PMC Lecture 02

Gianmaria Romano

24 September 2025

Appendix A

Terminals

A.1 Terminal instructions

It can come in handy to run programs inside terminals as they tend to be quicker compared to other text editors.
Generally speaking, the most used terminal commands are the following:

- **pwd:** this command provides the current working directory.
- **ls:** this command, often used with the grep command as well, shows the files or subdirectories present within the current directory.
- **cd:** this command allows to change the current directory.
  N.B.: To go backwards, it is necessary to use the **cd..** command instead.
- **rm:** this command allows to remove a file or directory.

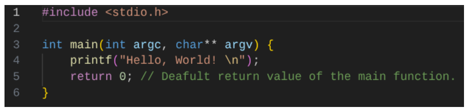
Most particularly, running a program in a terminal, an output file will be produced as well.

Appendix B

The C programming language

B.1 The basics of the C programming language

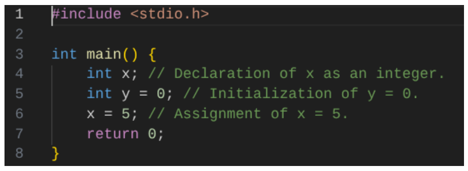
Every C program requires a header, which invokes the preprocessor before compiling time and declares input/output functions, and the main() function, which executes the program, eventually calling other functions to perform sub-tasks.
Example: The following code prints "Hello, World!":



```c
#include <stdio.h>

int main(int argc, char** argv) {
    printf("Hello, World! \n");
    return 0; // Deafult return value of the main function.
}
```

B.2 Declaring variables

Before being used, a variable must be declared by specifying a name and a type, although it is a good practice to initialize a value as well.
In addition, it is also possible to assign values in order to modify the contents of a variable.
Example: The following code declares a variable x, which is then set to store the value 5, and initializes another variable $y = 0$:



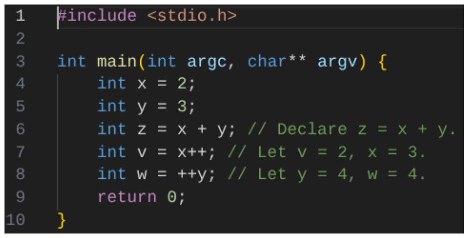
```c
#include <stdio.h>

int main() {
    int x; // Declaration of x as an integer.
    int y = 0; // Initialization of y = 0.
    x = 5; // Assignment of x = 5.
    return 0;
}
```

B.2.1 Defining macros

Using the #define command, it is possible to create macros, which act as ”symbolic constants” whose value will always stay associated to the name of the variable.

B.3 Arithmetic operators

In order to perform computation involving variables, it is possible to use arithmetic operators, which include the standard mathematical operations, along with the modulo division, and self-increments.
Example: The following code declares two integer variables x and y that are used to declare new variables z, which is implemented as the sum between x and y, and v and w, which are implemented through self-increments:



```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 2;
    int y = 3;
    int z = x + y; // Declare z = x + y.
    int v = x++; // Let v = 2, x = 3.
    int w = ++y; // Let y = 4, w = 4.
    return 0;
}
```

B.4 Relational operators

Generally speaking, relational operators are used to check conditions among variables, typically for comparison purposes.

B.5 Logical operators

Logical operators deal with combining more conditions, which, through dedicated commands, can be evaluated over entire words or bit-by-bit.

B.6 Input/Output activities

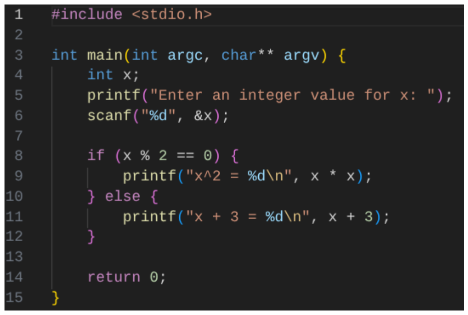
Input/Output activity mainly deals with showing values through the printf() function, which allows to print strings or, through dedicated placeholders, numerical values.
Alternatively, in the case of user-inputted values, it is possible to print values through the scanf() function.

B.7 Selective codes

A selective code branches towards different actions depending on whether a certain condition is true or not.

B.7.1 If/else structures

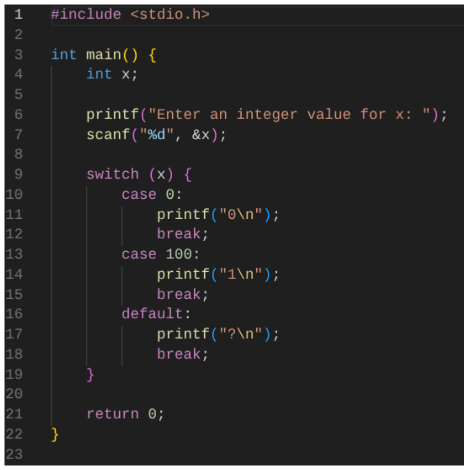
An if/else structure creates a hierarchical structure that gradually checks whether each condition is true or not, going on until the default case.
Example: The following code takes as input a positive integer x and, if x is even, it prints x^2, otherwise, if x is odd, it prints $x + 3$:



```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x;
    printf("Enter an integer value for x: ");
    scanf("%d", &x);

    if (x % 2 == 0) {
        printf("x^2 = %d\n", x * x);
    } else {
        printf("x + 3 = %d\n", x + 3);
    }

    return 0;
}
```

B.7.2 Case structures

A case structure branches according to the value of a control variable, meaning that, to avoid issues, a default case must always be specified.
Example: The following code prints a certain string depending on the received input:



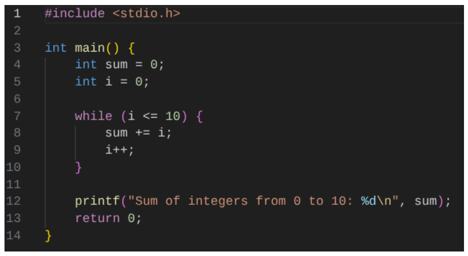
```c
#include <stdio.h>

int main() {
    int x;

    printf("Enter an integer value for x: ");
    scanf("%d", &x);

    switch (x) {
        case 0:
            printf("0\n");
            break;
        case 100:
            printf("1\n");
            break;
        default:
            printf("?\n");
            break;
    }

    return 0;
}

```

B.8 Iterative codes

An iterative code repeats some actions as long as a certain condition is true.

B.8.1 While loops

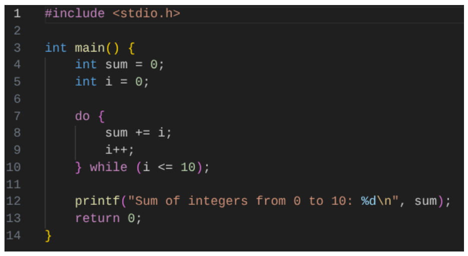
A while loop builds the iterative condition using an external control variable that gets updated during the iterations.
Example: The following while loop prints the sum of all integers from 0 to 10:



```c
#include <stdio.h>

int main() {
    int sum = 0;
    int i = 0;

    while (i <= 10) {
        sum += i;
        i++;
    }

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

B.8.2 Do/while loops

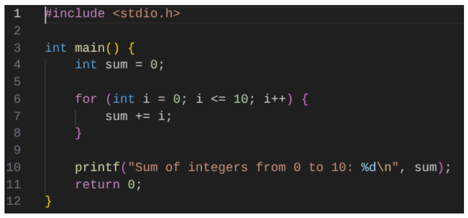
A do/while loop works similarly to a standard while loop, but, since it first executes the loop and then updates the control variable, it is always guaranteed to run at least once.
Example: The following do/while loop prints the sum of all integers from 0 to 10:



```c
#include <stdio.h>

int main() {
    int sum = 0;
    int i = 0;

    do {
        sum += i;
        i++;
    } while (i <= 10);

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

B.8.3 For loops

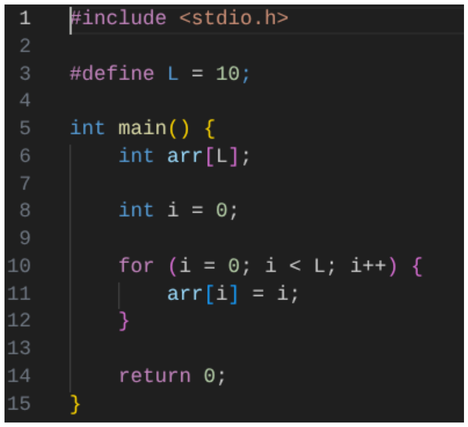
A for loop relies on a control variable that can be initialized and updated in a dedicated command.
Example: The following for loop prints the sum of all integers from 0 to 10:



```c
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 0; i <= 10; i++) {
        sum += i;
    }

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

B.9 Arrays

An array is a vector of elements of the same type (which are accessed by index slicing) that is declared by specifying its size and type, although, generally speaking, it is a good practice to initialize its values as well.
Example: The following code initializes an array where $arr[i] = i$:



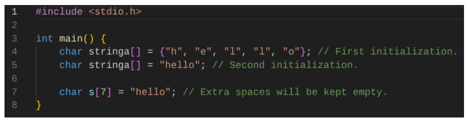
```c
#include <stdio.h>

#define L = 10;

int main() {
    int arr[L];

    int i = 0;

    for (i = 0; i < L; i++) {
        arr[i] = i;
    }

    return 0;
}
```

B.9.1 Generalizing arrays to build matrices

Arrays are often used to implement matrices as two-level arrays, whose elements are accessed using the $[i][j]$ notation, which allows to access the value located at the i^{th} row and j^{th} column.

B.9.2 Using arrays to represent strings

A string is a particular array of characters that is delimited using a dedicated terminator character.
Example: The following code provides some ways to implement a string as a character array:



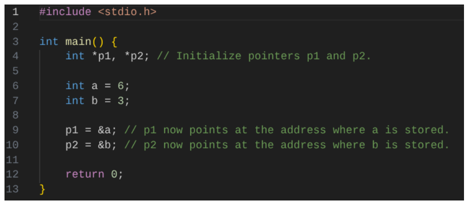
```c
#include <stdio.h>

int main() {
    char stringa[] = {"h", "e", "l", "l", "o"}; // First initialization.
    char stringa[] = "hello"; // Second initialization.

    char s[7] = "hello"; // Extra spaces will be kept empty.
}
```

B.10 Implementing functions

A function is a piece of code that can work independently of other functions and, if needed, it can also declare local variables within its scope.
Generally speaking, a function takes parameters by value, meaning that it will work on a copy of the original parameters, which are instead left untouched, although structures like arrays and pointers take parameters by reference, meaning that it will refer to the actual instance of the input values.

B.11 Pointers

A pointer is a particular variable that contains the memory address to another variable of the same type, eventually allowing to retrieve its value as well through dedicated commands.
Example: The following code, after initializing two pointers and two integers, sets p_1 to point towards a's memory address and p_2 to point towards b's memory address:



```c
#include <stdio.h>

int main() {
    int *p1, *p2; // Initialize pointers p1 and p2.

    int a = 6;
    int b = 3;

    p1 = &a; // p1 now points at the address where a is stored.
    p2 = &b; // p2 now points at the address where b is stored.

    return 0;
}
```