1 Structure of C programs

C programs consist of functions. Functions are collections of C statements that use data stored in C variables to perform operations specified in expressions. C functions are like the functions in R. The function named main is special because it signifies where a program begins execution and therefore must be present in every program. There are usually calls to other functions, either user-written or available in various libraries, to perform various tasks. An example of a simple program in which the function printf is called in main to print a string of characters, follows:

```c
main ()
{
    printf("hello, world\n");
}
```

1. main () is a function declaration.
2. Functions must be followed by parentheses, even if there are no arguments.
3. Braces {} delineate a block of statements, to be executed together.
4. printf is a function call. It is defined in the Standard C Library and is linked with hello.c when the program is compiled and linked.
5. Double quotes enclose a string constant.
6. \n is a new line character and results in a line-feed.
7. The string is output to the standard output (in this case the window).

More information on how to write functions will be given later.

2 Data types and sizes

Expressions written in the C language combine variables and constants to produce new values which may be assigned to other variables. The data type of these objects determine the results of operations performed and values stored in them. The basic type declaration statements in C are:

- char A single byte capable of holding one character in the local character set.
- int An integer, reflecting the natural size of integers on the host machine.
- float Single-precision floating point.
- double Double-precision floating point.

Variables

Some syntax rules concerning variables are:
Variable names can contain letters, digits, and underscores.

- The first character must be a letter or underscore.
- C is case sensitive! x is not the same as X.
- Normally, only the first 8 characters are significant.
- Reserved words (e.g. if, else, int etc.) cannot be used as variable names.

**Constants**

Constants in C can be any of the data types described above. For example, 2367 is an int type constant and 236.6 may be a float or a double type constant. A variable may be initialized with a value when declared:

```c
int lower = -52791;
float val = 371.9425, e1 = 3.719425E2, e2 = 3719425E-4;
char letter = 'x', state[] = "New Jersey";
double e3 = 3719425e-4, eps = 2.71828182846182e-15;
```

state[] defines a character array to store the character string shown. Arrays are discussed below.

**Array variables**

In C, the statement

```c
int ndigit[10];
```

declares ndigit to be an array of 10 contiguous memory locations. The elements ndigit[0], ndigit[1], ..., ndigit[9] are of the integer type. Notice that in C the subscripts begin at 0 compared to 1 as in Fortran. The array could be declared to be of any type as including character arrays used to store string constants. C also provides multi-dimensional arrays. Here is an example of an array initialization:

```c
char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

This two-dimensional array has two rows and thirteen columns. The elements of the array may be accessed or replaced using subscripts as in daytab[i][j], where i refers to the row and j to the column, respectively.

**Expressions**

C expressions combine values of constants, variables, array elements, function references etc. using operators and rules of precedence and associativity of the operators to form a value that may be assigned to other variables etc. In C, there are many types of expressions: primary, arithmetic, conditional etc.

The rules of precedence and associativity of the operators are given in the Table 1. In the table, operators are listed in the order of decreasing precedence and operators on the same level have the same precedence. For example, * has a higher precedence than + or -- which have the same precedence. Some examples are found in Table 2.
Operators and their associativity are as follows:

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ] - &gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>tt ! ++ - + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= =&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>:= = += -= % ^ = &amp;= ^= &gt;&gt; = &gt; &lt; &lt; = right to left</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Unary +, -, and * have higher precedence than the binary forms.

Table 1: Precedence and Associativity of Operators

3 A Slightly More Complex C Program

Although, it is not a requirement, for ease of readability it is recommended that C statements be written on separate lines and indented to show groups of statements to be executed together. Now that we have introduced C data types and C expressions, a more complex example can be presented. Here note that in the newer standards require that the `main()` function be declared to be of an `int` type and and argument of `void` be specifically used.

<table>
<thead>
<tr>
<th>Original expression</th>
<th>Equivalent expression</th>
<th>Reason for equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a*b+c</td>
<td>(a*b)+c</td>
<td>* has higher precedence than +</td>
</tr>
<tr>
<td>a+=b!=c</td>
<td>a+=(b!=c)</td>
<td>+= and != are right-associative</td>
</tr>
<tr>
<td>a-b+c</td>
<td>(a-b)+c</td>
<td>- and + are left-associative</td>
</tr>
<tr>
<td>sizeof(int)*p</td>
<td>(sizeof(int))*p</td>
<td>sizeof has higher precedence than cast</td>
</tr>
<tr>
<td>*p-&gt;q</td>
<td>*(p-&gt;q)</td>
<td>-&gt; has higher precedence than *</td>
</tr>
</tbody>
</table>

Table 2: Examples of precedence and associativity

The standard also requires that `main()` produce a `return value`. The zero in `return 0;` indicates successful (or normal) termination of the program. A non-zero integer is used to signal an error of some sort. The caller of the program receives that value as the program’s exit status. The exit status of a simple program such as as `hello.c` does not serve much purpose, but in more complex program, especially those which can be run non-interactively or unsupervised, the exit status can be examined and automated decisions can be made based on its value.
/* print Fahrenheit-Celsius Table */
int main(void)
{
    int lower, upper, step;
    float fahr, celsius;
    lower = 0; /* lower limit of table */
    upper = 300; /* upper limit */
    step = 20; /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f
", fahr, celsius);
        fahr = fahr + step;
    }
    return 0;
}

Here fahr and celsius are declared to be float variables and the conversion of values of fahr starting from 0 through 300 in steps of 20 (degrees fahrenheit) to celsius scale is performed in a while loop. As long as the condition in the parentheses in the while expression remains true, the three statements enclosed in the braces are executed; otherwise, the loop ends and execution continues with the next statement. Since there are no further statements in this program it terminates. We will later look at kinds of loops available in C.

Notice that the division 5.0/9.0 is carried out in floating-point arithmetic giving a floating-point result. But if this is written as 5/9 then the result will be zero since an integer division will take place. However, 5/9.0 or 5.0/9 will also result in the correct floating-point result as the integer will be coerced to floating-point before the division is performed. When an operator has operands of different types they are converted to a common type following this general rule: if an operand can be converted to a wider operand to match the types of the two operands without losing information, then such conversions will be performed automatically so that the operation can be performed more precisely.

4 Conditional Execution of C Statements

if-else

The if-else construct is used to execute one of two statements conditionally. The syntax is:

    if expression
        statement-1
    else
        statement-2

where the else portion is optional.
If an else branch is present it is associated with the nearest preceding if statement. For example, in the following, the else statement belongs to the second if statement:

```c
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

To alter this behavior, you must use braces. Thus the following associates the else statement with the first if statement:

```c
if (n > 0) {
    if (a > b)
        z = a;
} else
    z = b;
```

if-else expressions can be nested using the else-if construct as follows:

```c
else-if
    if (expression)
        statement
else if (expression)
    statement
else if (expression)
    statement
else (expression)
    statement
```

The 'expression's are evaluated in order until one is found which evaluates to true. If a true expression is found, its statement is executed. If none of the expressions are true, then the final statement is executed.

## 5 Loops in C

We encountered a while loop in the above example. The syntax is:

```c
while (expression)
    statement
```
Here the *expression* is evaluated; if it is non-zero (i.e. the condition is true) then the *expression* is executed. This process continues until *expression* becomes zero (i.e. the condition is true), at which point the loop terminates and execution transfers to next statement following the **while** loop. The body of the loop could be a single statement or a group of statements enclosed in braces. In that case the statements in braces are all executed as a single statement.

By contrast to the **while** loop, the **do-while** loop tests the termination condition at the bottom after each pass through the loop body which is therefore executed at least once.

```
    do
        statement
    while (expression)
```

Instead of the **while** loop

```
expr1;
while (expr2){
    statement
    expr3;
}
```

the following **for** loop could be used:

```
for (expr1; expr2; expr3)
    statement
```

Usually, **expr2** is a terminating condition and **expr1** and **expr3** are either assignments or function calls with **expr1** serving the function of initializing the loop variable and **expr3** incrementing it.

If **expr2** is missing, it is assumed to be true. Thus the following is an infinite loop:

```
for(;;) {
    ...
}
```

Here is the previous example reconstructed using a **for** loop:

```
/* Fahrenheit-Celsius Table */
int main (void)
{
    int fahr;
    for (fahr=0 ; fahr<=300 ; fahr=fahr+20)
        printf("%4d %6.1f\n", fahr,
               (5.0/9.0)*(fahr-32)) ;
    return 0;
}
```
6 Functions

In the introduction it was mentioned that C programs consist of functions and the special functions `main` and `printf` were introduced.

Here an example is given where a function `power()` is constructed and called from `main()`:

```c
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;
}

int power(int x, int n)
{
    int i, p;
    p = 1;
    for (i = 1, i <= n; i++)
        p = p * x;
    return p;
}
```

The general structure of a C function is as follows:

```c
<typename (arg list with associated type declarations, if any )
{
    declarations for local variables
    statements
    {
        statements
    }
}
```

The following points are not discussed here but are very important and must be looked up in the C text if needed.

- C functions are never nested
- Function arguments are always passed by value
- If `type` is omitted it is assumed to be `int`
- All functions return a value but the value needn’t be used.
- ANSI C allows the type to be `void` indicating no useful value is returned.
- Variables declared within a function are local to that function.
- Variables declared outside all functions are globally accessible.
7 Compiling C programs under Linux

C programs are usually edited into a file with an extension of .c (for example, hello.c) using an editor such as emacs. The line #include <stdio.h> at the top of hello.c before it is compiled. It is a C preprocessor directive. Its effect is to insert the contents of stdio.h or the header file in hello.c. The header file stdio.h contains, among other things, a declaration of the function printf. It’s the programmer’s responsibility to #include the necessary header files in the program so that every external function is supplied with a proper declaration. We will see examples of function declarations later. At this point it suffices to know that a function declaration in C is needed if C hasn’t encountered the function before it is called.

/* program hello.c */
#include <stdio.h>
int main(void) {
    printf("Hello World!\n");
    return 0;
}

As mentioned earlier, printf is a function defined in the Standard C Library and is linked with hello.c when the program is compiled and linked. The purpose of the Standard C Library is to free the user of the burden of writing functions to perform frequently needed operations. In the case of input/output operations, which are heavily operating system dependent, the caller of printf is relieved of the burden of dealing with dependency.

On a computer running the Unix operating system or a look-alike, once a program is edited into a file, such as hello.c, it can be compile by executing the command cc hello.c. If there are no syntax errors, the compiler will produce an executable file named a.out. When the program is executed by entering a.out or ./a.out on the command line, the words Hello World! will be printed to the terminal. On GNU Linux, the following command program:

gcc -o hello.exe hello.c

where hello.c is the file containing the C program. gcc is the compiler in most Linux and GNU Public License Operating Systems. The -o flag names the file where the executable will be saved. In this case, it is the binary file hello.exe. As mentioned earlier, the default is a.out. Other options that may be included on this command line will be discussed later.

8 Input-Output Operations in C

Recall that, to perform standard input and output, the pre-processor directive include<stdio.h> must be included in the the program file. This directive provides the compiler the information it needs to check the input/output functions in the standard C library.

8.1 The printf function

We have already seen the printf function which is used for formatted printing. This function allows us to print values and variables to the standard output (in most cases, the screen). For instance, to print the mean ave and standard deviation sd of n observations, the following
would be an appropriate print statement:

```c
printf("Sample size = %d; Mean = %f; SD = %f\n", n, ave, sd);
```

The above `printf` statement contains two sets of arguments: the `control string` enclosed in double quotes, and a set of `identifiers` to specify the value to be printed. The control string contains text, conversion specifiers, or both. Above the control string specifies that the string `Sample size =` is to be printed followed by the value of `n`. The characters `%d` represents a conversion specifier (to convert the value of the `int` variable `n` so that it can be printed). This is followed by the string `Mean =`, which is followed by the value of `ave`. The characters `%f` is a specifier to convert the floating-point variable `ave` for printing. The next string `SD =` follows, after which we have another conversion specifier `%f` to print the floating-point variable `sd`. Finally, the characters `\n` causes a skip to the next line, after the current line as described above has been printed.

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Output Type</th>
<th>Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>short, int</td>
<td>int</td>
<td>%i, %d</td>
</tr>
<tr>
<td>int</td>
<td>short</td>
<td>%hi, %hd</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>%li, %ld</td>
</tr>
<tr>
<td>int</td>
<td>unsigned int</td>
<td>%u</td>
</tr>
<tr>
<td>int</td>
<td>unsigned short</td>
<td>%hu</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
<td>%lu</td>
</tr>
<tr>
<td>float, double</td>
<td>double</td>
<td>%f, %e, %E, %g, %G</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>%LF, %LE, %Lg, %LG</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>%c</td>
</tr>
</tbody>
</table>

Table 3: Conversion Specifiers for Output Statements

Table 8.1 provides a listing of the conversion specifiers. To select a conversion specifier, we can select the type of specifier as per Table 8.1. To print an integer or a short integer, we use either the `%i` (integer) or the `%d` (decimal) specifier. Either of these specifiers provides the same result. For single-precision or double-precision floating point numbers, we use `%f` (floating-point format) or `%e` or `%E` (exponential format, depending on whether the `e` is written as `e` or `E`. The `%g`/%G (general) specifier prints the value using `%f` or `%e/%E` depending on the size of the number being printed.

The backslash in `\n` is called an `escape character` when it is used in a control string. The compiler combines it with the character that follows it, and attaches a special meaning to the combination of the characters. Table 8.1 presents the escape sequences recognized by C. Finally, it should be noted, that while the main purpose of `printf` is to print information, it also returns the number of characters printed.

### 8.2 The `scanf` function

The `scanf` functions allows us to enter values from the keyboard while the program is being executed. Suppose we wanted to read in three values, an integer, a character string, and a floating-point number. We would include the following statement:

```c
scanf("%d %c %f\n", &id, &name, &weight);
```

As in the `printf` case, the `scanf` function has two sets arguments, the first being the control
<table>
<thead>
<tr>
<th>Sequence</th>
<th>Character Represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alert character (ring a bell)</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\f</td>
<td>formfeed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
</tbody>
</table>

Table 4: Escape sequences recognized by the `printf` statement

String that specifies the types of the variables whose values are being entered. Table 8.2 provides the conversion specifiers for the different types of variables. In our case, we want to read in the three integer values `ix`, `iy`, `iz`. So we include the format in which these three variables should be read in. The second part of the statement contains the memory locations of the variables where the read in variables will be stored. Like everything else, memory locations have specific addresses and the unary operator `&` specifies addresses to which the respective variables are associated. Providing the address for each variable being read in is absolutely imperative.

Once again, as with the `printf` function, `scanf` returns an integer value which provides the number of successful conversions.

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>{%i, %d}</td>
</tr>
<tr>
<td>short</td>
<td>%hi, %hd</td>
</tr>
<tr>
<td>long int</td>
<td>%li, %ld</td>
</tr>
<tr>
<td>unsigned int</td>
<td>%u</td>
</tr>
<tr>
<td>unsigned short</td>
<td>%hu</td>
</tr>
<tr>
<td>unsigned long</td>
<td>%lu</td>
</tr>
<tr>
<td>float</td>
<td>%f, %e, %E, %g, %G</td>
</tr>
<tr>
<td>double</td>
<td>%lf, %le, %lE, %lg, %lG</td>
</tr>
<tr>
<td>long double</td>
<td>%Lf, %Le, %LE, %Lg, %LG</td>
</tr>
<tr>
<td>char</td>
<td>%c</td>
</tr>
</tbody>
</table>

Table 5: Conversion Specifiers for Input Statements

9 Reading from/Writing to Files

We have discussed how data are read from the keyboard and printed to a screen. We now turn our attention to the far more useful case of actually reading from and writing to a file.
We will discuss binary files later – because they efficiently store data, they are perhaps the most useful of file types when storing and reading in vast quantities of data.

Each data file in a program is accessed through a pointer. We do this with a FILE declaration (in upper case) which is a data type declared in stdio.h. Therefore, our program should include this header file (we also need it for operations on reading from and writing to files). The following declaration specifies the file pointer.

```c
FILE *ffile;
```

This means that the identifier finput is of the FILE data type, and the identifier (with the asterisk) *ffile is a pointer. When we declare FILE *fin; then fin points to a stream associated with a file. Although a stream is often associated with a file on the computer’s disk, it may also be associated with a magnetic tape, a network socket, a pipe, the keyboard, among other things. After this is defined, it must be associated with a specific file. This is done via the fopen function, which also specifies whether the file is to be opened in read, write or append mode. (There are other modes, too.) So the statement,

```c
fin=fopen("filename.txt","r");
```

means that the file pointer fin is to be used with a file named filename.txt from where we will read in the data. For example, the C statement

```c
fscanf(fin,"%d,%lf",n,weight);
```

The declaration FILE *fout; , say, and the statement

```c
fout=fopen("filename.txt","w");
```

would indicate that the file pointer fout to be used with a file named filename.txt to which we will write data, using a statement like

```c
fprintf(fout,"%d, %5.2g ",p,a[i][n-1]);
```

The statement

```c
ffile=fopen("filename.txt","a");
```

will indicate that the pointer ffile is going to be used with an existing file named filename.txt to which we will append information.

For reading as well as appending data, we should run a check as to whether these files exist and can be opened by the program. The function fopen returns a value NULL (whose symbolic constant value of character zero is defined in stdio.h). We should therefore run a check which would generate an error message if the file cannot be opened. Otherwise, we would open the file, and read in the data. A construct of the type shown below may be used to handle this situation:

```c
FILE *din;

if(din== (FILE *) NULL)
{
    fprintf(stderr,"**** Data file could not be opened. \n");
    exit(1);
}
```

Here stderr is a stream similar to stdout except it will appear on standard output even
when the output is redirected to a file, The exit function will terminate the program with a return value of 1.

10 Addresses and Pointers

What sets aside C from many other programming languages is the concept of pointers. Pointers allow the user to represent complex data structures effectively and efficiently, to change values passed as arguments to functions, return values through variables used as arguments, to work with memory that has been allocated dynamically, and to efficiently access array elements.

10.1 Addresses

Declarations like int n; and double x; set aside memory slots to hold values to be assigned to n and x. Each memory slot is identified by a numerical address, much in the same way that houses have are identified by their street numbers. The address of the slot assigned to a variable can be obtained by applying the address operator & as in &x. The following program illustrates how to print the values and addresses of variables:

```
#include <stdio.h>

int main(void)
{
    int n = 7;
    double x = 3.14;
    printf("n = %d, x = %g\n", n, x); /* print values of n and x */
    printf("address of n = %p\n", (void *)&n); /* print address of n */
    printf("address of x = %p\n", (void *)&x); /* print address of x */
    return 0;
}
```

The output of the program is something like this:

```
n = 7, x = 3.14
address of n = 0xbffff9f4
address of x = 0xbffff9ec
```

Addresses printed for n and x will vary on from one invocation to the next—the compiler and the operating system decide the most appropriate place for storing a given object. A C programmer should be acutely aware of the four attributes attached to an object: name, type, value, address. Note that (void *), called a cast, is required by printf function when using the p conversion specifier.

---

1 The %p conversion in printf is designed to print an address in hexadecimal (i.e., base 16). Hexadecimal digits are shown as 0, 1,...,9, a, b, c, d, e, f. The 0x prefix signals that this is a hexadecimal representation.
10.2 Pointers

It is possible to store the address of a variable in another variable. The code fragment:

```c
double x;
double *p;
p = &x;
```

stores the address \&x of x in the variable p. The declaration `double *p;` indicates that p will hold the address, not the value, of a variable of the type `double`. A variable like p that is intended to hold an address is called a pointer variable or a pointer for short. This is because the value stored in such a variable points to a value stored elsewhere. When p holds the address of x, we say that p points to x. The following program illustrates the idea:

```c
#include <stdio.h>

int main(void)
{
    double x = 3.14;
double *p = &x; /* p holds address of x */
    printf("x = %g\n", x); /* print value of x */
    printf("address of x = %p\n", (void *)&x); /* print address of x */
    printf("address of x = %p\n", (void *)p); /* print address of x */
    return 0;
}
```

The last two `printf` functions both print the address of x in hexadecimal base. If p is a pointer variable, then value that p points to is obtained by *p. Should we put the line

```c
printf("x = %g\n", *p);
```

in the program above, it will print the value of x, that is 3.14. The asterisk is called the dereferencing operator.

10.3 Arrays and Pointer Arithmetic

The statement `double a[100];` declares an array of length 100 of the type `double`. This array can hold a sequence of 100 double-precision floating point numbers. The i\textsuperscript{th} element of the array is accessed by a[i]. The array index begins with zero, therefore the array elements are a[0], a[1], a[2], ..., a[99]. This is in variance with the common usage in mathematics where indices of vectors begin with one, therefore some care is necessary when implementing mathematical algorithms in a C program. Experience shows that this cultural difference is easy to overcome. The code fragment:

```c
sum = 0.0;
for (i=0; i<100; i++)
    sum += a[i];
```

computes the sum of the array’s 100 entries.

An array is stored in a contiguous area of the memory, that is, the addresses of the array entries form an arithmetic progression. The following program illustrates this:
```c
#include <stdio.h>
int main(void)
{
    double a[4];
    printf("address of a[0] = %p\n", (void *)&a[0]);
    printf("address of a[1] = %p\n", (void *)&a[1]);
    printf("address of a[2] = %p\n", (void *)&a[2]);
    printf("address of a[3] = %p\n", (void *)&a[3]);
    return 0;
}
```

The output from the above should look something like this:

```
address of a[0] = 0xbffff6d8
address of a[1] = 0xbffff6e0
address of a[2] = 0xbffff6e8
address of a[3] = 0xbffff6f0
```

Note that the addresses are incremented by the constant value 8 because on these machines, a variable of type `double` is 8 bytes wide. For an array `a`, the symbol `a` by itself, without the square brackets, is a pointer to the first element of the array. Had we included the line `printf("%p\n", (void *)a);` in the program above, it would have printed the same value as that printed for `&a[0]`.

Pointers to arrays have a well-defined arithmetic. If `p` is a pointer to a cell of an array, then `p + i` (same as `i + p`) for an integer `i` is a pointer to `i`th cell further down the array. In particular, since `a` is the address of the first cell of the array `a[]`, then `a + i` is the address of the `i`th cell. That is:

```
a + i = &a[i]
```

This implies the following important observation: by dereferencing the pointer `a + i` i.e., by using `*(a + i)`, we obtain the value stored in the `i`th cell. However note that the we can get the value of the `i`th cell also using `a[i]`. Therefore we have the identity:

```
*(a + i) ≡ a[i]
```

C provides the notation `a[i]` for the convenience of programmers. Internally it treats it as `*(a + i)`. The latter notation, although abstruse in the first sight, makes quite a bit of sense. It says: to get the value of `a[i]`, start at the address of the first cell, move forward `i` units, then look up the value stored in that address.

Neither the `*(a + i)` nor the `a[i]` notation have built-in error-checking mechanisms. It is your responsibility, as a programmer, to make sure that the address `a + i` lies within the array bounds. Trying to access a memory address outside the array bounds is a fatal error and almost certainly will crash the program with the dreaded `segmentation fault` message.

**Remark:**
The C99 standard allows variable-size array declarations both in one-dimensional and multi-dimensional arrays. For instance,
void f(int n)
{
    double a[n]; /* a variable-size array*/
    .....etc....
}

is a valid code in C99. In C89 the size of an array must be prescribed at compilation time.

Another remark:

Dynamically allocated arrays, a topic that we will discuss in ?? and subsequent sections provides a better way to set aside storage for arrays. Explicit declarations of arrays, as described up to this point, are recommended only for small and fixed-sized arrays. For instance, the elasticity tensor mentioned above is a good candidate for such usage. Large arrays, for instance the pixel data of a large image, are best handled through dynamically allocated arrays.

10.4 Function Pointers

In this section, we discuss the use of function pointers. As we shall see, function pointers are essential in applied mathematics and statistical applications, e.g. for numerical integration, function maximization, root-finding, simulation from a general density, and so on. In all these cases, we need to pass a function as an argument to the concerned routine. For instance, if we want to optimize a function fun over an interval given by \([a, b]\). Recall that in our C function newton(x0,eps,maxiter,&ifail) we got over the problem by defining fun() and defun() as external functions and using function declarations when our main program was compiled. Recall that we passed function as arguments to newton() in R by. We would like to pass functions as argument to a function like newton(). We also need to tell newton() that fun and defun are functions in one variable, respectively. One way to do that is to use what are called function pointers – a concept we discuss here, and illustrate via the simple example of newton().

So, what are function pointers? By definition, they are pointers, just like data pointers discussed earlier, but instead in this case, these pointers point to the address of a function. So what does all this mean? Well, one way to understand all this is to appreciate that a running program allocates space in a certain part of the main memory (RAM). Both the executable compiled code as well as the declared variables have space allocated in this memory, and these spaces are allocated by address. Thus, a function in the program code has also an address.

Similar to data pointers, we have to declare function pointers by first declaring a variable which details the pointer to a function. This can be somewhat complicated. Here is a declaration to a simple function pointer declaration:

\[
declare double (*function)(double);
\]

This declares function as a pointer to a function which will return a double. The * indicates the pointer to the function and the parentheses around the word (double) indicates that function is a function of a double-precision floating point argument. Note however, that there is an extra pair of parentheses around *function. This sets the precedence, because the () around the pointer to the function means that it is a pointer first, pointing to a function taking an argument double and returning a value as a double.
Note To streamline code, we can also make use of *typedef*. For example, we could define

```c
typedef double (*ptrfn)(double);
```

which would mean an identifier *ptrfun* synonymous for the variable type “pointer to a function returning a double.” Then, we could just declare *function* as above, by using

```c
ptrfn function;
```

and the effect of such a declaration would look cleaner, but be identical to the one above.

### 10.5 Example of the use of Function Pointers

Here we rewrite the functions *newton()*, *fun()*, and *defun()* we used in an earlier example using function pointers. The structure of these functions and the main program are the same as before; however, there are *two additional arguments* to the function *newton()* that are function pointers:

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

double newton(double (*)(double *),double (*)(double *),double ,double ,
              int , int *);

double fun(double *);

double defun(double *);

int main(void)
{
    double ans;
    double x0=0.2, eps=.1e-7;
    int maxiter=30, ifail;
    ifail=0;
    ans=newton(fun, defun, x0,eps,maxiter,&ifail);
    if (ifail==0)
        printf("The zero is = %14.6f\n",ans);
    else
        printf("**** Number of iterations exceeded.\n");
    return 0;
}

double newton(double (*fun)(double *),double (*defun)(double *),double x0,
              double eps, int maxiter, int *ifail)
{
    double x, xlast;
    int iter;
    printf("Start Value= %14.6f Stopping Rule=%15.8e\n",x0,eps);
    x=x0;
    for (iter=0; iter < maxiter; iter++){
        xlast=x;
        x-=fun(&x)/defun(&x);
        printf("Iteration No= %4d Iterate= %15.7f\n",iter,x);
        if(fabs(xlast-x)<eps)
            return(x);
    }
    *ifail=1;
    return;
}
double fun(double *x)
{
    return(*x*exp(-*x)-.2);
}

double defun(double *x)
{
    return((1.e0-*x)*exp(-*x));
}

11 Structures

A structure provides a means of grouping variables under a single name for easier handling and identification. Complex hierarchies can be created by nesting structures. Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

11.1 Declaring Structures

A structure is declared by using the keyword `struct` followed by an optional structure tag followed by the body of the structure. The variables or members of the structure are declared within the body. Here is an example of a structure that would be useful in representing the Cartesian coordinates of a point on a computer screen, that is, the pixel position.

```c
struct point {
    int x;
    int y;
};
```

The `struct` declaration is a user defined data type. Variables of type `point` may be declared in the same way as variables of a built in type are declared.

```c
struct point {
    int x;
    int y;
} upperLeft;
```

is analogous to

```c
float rate;
```

The structure tag provides a shorthand way of declaring structures:

```c
struct point {
    int x;
    int y;
};
```

```c
struct point left,right;
struct point origin;
```
However, the C language allows data types to be named using the keyword `typedef`. For example:

```c
typedef double Money;
typedef unsigned long int ulong;

Money paycheck;
    /* This declares paycheck to be of type Money, or double */
ulong IDnumber;
    /* This declared IDnumber to be of type ulong, or unsigned long */
```

User defined data types such as `struct` may also be named using `typedef`.

```c
typedef struct point {
    int x;
    int y;
} Dot;

Dot left, right;
    /* Declares left and right to be Dots, or structures of type point */
```

The examples in the next section will present the different ways of declaring structures.

### 11.2 Using Structures

The individual members of a structure can be accessed using ".", the member access operator. Given

```c
typedef struct point {
    int x;
    int y;
} Dot;

....

Dot location;

The x coordinate is `location.x`
The y coordinate is `location.y`

It is also possible to create hierarchies by nesting structures. A rectangle could be represented as follows.

```c
typedef struct rect {
    Dot upperLeft;
    Dot upperRight;
    Dot lowerLeft;
    Dot lowerRight;
} Box;

Box myRectangle;
```
Pointers to structures may also be declared and assigned. Given the above `typedef` of `Dot`, a pointer may be declared and initialized as follows.

```c
Dot location; /* Defines an object of type Dot */
Dot *lpt; /* Declares a pointer of type Dot */
....
lpt = &location;
/* Assigns the address of location to the pointer, pt */
```

A special member access operator `->` is used to access the members of a structure via a pointer.

- The x coordinate of location is given by `lpt->x`
- The y coordinate of location is given by `lpt->y`

Here is an example illustrating the use of structures.

```c
#include <stdio.h>

typedef struct point {
    int x;
    int y;
} DOT;

struct rect {
    DOT ul;
    DOT ur;
    DOT ll;
    DOT lr;
};

int main(void)
{
    DOT location; /* One way to declare structures */
    DOT *lpt;

    struct rect myRect;
    /* Second way to declare structures */

    struct circle {   /* Third way */
        DOT origin; /* Third way */
        int radius;
    } myCircle;

    location.x = 5;
    location.y = 4;

    lpt = &location;

    printf("Starting point %d %d\n", location.x, lpt->y);

    return 0;
}
```
11.3 Arrays of Structures

The data structures needed to solve some problems are best represented as an array of structures. Consider, for instance, the problem of counting the occurrence of each letter in a string. One possible approach would be to declare two separate arrays. One would hold the letters to compare against. The second would hold a count of the occurrences of each letter. Their declarations would be:

```c
#define NUMLETTERS 26
....
char letter[NUMLETTERS];
int count[NUMLETTERS];
```

This approach would work, but each letter and its count are closely related. It would be convenient and less error prone to keep this closely linked data together in a structure.

```c
#define NUMLETTERS 26
....
/* Declares a structure with a structure tag lettertype */
struct lettertype {
  char letter;
  int count;
};

typedef struct lettertype Letter;
  /* Declares the structure to be a new data type, Letter */
Letter alphabet[NUMLETTERS];
  /* Defines an array, alphabet, of type letter */
```

Note that the above declarations could also be written as follows.

```c
typedef struct lettertype {
  char letter;
  int count;
} Letter;

Letter alphabet[NUMLETTERS];
```

11.4 Sample Program

Here is a program using the `Letter` data structure to count letters in a string.

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

#define NUMLETTERS 26

typedef struct lettertype {
  char letter;
  int count;
} Letter;
```
int main(void)
{
    /* Variable Declarations */
    int i, j;
    int length;
    Letter alphabet[] = {
        {'a',0},
        {'b',0},
        {'c',0},
        {'d',0},
        {'e',0},
        {'f',0},
        {'g',0},
        {'h',0},
        {'i',0},
        {'j',0},
        {'k',0},
        {'l',0},
        {'m',0},
        {'n',0},
        {'o',0},
        {'p',0},
        {'q',0},
        {'r',0},
        {'s',0},
        {'t',0},
        {'u',0},
        {'v',0},
        {'w',0},
        {'x',0},
        {'y',0},
        {'z',0}
    };
    char sampleString[] =
        "now is the time for all good men to come to the aid of their country";

    /* Find length of string */
    length = strlen(sampleString);

    for (i = 0; i < length; i++) {
        for (j = 0; j < NUMLETTERS; j++) {
            if (sampleString[i] == alphabet[j].letter) {
                alphabet[j].count++;
            }
        }
    }

    for (j = 0; j < NUMLETTERS; j++)
        printf("%c found %d times\n",alphabet[j].letter,alphabet[j].count);
    return 0;
}