Reading Material

From *Teach Yourself C*:

- Chapter 1: Sections 1.7 - 1.9
- Chapter 2: Sections 2.1 - 2.6
- Chapter 3: Section 3.2-3.4, 3.6, 3.7
A. Mathematical functions

To use mathematical functions it is necessary to include the header file `math.h`. Here are some of the most commonly used functions:

- `sqrt( argument )` \hspace{1cm} square–root
- `pow( argument, argument )` \hspace{1cm} power
- `exp( argument )` \hspace{1cm} exponential
- `log( argument )` \hspace{1cm} natural logarithm
- `log10( argument )` \hspace{1cm} logarithm base 10
- `cos( argument )` \hspace{1cm} cosine
- `acos( argument )` \hspace{1cm} arc cosine
- `sin( argument )` \hspace{1cm} sine
- `asin( argument )` \hspace{1cm} arc sine
- `tan( argument )` \hspace{1cm} tangent
- `atan( argument )` \hspace{1cm} arc tangent

Arguments can be of double or float type and so is the output and function type.

When angle variables are used with trigonometric functions (as input or output), they are given in radians and not degrees.
B. Functions

The more you break down your code to separate parts and components, – each of which does only one thing, i.e., has one function – the easier it is to understand, debug, develop, and expand your code.

Functions must be defined before they can be used, i.e., before they can be called from a part of the program. The data type of a function’s output variable is the type of the function itself.

```c
type-of-function-output name-of-function(type-of-input-variable name-of-input-variable, ... ) {
    ...
    return output-variable or value;
}
```

NOTE: You have to define the data type of each input variable separately, even if they are all of the same data type.

If a function is called in main, then it must be defined before that (after the #include statements). If it is called from another function, then it must be defined before that other function, etc.

If the function uses variables other than the input and output variables, they MUST be defined just after the function declaration, in exactly the same way variables are declared in the main function of the program.

Function calls can be used in printf statements or in general anywhere a variable can be used. For example:

```c
... /* function sum calculates the sum of two real numbers */
float sum (float a, float b) {
    return a+b;
}
...
SumTotal=sum(3.,5.);
printf("The sum is %f", SumTotal);

OR

printf("The sum is %f", sum(3.,5.));
```

Some functions return no output. For example on can build a function with the sole purpose of printing output. In these cases, we use a special data type called void:

```c
void PrintFunction (input variables) {
    printf("text");
}
```
C. Numerical Predicates

Often in programs we need to apply *branching*: different parts of the code are executed depending on the results of certain tests. Programming languages allow such testing with the use of operators that examine whether statements (*Boolean expressions*) are TRUE or FALSE. Such operators return a value equal to 1 if the statement is TRUE, and 0 if the statement is FALSE.

Predicates used with pairs of numbers:

- `==` equal ?
- `!=` not equal ?
- `>` greater than ?
- `<` lower than ?
- `>=` greater than or equal ?
- `<=` lower than or equal ?

The two variables used with numerical predicates must be of the same data type. Data-type conversion operators can be used in the same line as the predicate:

```plaintext
int i=50;
float r=5.6;
... (float) i == r ...
```
D. Conditional Statements

One-way conditional statement:

```cpp
if (Boolean expression) {
    if-true statement(s)
}
```

Two-way conditional statement:

```cpp
if (Boolean expression) {
    if-true statement(s)
} else {
    if-false statement(s)
}
```

IF-blocks can be nested:

```cpp
if (Boolean expression) {
    if (Boolean expression) {
        if-true statement(s)
    } else {
        if-false statement(s)
    }
}
```

E. Combining Boolean Expressions

Boolean expressions can be combined to form a single new expression and be used in conditional statements.

Boolean operators:

- `&&` AND
- `||` OR
- `!` NOT
F. Local and Global Variables

All variables declared within a function are local variables, in the sense that their values are available only within that function. So far, whenever we’ve dealt with variables they were local. This includes the ones declared within main.

When variables are passed from one function to the other as input parameters, they work as local variables: their values are available only within the function while it is executed. Input Parameters are isolated from the rest of the program and changes of their values within the executing function do not affect the value of the variable with the same name in the calling function. This means that C is a call-by-value language (instead of “call-by-reference”) and a new chunk of memory is reserved for each input parameter every time the corresponding function is called.

NOTE: It is safer to avoid using variables and parameters with the same name in different functions.

Global variables have the advantage that their value is accessible to and changeable by all functions that use them after the variable has been declared. Global variables are declared after the include lines and before function declarations.

To declare and define a constant, i.e., a global variable with a specific value, which however is not allowed to be changed by any part of the program under any circumstances (for example, constants like π, the gravitational constant $G$, the speed of light $c$, etc.), we use:

```
#define constant-name value
```

Constant definitions are placed after include lines and before the declaration of any global variables and the functions of the program.
G. Loops: while Statement

Very often in programming a set of statements have to be repeated a large number of times. In such cases we use loop structures, which allow us to repeat computations again and again provided that a certain condition is satisfied.

One of the most commonly used loop structures involves the while statement:

\[
\text{while ( Boolean expression ) } \{
\text{statements to be executed while Boolean expression is TRUE...}
\}
\]

For example, the following structure repeatedly prints and decreases the value of a positive integer N until N=0 (or in other words while the expression “N is different from 0” is TRUE):

\[
\text{if ( N>0 ) } \{
\text{while ( N ! = 0 ) } \{
\text{printf ( "\%i \n",N);}
\text{N = N-1;}
\}
\}
\]

A loop structure can enclose all possible other statements to be repeated, e.g., scanf, printf, if–else, and any function-calling statements.

Note that, if the Boolean expression used in a while statement is FALSE when the while statement is reached for the first time, then the statements enclosed in the loop are never executed.
H. Loops: for Statement

Another way of setting up loop structures is with the use of a *loop-control variable* and the *for* statement. The loop ends when a control statement that involves the control variable is no longer satisfied.

The very first line of a *for* statement typically contains three pieces of information: (i) the initial value of the loop-control variable, (ii) the Boolean expression checked at the beginning of each iteration, and, if FALSE, results in the termination of the loop, and (iii) an arithmetic expression that changes the value of the control variable at the end of every iteration. The basic structure of the *for* statement is:

```
for ( control-initialization expression ;
     loop-testing expression ;
     control-reassignment expression ) {
     statements to be executed while loop-testing expression is TRUE...
}
```

For example, the following loop structure calculates $2^N$, for $N>0$:

```
result=1;
for (counter = 1; counter != N+1; counter = counter + 1) {
    result = 2 * result;
}
```

Note that (i) the control-initialization statement is executed only once, at the very start of the loop, (ii) the loop-testing statement is executed at the very start of the loop and every time the loop is repeated; therefore if the testing expression is FALSE, then the loop is *never* executed.

A *for* statement can also be written as a *while* statement (see example below) and vice versa.

```
result = 1;
while (N != 0) {
    result = 2 * result;
    N = N - 1;
}
```
NOTE 1: Increasing or decreasing the value of an integer variable by 1 can be achieved with the use of the *increment* or *decrement* operators:

\[ i = i++ \text{ or } i = ++i \] have the same result as \( i = i + 1 \)

\[ i = i-- \text{ or } i = --i \] have the same result as \( i = i - 1 \)

but are more concise. It is much safer to avoid using the above increments embedded in arithmetic expressions (in fact, setting the operator before or after the variable makes a difference, if the operator is embedded in an arithmetic expression!).

NOTE 2: A loop structure can be interrupted (“broken”) using the *break* command at any point in the loop structure. It takes no argument and it usually appears within a conditional statement.

NOTE 3: Loop structures can be nested (one loop can be written as part of another loop), just like *if* blocks can be nested.
Programming Hints

- Always use comments in the code to explain what a function does and how.
- When you use multi-word variables, write them all as one word using upper case for the first letter of each word, e.g., MultiWordVariable=3.
- For simplicity and safety, avoid using the same name for variables that are passed from one function to the other.
- Align first and last line of conditional statements and indent successively each block of statements following if and else lines.
- Make it a habit to use parentheses to enclose Boolean expressions and curly brackets to enclose code lines of an if-else statement.