Some C Essentials

1 A Word About C

This resource aims to summarize the main features of the C language. It is intended to be adequate for the purpose of programming in ECE251. With care and experimentation, it can be used as an introduction to the main features of the language. If you are a C novice, it’s well worth utilizing another reference source as well, such as Programming in C Language by Kernighan and Ritchie.

2 Elements of a C Program

2.1 Keywords

C has a number of keywords whose use is defined. A programmer cannot use a keyword for any other purpose, for example, as a data name. Keywords are summarized in Tables 1–3.

2.2 Program Features and Layout

Simply speaking, a C program is made up of the following:

Declarations

All variables in C must be declared before they can be applied, giving as a minimum variable name and its data type. A declaration is terminated with a semicolon. In simple programs, declarations appear as one of the first things in the program. They can also occur within the program, with significance attached to the location of the declaration.

For example:

    float exchange_rate;
    int new_value;
**Table 1: C keywords associated with data type and structure definition.**

<table>
<thead>
<tr>
<th>Word</th>
<th>Summary Meaning</th>
<th>Word</th>
<th>Summary Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>A single character, usually 8 bit</td>
<td>signed</td>
<td>A qualifier applied to char or int (default for char and int is signed)</td>
</tr>
<tr>
<td>const</td>
<td>Data that will not be modified</td>
<td>sizeof</td>
<td>Returns the size in bytes of a specified item, which may be variable, expression, or array</td>
</tr>
<tr>
<td>double</td>
<td>A “double precision” floating-point number</td>
<td>struct</td>
<td>Allows definition of a data structure</td>
</tr>
<tr>
<td>enum</td>
<td>Defines variables that can only take certain integer values</td>
<td>typedef</td>
<td>Creates new name for existing data type</td>
</tr>
<tr>
<td>float</td>
<td>A “single precision” floating-point number</td>
<td>union</td>
<td>A memory block shared by two or more variables, of any data type</td>
</tr>
<tr>
<td>int</td>
<td>An integer value</td>
<td>unsigned</td>
<td>A qualifier applied to char or int (default for char and int is signed)</td>
</tr>
<tr>
<td>long</td>
<td>An extended integer value; if used alone, integer is implied</td>
<td>volatile</td>
<td>A variable which can be changed by factors other than the program code</td>
</tr>
<tr>
<td>short</td>
<td>A short integer value; if used alone, integer is implied</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: C keywords associated with program flow.**

<table>
<thead>
<tr>
<th>Word</th>
<th>Summary Meaning</th>
<th>Word</th>
<th>Summary Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>Causes exit from a loop</td>
<td>for</td>
<td>Defines a repeated loop—loop is executed as long as condition associated with for remains true</td>
</tr>
<tr>
<td>case</td>
<td>Identifies options for selection within a switch expression</td>
<td>goto</td>
<td>Program execution moves to labeled statement</td>
</tr>
<tr>
<td>continue</td>
<td>Allows a program to skip to the end of a for, while, or do statement</td>
<td>if</td>
<td>Starts conditional statement; if condition is true, associated statement or code block is executed</td>
</tr>
<tr>
<td>default</td>
<td>Identifies default option in a switch expression, if no matches found</td>
<td>return</td>
<td>Returns program execution to calling routine, causing also return of any data value specified by function</td>
</tr>
<tr>
<td>do</td>
<td>Used with while to create loop, in which statement or code block following do is repeated as long as while condition is true</td>
<td>switch</td>
<td>Used with case to allow selection of a number of alternatives; switch has an associated expression which is tested against a number of case options</td>
</tr>
<tr>
<td>else</td>
<td>Used with if, and precedes alternative statement or code block to be executed if if condition is not true</td>
<td>while</td>
<td>Defines a repeated loop—loop is executed as long as condition associated with while remains true</td>
</tr>
</tbody>
</table>
declare a variable called \texttt{exchange\_rate} as a floating-point number, and another variable called \texttt{new\_value} as an integer. The data types are keywords seen in the preceding tables.

\textit{Statements}

Statements are where the action of the program takes place. They perform mathematical or logical operations and establish program flow. Every statement which is not a block (see below) ends with a semicolon. Statements are executed in the sequence they appear in the program, except where program branches take place.

For example, this line is a statement:

\begin{verbatim}
    counter = counter + 1;
\end{verbatim}

\textit{Space and layout}

There is not a strict layout format to which C programs must adhere. The way the program is laid out and the use of space are both used to enhance clarity. Blank lines and indents in lines, for example, are ignored by the compiler, but used by the programmer to optimize the program layout.

As an example, the program that the mbed compiler always starts up with, shown as Program Example 2.1, \textit{could} be written as shown here. It wouldn’t be easy to read, however. It’s the semicolons at the end of each statement, and the brackets, which in reality define much of the program structure.

\begin{verbatim}
    #include "mbed.h"
    DigitalOut myled(LED1); int main() {while(1) {myled = 1; wait(0.2); myled = 0;
    wait(0.2);}}
\end{verbatim}

\textit{Comments}

Two ways of commenting are used. One is to place the comment between the markers /* and */. This is useful for a block of text information running over several lines.
Alternatively, when two forward slash symbols (//) are used, the compiler ignores any text which follows on that line only, which can then be used for comment.

For example:

/*A program which flashes mbed LED1 on and off,
Demonstrating use of digital output and wait functions. */
#include "mbed.h"  //include the med header file as part of this program

2.3 Compiler Directives

Compiler directives are messages to the compiler and do not directly lead to program code. Compiler directives all start with a hash, #. Two examples follow.

#include

The #include directive directly inserts another file into the file that invokes the directive. This provides a feature for combining a number of files as if they were one large file. Angled brackets (<>) are used to enclose files held in a directory different from the current working directory, hence often for library files not written by the current author. Quotation marks are used to contain a file located within the current working directory, hence often user defined.

For example:

#include "mbed.h"

#define

The #define directive allows use of names for specific constants. For example, to use the number $\pi = 3.141592$ in the program, we could create a #define for the name “PI” and assign that number to it, as shown:

#define PI 3.141592

The name “PI” is then used in the code whenever the number is needed. When compiling, the compiler replaces the name in the #define with the value that has been specified.
3 Variables and Data

3.1 Declaring, Naming, and Initializing

Variables must be named, and their data type defined, before they can be used in a program. Keywords from Table 1 are used for this. For example:

```c
int MyVariable;
```

defines “MyVariable” as a data type int (integer).

It is possible to initialize the variable at the same time as declaration, for example:

```c
int MyVariable = 25;
```

initializes `MyVariable` and sets it to an initial value of 25.

It is possible to give variables meaningful names, while still avoiding excessive length, for example, “Height”, “InputFile”, “Area”. Variable names must start with a letter or underscore; no other punctuation marks are allowed. Variable names are case sensitive.

3.2 Data Types

When a data declaration is made, the compiler reserves for it a section of memory, whose size depends on the type invoked. Examples of the link between data type, number range, and memory size are shown in Table 4. It is interesting to compare these with information on number types given in Appendix A. Note that the actual memory size applied to data types can vary between compilers. A full listing for the mbed compiler can be found in Ref. [3].

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Length (bytes)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Character</td>
<td>1</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>Character</td>
<td>1</td>
<td>-128 to +127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>Character</td>
<td>1</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>Integer</td>
<td>2</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer</td>
<td>2</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>4</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>long</td>
<td>Integer</td>
<td>4</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Integer</td>
<td>4</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>float</td>
<td>Floating point</td>
<td></td>
<td>1.17549435 × 10⁻³⁸ to 3.40282347 × 10⁺³⁸</td>
</tr>
<tr>
<td>double</td>
<td>Floating point, double precision</td>
<td></td>
<td>2.22507385850720138 × 10⁻⁶³⁰⁸ to 1.79769313486231571 × 10⁺⁶³⁰⁸</td>
</tr>
</tbody>
</table>
3.3 Working With Data

In C we can work with numbers in binary, fixed or floating-point decimal, or hexadecimal format, depending on what is most convenient, and what number type and range is required. For time critical applications it is important to remember that floating-point calculations can take much longer than fixed point. In general, it’s easiest to work in decimal, but if a variable represents a register bit field or a port address, then it’s usually more appropriate to manipulate the data in hexadecimal. When writing numbers in a program, the default radix (number base) for integers is decimal, with no leading 0 (zero). Octal numbers are identified with a leading 0. Hexadecimal numbers are prefixed with 0x.

For example, if a variable MyVariable is of type char we can perform the following examples to assign a number to that variable:

```c
MyVariable = 15; // a decimal example
MyVariable = 0x0E; // a hexadecimal example
```

The value for both is the same.

3.4 Changing Data Type: Casting

Data can be changed from one data type to another by type casting. This is done using the cast operator, seen at the bottom of Table 5. For example, in the line that follows, size has been declared as char, and sum as int. However, their division will not yield an integer. Therefore the result is cast to floating point.

```c
mean=(float)sum/size;
```

Some type conversions may be done by the compiler implicitly. It is, however, better programming practice not to depend on this, but to do it explicitly.

4 Functions

A function is a section of code which can be called from another part of the program. So if a particular piece of code is to be used or duplicated many times, we can write it once as a function and then call that function whenever the specific operation is required. Using functions saves coding time and improves readability by making the code neater.

Data can be passed to functions and returned from them. Such data elements, called arguments, must be of a type which is declared in advance. Only one return variable is allowed, whose type must also be declared. The data passed to the variable is a copy of the original. Therefore, the function does not itself modify the value of the variable named. The impact of the function should thus be predictable and controlled.
A function is defined in a program by a block of code having particular characteristics. Its first line forms the function header, with the format:

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2 variable_name_2, ...)
```

An example is shown in Fig. 1. The return type is given first. In this example, the keyword `float` is used. After the function name, in brackets, one or more data types may be listed, which identify the arguments which must be passed to the function. In this case, two arguments are sent, one of type `char` and one of type `float`. Following the function header, a pair of braces encloses the code which makes up the function itself. This could be anything from a single line to many pages. The final statement of the function may be a `return`, which will specify the value returned to the calling program. This is not essential if no return value is required.

### 4.1 The main Function

The core code of any C program is contained within its “main” function. Other functions may be written outside `main()` and called from within it. Program execution starts at the beginning of `main()`. It must follow the structure just described. However, as `main()` contains the central program, one expects to send nothing to it, nor receive anything from it. Therefore usual patterns for `main()` are:

```
void main (void){
void main (){}
int main (){}
```

The keyword `void` indicates that no data is specified. The mbed `main()` function applies the third option, as in C++ `int` is the return type specified for `main()`.
4.2 Function Prototypes

Just like variables, functions must be declared at the start of a program, before the main function. The declaration statements for functions are called prototypes. Each function in the code must have an associated prototype for it to run. The format is the same as for the function header.

For example, the following function prototype applies to the function header seen above:

```c
float conversion(char currency, float number_of_pounds)
```

This describes a function that takes inputs of a character value for the selected currency and a floating-point (decimal) value for the number of pounds to be converted. The function returns the decimal monetary value in the specified currency.

4.3 Function Definitions

The actual function code is called the function definition. For example:

```c
float conversion(char currency, float number_of_pounds) {
    float exchange_rate;
    switch(currency) {
        case 'U': exchange_rate = 1.50; // US Dollars
            break;
        case 'E': exchange_rate = 1.12 ); // Euros
            break;
        case 'Y': exchange_rate = 135.4); // Japan Yen
            break;
        default: exchange_rate = 1);
    }
    exchange_value=number_of_pounds*exchange_rate;
    return(exchange_value);
}
```

This function can be called any number of times from within the main C program, or from another function, for example, in this statement:

```c
ten_pounds_in_yen=conversion('Y',10.45);
```

The structure of this function is explained in Section 6.2.

4.4 Using the static Storage Class With Functions

The static data type is useful for defining variables within functions, where the data inside the function must be remembered between function calls. For example, if a function within a real time system is used to calculate a digital filter output, the function should
always remember its previous data values. In this case, data values inside the function should be defined as static, for example, as shown below.

```c
float movingaveragefilter(float data_in) {
    static float data_array[10]; // define static float data array
    for (int i=8;i>=0;i--) {
        data_array[i+1]=data_array[i]; // shift each data value along
    } // (the oldest data value is discarded)
    data_array[0]=data_in; // place new data at index 0
    float sum=0;
    for (int i=0;i<=9;i++) {
        sum=sum+data_array[i]; // calculate sum of data array
    }
    return sum/10; // return average value of array
}
```

5 Operators

C has a wide set of operators, shown in Table 5. The symbols used are familiar, but their application is not always the same as in conventional algebra. For example, a single “equals” symbol, “=”, is used to assign a value to a variable. A double equals sign, “==”, is used to represent the conventional “equal to.”

Operators have a certain order of precedence, shown in the table. The compiler applies this order when it evaluates a statement. If more than one operator at the same level of precedence occurs in a statement, then those operators are evaluated in turn, either left to right or right to left, as shown in the table. For example, the line

```c
counter = counter + 1;
```

contains two operators. Table 5 shows that the addition operator has precedence level 4, while all assign operators have precedence 14. The addition is therefore evaluated first, followed by the assign. In words we could say, the new value of the variable `counter` has been assigned the previous value of `counter`, plus one.

6 Flow Control: Conditional Branching

Flow Control is the title which covers the different forms of branching and looping available in C. As branching and looping can lead to programming errors, C provides clear structures to improve programming reliability.

6.1 If and Else

If statements always start with use of the `if` keyword, followed by a logical condition. If the condition is satisfied, then the code block which follows is executed. If the condition is
<table>
<thead>
<tr>
<th>Precedence and Order</th>
<th>Operation</th>
<th>Symbol</th>
<th>Precedence and Order</th>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parentheses and Array Access Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1, L to R</td>
<td>Function calls</td>
<td>( )</td>
<td>1, L to R</td>
<td>Point at member</td>
<td>➞ ➞ Y</td>
</tr>
<tr>
<td>1, L to R</td>
<td>Subscript</td>
<td>[ ]</td>
<td>1, L to R</td>
<td>Select member</td>
<td>X.Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arithmetic Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4, L to R</td>
<td>Add</td>
<td>X+Y</td>
<td>3, L to R</td>
<td>Multiply</td>
<td>*Y</td>
</tr>
<tr>
<td>4, L to R</td>
<td>Subtract</td>
<td>X−Y</td>
<td>3, L to R</td>
<td>Divide</td>
<td>/Y</td>
</tr>
<tr>
<td>2, R to L</td>
<td>Unary plus</td>
<td>+X</td>
<td>3, L to R</td>
<td>Modulus</td>
<td>%</td>
</tr>
<tr>
<td>2, R to L</td>
<td>Unary minus</td>
<td>−X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relational Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6, L to R</td>
<td>Greater than</td>
<td>X&gt;Y</td>
<td>6, L to R</td>
<td>Less than or equal to</td>
<td>&lt;=Y</td>
</tr>
<tr>
<td>6, L to R</td>
<td>Greater than or equal to</td>
<td>X&gt;=Y</td>
<td>7, L to R</td>
<td>Equal to</td>
<td>== =Y</td>
</tr>
<tr>
<td>6, L to R</td>
<td>Less than</td>
<td>X&lt;Y</td>
<td>7, L to R</td>
<td>Not equal to</td>
<td>!=Y</td>
</tr>
<tr>
<td></td>
<td>Logical Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11, L to R</td>
<td>AND (1 if both X and Y are not 0)</td>
<td>X&amp;&amp;Y</td>
<td>2, R to L</td>
<td>NOT (1 if X=0)</td>
<td>!X</td>
</tr>
<tr>
<td>12, L to R</td>
<td>OR (1 if either X or Y are not 0)</td>
<td>X</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bitwise Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, L to R</td>
<td>Bitwise AND</td>
<td>X&amp;Y</td>
<td>2, L to R</td>
<td>Ones complement</td>
<td>~X</td>
</tr>
<tr>
<td>10, L to R</td>
<td>Bitwise OR</td>
<td>X</td>
<td>Y</td>
<td>5, L to R</td>
<td>Right shift. X is shifted</td>
</tr>
<tr>
<td>9, L to R</td>
<td>Bitwise XOR</td>
<td>XY</td>
<td>5, L to R</td>
<td>Left shift. X is shifted</td>
<td>&lt; &lt; Y</td>
</tr>
<tr>
<td></td>
<td>Assignment Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14, R to L</td>
<td>Assignment</td>
<td>X=Y</td>
<td>14, R to L</td>
<td>Bitwise AND assign</td>
<td>X&amp;=Y</td>
</tr>
<tr>
<td>14, R to L</td>
<td>Add assign</td>
<td>X+Y=Y</td>
<td>14, R to L</td>
<td>Bitwise inclusive OR assign</td>
<td>X</td>
</tr>
<tr>
<td>14, R to L</td>
<td>Subtract assign</td>
<td>X−Y=Y</td>
<td>14, R to L</td>
<td>Bitwise exclusive OR assign</td>
<td>X′=Y</td>
</tr>
<tr>
<td>14, R to L</td>
<td>Multiply assign</td>
<td>X*=Y</td>
<td>14, R to L</td>
<td>Right shift assign</td>
<td>X&gt;&gt;&gt;=Y</td>
</tr>
<tr>
<td>14, R to L</td>
<td>Divide assign</td>
<td>X/=Y</td>
<td>14, R to L</td>
<td>Left shift assign</td>
<td>X&lt;&lt;&lt;=Y</td>
</tr>
<tr>
<td>14, R to L</td>
<td>Remainder assign</td>
<td>X%Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Increment and Decrement Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2, R to L</td>
<td>Preincrement</td>
<td>++X</td>
<td>2, R to L</td>
<td>Postincrement</td>
<td>X++</td>
</tr>
<tr>
<td>2, R to L</td>
<td>Predecrement</td>
<td>−−X</td>
<td>2, R to L</td>
<td>Postdecrement</td>
<td>X−−</td>
</tr>
<tr>
<td></td>
<td>Conditional Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13, R to L</td>
<td>Evaluate either X (if Z≠0) or Y (if Z=0)</td>
<td>Z?X:Y</td>
<td>15, L to R</td>
<td>Evaluate X first, followed by Y</td>
<td>X,Y</td>
</tr>
<tr>
<td></td>
<td>“Data Interpretation” Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2, R to L</td>
<td>The object or function pointed to by X</td>
<td>*X</td>
<td>2, R to L</td>
<td>The address of X</td>
<td>&amp;X</td>
</tr>
<tr>
<td>2, R to L</td>
<td>Cast—the value of X, with (scalar) type specified</td>
<td>(type) X</td>
<td>2, R to L</td>
<td>The size of X, in bytes</td>
<td>sizeof X</td>
</tr>
</tbody>
</table>
not satisfied, then the code is not executed. There may or may not also be following \texttt{else} or \texttt{else if} statements.

\textbf{Syntax:}

\begin{verbatim}
if (Condition1){
   \hspace{1em}C statements here
}else if (Condition2){
   \hspace{1em}C statements here
}else if (Condition3){
   \hspace{1em}C statements here
}else{
   \hspace{1em}C statements here
}
\end{verbatim}

The \texttt{if} and \texttt{else} statements are evaluated in sequence, i.e.,

\begin{itemize}
\item \texttt{else if} statements are only evaluated if all previous \texttt{if} and \texttt{else} conditions have failed;
\item \texttt{else} statements are only executed if all previous conditions have failed.
\end{itemize}

For example, in the above example, the \texttt{else if (Condition2)} will only be executed if Condition1 has failed.

Example:

\begin{verbatim}
if (data > 10){
   data += 5; \hspace{1em} //If we reach this point, data must be > 10
}else if(data > 5){ \hspace{1em} //If we reach this point, data must be <= 10
   data -= 3;
}else{ \hspace{1em} //If we reach this point, data must be <= 5
   nVal = 0;
}
\end{verbatim}

\subsection*{6.2 Switch Statements and Using break}

The \texttt{switch} statement allows a selection to be made of one out of several actions, based on the value of a variable or expression given in the statement. An example of this structure has already appeared, in the example function in Section 4.3. The structure uses no less than four C keywords. Selection is made from a list of \texttt{case} statements, each with an associated label—note that a colon following a text word defines it as a label. If the label equals the \texttt{switch} expression then the action associated with that \texttt{case} is executed. The \texttt{default} action (which is optional) occurs if none of the \texttt{case} statements are satisfied. The \texttt{break} keyword, which terminates each \texttt{case} condition, can be used to exit from any loop. It causes program execution to continue after the \texttt{switch} code block.
7 Flow Control: Program Loops

7.1 while Loops

A while loop is a simple mechanism for repeating a section of code, until a certain condition is satisfied. The condition is stated in brackets after the word while, with the conditional code block following. For example:

```c
i=1
while (i<10) {
    ... C statements here
    i++ //increment i
}
```

Here the value of `i` is defined outside the loop; it is then updated within the loop. Eventually, `i` increments to 10, at which point the loop terminates. The condition associated with the while statement is evaluated at the start of each loop iteration; the loop then only runs if the condition is found to be true.

7.2 for Loops

The for loop allows a different form of looping, in that the dependent variable is updated automatically every time the loop is repeated. It defines an initialized variable, a condition for looping, and an update statement. Note that the update takes place at the end of each loop iteration. If the updated variable is no longer true for the loop condition, the loop stops and program flow continues. For example:

```c
for(j=0; j<10; j++) {
    ... C statements here
}
```

Here the initial condition is `j=0` and the update value is `j++`, i.e., `j` is incremented. This means that `j` increments with each loop. When `j` becomes 10 (i.e., after 10 loops), the condition `j<10` is no longer satisfied, so the loop does not continue any further.

7.3 Infinite Loops

We often require a program to loop forever, particularly in a super-loop program structure. An infinite loop can be implemented by either of the following loops:

```c
while(1) {
    ... continuously called C statements here
}
```

Or
for(;;) {
    ... continuously called C statements here
} 

7.4 Exiting Loops With break

The `break` keyword can also be used to exit from a `for` or `while` loop, at any time within the loop. For example:

```c
while(i>5) {
    ... C statements here
    if (fred == 1)
        break;
    ... C statements here
} //end of while
//execution continues here loop completion, or on break
```

8 Derived Data Types

In addition to the fundamental data types, there are further data types which can be derived from them. Example types that we use are described in this section.

8.1 Arrays and Strings

An array is a set of data elements, each of which has the same type. Any data type can be used. Array elements are stored in consecutive memory locations. An array is declared with its name and the data type of its elements; it is recognized by the use of the square brackets which follow the name. The number of elements and their value can also be specified. For example, the declaration

```c
unsigned char message1[8];
```

defines an array called `message1`, containing eight characters. Alternatively, it can be left to the compiler to deduce the array length, as seen in the two examples here:

```c
char item1[] = "Apple";
int nTemp[] = {5,15,20,25};
```

In each of these the array is initialized as it is declared.

Elements within an array can be accessed with an index, starting with value 0. Therefore, for the first example above, `message1[0]` selects the first element and `message1[7]` the last. An access to `message[8]` would be outside the boundary of the array and would give invalid data. The index can be replaced by any variable which represents the required value.
Importantly, the name of an array is set equal to the address of the initial element. Therefore, when an array name is passed in a function, what is passed is this address.

A string is a special array of type `char` that is ended by the NULL (\0) character. The null character allows code to search for the end of a string. The size of the string array must therefore be one byte greater than the string itself to contain this character. For example, a 20 character string could be declared:

```c
char MyString[21]; // 20 characters plus null
```

### 8.2 Pointers

Instead of specifying a variable by name, we can specify its address. In C terminology, such an address is called a pointer. A pointer can be loaded with the address of a variable by using the unary operator “&”, like this:

```c
my_pointer = &fred;
```

This loads the variable `my_pointer` with the *address* of the variable `fred`; `my_pointer` is then said to *point* to `fred`.

Doing things the other way round, the value of the variable pointed to by a pointer can be specified by prefixing the pointer with the “*” operator. For example, `*my_pointer` can be read as “the value pointed to by `my_pointer`”. The * operator, used in this way, is sometimes called the dereferencing or indirection operator. The indirect value of a pointer, for example, `*my_pointer`, can be used in an expression just like any other variable.

A pointer is declared by the data type it points to. Thus,

```c
int *my_pointer;
```

indicates that `my_pointer` points to a variable of type `int`.

We can also use pointers with arrays, because an array is really just a number of data values stored at consecutive memory locations. So if the following is defined:

```c
int dataarray[]={3,4,6,2,8,9,1,4,6}; // define an array of arbitrary values
int *ptr; // define a pointer
ptr = &dataarray[0]; // assign pointer to the address of
                   // the first element of the data array
```

Given the previous declarations, the following statements will therefore be true:

```c
*ptr == 3; // the first element of the array pointed to
*(ptr+1) == 4; // the second element of the array pointed to
*(ptr+2) == 6; // the third element of the array pointed to
```
So array searching can be done by moving the pointer value to the correct array offset. Pointers are required for a number of reasons, but one simple reason is because the C standard does not allow us to pass arrays of data to and from functions, so we must use pointers instead to get around this.

### 8.3 Structures and Unions

Structures and unions are both sets of related variables, defined through the C keywords `struct` and `union`. In a way they are like arrays, but in both cases they can be of data elements of different types.

Structure elements, called members, are arranged sequentially, with the members occupying successive locations in memory. A structure is declared by invoking the `struct` keyword, followed by an optional name (called the structure tag), followed by a list of the structure members, each of these itself forming a declaration. For example:

```c
define resistor {int val; char pow; char tol;};
```

declares a structure with tag `resistor`, which holds the value (`val`), power rating (`pow`), and tolerance (`tol`) of a resistor. The tag may come before or after the braces holding the list of structure members.

Structure elements are identified by specifying the name of the variable and the name of the member, separated by a full stop (period). Therefore, `resistor.val` identifies the first member of the example structure above.

Like a structure, a union can hold different types of data. Unlike the structure, union elements all begin at the same address. Hence the union can represent only one of its members at any one time, and the size of the union is the size of the largest element. It is up to the programmer to track which type is currently stored! Unions are declared in a format similar to that of structures.

Unions, structures, and arrays can occur within each other.

### 9 C Libraries and Standard Functions

#### 9.1 Header Files

All but the simplest of C programs are made up of more than one file. Generally, there are many files which are combined together in the process of compiling, for example, original source files combining with standard library files. To aid this process, a key section of any library file is detached and created as a separate header file.

Header file names end in `.h`; the file typically includes declarations of constants and function prototypes and links on to other library files. The function definitions themselves
stay in the associated .c or .cpp files. To use the features of the header file and the file(s) it invokes, it must be included within any program accessing it, using #include. We see mbed.h being included in almost every program in the book. Note also that the .c file where the function declarations appear must also include the header file.

9.2 Libraries and the C Standard Library

Because C is a simple language, much of its functionality derives from standard functions and macros which are available in the libraries accompanying any compiler. A C library is a set of precompiled functions which can be linked into the application. These may be supplied with a compiler, available in-company, or be public domain. Notably there is a Standard Library, defined in the C ANSI standard. There are a number of standard header files used for different groups of functions within the standard library. For example, the <math.h> header file is used for a range of mathematical functions (including all trigonometric functions), while <stdio.h> contains the standard input and output functions, including, for example, the printf( ) function.

9.3 Using printf

This versatile function provides formatted output, typically for sending display data to a PC screen. Text, data, formatting, and control formatting can be specified. Only summary information is provided here, a full statement can be found in [1]. Examples below are taken from chapters in this book. In each case the function appears in the form pc.printf( ), indicating that printf( ) is being used as a member function of a C++ class pc created in the example program. This doesn’t affect the format applied.

Simple text messages

    pc.printf("ADC Data Values\n\r"); \send an opening text message

This prints the text string “ADC Data Values...” to screen and uses control characters \n and \r to force a new line and carriage return respectively.

Data messages

    pc.printf("%1.3f",ADCdata);

This prints the value of the float variable ADCdata. A conversion specifier, initiated by the % character, defines the format. Within this the “f” specifies floating point, and the .3 causes output to three decimal places.

    pc.printf("%1.3f \n\r",ADCdata); \send the data to the terminal

As above, but includes \n and \r to force a new line and carriage return.
**Combination of text and data**

```c
pc.printf("random number is \%i\n\r", r_delay);
```

This prints a text message, followed by the value of `int` variable (indicated by the “i” specifier) `r_delay`.

```c
pc.printf("Time taken was \%f seconds\n", t.read()); //print timed value to pc
```

This prints a text message, followed by the return value of function `t.read()`, which is of type `float`.

This Tutorial is highly leveraged from Appendix B of

**Fast and Effective Embedded Systems Design** by Rob Toulson and Tim Wilmshurst.

This text is an excellent tool for learning about a different development environment: ARM mbed.