

Notes on C Programming

L. R. Schaeffer

May 2000

Introduction

These notes are the basics of the C language that are needed for this course. For more details and examples see the recommended texts. There are also web sites of C tutorial notes that may be of help.

The Main Program

The C language was developed to promote the writing of well-structured programs. C programmers like to be concise and elegant in their programs which sometimes makes C programs difficult to understand. A C program consists of a main program and often functions, which are subroutines or subunits of the main program. The idea is to keep the main program as short, concise and easy to follow as possible by using many functions.

```
main( int argc, char *argv[] )
{
    statements;
}
```

All C program statements end in a semicolon, ;. If a semicolon is missing, the C compiler can generate many pages of error messages. Statements of the program are grouped between braces, { }. The main program may have arguments that are passed to it, such as a random number seed, file names, or program options. `argc` will indicate the number of arguments on the command line, and `argv[]` will point to the first character of each argument. Thus, the first part of a C program is the processing of the arguments, and this could be the job of a C function or could be part of the main program. If no arguments are to be passed, then one writes `main()`.

Include Statements

Programs usually require functions that are used repeatedly, and these are stored in libraries. They are added to a program through `#include` statements prior to the main program.

`#include<stdio.h>` contains functions for standard input and output, definitions of EOF and NULL, and how to handle errors.

`#include<string.h>` contains functions for manipulation of strings of characters, and converting characters to integer or floating point numbers.

`#include<math.h>` contains functions for mathematical operations.

`#include "myfuncs.h"` contains functions written by the programmer, and other information.

The ".h" signifies a header file. Header files contain information needed by the program. For example, a simulation program may need parameters like heritability, variances, number of generations, number of animals, or number of Gibbs samples, and these parameters could change each time the program is run. By putting the parameters in a header file, only the header file needs to be changed and not the program code. These parameters are known as global or external variables, and can be used by any function in the program.

Header files also contain *prototypes*, which are like examples of the functions that are used in the program. The example tells the C compiler to look and plan for a function that will be of a particular type and a function that will expect certain types of arguments. Header files contain structure definitions, which are typically the data structures of input and output files. The programmer may also define new variable types.

Variable Types

All variables in a C program must be declared before they are used. Declaration causes storage space to be reserved. A declaration gives the variable type and the identifier. Below are some examples:

```
int          lower, upper, i, k, ndigit[10];
char        label, line[256], sc='%';
float       average=0.0;
double      variance, *x;
struct anode animal;
```

`int` = integers or non-decimal numbers in the range of -2147483647 to +2147483647.
`ndigit[10]` is an array of ten integer elements.

`char` = characters. `label` will hold a single character while `line` is a character array that will hold up to 256 characters. The variable `sc` has been initialized to be a % sign. Initialization is done only once before the program starts executing, unless it is declared as

```
auto char sc='%';
```

in which case `sc` would be re-initialized to % each time that function is entered. Declaration as

```
const char sc='%';
```

means that `sc` would not be allowed to change in value.

`float` = single precision, floating point numbers.

`double` = double precision, floating point numbers. Here `*x` refers to a pointer to a double precision number. A pointer has to have the same variable type as the object to which it points.

`struct anode` = a user defined structure. `animal` refers to a variable that has the structure defined by `anode`.

The identifiers, or names of variables, are sequences of letters, digits, and underscores, but must not begin with a digit. Up to 31 characters can be used in an identifier. C distinguishes between upper and lower case letters. Therefore,

```
int    mu, MU, Mu, mU;
```

would declare four different variables as integer type. There are reserved words which should not be used as identifiers. Some of these words are as follows:

<code>asm</code>	<code>default</code>	<code>float</code>	<code>register</code>	<code>switch</code>
<code>auto</code>	<code>do</code>	<code>for</code>	<code>return</code>	<code>typedef</code>
<code>break</code>	<code>double</code>	<code>fortran</code>	<code>short</code>	<code>union</code>
<code>case</code>	<code>else</code>	<code>goto</code>	<code>signed</code>	<code>void</code>
<code>char</code>	<code>entry</code>	<code>if</code>	<code>sizeof</code>	<code>volatile</code>
<code>const</code>	<code>enum</code>	<code>int</code>	<code>static</code>	<code>while</code>
<code>continue</code>	<code>extern</code>	<code>long</code>	<code>struct</code>	

Variables may also be identified through `#define` statements, which occur outside of the main program, usually in a user defined header file. For example,

```
\#define MaxLine 1024
```

Now `MaxLine` can be used anywhere in the program. The compiler replaces `MaxLine` with the number 1024 wherever it occurs in the program.

Variables have *scope*. Some variables are used only within a particular function, while other variables need to be available to several or all functions. In general, if the variables are declared inside a function, that is within the braces of a function, then the variable is local and its use is limited to within that function. If a variable is declared outside of a function and the main program, then the variable is global and can be used by all functions. The identifier should not be duplicated in any of the functions. For example,

```
int    Number_of_animals=0;

main() { ... }
```

then none of the functions should have that identifier declared within the function.

Structures

A structure is a variable type, but it consists of one or more variables that are grouped together under a single name. For example, a pedigree file contains identifications of the animal, it's sire and it's dam. Each entry in this file is a record, and a record can be conveniently handled as a structure in C. Define the structure pedigree as

```
struct pedigree {
    int aid;    /* animal's ID */
    int sire;   /* sire's ID  */
    int dam;    /* dam's ID   */
};
```

Structures are usually defined in a header file prior to the main program. Within the main program, then variables can be declared to be of type `struct pedigree`.

```
struct pedigree anim;
```

To reference the sire in the program, one enters `anim.sire`.

Structures can contain other structures. Now add the animal's birth date, breed code, birth weight, blood type, and sex code.

```
struct details {
    int year;
    int month;
    int day;
    int birth_weight;
    char sex_code;
    char breed_code;
    char blood_type;
}
struct pedigree {
    int aid;
    int sire;
    int dam;
    struct details da;
}
```

In the main program, a reference to the breed code would be made as `anim.da.breed_code`.

Given that there are usually thousands of animals in a pedigree file, then we could have an array of structures.

```
struct pedigree anim[Nanimals];
```

A reference to the breed code of animal `i` would be accomplished by `anim[i].da.breed_code`.

Operations on Variables

Comments can and should be added to programs to improve readability and to explain what the program is doing. Any characters that appear between `/*` and `*/` are ignored by the C compiler. Whitespace is a term to indicate blanks inserted between variables or lines of code. C code can be made obscure by removing blanks between identifiers and operators. However, adding whitespace and using indentations can help make a program less obscure. This could help in debugging a program or making changes to a program a year or more after the original code was written.

Assignments

```
x = 5;
n = x + 2;
y = n * (x - 3);

y *= 10;    /* equivalent to y = y * 10; */
x += 5;     /* equivalent to x = x + 5;   */
n++;       /* equivalent to n = n + 1;   */
++n;       /* equivalent to n = n + 1;   */
x--;       /* equivalent to x = x - 1;   */
g = p / 3;
y -= 10;
```

Conditional Expressions

Suppose the minimum of `a` or `b` are to be found. The following alternatives exist to accomplish that task.

```
if( a < b )
{
    z = a;    /* Note the whitespaces here */
}
else
{
    z = b;    }
}
```

or

```
z = ( a < b ) ? a : b;
```

or

```
z = b;  
if( a < b ) z = a;
```

The second alternative is concise and takes only one line of code. FORTRAN programmers may prefer the third alternative. The first alternative is required if other computations are needed besides assigning a value to z.

A multi-way decision process could follow another conditional expression with the following syntax:

```
if( a < b )          /* Could also be shortened as follows */  
  {                 /*   if( a < b )                               */  
    z = b;          /*     z = b;                               */  
  }                 /*   else if( a < c )                               */  
else if( a < c )     /*     z = c;                               */  
  {                 /*   else if( a < d )                               */  
    z = c;          /*     z = d;                               */  
  }                 /*   else z = a;                               */  
else if( a < d )  
  {  
    z = d;      }  
else  
  {  
    z = a;      }
```

If any of the expressions are true, then the statements associated with it are executed and the entire chain is terminated. Thus, if a is less than b, then z = b; and the chain is terminated, even if a is less than c or d. Ordering and forming the correct conditional expressions could be critical to the outcome of the program. If possible, the if expressions should be arranged in descending order of probability of truth. This will reduce the amount of time taken to check the conditions.

The else part handles the default case if none of the other expressions are true. In the above example, if a had to be less than b, c, or d, then the else part would not be needed, but to be safe, the else part should be present. Instead of assigning z = a; in

the `else` part, an error message could be printed and the program could be terminated because `a` being greater than `b`, `c`, and `d` was not expected.

The `switch` statement is another conditional, multi-way decision expression. Suppose `char choice` has been read in as input, and depending on its value different statements must be executed. An example follows:

```
switch (choice)
{
  case 'A' : nA++; break;
  case 'B' : nB++; nA++; break;
  case 'C' : nC++; nB++; nA++; break;
  default : nwrong++; break;
}
```

The `break;` statements cause an immediate exit from the `switch` statement, otherwise execution continues to the next `case`, and this may not be desired in most situations. Again, if `choice` is only expected to be `'A'`, `'B'`, or `'C'`, then the `default:` case may not be needed, but to be safe, `default:` should always be given. A `switch` statement is useful in processing the arguments from the command line of a program. For example, to compile a C program, the command line might be

```
cc -Ae -O myprogram.c -o gibbs
```

where `cc` is the command to invoke the C compiler, `-Ae` is an option to utilize particular libraries for the HPUNIX operating system, `-O` is to optimize the program execution, `myprogram.c` is the file containing the C program, and `-o gibbs` assigns the name `gibbs` to the executable file for running the program. Thus, `A`, `O`, and `o` would be cases in the `switch` statement.

Loops

Loops are very important in programming. In genetic evaluation, for example, processing needs to proceed from 1 to `TotAnim`, the total number of animals, many times during the iterative solving of mixed model equations. There are three kinds of loops in C. The two most frequently used are the `while` loop and the `for` loop.

A `while` loop is a block of statements that is re-executed until an expression becomes 0 or `FALSE`. A `while` loop is often used while reading an input file of data. Data are processed `while` the file still has data in it. Once the end of file (`EOF`) condition is reached, then the `while` loop is terminated. The number of records in a file is often different from one file to another and the `while` loop allows a degree of generality. An example of the syntax is

```

while ( a == b )
{
    v += obs;
    tss += (obs * obs);
    n++;
    b = get_next(obs);
    if( b == 0 ) break;
}

```

where `get_next(obs)` would be a function that retrieves the next observation and `b`-value, which could be a sire identification number. When all the data for a sire are finished, a new sire identification will cause `b` to be different from `a`. Outside of the loop, the mean and variance can be calculated and `a` can be set to `b` and the variables `v`, `tss`, and `n` would also be reset to zero before the loop is executed again. If `b` is equal to 0, then `break`; causes the loop to terminate, and this is because there are no further sires or observations.

A `for` loop is analogous to a `DO` loop in FORTRAN. Usually there are start and stop limits for the loop, such as from 1 to `TotAnim`. An example of the syntax is

```

for( i=0; i<TotAnim; i++)
{
    v += obs[i];
    tss += (obs[i] * obs[i]);
}

```

Notice that `i=0;` is the initial value of `i`. The loop is executed if `i<TotAnim:` is true. After the statements are executed, then `i++;` is performed, and then the loop is executed again as long as `i<TotAnim;` is true. When `i` equals `TotAnim` the loop is terminated, and the statements following the loop are executed. Thus, the range of acceptable subscripts for the array `obs` go from 0 to `TotAnim - 1`. This is a difference from FORTRAN which does not allow 0 to be an acceptable subscript. This little difference can be confusing at times to FORTRAN programmers that are switching to C.

A `for` loop could be rewritten as a `while` loop. Using the previous example,

```

i = 0;
while( i<TotAnim )
{
    v += obs[i];
    tss += (obs[i] * obs[i]);
    i++;
}

```

The third type of loop is a `do-while` loop, but this is used very infrequently. The above example would be coded as

```
i = 0;
do {
    v += obs[i];
    tss += (obs[i] * obs[i]);
    i++;
} while ( i < TotAnim );
```

The `continue`; statement inside a loop causes control to be passed to the expression within the `while()` statement or to the `i++`; element in the `for` loop, and the next loop through would be processed if the condition is true. On the other hand, the `break`; statement causes the program to leave the loop entirely.

Pointers and Arrays

A pointer contains the address (i.e. storage location in memory) of a variable. If the variable happens to be an array, then incrementing the pointer is another way of going through an array without using subscripts. The following examples will illustrate the use of pointers.

```
int *ip          /* ip is a pointer to an int variable */
int a[TotAnim], x, y, j;

ip = &a[0];      /* ip holds the address of a[0]          */
ip++;           /* ip holds the address of a[1]          */
x = *ip;        /* x = a[1]; equivalent                                  */
j = 200;
ip--;           /* ip points to a[0] again                             */
x = *(ip+j)     /* x = a[200]; equivalent                               */
y = (*ip)++;    /* y = a[0]++; equivalent                               */
```

From these examples it seems like little is gained by using a pointer over subscripts except confusion. However, the arguments of function calls are passed by values. That is, a copy of the contents of a variable are passed. In `f(x)` the number stored in `x` is passed to the function `f`. The function can do whatever it likes to that number, but the contents of `x` in the calling program are not changed. If the function requires an entire array of numbers, then obviously each one can not be put in separately within the function call statement, `f(a[0], a[1], a[2], ...)`. Instead the address of `a[0]` is sufficient. Suppose we have an array, `cov[Nitems]`, and we want to determine the mean and subtract the mean from all elements in the array. Let the function be called `zerom`. The function will

require the address of the first element in the array and the number of elements in the array. The main program would be as follows:

```
void zerom( double *, int * );

main( )
{ double cov[MaxSize], *pcov;
  int Nitems=0, *pN;
/* Statements to form cov[] and Nitems are not shown */
  pcov = &cov[0];
  pN = &Nitems;
  zerom( pcov, pN );
/* Statements after are not shown */
}
```

The first line above is the prototype that tells the compiler about the function that will appear somewhere. The actual function statements will follow the main program or will be in a separate file. The function is as follows:

```
void zerom( double *pv, int *ip )
{
  double sum, ave;
  int i;
  sum = 0.0;
  for( i=0; i<(*ip); i++ )
  {
    sum += *(pv+i);
  }
  ave = sum / (*ip) ;
  for( i=0; i<(*ip); i++ )
  {
    *(pv+i) -= ave;
  }
  return;
}
```

Note that it was not necessary to pass a pointer for `Nitems`. Use of pointers is faster than the use of subscripts (so it says in the book).

Structures

There can be pointers to structures, which allows entire structures to be used within functions. Using the simple pedigree structure from earlier, that is,

```

struct pedigree {
    int aid;    /* animal's ID */
    int sire;  /* sire's ID  */
    int dam;   /* dam's ID   */
};

```

then within the main program the declaration might be

```

struct pedigree anim[Nanimals], *p;

```

where `*p` is a pointer to a variable of type `struct pedigree`, and `anim` is an array of structures. To start, let `p = &anim[0]`; then `p+1` would point to the location of `anim[1]`. The storage address is incremented by the size of the structure for one animal, which, in this case, is the space occupied by three `int` variables. The contents of a location are accessed by

```

la = p->aid; /* equivalent to la = anim[0].aid */
ls = p->sire;
ld = p->dam;
if( p->sire == 0 ) p->sire = new_ID;

```

Input and Output

Input and output (I/O) in C is based on the concept of a stream of characters coming from a file or the keyboard.

PC I/O

The output function to the monitor is `printf`. Examples are as follows. Let `pa` be a pointer to a structure, like `pedigree`, then

```

printf(" %d %d %d %10.6f %-16s\n", pa->aid, pa->sire,
    pa->dam, coef, name);

```

where `%d` indicates an `int` variable is to be printed, `%10.6f` is the format for `coef` which might be an inbreeding coefficient, so that `coef` will be printed within a field of 10 spaces with six decimal places, and `%-16s` is the format for `name` within a field of 16 spaces and the `-` indicates left justified within the field. There must be matching formats for each variable to be printed. Note the `\n` at the end of the formats. This is the new line indicator, such that the next time this `printf` is used the output will begin on a new line. Without the new line indicator, the next `printf` will continue writing where the last `printf` finished.

The input function from the keyboard is `scanf`. For input, the user needs to provide the addresses of where the input items are to be stored.

```
int day, year;
char monthname[20];

scanf(" %d %s %d", &day, monthname, &year);
```

`monthname` is a character array and its name acts as a pointer. Also acceptable would be the following.

```
int *day, *year;
char monthname[20];

scanf(" %d %s %d", day, monthname, year);
```

File I/O

Before information is read from or written to a data file, the file must be opened. A file pointer is needed using a structure declaration called `FILE`.

```
FILE *fp;

fp = fopen( name, "r" );
if( fp == NULL ) exit(-1);
```

where `name` is a character string containing the name of the file, and `"r"` is the read mode argument. The write mode argument would be `"w"`. If `fopen` has any problem in opening the file (because the file does not exist, if reading), then `fp` would be set to `NULL`. Thus, it is advisable to check that `fp` is not `NULL` before trying to read any data.

Reading and writing can be achieved by using `fscanf` and `fprintf`, respectively.

```
int day, year;
char monthname[20];

fscanf( fp1, "%d %s %d", &day, monthname, &year);

fprintf( fp2, "%d %d\n", day, year);
```

These are the general I/O routines, but in C the user may also write their own read and write functions for specific situations, but this is usually cumbersome. The function `fgetc(fp)`, for example retrieves one character from the stream at a time, and the user must check for the new line character, for blank spaces, and all other contingencies. Writing would use `fputc(fp)`. With large files, writing them in binary, `fwrite`, may be more efficient than `fprintf`, and then they would need to be read with `fread`. Binary reading and writing involves structures and buffers. Binary files typically take up less disk space and are faster in reading and writing. The downside is that the file can not be browsed or edited by usual UNIX commands like `more` or `emacs`.

Files can be read more than once in a program. After reaching the end of a file (EOF), then `rewind(fp)` will reset the file to the first record. If a data file has been read and will no longer be needed, then the file should be closed with `fclose(fp)`. Similarly, when writing a file, it is a good idea to close the file before exiting the program.

Storage Allocation

In FORTRAN 77 programs the programmer had to know the maximum sizes of any arrays or matrices that would be encountered by the program. Thus, an array dimensioned at 1 million elements would always take up the same amount of space even if the problem being solved used only 1000 elements of the array. Memory space is needed by all programmers and on one system there could be competition for that space. Thus, if only 1000 elements would be needed this time, it would be friendly to only use memory for 1000 elements rather than 1 million. In C there is the possibility to only use the memory space that is necessary through dynamic allocation. There are two functions in C for requesting fresh regions of memory, called `malloc` and `calloc`. Suppose we have a structure and we need memory to store another instance of that structure.

```
struct pedigree anim, *p;

p = (struct pedigree *) malloc(sizeof(struct pedigree));
if (p == NULL)
{
    printf(" Ran out of memory\n");
    exit(-1);
}
```

Note the use of pointers to point to the new memory location that has been allocated. Also note the check that is needed to verify that there was enough space. The programmer needs to keep track of the memory locations because they will not necessarily be sequential. They can be stored in an array.

`malloc` only finds the free space for you, but that space will not be initialized to any specific values. The user must then initialize the space to zeros or whatever values are

appropriate for the program. An alternative is `calloc` which sets that region of memory to all zero bits.

```
struct pedigree anim, *p;
m = 1;
p = (struct pedigree *) calloc(m, sizeof(struct pedigree));
if (p == NULL)
{
    printf(" Ran out of memory\n");
    exit(-1);
}
```

`calloc` could also find sequential memory for `m` occurrences of `struct pedigree`, rather than just one at a time. The `free(pointer)` statement is used to de-allocate that memory space. If the `pointer` was to an array, then if the `free` statement has occurred, then the program should no longer make reference to that memory space. All or part of the `free`-ed memory space may be re-allocated by another `malloc` or `calloc` statement.

Math Functions

Access to math functions are declared by including the library header file, `math.h`, but may also require the `-lm` option on the command line when compiling the program. All operations of the math functions are on variables of type `double`. The functions that are available are listed below, but the reader is referred to books on C to determine their usage.

<code>abs</code>	<code>cosh</code>	<code>HUGE_VAL</code>	<code>pow</code>	<code>tanh</code>
<code>acos</code>	<code>div</code>	<code>labs</code>	<code>rand</code>	
<code>asin</code>	<code>exp</code>	<code>ldexp</code>	<code>sin</code>	
<code>atan</code>	<code>fabs</code>	<code>ldiv</code>	<code>sinh</code>	
<code>atan2</code>	<code>floor</code>	<code>log</code>	<code>sqrt</code>	
<code>ceil</code>	<code>fmod</code>	<code>log10</code>	<code>srand</code>	
<code>cos</code>	<code>frexp</code>	<code>modf</code>	<code>tan</code>	