) Note the importance of brackets.

10. Character Counting. Objective: count the characters in the input stream, stop at EOF.

```
/*-------------------------------------------------
 ctch1.c - counts input chars; version 1
 j.g.c.
 3/10/89.
 copied from K&R p.18.
-------------------------------------------------*/
#include <stdio.h>

int main(void)
{
  long nc;

  nc=0;
  while(getchar() != EOF)
    ++nc;
  printf("%ld\n", nc);
  return 0;

}
```

Dissection:

a) ++ operator: increment by one; −−: decrement by one. Note: As single statements, ++nc and nc++ are equivalent, but give different values in expressions, *e.g.*,

```
int i, n;
n = 6;
i = ++n; /* POST-increment gives i == 7, and
            n == 7; the increment is done BEFORE
            the assignment */
```

wheras,

```
n = 6;
i = n++; /* POST-increment gives i == 6, and
            n == 7 */
```

b) long (integer) is at least 32 bits long.

c) %ld tells printf to expect a long.

11. Line Counting.

```
/*-----------------------------------------------*/
#include <stdio.h>

/* count lines in input*/
int main(void)
{
  int c, nl;

  nl = 0;
  while((c=getchar()) != EOF)
    if(c == '\n')
      ++nl;
  printf("%d\n", nl);
  return 0;
}
/*-------------------------------------------------*/
```

Dissection:

a) if statement:

tests condition in (...)
if true executes statement or group { ... } following,
otherwise (false) skip them.

b) == means *is-equal to*. CAUTION: = in place of == can be syntactically correct and so not trapped by compiler — a nasty hard-to-find error results!

c) Character constants, *e.g.*, '\n' represents an integer value equal to the value of the character in the machine's character set; In ASCII: '\n' == 10 decimal; 'A' == 65 decimal.

12. Word Counting.

Word = any sequence of chars that does not contain a *white-space* char. *i.e.*, blank, tab, newline.

```
/*-------------------------------------------------*/
#include <stdio.h>

#define IN 1  /*inside a word*/
#define OUT 0 /*outside a word*/

/* count words, lines and chars in input*/
```

```
int main(void)
{
  int c, nl, nw, nc, state;

  state = OUT;
  nl = nw = nc = 0;
  while((c=getchar()) != EOF){
    ++nc;
    if(c == '\n')
      ++nl;
    if(c==' ' || c=='\n' || c=='\t')
      state = OUT;
    else if(state == OUT){
      state = IN;
      ++nw;
    }
  }
  printf("%d %d %d\n", nl, nw, nc);
  return 0;
}
/*-----------------------------------------------*/
```

Dissection:

a) nl = nw = nc = 0; recall: an assignment has a value, *i.e.*, nc = 0; has the value '0'.

b) || means OR, && means AND

c) Expressions with || and && are evaluated left to right — evaluation is stopped as soon as the result is known. *e.g.*,

```
int i=1, j=2, k=3;
if (i==1 || j==3 || k==3) { ... }
```

the evaluation of the condition can, and does, stop as soon as j==3 evaluates to FALSE (why?)

d) if(*expression*)
statement1
else
statement2.

As usual *statement* can be a group in brackets { ... }

13. Arrays. Below is a program to count the occurrences of each numeric digit, of white-spaces and of all other characters (together) — without using 12 named counters!

```
/*-------------------------------------------------*/
#include <stdio.h>

/*counts digits, white-spaces, others*/
int main(void)
{
  int c, i, nwhite, nother;
  int ndigit[10];

  nwhite = nother = 0;
  for(i=0; i<10; i++)
    ndigit[i] = 0;

  while((c=getchar()) != EOF)
    if(c>='0' && c<='9')
      ++ndigit[c-'0'];
    else if(c==' ' || c=='\n' || c=='\t')
      ++nwhite;
    else
      ++nother;

  printf("digits =");
  for(i=0; i<10; ++i)
    printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n",
          nwhite, nother);
  return 0;
}
```

Dissection:

a) int ndigit[10]; an array of 10 ints.

b) The for(...) loops must go 0, 1, ... 9, since, in C, array subscripts must start at 0.

c) The test if(c>='0' && ... : && denotes logical AND.

d) c–'0' assumes that '0','1'...'9' have consecutive, increasing values.

e) c–'0' is an integer expression $\Longrightarrow$ it's OK for a subscript.

f) if(*condition1*)
statement1
else if(*condition2*)
statement2
.....
else
statement

This is the model for a multiway decision; you can have any number of

else if (*cond*)
stmt

groups between first if and final else; the final else catches anything that didn't satisfy any of the previous conditions.

g) Indenting style: Again, please note that we don't want to run off the right-hand edge of the paper.

F. Functions.

1. We have already encountered functions that have been written for us, *e.g.*, printf, getchar, putchar.

2. Here is a program that reads ints and floats and does some arithmetic with them. We'll use it as a case-study of many of the aspects of functions.

```
/*-----------------------------------------
  math1.c
  j.g.c. 6/1/96
  demo of functions -- in same file
-----------------------------------------*/
#include <stdio.h>

int getInt(void)
{
  int i;
```

```
  scanf("%d", &i);
  return i;
}

float getFloat(void)
{
  float f;

  scanf("%f", &f);
  return f;
}

float addf(float a, float b)
{
  float c;

  c = a + b;
  return c;
}

int main()
{
  float x, y, z, w;
  int i, j, k;

  printf("enter first int:");
  i = getInt();
  printf("enter second int:");
  j = getInt();
  k = i + j;
  printf("%d + %d = %d\n", i, j, k);

  printf("enter first float:"); x = getFloat();
  printf("enter second float:"); y = getFloat();
  z = x + y;
  printf("%f + %f = %f\n", x, y, z);

  w = addf(x, y);
  printf("%f + %f = %f\n", x, y, w);

  printf("%f + %f = %f\n", x, y, addf(x, y));

  return 0;
}
```

Dissection:

a) A function definition consists of:

return-type function-name(parameter list if any)
{
  definitions of internal variables (locals);
  statements
}

b) The functions may be all in the same file, along with main, or may be spread across many files.

c) The variables a, b, c are local to addf; they are invisible elsewhere.

d) a, b, c are called **parameters**, or *formal parameters*, or, colloquially, *formals*.

e) In the call: addf(x, y), x, y are called **arguments**, or *actual parameters*, or colloquially *actuals*.

f) return <*expression*> returns a value to main and passes control back to main.

g) Unless the function returns a value (some don't) return is not essential, but if the function declaration indicates that the function returns a value, then there must be an appropriate return statement.

h) The calling function can ignore the returned value.

3. Here is another version with different layout; here we've kept main at the beginning, and functions at the end; consequently, we've had to declare functions getInt, etc. before they are called.

```
/*-----------------------------------------
  math2.c
  j.g.c. 6/1/96
  demo of functions -- in same file
  -- now with functions *after* main(), and with
  declarations.
-----------------------------------------*/
#include <stdio.h>
```

```
/*-- must declare functions before calls --*/

int getInt(void);
float getFloat(void);
float addf(float a, float b);

int main()
{
  float x, y, z, w;
  int i, j, k;

  printf("enter first int:");
  i = getInt();
  printf("enter second int:");
  j = getInt();
  k = i + j;
  printf("%d + %d = %d\n", i, j, k);

  printf("enter first float:"); x = getFloat();
  printf("enter second float:"); y = getFloat();
  z = x + y;
  printf("%f + %f = %f\n", x, y, z);

  w = addf(x, y);
  printf("%f + %f = %f\n", x, y, w);

  /*- function call can replace variable
       addf returns a float value  -*/
  printf("%f + %f = %f\n", x, y, addf(x,y));

  /*- you can 'mix' types -- if the function has been
      declared, and if types are compatible; here it
      is possible to 'coerce' 'i' to float; essentially
      the compiler inserts a 'cast' -- (float)i; usually
      it is best for the programmer to do this explicitly
   -*/
  printf("%d + %f = %f\n", i, y, addf(i,y));

  return 0;
}

int getInt(void)
{
  int i;

  scanf("%d", &i);
```

```
  return i;
}

float getFloat(void)
{
  float f;

  scanf("%f", &f);
  return f;
}

float addf(float a, float b)
{
  return a + b;
}
```

Dissection:

a) int getInt(void); declares the *type* of getInt(); in C parlance, it is called the *prototype* for getInt.

b) getInt has type: void –> int.

c) addf has type: float, float –> float.

d) Parameter names in prototypes are neither significant nor neccessary. But, can provide good documentation.

e) Prototypes and the way functions are declared are the biggest change between ANSI C and the earlier versions. Easier for compiler to check errors.

4. Normally, it is a good idea to split a program into **modules**, *e.g.*, the arithmetic functions we've written may be tested an stable, whilst the main program is subject to change; it's nonsensical to have to re-compile the functions each time, so we'll put them in a separate file funs.c.

5. Actually, there are bigger and better reasons for such *modularity*, but we'll come to them later. Thus, math3.c:

```
/*-------------------------------------------
  math3.c
  j.g.c. 7/1/96
  demo of functions -- in *separate* file / module
   funs.c
  and with declarations in
        funs.h
-------------------------------------------*/
#include <stdio.h>

#include "funs.h"

int main()
{
  float x, y, z, w;
  int i, j, k;

  printf("enter first int:");
  i = getInt();
  printf("enter second int:");
  j = getInt();
  k = i + j;
  printf("%d + %d = %d\n", i, j, k);

  printf("enter first float:"); x = getFloat();
  printf("enter second float:"); y = getFloat();
  z = x + y;
  printf("%f + %f = %f\n", x, y, z);

  w = addf(x,y);
  printf("%f + %f = %f\n", x, y, w);

  printf("%f + %f = %f\n", x, y, addf(x,y));


  return 0;
}
```

and, funs.c:

```
/*-----------------------------------------------
  funs.c
  j.g.c. 7/1/96
  demo of functions in separate file
  called by math3.c
-----------------------------------------------*/
#include "funs.h"
int getInt(void)
```

```
{
  int i;

  scanf("%d", &i);
  return i;
}
float getFloat(void)
{
  float f;

  scanf("%f", &f);
  return f;
}
float addf(float a, float b)
{
  return a + b;
}
```

and, we need funs.h:

```
/*-----------------------------------------------
  funs.h
  j.g.c. 7/1/96
  demo of functions in separate file
  called by math3.c
-----------------------------------------------*/
#include <stdio.h>

int getInt(void);
float getFloat(void);
float addf(float a, float b);
```

Dissection:

a) When using a library or external *module*, you must get into the habit of producing a *header* (*.h) file that contains declarations of the functions; and, declarations of other *things* required by the functions, *e.g.*, symbolic constants.

b) If you do not include declarations, the compiler reverts to pre-ANSI habits: it assumes, implicitly, that all functions return an int, and that actual and formal arguments agree;

in this pre-ANSI mode, the line in math2.c above,

printf("%d + %f = %f \n", i, y, addf(i,y));

would fail miserably. Fortunately, gcc (GNU C compiler) seems to issue warnings in such cases; other compilers may not be so helpful; gcc -Wall is better. (Note that on all Unixes except Linux, the standard C compiler executable is ran with the 'cc' command. On Linux, the standard is the GNU C compiler which is ran with the 'gcc' command.)

c) The prototypes of standard library functions are contained in header files, *e.g.*, stdio.h is just an ordinary text file containing prototypes of printf, etc.; (in addition, stdio.h contains #defines).

d) '.h' files are <u>not</u> compiled — they get compiled as part of any file in which they are included (recall, CPP – the C PreProcessor executes all '#' commands before the compiler proper.

e) When you have separate *modules* (or *compilation units*), obviously, the compilation & linking procedure must be modified. The separate units can be compiled to object, as before:

```
gcc -c math3.c          # yields math3.o
gcc -c funs.c           # –> funs.o
```

f) Now link

```
gcc -o math math3.o funs.o          # yields executable math
```

g) Actually, you can do all this in one line

```
gcc -o math math3.c funs.c
```

6. Raise an integer m to a positive integer power n: power(m, n):

```
/*-----------------------------------------------------
  tstpow  - tester for power
```

```
  j.g.c.
  3/10/89
  copied from K&R p. 24,25.
-------------------------------------------------------*/
#include <stdio.h>
int power(int m, int n);
int main(void)
{
  int i;
  for(i=0; i<10; ++i){
    printf("%d %d %d\n", i, power(2,i), power(-3,i));
  }
  return 0;
}
/*-------------------------------------------------------
  power - raise base to n-th power; n not negative.
  j.g.c.  3/10/89
  copied from K&R p.25
-------------------------------------------------------*/
int power(int base, int n)
{
  int i, p;
  p = 1;
  for(i=1 ;i<=n; ++i)
    p = p * base;
  return p;
}
```

G. Parameters — Pass by Value.

1. *Call by value* means that the arguments are copied into temporary storage for the function; essentially, the formal parameters are local variables that get created, then initialised to the value of the arguments; thus, above, in the call power(2, i), first a local int called *base* is created, and initialised to value '2'; likewise, a local 'n' gets created and initialised to the value that 'i' has.

2. Contrast: Subroutine/function parameters in Fortran, where the subroutine/function has access to the original argument variable — *pass by reference.*

3. We have no explicit *call by reference* in C (must be done using pointers). But, conversely, in C, arrays are passed by reference, and you cannot pass them by value!

4. Another version of power to demonstrate call by value:

```
/*-----------------------------------------------------
  power - raise base to n-th power, n non-neg., ver.2
          to show use of call by value
-----------------------------------------------------*/
int power(int base, int n)
{
  int p;
  for (p=1; n>0; --n)
    p = p * base;
  return p;
}
```

Dissection: Note that parameter 'n' can be modified, without affecting the argument; 'n' is local, and, like all local variables, temporary.

H. Character Arrays.

1. To read a set of lines, find and print the longest. Pseudo-code:

```
while( there is another line )
  if( it is longer than he current longest )
  save it
  save its length
print the longest line.
```

2. Solution: Break into functions — following the pseudo-code specification: (1) *getline*: fetch the next line, work out its length, (2) *copy*: use for saving a line — cannot assign strings in C.

```
/*-----------------------------------------------------
  lil.c - find and print longest line
  j.g.c.
  4/10/89.
  Copied from K&R p29.
```

```
-------------------------------------------------------*/
#include <stdio.h>
#define MAXLINE 1000  /*size of buffer for line*/

int getline(char line[],int maxline);
void copy(char to[],char from[]);

int main(void)
{
  int len; /*current line length*/
  int max; /*max. length so far*/
  char line[MAXLINE]; /*current input line*/
  char longest[MAXLINE]; /*buffer for longest line*/

  max=0;
  while((len=getline(line,MAXLINE))>0)
    if(len>max){
      max=len;
      copy(longest,line);
    }
  if(max>0) /*there was at least one line*/
    printf("%s,longest);
  return 0;
}

/*-----------------------------------------------------
  getline - read a line into s, RETURN length
-------------------------------------------------------*/
int getline(char s[],int lim)
{
  int c,i;

  for(i=0;(i<lim-1)&&((c=getchar())!=EOF)&&(c!='\n');++i)
    s[i]=c;
  if(c=='\n'){
    s[i]=c;
    ++i;
  }
  s[i]='\0';
  return i;
}
/*-----------------------------------------------------
  copy - copy "from" into "to".
-------------------------------------------------------*/
void copy(char to[],char from[])
{
  int i;
```

```
  i=0;
  while((to[i]=from[i])!='\0')
    ++i;
}
```

Dissection:

a) It is not neccessary to declare the length of 's' in getline; storage has already been allocated in main.

b) void copy(..) states explicitly that copy does not return a value — allows compiler to check inconsistent usage.

c) Null terminated strings; convention in C;

"hello\n" stored as —
h e l l o \n \0 — 7 chars.

d) NB. the issue of buffer overflow is ignored, like many of these programs, we have not provided full 'bullet-proofing',

I. External Variables and Scope.

1. Local variables, *e.g.*, 'line[]', 'longest[]' in 'main()' are private or LOCAL to main.

2. The 'i' in copy is unrelated to 'i' in getline; *i.e.*, it has LOCAL SCOPE.

3. Such variables are also called AUTOMATIC — they only come into existence when the function is called, they disappear when the function is exited.

4. Contrast STATIC — still local, but values are retained from call to call. (Side-issue: STATIC, when applied to external variables or functions, also has another effect (see K&R Chap. 4.6) namely,

that their names are invisible outside the file in which they are declared.

5. Automatic assumed unless:

```
static int c; /* c declared static */
```

6. Declaration:

```
auto int c; /* equivalent to int c; */
```

$\Longrightarrow$ hardly ever used — since defaults to auto.

7. External Variables:

a) Similar to variables in a Fortran COMMON block.

b) They are globally accessible. [GLOBAL SCOPE]

c) Remain in existence permanently. Their LIFETIME (or DURATION or SPAN) is for the duration of the execution of the program. $\Longrightarrow$ *Scope* and *lifetime* are important issues in programming languages.

d) Thus, externals/globals can be used to communicate data between functions; but, there are all sorts of reasons why you should never use them — mostly to do with reduction of coupling between software units.

e) Rules of use of externals.

i) An external variable must be DEFINED, exactly once, OUTSIDE any function.

ii) It must be DECLARED in each function that accesses it; and the declaration must announce that it is external:

extern int max;

Example E–1. lil.c rewritten to use external variables.

```
/*-------------------------------------------------
  lil1.c - find and print longest line, version 2
           to demo extern variables.
  j.g.c.
  4/10/89.
  Copied from K&R p32.
-------------------------------------------------*/
#include <stdio.h>
#define MAXLINE 1000  /*size of buffer for line*/

/* external definitions follow N.B. outside any function*/

int max; /*max. length so far*/
char line[MAXLINE]; /*current input line*/
char longest[MAXLINE]; /*buffer for longest line*/

int getline(void);
void copy(void);

int main(void)
{
  int len; /*current line length*/
  extern int max;
  extern char longest[];

  max=0;
  while((len=getline())>0)
    if(len>max){
      max=len;
      copy();
    }
  if(max>0) /*there was at least one line*/
    printf("%s,longest);
  return 0;
}
/*-------------------------------------------------
  getline - read a line into external line, RETURN length
-------------------------------------------------*/
int getline(void)
{
  int c,i;
  extern char line[];
```

```
  for(i=0;(i<MAXLINE-1)
      &&((c=getchar())!=EOF)&&(c!='\n');++i)
    line[i]=c;
  if(c=='\n'){
    line[i]=c;
    ++i;
  }
  line[i]='\0';
  return i;
}
/*----------------------------------------------------
  copy - copy extern line into extern longest.
----------------------------------------------------*/
void copy(void)
{
  int i;
  extern char line[],longest[];

  i=0;
  while((longest[i]=line[i])!='\0')
    ++i;
}
```

Dissection:

i) Externals max, line[], longest[] are DEFINED outside any function.

ii) Externals are DECLARED before they are used in a function.

iii) Exception to 2: if definition appears in the source file before use, then extern need not be stated explicitly; but it is much better for documentation to state it explicitly.

iv) As they are typed, 'getline' and 'copy' could be placed in separate files from 'main'; the LINKER would connect the *external* functions.

v) The *external* data could likewise be defined in a separate file, which could be compiled and linked

(even though it only contains data).

vi) Empty argument list must use explicit 'void'. If you defined the function 'copy' as

```
copy(){
  ....
}
```

the compiler would assume pre-ANSI C; the major difference with pre-ANSI was much more relaxed argument/parameter consistency checking. In fact there was no consistency checking $\Longrightarrow$ the *called* function just assumed that the *caller* had passed the correct argument types; so, you had all sorts of nasties, like integers being interpreted as floating-point!

vii) External variables are against most of the principles of software engineering: modularity, information hiding, lack of coupling.

---

f) I recommend that you adopt the following standard: all uses of globals / extern must be accompanied with a justification, contained in a comment, /*..*/, placed nearby; it is surprising how, with a little thought they can be completely avoided; and, you really appreciate it when it comes to testing.

J. Definition, Declaration.

1. We will adopt the same convention as K&R for the use of these terms.

2. DEFINITION: the variable is created (or function body given)

— including allocation of storage.

3. DECLARATION: The properties of the variable (or function) are announced to the program — **but no storage allocated**.

K. Creation of Executables.

1. Basics: Compiling and Linking.

a) Source programs like hello.c, math3.c etc. are not directly executable. There are a number of stages in creation of the executable, and executing it.

b) The first stage of creating an executable is to *compile* hello.c into *object* code.

```
gcc –c hello.c
```

Doing this, and puts the hello object code into a file 'hello.o'. This object code is essentially *machine* code, with code to initialize main, a call to printf, and code to return 0. But the code for 'printf' is not present.

c) The second stage is to *link* the machine code in hello.o with appropriate object code for printf:

```
gcc –o hello hello.o
```

This produces the executable file hello (no extension).

d) Even though we use gcc (or cc), gcc actually invokes a command called ld to do the linking-loading (*loading* refers to loading of external code).

e) Here, the extraction of the code for printf from a **library** is kept implicit; however, a library is nothing more than an object file, with appropriate indexes to each function.

f) Finally, to execute hello, you type

```
hello
```

at the Unix prompt. This reads the contents of hello into memory, and starts execution at an appropriate start address.

g) The situation may be made more clear-cut if we look at the two module program math3.c, funs.c, introduced above. Recall,

```
gcc –c math3.c                  # yields math3.o
gcc –c funs.c                   # → funs.o

gcc –o math math3.o funs.o      # yields executable math
```

However, you can do all this in one line

```
gcc –o math math3.c funs.c
```

This process is diagrammed in Figure B-1.

2. Other Libaries.

a) Not all *system* functions are in the default libraries that are searched by gcc/ld. For example, math functions like sqrt — take math4.c, if you try to link using:

```
gcc –ansi –pedantic –Wall –Wstrict–prototypes –o math \
   math4.c funs.c
```

The linker will complain that the call to sqrt is *unresolved*. To correct this, you must explicitly mention the (m)aths library; thus,

```
gcc –ansi –pedantic –Wall –Wstrict–prototypes –o math \
   math4.c funs.c –lm
```

```
/*----------------------------------------
  math4.c
  j.g.c. 7/1/96
  extension of  math3.c, but showing use of 'math library'
```

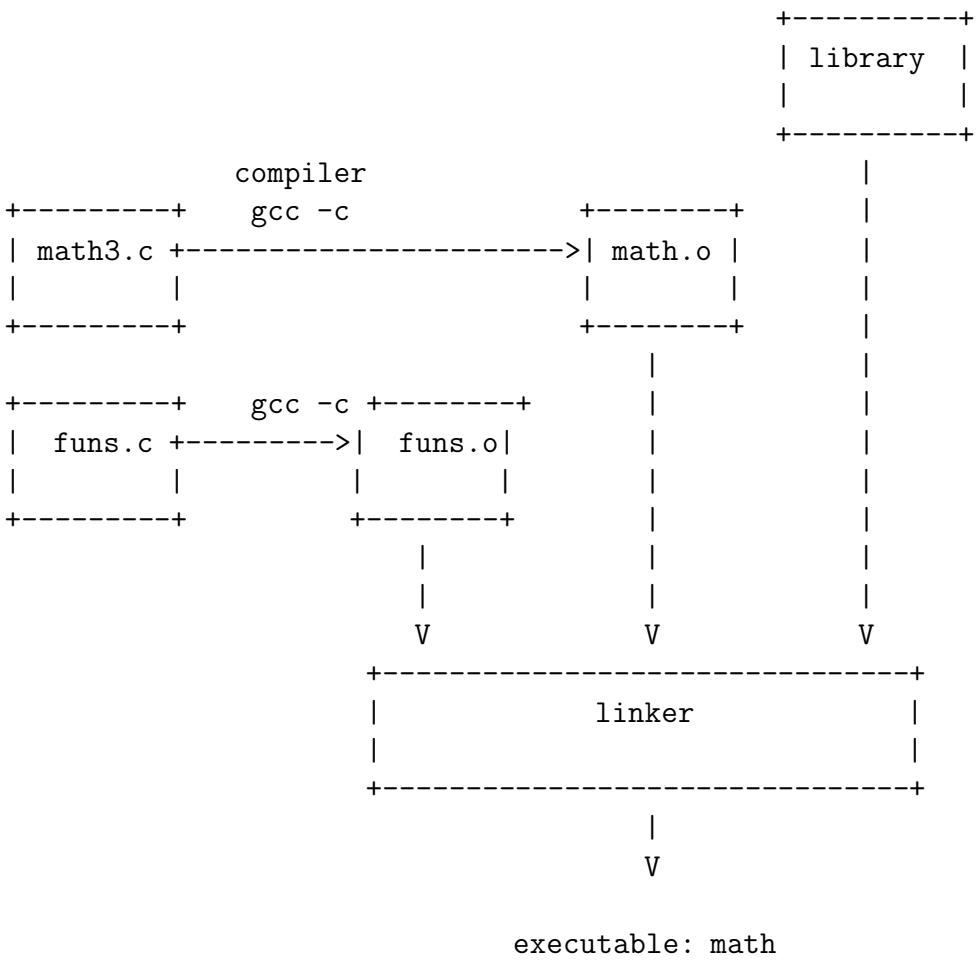

Figure E–1: Compiling and Linking

```
  function -- 'sqrt'
  i.e. need:  (1) #include <math.h>
   (2) put -lm in linker command
-----------------------------------------*/
#include <stdio.h>
#include <math.h>

#include "funs.h"

int main()
{
  float x, y, z, w, v;
  int i, j, k;

  printf("enter first float:"); x = getFloat();
  printf("enter second float:"); y = getFloat();
  w = addf(x, y);
  printf("%f + %f = %f\n", x, y, w);

  /* make sure we have a non-negative number! */
  w = fabs(w);
  v = sqrt(w);
  printf("square-root of %f is %f\n", w, v);

  return 0;
}
```

b) Also, libraries exist to carry out a variety of X-Window functions.

3. Static vs. Shared Libraries. The code for addf, getInt, and getFloat is linked *statically*, *i.e.*, the object code for each of the functions is copied into the file math. For common functions like printf, this can become wasteful; hence **shared** libraries, in which the linker inserts just a *pointer* to shared code that held in the operating system, or maybe somewhere on disk.

4. Make.

a) When you get to more complicated systems of multi-module programs, make can become useful.

b) Here is a *makefile* for math4.c:

```
# makefile for math4
# j.g.c. 7/1/96
#

# here we define a 'macro' CC as gcc, the GNU
# C-compiler

CC=gcc -ansi -pedantic -Wall -Wstrict-prototypes

# how to create 'math' (the executable):
# (1) math4.o, and funs.o  must be up to date

math:   math4.o funs.o

#
# (2) create prog by linking math4.o, funs.o
#  note: *must* use TAB here, spaces no good

(CC) -o math math4.o funs.o -lm

# specify what funs.o depends on
# don't forget the TAB for the cc command!

funs.o: funs.c funs.h
(CC) -c funs.c

# dependencies for math4.o
math4.o: math4.c
(CC) -c math4.c

# clean-up directory of .o(bject) files

# invoke as: make clean
# i.e. a separate shell command, typically
# after you're finished developing \& executable 'math' is OK

# note that the 'rm' command is on the line
# *after* the 'clean' target, and, as usual, TABbed
clean:
rm -f funs.o math4.o
```

c) The standard practice is to name this file Makefile, then invoking

```
make
```

causes math to be *made*: all programs compiled and linked. Make takes are of dependencies, *e.g.*, if math4.o and funs.o already exist, make will not recompile them. On the other hand, if they exist, but if, say, math4.c has changed since the last compilation to math4.o, then make will recompile math4.c (but not func.c).

d) In the example above,

```
make clean
```

will delete all the '.o' files.

e) If you want to call the *makefile* something else, *i.e.*, math4.m, maybe you want to keep many make files in one directory — though advanced make users have other solutions to that problem — then you can invoke make.m as

```
make –f make.m
```

f) Make is very much part of UNIX, and is used for very much more than just C programs.

g) Another point, because of its fiddly syntax, nobody ever creates a makefile from scratch; find a *template* makefile (one that works!) that is close to what you require and change bits to suit. Be careful with the places where the TAB syntax matters.

5. Warnings for gcc and cc.

a) As I have said earlier, gcc and cc, without warnings switched on, is positively dangerous.

b) I have suggested the following for either gcc or cc:

gcc –ansi –pedantic –Wall –Wstrict–prototypes

The following list explains the various switches on GNU cc (*e.g.*, gcc):

```
# -pedantic:  issue warnings for non-ANSI standard C
# -pedantic-errors:  issue errors for non-ANSI standard C
# -Wall:  enable a host of warning messages
# -Wpointer-arith:  warn of dependency on "size_of" function or void
# -Wcast-qual:  warn of pointer cast that removes type qualifier
# -Wcast-align:  warn of pointer cast that increases alignment
# -Wwrite-strings:  make string constants type "const char[]" and warn
about non-const char
* mix
# -Wconversion:  warn when prototype causes different type conversion
# -Waggregate-return:  warn of functions returning structures
# -Wstrict-prototypes:  warn if argument types missing
# -Wmissing-prototypes:  warn if global function has no prototype
# -Wredundant-decls:  warn of multiple declarations in same scope
# -Wnested-externs:  warn of externs within functions
# -Winline:  warn if function cannot be inlined
# -Werror:  make all warnings errors
# -ansi:  ANSI standard C
```

c) It is recommended that the following be used with gcc:

–ansi
–pedantic
–Wall
–Wpointer-arith
–Wcast-qual
–Wwrite-strings
–Wconversion
–Wno-strict-prototypes
–Wmissing-prototypes
–Wnested-externs
–Winline
–Wno-error

d) Jamie Blustein <jamie@csd.uwo.ca> has notes about using gcc at http://www.csd.uwo.ca/∼jamie/.Refs/.Footnotes/gcc.html

L. Interactions with the Operating System.

1. Since Unix and C are closely related (indeed, Unix is written in C), C was built to allow easy access to the operating system.

2. The function int system (const char *s) (which resides in <stdlib.h>) allows a C program to pass a string to the operating system to be processed. For instance,

```
system ("date");
```

causes the Unix program date to be run; it prints the date and time of day on the standard output. system returns a system-dependent integer status from the command executed. In the Unix system, the status returned is the value returned by exit.

3. The function void exit (int status) (also in <stdlib.h>) causes normal program termination. Open files are flushed, open streams are closed, and control is returned to the environment. How status is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values EXIT_SUCCESS and EXIT_FAILURE may also be used. Note that if one used exit to terminate a program, one **would not** use return in these cases.

4. The function *getenv (const char *name) (in <stdlib.h>) returns the environment string associated with name, or NULL if no string exists.

5. There are various *time* functions defined in <time.h> to return time information from the operating system — see Appendix B10 in Kernigham & Ritchie.

6. The function int remove(const char *filename) (<stdio.h>) removes the named file. It returns non-zero if the attempt fails. This is the function called by the Unix command rm.

7. The function int rename(const char *oldname const char *newname) (<stdio.h>) changes the name of a file. It returns non-zero if the attempt fails. This is the function called by the Unix command mv.

8. The following commands can give useful information about the processes that are running on the machine. They are all in <stdio.h>.

a) int getpid() and int getppid() returns the process's ID and the parent's process ID.

b) int chdir(char *pathname) sets the process's *current working directory* to the directory stored in pathname. The process must have execute permission from the directory to succeed. chdir() returns 0 if successful, –1 if not.

c) int nice(int delta) adds delta to a process's current priority value. Only a super-user may specify a delta that leads to a negative priority value. Legal priority values lie between –20 and +19, the lower the value, the larger amount of time the CPU spends on the process. If a delta is specified that takes a priority value beyond a user's limit, the priority value is truncated to the limit. If nice() succeeds, it returns a new nice value; otherwise, it returns –1. Note that this can cause problems, since a nice value of –1 is legal.

d) int getuid() and int getgid() returns the *user*'s and the user's *group* ID.