Achieve Target 1

The following program is a simple program on array data adding (insertion) and getting (retrieval). You have studied a similar program in the previous Perform Exercise.

```c
#include <stdio.h>

void main() {
    float anArray[5];
    float sum = 0;
    int i;
    printf("Enter %d numbers:\n", 5);
    for (i=0; i<5; i++)
        scanf("%f", &anArray[i]);
    for (i=0; i<5; i++)
        sum = sum + anArray[i];
    printf("The sum is %f\n", sum);
}
```

The requirements of a program often change and so making modifications to programs is inevitable. For example, we may now want the array size to hold 10 numbers. To achieve this, we will need to make changes to 4 different places (the blue bolded) in the above program. The language C offers a simpler solution.

```c
#include <stdio.h>

#define SIZE 5

void main() {
    float anArray[SIZE];
    float sum = 0;
    int i;
    printf("Enter %d numbers:\n", SIZE);
    for (i=0; i<SIZE; i++)
        scanf("%f", &anArray[i]);
    for (i=0; i<SIZE; i++)
        sum = sum + anArray[i];
    printf("The sum is %f\n", sum);
}
```

The above program is equivalent of the previous one, but it makes use of the `#define` feature in C that allows the definition of a text label. The program defines the label "SIZE" to equal to "5". With rare exceptions, all occurrences of `SIZE` in the program will be replaced by `5`. The advantage of this
feature is that changing the "5" to "10" will cause the change be made to all the "SIZE" labels in the program. Learn it.

Achieve Target 1 Cont

Study the following program that uses a char array.

```c
#include <stdio.h>
void main() {
    char charArray[256];
    int length;
    int i;
    printf("Enter a line of text (press Enter at the end):
" );
    for (i=0; i<256; i++) {
        scanf("%c", &charArray[i]);
        if (charArray[i] == '\n') {
            length = i;
            break;
        }
    }
    printf("The text entered is:
" );
    for (i=0; i<length; i++)
        printf("%c", charArray[i]);
    printf("\n");
}
```

Modify the above program so that it takes advantage of the #define feature of C. Work on the program to make it happen.

Note that the #define line is not a C statement and therefore no semicolon is required. In fact, adding a semicolon there will cause syntax errors. For example, if we had defined the following.

```
#define SIZE 5;
```

It would cause a syntax error in the following statement.

```c
for (i=0; i<SIZE; i++)
    scanf("%f", &anArray[i]);
```

The above statement would be processed to become the following one.

```c
for (i=0; i<5; i++)
    scanf("%f", &anArray[i]);
```

The line replaces the label SIZE with whatever text following it. Therefore the label "SIZE" would be mapped to "5;" and it would cause a syntax error with the for statement. Remember it.

The #define line is called a pre-processor directive. The #include line is another pre-processor directive. Remember it.
Achieve Target 2

The following program detects how many punctuation characters are there in the input. Only 5 punctuation types are included.

```c
#include <stdio.h>

#define PUNCTSIZE 5

void main() {
    char punctArray[PUNCTSIZE] = {'.', ',', ';', '?', '!'};
    char data;
    int i;
    int numPunct = 0;

    printf("Enter a line of text (Enter to terminate):\n");

    while (1) {
        scanf("%c", &data);
        if (data == '\n')
            break;
        for (i=0; i<PUNCTSIZE; i++) {
            if (data == punctArray[i]) {
                numPunct++;
                break;
            }
        }
    }
    printf("Number of punctuations %d\n", numPunct);
}
```

Enter a line of text (Enter to terminate):
C is a programming language; I like it.
Number of punctuations 2

There is no reason why we cannot have more than one array in a program. Add code to the above program so that all entered characters are also stored in a separate char array. You will need to do the following.

1. Declare a char array with at least 256 array elements.
2. Add code that store all entered characters into the array. Remember to keep record of how many characters have already been stored in the array.
3. At the end, add code to print all the characters stored in the char array.

Work on the program to make it happen.
Achieve Target 2 Cont

Study the following program which is a minor modification to the solution of the previous exercise.

```c
#include <stdio.h>
#define CHARSIZE 256
#define PUNCTSIZE 5

void main() {
    char punctArray[PUNCTSIZE] = {'.', ',', ';', '?', '!'};
    char charArray[CHARSIZE];
    char data;
    int i;
    int length = 0;

    printf("Enter a line of text (Enter to terminate):
");
    while (1) {
        scanf("%c", &data);
        if (data == '\n')
            break;
        for (i=0; i<PUNCTSIZE; i++) {
            if (data == punctArray[i])
                break;
        }
        if (i==PUNCTSIZE)
            charArray[length++] = data;
    }
    for (i=0; i<length; i++)
        printf("%c", charArray[i]);
    printf("\n");
}
```

This program does not store all entered characters. It discards all the five punctuation types and stores all other characters.

The `if` statement in blue is the key of the logic. Note that the `for` loop (with a red circle) would continue running until `i` becomes `PUNCTSIZE`, unless the `break` statement is executed and the loop terminates pre-maturely. The former case would happen if the character is not one of the punctuation types. So if `i` is equal to `PUNCTSIZE`, then the `for` loop must have been running until the end and the character is not one of the punctuation types. In such case we store the character into the array `charArray`. Learn it.

Modify the above program so that the punctuation characters are stored in another `char` array. You will need to do the following.

1. Declare a char array with at least 256 array elements.
2. Add code that stores the punctuation characters into the new char array. You will need another variable for keeping record of how many punctuation characters have been stored.

Work on the program to make it happen.
Achieve Target 3

The following program uses pre-defined char array for matching "password".

```
#include <stdio.h>
#define PASSWORDSIZE 6
#define ARRSIZE 256

void main() {

    char passwordArray[PASSWORDSIZE] = {'t', 'o', 'p', 'g', 'u', 'n'};
    char charArray[ARRSIZE];

    int matched = 0; /* keep track of how many characters matched the passwd */
    int i;
    int length = 0;

    printf("Enter password (Enter to terminate)\n");
    for (length=0; length<ARRSIZE; length++) {
        scanf("%c", &charArray[length]);
        if (charArray[length] == '\n')
            break;
    }

    for (i=0; i<length; i++)  {
        if (charArray[i] == passwordArray[i])
            matched++;
        else {
            matched = 0;
            break;
        }
    }

    if (matched == PASSWORDSIZE)
        printf("The password is correct\n");
    else
        printf("The password is wrong\n");
}
```

In the above program, the entered password is compared with the passwordArray character by character. Only if "topgun" is entered exactly at the input, then this password is considered correct. All other character sequences are considered wrong.

The first for loop (in green colour) is the standard code for reading and storing characters into a char array. The variable length records the number of characters stored.

The variable matched plays a pivotal role in the program. It keeps track of how many characters, starting from the first, match the passwordArray. The second for loop compares the two arrays of characters. The variable matched is increased by one for every match pair of characters. The for loop has two exits – one when a mismatch is found (break statement) and another when all the entered characters are compared. Password checking need not continue in either of these cases.

Finally, the variable matched is checked against the length of the passwordArray. Matching the length means that the password entered is correct. This is a standard method to compare two arrays of data. Learn it.

It is easy to change the password. Modify the program so that the password is now "forrestgump". Work on the program to make it happen.
Achieve Target 3 Cont

Let's apply the array matching method to this program.

```c
#include <stdio.h>
#define NUMARRAYSIZE 6

void main() {
    int numArray[NUMARRAYSIZE] = {12, 45, 31, 67, 89, 2};
    int inputArray[NUMARRAYSIZE];
    int i;
    printf("Enter %d numbers:\n", NUMARRAYSIZE);
    for (i=0; i<NUMARRAYSIZE; i++) {
        scanf("%d", &inputArray[i]);
    }
    printf("The entered numbers are: ");
    for (i=0; i<NUMARRAYSIZE; i++)  {
        printf("%d ", inputArray[i]);
    }
    printf("\n");
}
```

Output

Enter 6 numbers:
3 4 5 6 7 8
The entered numbers are: 3 4 5 6 7 8

The above program reads in a series of integers and prints them out.

We studied how to compare two arrays in the previous exercise. Rewrite the above program so that the entered integers are compared with the integers defined in numArray. The program should print the message "Matched Sequence" if the entered integer matched the numArray exactly. Otherwise the program should print "No Match". The following reminders may be useful.

1. Refer to the previous exercise for useful code skeletons. Adapt them to this problem.
2. The statements in green in the above program should be removed.
3. Declare variables if necessary.

Work on the program to make it happen.
Achieve Target 4

The following program allows up to 128 integers to be entered and stored in an array.

```c
#include <stdio.h>
#define INPUTARRAYSIZE 128

void main() {
    int inputArray[INPUTARRAYSIZE];
    int length = 0;
    int i;
    printf("Enter numbers (-1 to terminate):
");

    for (i=0; i<INPUTARRAYSIZE; i++) {
        scanf("%d", &inputArray[i]);
        if (inputArray[i] == -1) {
            length = i;
            break;
        }
    }

    printf("The entered numbers are: ");
    for (i=0; i<length; i++)  {
        printf("%d ", inputArray[i]);
    }
    printf("\n");
}
```

Again there are many familiar features in the program.

1. The use of −1, a sentinel value, to terminate the input. It is assumed that the sentinel value is not useful as an input data.
2. The use of a variable (length) to record how many integers are stored in the array.
3. The use of #define to define a label for specifying array size.

We have learned that the variable length is used to keep record of how many integers are stored in the array. We will study another method that does not require an extra variable for this purpose.

```c
... void main() {
    int inputArray[INPUTARRAYSIZE];
    int i;

    printf("Enter numbers (-1 to terminate):
");
    for (i=0; i<INPUTARRAYSIZE; i++) {
        scanf("%d", &inputArray[i]);
        if (inputArray[i] == -1) {
            break; /* leaving the –1 at the end of the numbers */
        }
    }

    for (i=0; inputArray[i] != -1; i++) {
        printf("%d ", inputArray[i]);
    }
    printf("\n");
}
```

This program is equivalent to the previous program, but without the use of an extra variable to record how many integers are stored. The sentinel value −1 is placed after the entered integers in the array. So the sentinel value can indicate where is the end of the integers. The second for loop utilises the sentinel value in deciding where to stop when traversing the array. Learn it.
Achieve Target 4 Cont

Let’s apply the sentinel value method to a character array.

```c
#include <stdio.h>
#define CHARARRAYSIZE 1024

void main() {
    char charArray[CHARARRAYSIZE];
    int length;
    int i;

    printf("Enter a line of text (Enter to terminate):\n");
    for (i=0; i<CHARARRAYSIZE; i++) {
        scanf("%c", &charArray[i]);
        if (charArray[i] == '\n') {
            length = i;
            break;
        }
    }

    for (i=0; i<length; i++) {
        printf("%c", charArray[i]);
    }
    printf("\n");
}
```

The above program uses the variable length to keep record of how many characters are stored in the array `charArray`.

Modify the above program so that it uses the sentinel value ‘\n’ to indicate how many characters are stored in the array, instead of using the variable `length`. Work on the program to make it happen.

The sentinel value also has its disadvantage – the sentinel value cannot be part of the data. For example, if we use the newline character ‘\n’ as the sentinel value, then we cannot store ‘\n’ as part of the data. In the same way, we used –1 as the sentinel value in the previous exercise, we cannot accept –1 as part of the input data. Remember it.

For situations where we can find a good sentinel value that will not be part of the data, the sentinel value method is better than using the length variable method. This is particularly true for using character arrays to store text. Many special characters can be candidates for the sentinel value, such as newline (‘\n’), carriage return (‘\r’), backspace (‘\b’), tab (‘\t’), etc.

Rewrite your solution to the above exercise. Instead of using the newline character as the sentinel value, the program will use the NULL character ‘\0’. The NULL character ‘\0’ is a very suitable sentinel value because it is not a readable or printable, and normally will not be found in text. Work on the program to make it happen.
Achieve Target 5

The use of a sentinel value to terminate an array simplifies a program.

```c
#include <stdio.h>
void main() {
    char charArray[] = {'c', 'o', 'm', 'p', 'i', 'l', 'e', 'r', '\0'};
    int i;
    for (i=0; charArray[i] != '\0'; i++) {
        printf("%c", charArray[i]);
    }
    printf("\n");
}
```

We need not concern about the length of the array, because there is a NULL character '\0' at the end of data.

The NULL character '\0' is in fact very special in C programs. Using the NULL character in character arrays can make use of many features and functions in C. For example, the statements in the box above is designed to print the data of a NULL terminated character array. The statements can be replaced by a single call to printf as in the following program.

```c
#include <stdio.h>
void main() {
    printf("%s\n", charArray);
}
```

The conversion character "\%s" is for printing a NULL terminated character array. An error may occur if charArray is not terminated by a NULL character.

If a character array is terminated by a NULL character, we call it a string. A string is always a character array, but a character array is a string only if it is terminated by a NULL character.

Remember it.

String is a very common feature in programming. We have seen in the above program a method of specifying the string "compiler". However, the C language provides a simpler way to specify strings.

```c
#include <stdio.h>
void main() {
    char charArray[] = "compiler";
    printf("%s\n", charArray);
}
```

The above program shows an equivalent way to specify pre-defined strings. Pre-defined strings are enclosed within a pair of double quotes. These strings are called string literals. Remember it.

The above three programs do exactly the same with different methods. You should learn them all though the last method would be most frequently used. Learn it.
Achieve Target 5 Cont

Study the following program.

```c
#include <stdio.h>
#define CHARARRAYSIZE 1024

void main() {
    char charArray[CHARARRAYSIZE];
    int i;

    printf("Enter a line of text (Enter to terminate):
");
    for (i=0; i<CHARARRAYSIZE; i++) {
        scanf("%c", &charArray[i]);
        if (charArray[i] == '\n') {
            charArray[i] = '\0'; /* special sentinel value called NULL */
            break;
        }
    }
    printf("%s\n", charArray);
}
```

The input part is a standard loop that reads until a newline character is seen. The output part is a simple `printf` statement that prints the string `charArray`. The now complicated looking input part has a simpler alternative.

```c
#include <stdio.h>
#define CHARARRAYSIZE 1024

void main() {
    char charArray[CHARARRAYSIZE];

    printf("Enter a line of text (Enter to terminate):
");
    gets(charArray);
    printf("%s\n", charArray);
}
```

The function `gets` reads in a line of text into a character array. It also adds a NULL character at the end to make it into a string. An alternative is to use the `scanf` function.

```c
scanf("[\^n]", charArray); /* same as gets(charArray) */
```

The conversion character "[\^n]" is for reading text until the newline character ('\n'). The receiving variable (charArray) must be a character array. The function `scanf` will make it into a string by adding a NULL character at the end. Note that `charArray` has no ampersand (&) placed before it, unlike the cases of reading an integer, a floating point number or a character. Remember it.

There are two string literals exist in the above program, indicated by green bold. Remember that string literals are decorated by a pair of double quotes. String literals not only exist in array initialisation but also in other forms. Remember it.
Achieve Target 5 Cont 2

Let's apply what we have learned.

```c
#include <stdio.h>
#define PASSWORDSIZE 6
#define ARRAYSIZE 256

void main() {
    char passwordArray[PASSWORDSIZE] = {'t', 'o', 'p', 'g', 'u', 'n'};
    char charArray[ARRAYSIZE];
    int matched = 0; /* keep track of how many characters matched the passwd */
    int i;
    int length = 0;
    printf("Enter password (Enter to terminate):\n");
    for (length=0; length<ARRAYSIZE; length++) {
        scanf("%c", &charArray[length]);
        if (charArray[length] == '\n')
            break;
    }
    for (i=0; i<length; i++)  {
        if (charArray[i] == passwordArray[i])
            matched++;
        else {
            matched = 0;
            break;
        }
    }
    if (matched == PASSWORDSIZE)
        printf("The password is correct\n");
    else
        printf("The password is wrong\n");
}
```

Now with the techniques you have studied in this target, rewrite the password checking program above to use the new techniques including string declaration, string reading, and string processing. Note that string processing is basically character array traversal. Remember that strings are really character arrays but terminated with a NULL character. The following lists some suggested items.

1. Declare a string literal to hold the password.
2. Using a string reading function to read in the user password, instead of using the `for` loop.
3. Modify the `for` loop for matching two character arrays so that it handles two strings instead.
4. Modify the final condition checking for password correctness (the red circle). This is quite challenging because `PASSWORDSIZE` is no longer available. You may use the following code for checking out the length of the password string.

   ```c
   for (passwordLen=0; passwordArray[passwordLen] != '\0'; passwordLen++) {
   }
   ```

   **Work** on the program to make it happen.
Achieve Target 6

Study the following program.

```c
#include <stdio.h>
#define SIZE 256

void main() {
    char charArray[SIZE];
    int len;

    printf("Enter a line of text (Enter to terminate):
");
    scanf("%[^\n]", charArray);

    for (len=0; charArray[len] != '\0'; len++) {
    }

    printf("Length is %d\n", len);
}
```

The above program traverses through the string and count the length of the string. The `for` body is empty because no processing is needed other than increasing the counter variable `len`. We could add code in the `for` body so that each array element can be processed one by one.

```c
#include <stdio.h>
#define SIZE 256

void main() {
    char charArray[SIZE];
    int len;

    printf("Enter a line of text (Enter to terminate):
");
    scanf("%[^\n]", charArray);

    for (len=0; charArray[len] != '\0'; len++) {
        if (charArray[len] >= 'a' && charArray[len] <= 'z')
            charArray[len] = charArray[len] - 'a' + 'A'; /* convert to uppercase */
    }

    printf("%s\n", charArray);
}
```

The above program converts all lowercase letters into uppercase letters. The blue bold statement performs the conversion. Characters in C are represented using that ASCII standard. Each character is in fact a number between 0 and 255. For example, 'A' is 65 and 'g' is 103.

Subtracting the current character by 'a' gives a value that is the offset from 'a'. For example, 'b' – 'a' gives 1 and 'd' – 'a' gives 3. Adding the value to 'A' gives the original character in uppercase. For example, 1 + 'A' gives 'B' and 3 + 'A' gives 'D'. Study the Appendix on ASCII table for more information.

Modify the above program so that all uppercase letters are converted to lowercase letters. Work on the program to make it happen.
Achieve Target 6 Cont

More on string processing in this program.

```c
#include <stdio.h>
#define SIZE 256

void main() {
    char charArray[SIZE];
    int i;

    printf("Enter a line of text (Enter to terminate):
");
    scanf("%[^\n]", charArray);
    for (i=0; charArray[i] != '\0'; i++) {
        if (charArray[i] == '.') { /* found a full stop */
            charArray[i] = '\0';
            break;
        }
    }
    printf("%s\n", charArray);
}
```

Output

Enter a line of text (Enter to terminate):
C offers a set of libraries for programmers. They are useful.
C offers a set of libraries for programmers

The above program truncates (cut-short) a string at the first full stop ('.') found. The truncation is done by replacing the full stop character with the NULL character. This insertion of NULL character means setting this element as the end of the string. Learn it.

Modify the program so that only the first half of the string is kept and the remaining is truncated. You will need to find out the length of the string first, and then decide which element is to insert the NULL character. Work on the program to make it happen.
Achieve Target 6 Cont 2

The following program is a rewritten password-checking program. Study the program.

```c
#include <stdio.h>
#include <string.h>
#define ARRAYSIZE 256

void main() {
    char passwordArray[] = "topgun";
    char charArray[ARRAYSIZE];
    printf("Enter password (Enter to terminate):
");
    scanf("%[^\n]", charArray);

    if (strcmp(charArray, passwordArray) == 0)
        printf("The password is correct\n");
    else
        printf("The password is wrong\n");
}
```

The string matching part is now a single statement doing a function call to `strcmp`. The function compares two strings (`charArray` and `passwordArray`) and returns 0 if the two strings match exactly. The function call replaces many lines of code and greatly simplifies the program. It does much of the job for you as a programmer. **Remember** it.

C provides many such functions for processing strings (NULL terminated character arrays). Many of them can be found in the `<string.h>` library. The function `strcmp` is part of the library, and therefore the above program needs the following line to include the library.

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

The following program shows another useful function `strlen` for finding the length of a string.

```c
#include <stdio.h>
#include <string.h>
#define SIZE 256

void main() {
    char charArray[SIZE];
    int len;
    printf("Enter a line of text (Enter to terminate):
");
    scanf("%[^\n]", charArray);

    len = strlen(charArray);
    printf("Length is %d\n", len);
}
```

Again this simplifies the program with a call to library functions. The use of existing library functions in your program is important. You will need to be familiar with the library functions and understand their effect and limitations.
Achieve Target 7

In the last 6 targets, you have worked hard on the following ideas. Ponder upon these ideas and remember what you have learned.

1. Use the pre-processor directive `#define` to help manage the use of array size in your program. This requires a little bit more effort at the beginning to make sure the labels are placed appropriately. But if the array size is subject to change the effort is worth it. Changing one place at the `#define` line will see the change effects in specific places in your program. Using `#define` appropriately will make your program more **resilience to change**.

2. Arrays are declared with a **size**, however, that maybe only a part of the array is stored with data. The number of array elements actually stored with data is called often as the **data length**. The size of an array and the data length are two distinctive pieces of information. This information of data length is required for array traversal and it should be kept as a record. One method is to use a variable to record the data length.

3. An alternative to using a variable to record the data length is to use a **sentinel value**. By definition, sentinel value is not considered as part of the data. To indicate the length of the data part in an array, a sentinel value can be placed to indicate the end of the data part. In array traversal, the execution condition could be set to detect the sentinel value, and the traversal can stop if the sentinel value is seen.

4. The advantage of using the sentinel value is no extra variable is needed to keep record of the data length. A possible disadvantage is that the data set may be restricted because the sentinel value cannot be part of the data. However, a carefully chosen sentinel value will often avoid this possible problem. For example, in character arrays we could safely use the NULL character (`\0`) as the sentinel value.

5. A character array that uses the NULL character as the sentinel value to terminate the data part is known as a **string**. String is special in C because there are many functions and features supporting string processing.
Appendix – ASCII Table

The ASCII indexes are in decimal.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 9</td>
<td></td>
<td></td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 – 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 – 29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 – 39</td>
<td></td>
<td></td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>’</td>
<td></td>
</tr>
<tr>
<td>40 – 49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 – 59</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>60 – 69</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>70 – 79</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>80 – 89</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>90 – 99</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>100 – 109</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
</tr>
<tr>
<td>110 – 119</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>120 – 129</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td>)</td>
<td>~</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The shaded ASCII indexes are non-printable special characters. For example, NULL, newline, etc are non-printable special characters.
The following program also prints the ASCII table. Study this program to learn the relation between ASCII index and characters.

```c
#include <stdio.h>

void main() {
    int row, index;
    int sindex;
    int eindex;
    int i;

    /* printing the title */
    printf("ASCII Table\n\n");

    /* printing the top row of the table */
    printf("         ");
    for (i=0; i<=9; i++)
        printf("%3d", i);
    printf("\n");

    /* looping through the rows */
    for (row = 0; row <= 12; row++) {
        /* first calculate the starting and ending ASCII index of the row */
        sindex = row * 10;
        eindex = sindex + 9;
        /* print the range of the row */
        printf("%3d - %3d", sindex, eindex);
        /* looping through the columns */
        for (index = sindex; index <= eindex; index++) {
            if (index >= 33 && index <= 126) /* printable characters */
                printf("%3c", index);
            else
                printf("   ");
        }
        printf("\n");
    }
}
```

In the above program, only the printable characters are printed. The variable index is printed as a character using the conversion character "%c". The number 3 in the "%3c" is a formatting instruction that specifies at least 3 spaces for printing the character.

**Output**

```
ASCII Table

0 -  9
10 -  19
20 -  29
30 -  39           !  "  #  $  %  &  '
40 -  49  (  )  *  +  ,  -  .  /  0  1
50 -  59  2  3  4  5  6  7  8  9  :  ;
60 -  69  <  =  >  ?  @  A  B  C  D  E
70 -  79  F  G  H  I  J  K  L  M  N  O
80 -  89  P  Q  R  S  T  U  V  W  X  Y
90 -  99  Z  [  ]  ^  _  `  a  b  c
100 - 109  d  e  f  g  h  i  j  k  l  m
110 - 119  n  o  p  q  r  s  t  u  v  w
120 - 129  x  y  z  {  |  }  ~
```