An introduction to C

Bahjat F. Qaqish
(bahjat_qaqish@unc.edu)
The University of North Carolina at Chapel Hill

Introduction

This brief introduction to the C programming language is meant for those who know other programming languages such as FORTRAN, Pascal, SAS/IML, S, R, MATLAB, etc. The introduction is based on a series of short examples. A few comments are given after each example, but not everything is explained. It is definitely not the intent here to give detailed descriptions of the syntax and semantics of of the C language. It is expected that the reader will learn primarily by studying the examples and running them. All programs will be in ISO C, specifically the version known as C89. Features of C99 (e.g. variable-length arrays) will not be used. Save all programs you type.

To learn more about the C language, the books "C Programming: A Modern Approach" by K. N. King (1996) and "Pointers on C" by K. A. Reek (1997) are recommended. The book "C: A Reference Manual", 5th edition, by S. Harbison and G. Steele (2002) is recommended as a comprehensive reference, but, naturally, it is not a book from which one can learn the C language.

The first step

Create a file called `prog1.c` that contains the single line:

```
int main () { return 0; }
```

The details of how to compile and link a program are system-dependent. On UNIX systems that have the GNU compiler installed, the command is:

```
gcc prog1.c  -lm
```

Other common commands that invoke a compiler are `CC` and `cc`. To run the compiled code (on a UNIX system) issue the command

```
a.out
```

You have just run your first C program!

Developing a C program invloves the following steps:

1. Write or edit the program using a text editor,
2. Compile and link using the `gcc` or similar commands. The result of compiling (and linking) is a file called `a.out`. If there are compilation errors go to step 1.
3. Run the program by typing the command `a.out`. If there are runtime erros go to step 1.

There are variations, but these are the basic steps. Every C program must contain a function called `main()` and it is where execution begins. `int main()` means that `main` is a function that returns an integer. That integer value is passed back to the operating system upon exit. The above program returns the value `0`. The body of the function is between two braces {}. Most C programs contain other functions besides `main`.

C is case-sensitive; variables `x` and `X` are different.

Fundamantal Data Types, Output

Run (i.e. edit, compile and run) the following program, `prog2.c`:

```
#include <stdio.h>
int main ()
{
  int    i = 5;
  double x = 3.5;
  const char title[] = "Introduction to C\n";

  printf("%s", title);
  printf("i = %d\n",  i);
  printf("x = %g\n",  x);
  printf("i = %d, x = %g\n",  i, x);
  printf("x = %g, i = %d\n",  x, i);
  return 0;
}
```

There are three fundamental data types in C: *integer*, *floating point* and *pointer*. In the above program, `i` is of an integer type called `int`. `x` is of a floating point type called `double`. The character type (`char`) is basically an integer type. Strings (such as `title` in the above program) are represented in C as arrays of `char`s. Two important facilities that extend the fundamental types are *arrays* and *structures*; both will be discussed in later sections.

Note that it is possible to initialize variables at the time they are defined as in `int i = 5` and `double x = 3.5`. In fact, it is good practice to do so if possible.

The function `printf` prints values of variables according to *conversion specifications*. The specification `%d` requests that the value be printed as an `int`, `%g` specifies a `double` and `%s` specifies a string. More than one variable can be printed by a single `printf` but the formats must match the variable types. The `\n` instructs printf to go to a new line. Whenever `printf` or similar input/output functions are used, the `#include <stdio.h>` line must appear before `main`. Note that the `#` character must be in the first column on the line.

Output from `printf` goes to the *standard output file* named `stdout`, usually diverted to the screen or console, but it can be redirected to a file, for example (UNIX commands)

```
a.out > prog2.out
cat     prog2.out
```

Input

Run the following program, `prog3.c`:

```
#include <stdio.h>
int main ()
{
  int i;
  double x;

  printf("Enter int i:");
  scanf("%d", &i);
```

```
  printf("Enter double x:");
  scanf("%lf", &x);

  printf("i = %d, x = %g\n",  i, x);
  return 0;
}
```

The function `scanf` is used to read values of variables. The `%d` specifies an integer, while `%lf` specifies a double precision floating point value. More than one variable can be read but the formats must match the variable types. Notice that a `&` must preceed variable names in `scanf`. Notice also that `#include <stdio.h>` is required.

Input of `scanf` is read from the *standard input file* named `stdin`, usually diverted from the keyboard, but it can be redirected from a file. Create a file `prog3.dat` containing the numbers `5 3.5` (on either one or two lines), generate `a.out` from `prog3.c`, then issue the UNIX command

```
a.out < prog3.dat
```

---

if else

Run the following program, `prog4.c`:

```
#include <stdio.h>
int main ()
{
  int i, s;

  printf("Enter int i:");
  scanf("%d", &i);

  if (i < 0)  {
    printf("Value entered < 0\n");
  }
  else {
    printf("Value entered >= 0\n");
  }

  if (i < 0)  s = -1;
    else if (i > 0) s = 1;
    else s = 0;

  return 0;
}
```

Observe the following: There is no `then` in C. The logical expression after `if` must be enclosed in parentheses. When the condition is true, the statement, or { block of statements }, that follow are executed. Equality and non-equality are tested as follows

```
  if (i == 0)  printf("i is 0.\n");
  if (i != 0)  printf("i is not 0.\n");
```

Other comparison operators are: >= and <=.

Warning: A common error is to use a single = for testing equality. In C, = is for assignment and == is for testing equality. The error doesn't generate a compiler error since

```
if (i = 0)  printf("This will never print.\n");
```

is a valid C construct (assignments are expressions that have a value). It helps to write comparisons as

```
if (0 == i) ...
```

rather than

```
if (i == 0) ...
```

The reason is that if == is typed as = by mistake, the code

```
if (0 = i) ...
```

will generate a compilation error. However this obviously doesn't help when two variables are compared

```
if (x = y) ...  /* oops, meant: if (x == y) ... */
```

You've been warned!

Logical expressions

All logical expressions evaluate to either 0 or 1. We saw above simple logical expressions such as

```
(i > 0)
```

which evaluates to 1 if `i` is greater than zero, and to 0 otherwise. The resulting value can be used not only in `if` statements, but also in arithmetic expressions, e.g.

```
npositive = npositive + 5 * (i > 0);
```

has an effect similar to

```
if (i > 0)  npositive = npositive + 5;
```

Logical expressions can be combined. The lines

```
a = (i > 0 && j > 0);
b = (i > 0 || j > 0);
c = !(i+j == 0);
```

do the following:
set `a=1` if both `i` and `j` are positive, `a=0` otherwise,
set `b=1` if at least one of `i` and `j` is positive, `b=0` otherwise,
set `c=1` if `i+j` is not 0, `c=0` otherwise.
So, `&&` stands for *logical AND*, `||` stands for *logical OR*, and `!` stands for *logical NOT*,

Warning: The single-character operators `&` and `|` are *bitwise* operators. A common error is to type `&` instead of `&&` or `|` instead of `||`.

---

for

Run the following program, `prog5.c`:

```
#include <stdio.h>
int main ()
{
  int i;
  double x;

  for (i=1; i <= 5; i = i+1)  printf("%d*%d=%d\n",  i, i, i*i);

  for (x=0; x <= 1 ; x = x+0.2)  {
    printf("x   = %g, ",  x);
    printf("x*x = %g\n",  x*x);
  }
  return 0;
}
```

The first loop above runs as follows:

1. i = 1 (initialization).
2. If i $\leq$ 5 (testing the condition) then proceed, else go to step 6.
3. Print i and i^2 (body of the loop).
4. i = i + 1 (increment/decremnt).
5. Go to step 2.
6. End of loop.

Simply: Set i=1, then as long as i$\leq$ 5 execute the body of the loop and increment i by 1.

The body of a `for` loop may consist of a single statement or a { block of statements }. The braces are optional if the body consists of a single statement.

The body may be empty,

```
  for (i=1; i <= 10;  i = i + i);
```

This loop exits when `i` is 16.

The body of a `for` loop may not run at all, e.g. if $n == 0$ in this code

```
  /*  s <- 1 + 2 + 3 + ... + n  */
  s = 0;
  for (i = 1; i <= n;  i = i+1)    s = s + i;
```

---

break, continue

Run the following program, `prog6b.c`:

```
#include <stdio.h>
int main ()
{
  int i;
```

```
  for (i=1 ; i <= 10 ; i = i+1 ) {
      printf("Inside the loop, before break: i = %d\n",  i);
      if (i > 5) break;
      printf("Inside the loop, after  break: i = %d\n",  i);
  }
  printf("Outside the loop: i = %d\n",  i);
  return 0;
}
```

Run the following program, `prog6c.c`:

```
#include <stdio.h>
int main ()
{
  int i;

  for (i=1 ; i <= 10 ; i = i+1 ) {
      printf("Inside the loop, before continue: i = %d\n",  i);
      if (i > 5) continue;
      printf("Inside the loop, after  continue: i = %d\n",  i);
  }
  printf("Outside the loop: i = %d\n",  i);
  return 0;
}
```

The `break` statement is used to break out of blocks. The `continue` statement is used to go back to the top (beginning) of the loop (the loops continues to run). Compare output from the two programs above.

Operators

Instead of

```
  i = i + 1;
```

one can write

```
  ++i;
```

or

```
  i++;
```

The form `++i` is *pre-increment* while `i++` is *post-increment*. (In this section, it is assumed that `i` and `j` have been defined as `int`.) The difference is important when they are used in expressions. For example, after

```
  i = 3;
  j = (++i);    /* same as: i = i + 1; j = i;  */
```

`j` will have value `4`, because the expression `++i` returns the value of `i` after incrementing. However, after

```
  i = 3;
  j = (i++);   /* same as: j = i; i = i + 1;  */
```

`j` will have value `3`, because the expression `i++` returns the value of `i` before incrmenting.

Similarly `i--` means "decrement i by 1". Loops are usually written using `++` or `--`:

```
  for (i=1; i <= 50; ++i)  s += x[i];
  for (i=50; i >= 1; --i)  s += x[i];
```

The following table gives equivalent statements on each line:

```
i = i+1;   i += 1;   i++; ++i;
i = i-1;   i -= 1;   i--; --i;
i = i+5;   i += 5;
i = i-5;   i -= 5;
i = i*5;   i *= 5;
i = i/5;   i /= 5;
```

When either pre- or post-increment form will do, the choice is often a matter of habit. What habit should one develop? It turns out that the pre-increment form has an advantage in the C++ programming language. If you want to be ready when you move to C++, make pre-increment your default choice. For example, write your loops as

```
  for (i=1; i <= n; ++i)   ...
```

rather than

```
  for (i=1; i <= n; i++)   ...
```

Note that integer division yields the integer quotient, thus

```
  i = 11 / 3;
```

produces i=3. The remainder operator is `%`,

```
  i = 11 % 3;
```

produces i=2.

Arrays

Run the following program, `prog7.c`:

```
#include <stdio.h>
int main ()
{
  int i, s = 0, a[21];
  int n  =  sizeof(a) / sizeof(a[0]);

  for (i = 0; i < n; ++i)  a[i] = i;
  for (i = 0; i < n; ++i)  s += a[i];

  printf("0+...+%d = %d\n", n-1, s);
  return 0;
}
```

The array a[] contains 21 elements: a[0], a[1], a[2], ..., a[20]. The first element in array a[] is a[0], not a[1], and the last is a[20], not a[21]. Warning: A common error is to access beyond the array bounds, e.g. a[21].

Notice how variables s and n are initialized in the above program. The following lines show valid ways to define and initialize an array in a single statement:

```
int a[5] =  {2, 3, 5, 7, 11};
int b[5] =  {1, 4, 9};
int c[]  =  {2, 3, 5, 7, 11, 13, 17, 19};
const char t[] =  "Introduction to C\n";
int n_c  =  sizeof(c)/sizeof(c[0]);
int n_t  =  sizeof(t)/sizeof(t[0]);
```

All elements of a[], but only the first three elements of b[], are initialized. If the whole array is initialized it is better not to specify the number of elements, as in c[] and t[]. The number of elements in an array can be computed as shown above.

Functions

Run the following program, prog8.c:

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

double logit (double x)
{
  double t =  log(x/(1-x));
  return t;
}

int main ()
{
  double t, z;

  t = 0.5;
  z = logit(t);
  printf("logit(%g) = %g\n", t, z);
  return 0;
}
```

Note: Using the "-lm" compiler switch is necessary for the above program since it uses the math library function log. This switch links the program to the math library.

Functions can be defined very much like function main. In the above program the function logit is defined as a function that takes an argument of type double and returns a value of type double.

Notice that variable t in logit is local to logit, i.e. it is different from t in main. This is one of the advantages of using functions; local variables don't interfere with variables in the calling function.

The return value of a function need not be assigned to a variable; in that case the return value will be discarded. The logit function above can be called as follows

```
logit(t);
```

In the above program, the function logit was defined before function main. It is possible to reverse that order. However, in that case, logit must be declared before it can be used, for example:

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

double logit (double x);

int main ()
{
  double t, z;

  t = 0.5;
  z = logit(t);
  printf("logit(%g) = %g\n", t, z);
  return 0;
}

double logit (double x)
{
  double t =  log(x/(1-x));
  return t;
}
```

The line

```
double logit (double x);
```

is a *function prototype*. It gives the compiler enough information about `logit` to allow calling it in `main`. The lines

```
double logit (double x)
{
  double t =  log(x/(1-x));
  return t;
}
```

comprise the *definition* of the function `logit`. This definition may actually reside in a separate file - more on that later.

Note that the local variable `t` in `logit` is not absolutely essential; `logit` could have been equivalently defined as follows:

```
double logit (double x)
{
  return  log(x/(1-x));
}
```

The expression `log(x/(1-x))` will be evaluated and its value returned.

The keyword `void` can be used as a function return type to indicate that the function does not return a value. Functions may take no arguments and return no value:

```
double f1 (void);  /* no arguments, returns a double   */
void   f2 (int i); /* an int argument, no return value */
void   f3 (void);  /* no arguments, no return value    */
```

#include

#include instructs the C *preprocessor* to include other files, usually *header files*. Those files give the compiler information about functions (function prototypes) and possibly various other things. For example, `stdio.h` contains information on input/output

functions, `math.h` contains information on mathematical functions such as `log` and `exp`, `stdlib.h` contains information on memory allocation functions, `string.h` contains information on string handling functions. There are other important standard header files which will not be used in this introduction.

It is not a bad idea to include the following in all your programs:

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

More on Functions

Run the following program, `prog9.c`:

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

void f (double x)
{
  x = 0;
}

void g (double *p)
{
  *p = 0;
}

int main ()
{
  double t;

  t = 5;
  f(t);
  printf("after f(t), t = %g\n", t);

  g(&t);
  printf("after g(&t), t = %g\n", t);
  return 0;
}
```

Function arguments in C are passed by value. This means that in the call

```
  f(t);
```

a copy of t is made and only the copy is made available to function `f` as its first argument, `x`. When f(x) modifies x in the line

```
  x = 0;
```

only the local copy of t is modified. The value of `t` in `main` is not affected.

So, how can functions modify values of variables? To allow a function to modify the value of a floating point number or an integer, its address (memory address, the memory location where it is stored) must be passed to the function as in

```
  g(&t);
```

where "&t" is "the address of t", and the function must be written accordingly; function `g` takes as an argument a "pointer to double". To access the variable whose address is p, we use "*p" as in

```
  *p = 0;
```

which means "assign 0 to the variable whose address is p" or "assign 0 to the variable pointed to by p". The variable p is a *pointer*. The three basic data types in C are: integer, floating point and pointer.

Array arguments can be modified without using &. This is because *the name of an array is a pointer to its first element*. If an array is defined as

```
  double x[21];
```

then the name "x" refers to the address of x[0], i.e. "x" is equivalent to "&x[0]". The following function fills an array with zeros.

```
void zero (double x[], int n)  /* x[0:n-1] <- 0  */
{
  int i;

  for (i=0; i < n; ++i)  x[i] = 0;
}
```

It can be called as follows

```
  zero(a, 30);
```

assuming a[] has been defined as an array of 30 or more doubles.

Notice the /* comment */ that describes what the function does.

Arrays of variable size

Run the following program, `prog10.c`:

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

int main ()
{
  int n;
  double *x, *y;

  /* allocate */
  n = 10;
  x = malloc(n*sizeof(x[0]));
  if (x == NULL) exit(1);
  printf("x[] now contains %d elements\n", n );
  /* use x[] here */

  /* grow     */
  n = 100;
  y = realloc(x, n*sizeof(x[0]));
```

```
  if (y == NULL) exit(1);
  x = y;
  printf("x[] now contains %d elements\n", n );
  /* use x[] here */

  /* shrink   */
  n = 5;
  y = realloc(x, n*sizeof(x[0]));
  if (y == NULL) exit(1);
  x = y;
  printf("x[] now contains %d elements\n", n );
  /* use x[] here */

  free(x);
  return 0;
}
```

It is useful to have arrays whose size is unknown at compile time, and arrays whose size can change while a program is running. In this section we shall consider one-dimensional arrays only. The general strategy, illustrated in the above program, is as follows:

1. Declare the array simply as a pointer to the desired type.
2. Use function `malloc` to allocate as much memory as needed. `malloc` allocates a block of memory and returns a pointer to it. It should be assumed that the allocated memory contains garbage. If not enough memory is available `malloc` returns a special value, NULL. It is a good idea to always check the returned value and take appropriate action if it is NULL.
3. To resize (grow or shrink) an array while preserving its contents use function `realloc`. Also check for a NULL return value. `realloc` may move the array to another area in memory; the array doesn't necessarily grow or shrink in-place.

   Note that in the above program the result of `realloc` is assigned to a temporary variable y, rather than to x directly. This way, if `realloc` fails and returns NULL, the original value of x is not lost and the elements of x[] will still be accessable. Had we written

   ```
   x = realloc(x, n*sizeof(x[0]));
   ```

   the original value of x would be lost if `realloc` fails. Rule: Always use a "spare pointer", like `y` above, with `realloc`.
4. When the array is not needed anymore, use `free` to release the memory, i.e. return it back to the system. A common programming error in C is to allocate memory and never free it, even though it will never be used (within that program). That can occur in functions that allocate memory for temporary storage, but never free it. This error is known as "memory leakage".

#define

Sometimes a certain constant needs to appear two or more times within the same program. For example, suppose we want a program that can process up to 100 items, and we keep item data in two arrays; an array of 100 `int`s and an array of 100 `double`s.

```
#include <stdio.h>
int main ()
{
  const int capacity = 100;
  int       item_code[100];
  double    item_weight[100];  /* weight in Kg */
```

```
  int       i, n = 0;

  ... /*  read "n" and code and weight data */

  for (i = 0; i < n; ++i)
    printf("Item code: %d,  weight: %g Kg\n",
            item_code[i], item_weight[i]);

  return 0;
}
```

Now, suppose we want to increase the maximum capacity to 500 items. In editing the code, we may forget to replace all occurances of "100" with "500". One way to localize updates to a single location is as follows `prog11.c`:

```
#include <stdio.h>
#define CAPACITY 100
int main ()
{
  const int capacity = CAPACITY;
  int       item_code[CAPACITY];
  double    item_weight[CAPACITY];  /* weight in Kg */
  int       i, n = 0;

  ... /*  read "n" and code and weight data */

  for (i = 0; i < n; ++i)
    printf("Item code: %d,  weight: %g Kg\n",
            item_code[i], item_weight[i]);

  return 0;
}
```

Now all we need to change is the line with the #define directive, we change. "#define CAPACITY 100" to "#define CAPACITY 500".

Lines that start with # are *preprocessor directive*. The preprocessor goes through the source text `prog11.c`, replaces occurances of "CAPACITY" by the literal "100", then passes the resulting transformed file to the C compiler proper. Occurances of "CAPACITY" within comments or string constants are not transformed. Note that "CAPACITY" is not a C variable, but rather a *preprocessor macro*.

Keep in mind that the C preprocessor has its own language, and it is not C! Two warnings to emphasize that fact: 1) Don't use an equal sign after the name being #defined. 2) Don't put a semicolon at the end of #define or any other preprocessor command.

Another way to localize updates is to a use `enum`. Replace the #define line in `prog11.c` with

```
enum { N = 100 };
```

Several constants can be appear in a single `enum`:

```
enum {
  ROWS = 25,
  COLS = 80,
  SIZE = ROWS*COLS
};
```

enum has a technical advantage over #define, but it works only for integer constants (but not floating point types).

typedef

Run the following program, `prog12.c`:

```
#include <stdio.h>
enum { MAX_VEC_LEN = 100 };
typedef double VECTOR [1 + MAX_VEC_LEN];

double sum (VECTOR x, int n)     /* returns x[1]+...+x[n]  */
{
  int i;
  double s=0;

  for (i=1; i <= n; ++i)  s += x[i];
  return s;
}

int main ()
{
  int i, n = 20;
  VECTOR x;

  for (i=1; i <= n; ++i)  x[i] = i*i;
  printf("1+4+9+16+...+%d = %g\n", n*n, sum(x, n));
  return 0;
}
```

New data types can be defined using `typedef` (type definition). A new data type called VECTOR is defined in the above program. Once a new type is defined, it can be used just like the the standard types int and double.

struct

Run the following program, `prog13.c`:

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

typedef struct
{
  char *name;
  double x;
  double y;
} CITY;

double distance (CITY a, CITY b)
{
  double dx, dy;

  dx = a.x - b.x;
  dy = a.y - b.y;
  return sqrt(dx*dx+dy*dy);
}

int main ()
{
```

```
  CITY  c1, c2;

  c1.x = 1; c1.y = 2;  c1.name = "Chapel Hill";
  c2.x = 7; c2.y = 10; c2.name = "Hillsboro";
  printf("The distance between %s and %s is %g miles.\n",
          c1.name, c2.name, distance(c1, c2));
  return 0;
}
```

The above program defines a new data type called CITY. The CITY structure consists of three components (members, or fields); coordinates x and y and a name. Structures are a convenient way to keep together various pieces of information about an object. Structures are passed to functions by value and function return values can be structures.

Recursion

Suppose we want to print all permutations of array a[1], ..., a[n] (where n is not fixed in advance). Suppose further that we have a function that can permute elements a[i+1], ..., a[n], where i ($\leq$ n) is a fixed integer. If i+1 == n, the function simply prints a[], since there is only one permutation of the single element a[n]. One key idea is that such a function can be "extended" to permute a[i], ..., a[n] as follows: First call the function to permute a[i+1], ..., a[n]. Then swap a[i] with a[i+1], call the function again to permute the slighly modified a[i+1], ..., a[n], and swap back a[i] and a[i+1] (i.e restore the original array order). Next, swap a[i] with a[i+2], call the function again to permute a[i+1], ..., a[n], and swap back a[i] and a[i+2]. Proceed in this way until, in the last step, a[i] is swapped with a[n]. Another key idea is that the extension can be built into the function itself. This means that the function will call itself. This leads to the following implementation:

```
void perm   (int a[], int i, int n)
  /* output all permutations of a[i],...,a[n]. */
  /* a[] is not modified. */
  /* recursive */
{
  int j, t;

  if (i == n)     output(a, n);       /* <- recursion ends here */
  else  {
    for (j = i; j <= n; ++j) {
      t=a[i]; a[i]=a[j]; a[j]=t;      /* swap a[i], a[j] */
      perm(a, i+1, n);                /* permute a[i+1], ..., a[n] */
      t=a[i]; a[i]=a[j]; a[j]=t;      /* swap a[i], a[j] */
    };
  };
};
```

Functions that call themelves are called *recursive functions*. Recursion is a natural candidate when a method for solving a problem of "size" n is available and it can be extended easily to solve a problem of size n+1. The "extension" code is written into the body of the function, and the function calls itself. The core of function `perm` above is a function that can perform the trivial task of permuting a single element, a[n]. One level of recursion allows it to permute two elements; a[n-1], a[n]. Another recursion extends it further to permute a[n-2], a[n-1], a[n]. One more recursion extends it to permute a[n-3],..., a[n]. Repeated recursion allows it eventually to permute the whole array a[1], ..., a[n].

while, do while

In addition to for-loops, there are two other iteration constructs. The "do while" construct repeats an action (a block of code) while a condition is true:

```
do {ACTION}  while (CONDITION);
```

Note that ACTION will be executed *at least once*, since CONDITION is tested at the "bottom" of the loop.

The other iteration construct, "while", repeatedly tests a condition and executes an action if the condition is true:

```
while (CONDITION) {ACTION}
```

The ACTION *may not execute at all*, since the CONDITION is tested at the "top" of the loop.

The for-loop in function `sum`, `b012.c`, can be rewritten in the following forms:

```
i = 1;
while (i <= n) {s += x[i]; ++i;}

i = 1;
do {s += x[i]; ++i;} while (i <= n);  /* n > 0 assumed */
```

Examples: Each of the two functions below returns the smallest element of an array.

```
int min1 (int a[], int n)
  /* returns minimum of a[1:n], assumes n >= 1 */
{
  int smallest = a[1];
  int i;

  for (i = 2; i <= n; ++i)
     if (a[i] < smallest)  smallest = a[i] ;

  return smallest;
}

int min3 (int a[], int n)
  /* returns minimum of a[1:n], assumes n >= 1 */
{
  int temp = a[n];     /* a[n] will be used to store intermediate values */
  int smallest = a[n]; /* initial value */
  int  i = 0;          /* start at a[0] */

  for (;;) {
    while (a[++i]  >  smallest); /* scan forward  */
    if (i == n) break;           /* exit the loop if at a[n] */
    a[n] = smallest = a[i];      /* found a smaller element */
  }
  a[n] = temp;                   /* restore original value */

  return smallest;
}
```

Function Prototypes, Separate Compilation and Libraries

Function prototypes are statements that inform the compiler about the number and types of function arguments and about the function return type. The actual code comprising the function is not part of the prototype. Here is a collection of prototypes of functions that compute simple descriptive statistics:

```
/* stats.h:  descriptive stats library header file */
double mean( double x[], int n );
double stdev( double x[], int n );
double variance( double x[], int  n );
double corr( double x[], double y[], int n );
```

The first line above is the prototype of function `mean`. It declares `mean` to be a function that takes two arguments, an array of doubles and an integer, and returns a value of type double. This is all the information the compiler needs in order to use `mean` in a program. The actual code that executes when `mean` is called may appear later in the same file. It may instead appear in a separate file, and in that case the function prototypes are collected in a special *header file* `stats.h`. Header files contain function prototypes, typedefs and #defines, but no statements that would generate executable code.

The actual code for the statistical functions is in file `stats.c`.

```
/* stats.c:  descriptive stats library  */

#include <math.h>
#include "stats.h"

/*********************************************************/
double mean(double x[], int n)
  /* returns the mean of x[1],...,x[n] */
{
  double s=0;
  int i;

  for (i=1; i<=n; ++i) s += x[i];
  return s / n;
}
...
```

Now, `stats.c` can be compiled to generate an object file, "stats.o", by the UNIX command:

```
gcc -c  stats.c
```

where the "-c" means "compile only" (but don't link).

In order to use the statistical fucntions developed above in a program, two things are needed: 1) The header file "stats.h" must be #included in the program. 2) The program must be linked to the object file "stats.o". Example:

```
/* b017.c:  use external function libraries */

#include <stdio.h>
#include "stats.h"

#define N 100

int main ()
{
  int n=N;
```

```
  double x[1+N];

  ...

  printf("N = %d\n", n);
  printf("Mean = %f\n", mean(x, n));
  printf("Var  = %f\n", variance(x, n));
  return 0;
}
```

Notice that "stats.h" is enclosed in quotes rather than <> which is reserved for the standard library files such as stdio.h, stdlib.h, math.h, etc. The program is compiled and linked to "stats.o" by the UNIX command

```
gcc     b017.c   -lm   stats.o
```

Summary: 1) To implement a function library, put function prototypes in a header file and actual function code in one or more files that should #include the header file, and which are compiled with "-c" to generate object files. 2) To use a function library #include the header file and link to the object file(s).

There is a variation on the above. The four statistical functions may be placed in separate files `mean.c, stdev.c, variance.c, corr.c`. Each of these files must #include "stats.h". They are compiled into four object files, which are then aggregated into a single *library* or *archive* file with the `ar` command.

```
gcc -c mean.c
gcc -c stdev.c
gcc -c variance.c
gcc -c corr.c
ar  -r stats.o   mean.o stdev.o variance.o corr.o
```

The advantage of this approach is that individual modules in stats.o can be updated without affecting other modules. If `corr.c` was modified, stats.o could be updated by

```
gcc -c corr.c
ar  -r stats.o   corr.o
```

An important tool for automating updates of software libraries and programs is the "make" utility. It will be covered in a separate section.

Passing functions as arguments to other functions

Functions can be passed as arguments to other functions. Suppose we wish to compute y[i] = f(x[i]) for i=1,...,n, where `f` can be the `sqrt(), log(), sin()` or a any compatible user-defined function. This can be achieved as follows.

```
#include <math.h>
#define N 100

void apply (double y[], double f (double), double x[], int n)
  /* y[1:n] <- f(x[1:n])  elementwise */
{
  int i;

  for (i=1; i <= n; ++i)  y[i] = f(x[i]);
}

double logit (double p)  {  return log(p/(1-p)); }

int main ()
{
  double x[N+1], y[N+1];
  ...

  apply (y, sqrt,  x, N);
  apply (y, logit, x, N);
  return 0;
}
```

This facility is useful in writing general-purpose functions that solve f(x)=0, maximize f(x), differentiate or integrate f(x), etc. For another example, see b018.c.

The standard library function `qsort` has a function as one of its arguments, see b016.c. This allows qsort() to sort various types of arrays.

---

static data

The keyword "static" allows functions to retain or save values of local variables between calls. One such example is the function `factorial` shown below. It computes factorials only once, the first time it is called, and saves them in a local array. Without the "static" qualifier, local variables and arrays disappear and lose their values between calls.

The variable `ready` indicates whether the local array has been filled properly or not. Initially, `ready` is set to 0. That causes the block that initializes the array `fact` to execute the first time `factorial` is called. The same block also sets `ready` to 1 to indicate that `fact` has been initialized. Subsequent calls to `factorial` will find that `ready==1` and that block will not execute. The array `fact` is initialized only once, the first time `factorial` is called. Subsequent calls to `factorial` will simply use the previously computed values.

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

#define FACTMAX 100

double factorial (unsigned int i)
{
  static double fact[FACTMAX+1];
  static int ready = 0;
  int j;

  if (i > FACTMAX) return -1;   /* out of range */

  if (!ready)  {        /* initialize the array fact[] */
    fact[0] = 1;
    for (j=1; j <= FACTMAX; ++j) fact[j] = j*fact[j-1];
    ready = 1;          /* now we are ready */
  }

  return fact[i];
}

/*********************************************************/

int main()
{
  int i;

  for (i=0; i <= 105; ++i)
      printf ("factorial(%d) = %g\n", i, factorial(i) );

  return 0;
}
```

A similar use of static data occurs in implementing the polar method of simulating pseudo-random normal variates. In the polar method, two values are simulated at a time. On odd-numbered (first, third, fifth, etc) calls, pairs of values are generated; one value is output and the other saved for the next call. On even-numbered (second, fourth, sixth, etc) calls the saved value from the previous call is output, but nothing new is generated. See rannor.c.