EE102: Software Engineering I

Section 7 – Introduction to Programming

7.1 Definition and Notes:

- **Programming**
  - Significant component of the Software Engineering process
  - Includes Design, Coding and Testing
- **Design**
  - The design stage follows some principles in order to solve the given problem using the computer
  - These principles are independent from the programming language used for the implementation of the solution
  - The solution is regularly provided using **pseudo-code** or **diagrams**
- **Coding**
  - The solution is regularly implemented using that programming language which maximizes the advantages

  - These advantages could be in terms of execution speed (how fast the final code will execute), implementation speed (how fast the programmers will be able o write the code), cost (how much the implementation will cost), what language the programmers are more familiar with, etc.

- **Testing**
  - Very significant phase of programming
  - It is important to be sure that:
    - the right program was built and
    - the program was built right

- **Programming languages**
  - There are a large variety of programming languages, many of different types, with advantages and disadvantages
  - When one learns how to program, the principles are more important than the implementation language
  - e.g. C, C++, BASIC, PASCAL, JAVA, PROLOG, COBOL

- **Note**
  - During this course we will use C for exemplification

7.2 C Programming Language:

- **Characteristics**

    - C is a "***high level***" language (similar to human language, English in particular) that makes it easy to be read, modified and written by people
    - Programmers will use C to write their own programs (***source code***)
    - C requires ***translation*** into "low level", machine language that the CPU of the computer can understand and execute
    - Translation programs named ***compilers*** will translate the C programs into ***executable programs***
    - Executable programs are self contained (they do not need a copy of the C source program in order to run) and can be executed very fast by the computer
    - Often the programmers need to use some functions that are either provided by the programming language as an extension to the basic set of instructions or that have been written by other programmers
    - These functions are included in ***libraries*** that have to be added to the original executable in order to allow for the program execution
    - In order to use external written functions, the programmers have to add ***header files*** corresponding to the used functions to their source code

7.2 First C program

```
/*
hello.c: Program to display "Hello World"
*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello World!\n");
    return(EXIT_SUCCESS);
}

/*
Expected output:
Hello World!
*/
```

- **Notes (1)**
  - As any other C program this starts with a section for all **pre-processor directives**
  - These start with "#" and are instructions given to the program that will process your file named "***Compiler***"
  - Part of the Compiler called "***Pre-processor***" reads these directives, performs consequent actions and removes them from the program before allowing it to be further processed

  - The #include directive tells the Compiler to add another file in front of your code, file indicated using its name (e.g. stdio.h)
  - The included files are “***header files***” and will be presented later on

- **Notes (2)**

  - As any other C program this includes the **main() function**
  - This is the entry point into your program
  - When executing your code, the control will be given first to the main() function and then to the first of its ***statements*** (***instructions***)
  - The instructions will be then executed ***sequentially***
  - A C program can have any number of functions, but a single main() function

- **Notes (3)**

  - As any other function the main() function has:
    - A ***return type*** (e.g. “int”)
    - A ***name*** (e.g. “main”)
    - A ***list of parameters*** (between the brackets; e.g. empty in this case)
    - A ***body*** (includes instructions between { })

- This function has two statements:
- **printf("Hello World!\n");**
  - This ***prints*** in a ***formatted*** manner the ***text*** “Hello World!” on the screen
  - The only formatting here is that after the text a ***newline character*** (**‘\n’**) will be also printed, causing the caret to go to the next line
- **return(EXIT_SUCCESS);**
  - This statement ***returns the control to the calling function***
  - In this case ends the program
  - The instruction accepts one parameter which is the ***function’s return value***
  - EXIT_SUCCESS is a ***constant*** pre-defined in the stdlib.h header file
- ***Any statement ends with a semi-colon “;”***

7.3 Steps in Running an Own Written C Program

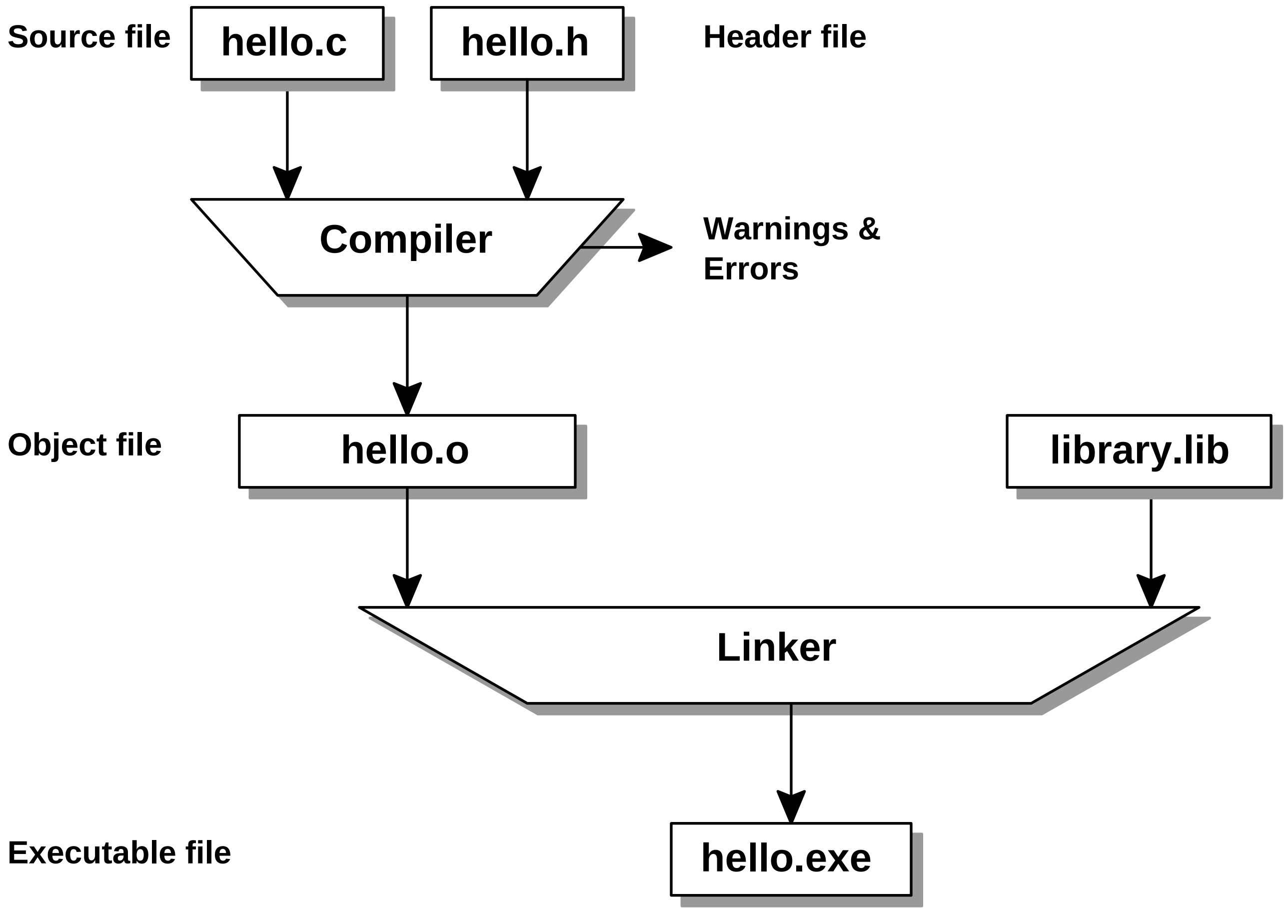
- **Writing the Code**
    - The program has to be written as a text file
    - Save the program with a suggestive name and .c extension
    - Make sure you have included the main() function

- **Compiling the Code**
    - Run the compiler with your source code as input
    - The compiler will translate the file with your program into ***an object file***
    - The object file contains binary instructions that can be processed by your computer's CPU

- **Linking**
    - The ***linker*** is a program that adds required libraries to your object file and builds the final executable file
    - The executable has often the same name as the file containing your original program, but the .exe extension

- **Running the Program**
    - Type the name of the executable and press ENTER
    - Your program will execute, instruction by instruction sequentially, starting with the first

- **Compiling Process**



7.4 Libraries and Header Files

- **Library**
    - **Definition:**
        - A collection of functions that can be called within a program without having them implemented again
    - **Fully specified by:**
        - **Header file(s)**
            - Provide specification of the external interface of each function of the library
            - Specify how to call each function
            - E.g. arithm.h, logic.h, etc.
        - **Source file(s)**
            - Contain the implementation of the library, including all the functions
            - E.g. arithm.c, logic.c, etc.
        - **Object file(s)**
            - Contain the compiled versions of the source code files
            - There is one object file for each source file
            - E.g. arithm.o, logic.o, etc.

- **Archive**
  - Includes all the object files packed together
  - When building an executable program, the linker scans each library, extracts and adds to the final code only those parts that are actually needed
  - E.g. math.lib or math.a

- **To use the library one needs:**
  - **The header files**
    - Such as when the compiler translates the program it can check that the interfaces to library functions are correct
  - **The archive file**
    - Such as the linker can extract the relevant information and build the executable program

- **Header Files**
  - **Definition:**
    - A collection of external interfaces for a set of functions, type declarations and constant definitions
  - **Reason:**
    - Simplifies the reuse of code

7.5 Basics of C Language

- **7. 5. 1. Data Types**
  - **Literals**
    - Representation in the language of a certain value
    - Are assigned an implicit type, based on their form
    - E.g. 123, ‘a’, “Dave”, - 43.27, \o34, \xd4
  - **Variables**
    - Location of memory to store certain values
    - Are assigned types explicitly (by programmer)
    - Variables can store only values of the assigned type
    - Are assigned names that are subject to rules:
      - Must begin with an alphabetic character
      - Other characters can be either alphabetic or alphanumeric
      - Can include any number of characters
      - ‘_’ is considered an alphabetic character
      - Cannot be one of the C keywords: “if”, “case”, “for”, “while”, “int”, “long”, “char”, “else”, “do”, “void”, “typedef”, “break”, “switch”, “float”, “double”, “continue”, “const”, “class”, “struct”, etc.
    - Regularly programmers choose suggestive names for variables: e.g. account_no, balance, stud_ID, ch

- It is good practice to initialize variables (in C/C++) and compulsory in other languages (Java)
- The assignment type-variable name is performed at **declaration**; e.g. float balance; int stud_ID; char ch;

- **Primitive Types**
  - Data is represented in binary
  - The **type** and **length** makes the difference between different interpretations of certain binary representation

  - **Integer numbers**

Type	**Length**	**Signed Range**	**Unsigned Range**
char	**8 bits (1 byte)**	-2^7 **to** $(2^7 - 1)$	**0 to** $(2^8 - 1)$
short	**16 bits (2 bytes)**	-2^{15} **to** $(2^{15} - 1)$	**0 to** $(2^{16} - 1)$
int	**16/32 bits (2/4 bytes)**	-2^{15} **to** $(2^{15} - 1)$ **or** -2^{31} **to** $(2^{31} - 1)$	**0 to** $(2^{16} - 1)$ **or 0 to** $(2^{32} - 1)$
long	**32 bits (4 bytes)**	-2^{31} **to** $(2^{31} - 1)$	**0 to** $(2^{32} - 1)$

Declaration:

int money;

Assignment:

money = 101;

Memory representation:

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1

- **Real numbers**

Type	Length	Signed Range
float	**32 bits (4 bytes) mantissa: 23 exponent: 8**	**$-1.7 * 10^{38}$ to $- 0.29 * 10^{-38}$ and $0.29 * 10^{-38}$ to $1.7 * 10^{38}$**
double	**64 bits (8 bytes) mantissa: 55 exponent: 8**	**N/A**

Declaration:

float balance;

Assignment:

balance = 1000.0;

Memory representation: $1.0 * 10^3$

0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

- **Characters**

Type	Length	Unsigned Range (ASCII)
char	**8 bits (1 byte)**	**0 to 255**

Declaration:

char ch;

Assignment:

ch = ‘A’;

Memory representation:

0	1	0	0	0	0	0	1

Notes:

- ASCII code for ‘A’ is $01000001_{(2)}$, $41_{(16)}$ or $64_{(10)}$
- Variables of char type can hold both integers and characters (as ASCII codes), difference is made at input/output
- There are some special characters represented in C as “escape codes”

Name	C representation
newline	**‘\n’**
backspace	**‘\b’**
alert bell	**‘\a’**

- **Complex Types**
  - Data is represented in binary
  - The **structure**, **type** and **length** makes the difference between complex types
  - **Array**

Complex type	Items type	Length
array	**Primitive type or user defined**	**Indicated by programmer**
string	**char**	**Indicated by programmer**

Declaration:

```
int coin_list[8];
char text[7];
```

Assignment:

```
coin_list[0] = 1;        text[0] = 'G';
coin_list[1] = 2;        text[1] = 'A';
coin_list[2] = 5;        text[2] = 'B';
coin_list[3] = 10;       text[3] = 'R';
coin_list[4] = 20;       text[4] = 'I';
coin_list[5] = 50;       text[5] = 'E';
coin_list[6] = 100;      text[6] = 'L';
coin_list[7] = 200;      text[7] = '\0';
```

Memory representation:

1
2
5
10
20
50
100
200

‘G’
‘A’
‘B’
‘R’
‘I’
‘E’
‘L’
‘\0’

- **Structure**

  To be presented in semester II

- **Union**

  To be presented in semester II

- **Class**

  To be introduced in year II

- **7. 5. 2. Expressions**
  - **Definition**
    - Combinations of literals, variables, **operators** and other expressions that can be evaluated and give a resultant value of certain type
  - **Operators**
    - **Arithmetic**

      + (addition), **-** (subtraction),

      / (division), ***** (multiplication), **%** (remainder)
    - **Logic**

      && (and), **||** (or), **!** (not)
    - **Relational**

      > (greater than), **>=** (greater or equal to)

      < (smaller than), **<=** (smaller or equal to)

      == (equal to), **!=** (different than)
    - **Bit-level**

      & (bitwise and), **|** (bitwise or), **^** (bitwise exclusive or)

      ~ (bitwise complement)

      << (shift left), **>>** (shift right)
  - **Expression**
    - [**(**] **Variable/Literal** [**Operator Expression**] [**)**]

      e.g. 27 * balance + (amount – 100.2)

- **7. 5. 3. Input and Output**

    - **Formatted input**

        - Simple types:

          # include <stdio.h>

          simple_type s_var;

          scanf("%flag", &s_var);

Simple Types	Flag
int	%d
float	%f
double	%lf
char	%c

        - Complex type:

          # include <stdio.h>

          complex_type c_var;

          scanf("%flag", c_var);

Complex Type	Flag
string	%s

- **Formatted output**

  - Simple types:

    # include <stdio.h>

    simple_type s_var;

    printf("%flag", s_var);

Simple Types	Flag
int	%d
float	%f
double	%lf
char	%c

  - Complex type:

    # include <stdio.h>

    complex_type c_var;

    printf("%flag", c_var);

Complex Type	Flag
string	%s

- **7. 5. 4. Summary**

  - **Simple Types**

Simple types	Declaration	Memory allocation	E.g.	Storage type
integer	**int i_num;**	**4 bytes**	**8**	**Binary**
real	**float f_num;**	**8 bytes**	**7.25**	**Binary**
boolean	**int b_val;**	**4 bytes**	**0**	**Binary**
char	**char c_val;**	**1 byte**	**‘a’**	**Binary (ASCII)**

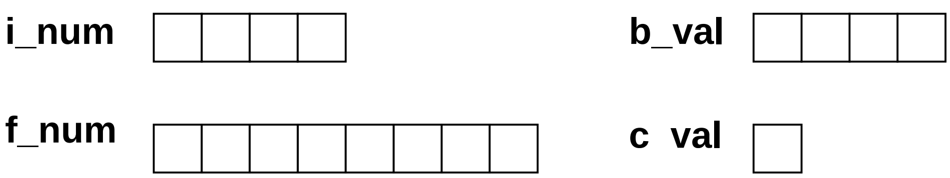
Declarations

int i_num;

float f_num;

int b_val;

char c_val;



Reading

scanf(“%d”, &i_num);

scanf(“%f”, &f_num);

scanf(“%d”, &b_val);

scanf(“%c”, &c_val);

Writing

```
printf("The integer number is: %d\n", i_num);
printf("The float number is: %f\n", f_num);
printf("The boolean value is: %d\n", b_val);
printf("The character is: %c\n", c_val);
```

Assignment

```
i_num = 20;
f_num = 2.35;
b_val = 0;
c_val = 'z';
```

Special Notes

- **i_num = 2.34;** results in the storage of the integer part of the float number only (i.e. 2.34 -> 2)
- **f_num = 3;** results in the transformation of the integer number in a float number (i.e. 3 -> 3.0)
- 0 stored in b_val variable is interpreted as **false**, whereas any value different than 0 is evaluated as **true**
- **c_val = 'z';** results in the storage of the ASCII code that corresponds to the character 'z' in the c_val variable, in binary format
- **c_val = 125;** is also valid since any value that can be represented in binary on 1 byte can be stored in a character-type variable

Operations

```
i_num = i_num + 20;
f_num = i_num * 1.53;
b_val = (i_num > 30);
c_val = 'a' +  2;
```

Special Notes

- **i_num = i_num/3;** results in an integer result since both operands are integer (e.g. if i_num's original value was 20, the result of this assignment is 6)
- **f_num = i_num + 3;** the result is converted in real number (e.g. if i_num's value was 20, the result of this assignment is 23.0)
- **b_val = 20 * 3;** since b_val is declared an integer, any operation defined for integers can be applied on b_val; only its value is interpreted as Boolean
- **c_val = 'a' + 2;** any operation can involve character variables; the result in interpreted as an ASCII code of a character (i.e. in this case character 'c')

Comparison

```
i_num >= 20
f_num < 3.75
b_val
c_val == 'c'
```

Special Notes

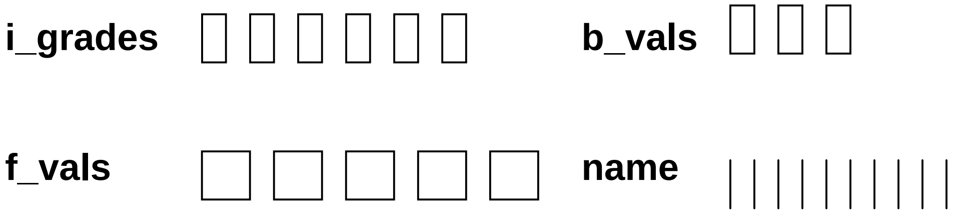
- **>, >=, <, <=, == and !=** are the comparison operators
- **||, && and !** can be used to combine the results of such comparisons
- **b_val** can be used as is in comparison as well as an integer value as term in a comparison
- **c_val** can be compared with other characters (e.g. ‘s’) as well as with their ASCII codes (e.g. 61)

o Complex Types

Complex types	Declaration	Memory allocation	E.g.	Storage type
string (array of chars)	char name[10];	10 * 1 byte	"John"	Run of binary (ASCII)
array of integers	int i_grades[6];	6 * 4 bytes	70, 60, …	Run of binary
array of booleans	int b_vals[3];	3 * 4 bytes	0, 1, 0	Run of binary
array of reals	float f_vals[5];	5 * 8 byte	7.2, 4.35, …	Run of binary

Declarations

```
int i_ grades[6];
float f_vals[5];
int b_vals[3];
char name[10];
```



Reading

```
for (index = 0; index < 6; index = index +1)
        scanf("%d", &i_grades[index]);
for (index = 0; index < 5; index = index +1)
        scanf("%f", &f_vals[index]);
for (index = 0; index < 3; index = index +1)
        scanf("%d", &b_vals[index]);
for (index = 0; index < 10; index = index +1)
        scanf("%c", &name[i]);
```

Special Notes:

- **Loops** are necessary in order to allow reading all the arrays' elements
- **A string-type variable** can be read in other ways:

```
scanf("%s", name);
        or
gets(name);
```

Writing

```
for (index = 0; index < 6; index = index +1)
        printf("The %d-th integer number is: %d\n", index,
                        i_grades[index]);

for (index = 0; index < 5; index = index +1)
        printf("The %d-th float number is: %f\n", index,
                        f_vals[index]);

for (index = 0; index < 3; index = index +1)
        printf("The %d-th boolean value is: %d\n", index,
                        b_vals[index]);

for (index = 0; index < 10; index = index +1)
        printf("The %d-th character is: %c\n", index, name[index]);
```

Special Notes:

- **Loops** are necessary in order to allow writing all the arrays' elements
- **A string-type variable** can be printed in other ways:

```
printf("The string variable is %s\n", name);
        or
puts(name);
```

Assignment

```
i_grades[3] = 75;
f_vals[1] = 2.45;
b_vals[0] = 1;
name[3] = 'z';
```

Special Notes

- The values of an array **cannot be assigned in block** with the exception of **string** type
- A special library was produced to allow for processing string type variables; in order to use it the **<string.h>** header must be included
- Assignment of a string-type variable is performed with the function: **strcpy(destination_string, source_string);** (e.g. strcpy(name, "John");)

Operations

```
i_grades[0] = (i_grades[3] + i_grades[2]) / 2;
f_vals[1] = i_grades[0] * 1.53;
b_vals[0] = (f_vals[0] > 30.5);
name[0] = 'a' + 3;
```

Special Note

The values of an array **cannot be processed in block**

Comparison

```
i_grades[0] >= 20
f_vals[1] < 3.75
b_vals[1]
name[2] == 'a'
```

Special Notes:

- Same operators **>, >=, <, <=, ==** and **!=** can be used as well as **||, && and !** to combine the results of individual comparisons
- If the header **<string.h>** is included**,** Comparisons of string-type variables can be performed in block with the function: **strcmp(string1, string2)** that returns a 0 in case of match and different than 0 otherwise. (e.g. strcmp(name, "John") == 0)

- **7. 5. 5. Examples**

```
/*
hello_u.c: Program to read a name and display "Hello" + name
*/

#include <stdio.h>

#include <stdlib.h>

int main()

{

    char my_name[20];

    printf("Enter your name: ");

    scanf("%s", my_name);

    printf("\nHello %s!\n!", my_name);

    return(EXIT_SUCCESS);

}

/*
Expected output:

Enter your name: John
Hello John!
*/
```

```
/*
io.c: Program that introduces and uses input/output functions
*/

#include <stdio.h>

#include <stdlib.h>

int main()

{

    float float_num;

    int int_num;


    printf("Enter two numbers (float and integer): ");

    scanf("%f%d", &float_num, &int_num);


    printf("\nThe inputted numbers are: %f and %d\n",
                                        float_num, int_num);

    return(EXIT_SUCCESS);

}

/*
Expected output:
Enter two numbers (float and integer): 3.5
6
The inputted numbers are: 3.500000 and 6
*/
```

```
/*
type_int.c: Program that introduces and uses the type integer (int)
*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int v, w, x, y, z;
    x = 4;
    y = 8;
    z = y/x;
    printf("z is equal to: %d\n", z);

    w = x/y;
    printf("w is equal to: %d\n", w);

    x = 3;
    y = 5;
    v = (y/x) * 10;
    printf("v is equal to: %d\n", v);

    return(EXIT_SUCCESS);
}
/*
Expected output:
z is equal to: 2
w is equal to: 0
v is equal to: 10
*/
```

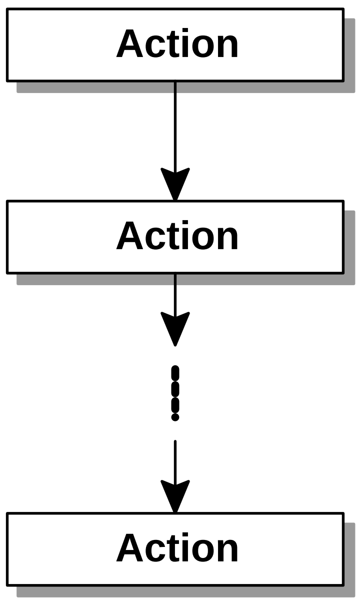
```
/*
type_float.c: Program that introduces and uses the type real (float)
*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
      float a, b, c, d, e, f;
      int x;

      b = 4.2;
      c = b/2 ;
      printf("c is equal to: %f\n", c);

      a = 10.0;
      d = (a/b) * c;
      printf("d is equal to: %f\n", d);

      x = 3;
      e = d/x;
      printf("e is equal to: %f\n", e);

      f = x/d;
      printf("f is equal to: %f\n", f);

      x = x*c;
      printf("x is equal to: %d\n", x);

      return(EXIT_SUCCESS);
}
/*
Expected output:
c is equal to: 2.100000
d is equal to: 5.000000
e is equal to: 1.666667
f is equal to: 0.600000
x is equal to: 6
*/
```

```
/*
type_bool.c: Program that introduces and uses boolean type (int)
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int b1, b2, x, y;

    x = 4;
    y = 8;
    b1 = (x>y);

    if (b1)
        printf("x is greater than y!\n");
    else
        printf("y is greater than x!\n");

    b2 = (x==y);

    if (!b2)
        printf("x and y are not equal!\n");
    else
        printf("y and x are equal!\n");

    return(EXIT_SUCCESS);

}
/*
Expected output:
y is greater than x!
x and y are not equal!
*/
```

```
/*
type_char.c: Program that introduces and uses character type (char)
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char ch;

    char name[20];

    printf("Are you student? (y/n): ");

    scanf("%c", &ch);

    printf("Enter your name: ");

    scanf("%s", name);

    printf("\nWelcome %s!\n", name);

    if (ch == 'y' || ch == 'Y')
        printf("%s is student!\n", name);
    else
        printf("%s is not student!\n", name);

    return(EXIT_SUCCESS);
}
/*
Expected output:
Are you student? (y/n): y
Enter your name: John
Welcome John!
John is student!
*/
```

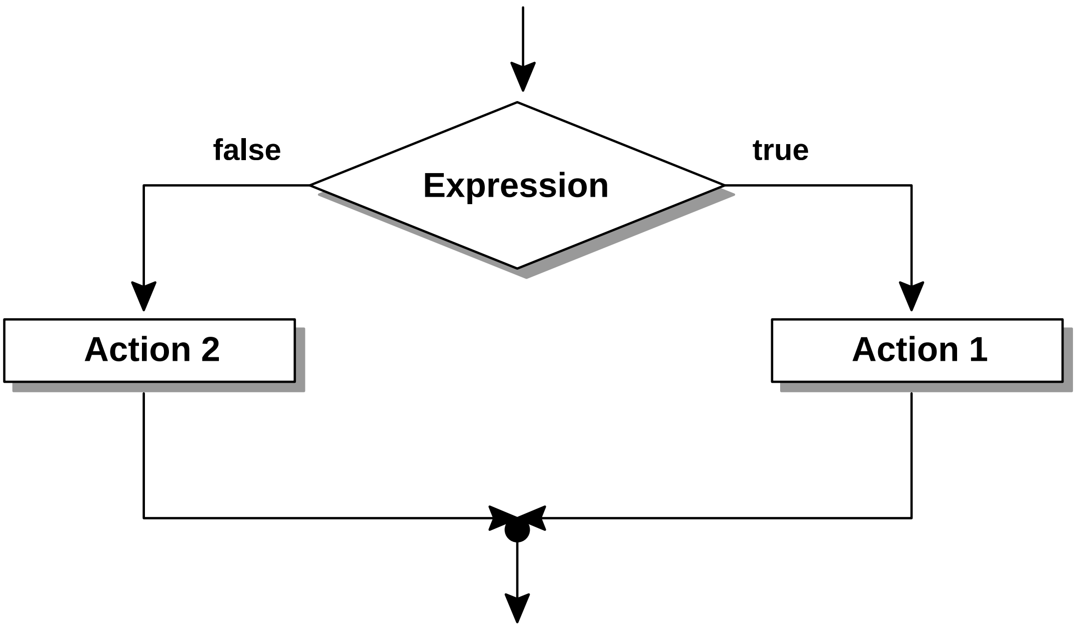
```
/*
rect1.c: Program that computes area and perimeter of a rectangle
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    float width, height, area, perim;

    printf("Welcome to rectangle!\n");
    printf("Enter the width: ");
    scanf("%f", &width);
    printf("The width entered is %f!\n", width);
    printf("Enter the height: ");
    scanf("%f", &height);
    printf("The height entered is %f!\n", height);

    area = width * height;

    printf("The area is %f!\n", area);

    perim = (width + height) * 2;

    printf("The perimeter is %f!\n", perim);

    return(EXIT_SUCCESS);
}
/*
Expected output:
Welcome to rectangle!
Enter the width: 2
The width entered is 2.000000!
Enter the height: 3.5
The height entered is 3.500000!
The area is 7.000000!
The perimeter is 11.000000!
*/
```

- **7. 5. 6. Flow Control**
  - **Sequence**
    - Control flows from the top down, sequentially performing every action
    - Actions could be:
      - **Definitions**
        e.g. int i;
      - **Declarations**
        e.g. void my_function();
      - **Statements**
        e.g. i = 32 * (i+2)
      - **Blocks**
        e.g.
```
{
        Statement 1;
        Statement 2;
        …
        Statement N;
}
```
      - **Function calls**
        e.g. my_function();



- **Decisions**
  - Control flows sequentially on different paths depending on the result of a test
  - There are two different types of decisions:
    - **Binary decision**

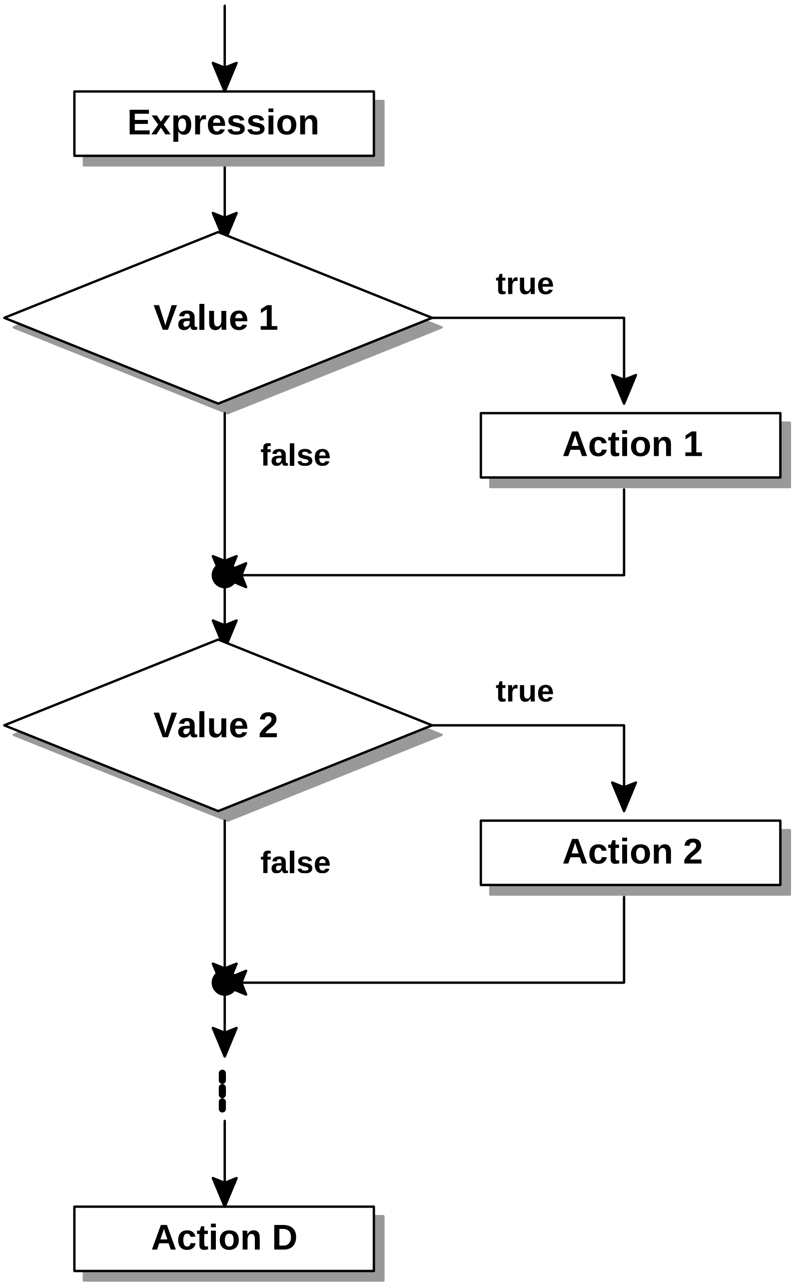
```
if (Expression)
        Action 1
else
        Action 2
```




```
e.g. if (i > 20)
                i = 20;
        else
                i = i + 1;

if (Expression)
                Action 1

e.g. if (i > 20)
                i = 20;
```

- **Multi-way decision**

```
switch (Expression)
{
        case Value 1: Action 1;
        case Value 2: Action 2;
        …
        default: Action D;
}
```



e.g.
```
switch (i)
{
case 0: printf("i is zero!\n");
case 1: printf("i is one!\n");
case 2: printf("i is two!\n");
default: printf("i is not 0,1 or 2!\n");
}
```

output for i equal to 0:

```
i is zero!
i is one!
i is two
i is not 0, 1 or 2!
```

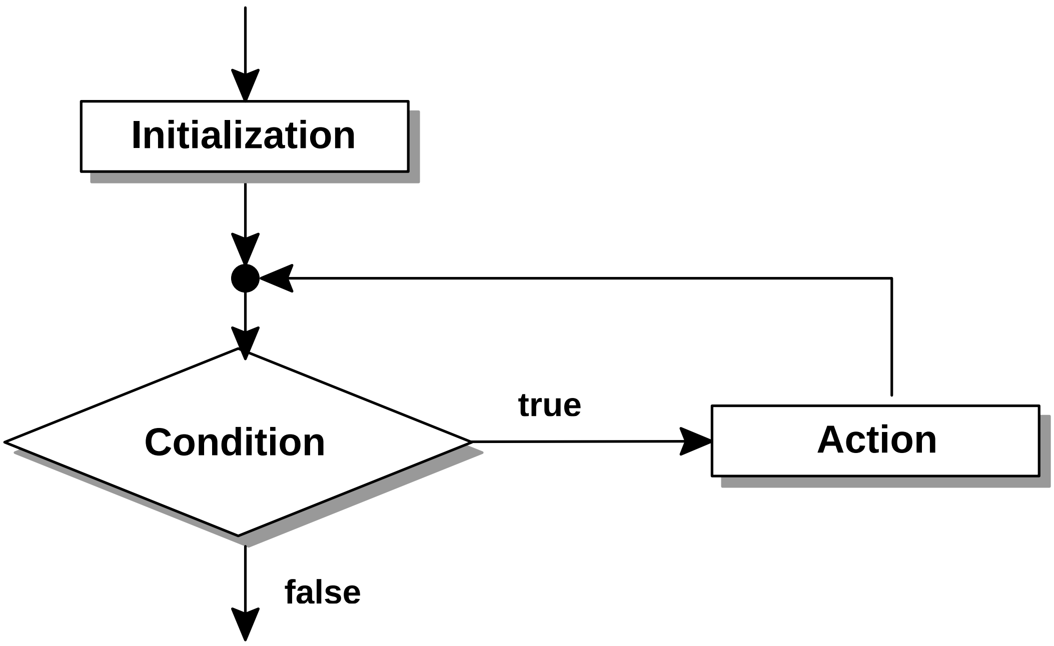
e.g.
```
switch (i)
{
case 0: printf("i is zero!"); break;
case 1: printf("i is one!"); break;
case 2: printf("i is two!"); break;
default: printf("i is not 0,1 or 2");
}
```

output for i equal to 0:

```
i is zero!
```

- **Iteration (Loops)**
  - Control flows repeatedly through the same sequence of actions
  - Iterations could be:
    - **Test-first iteration**

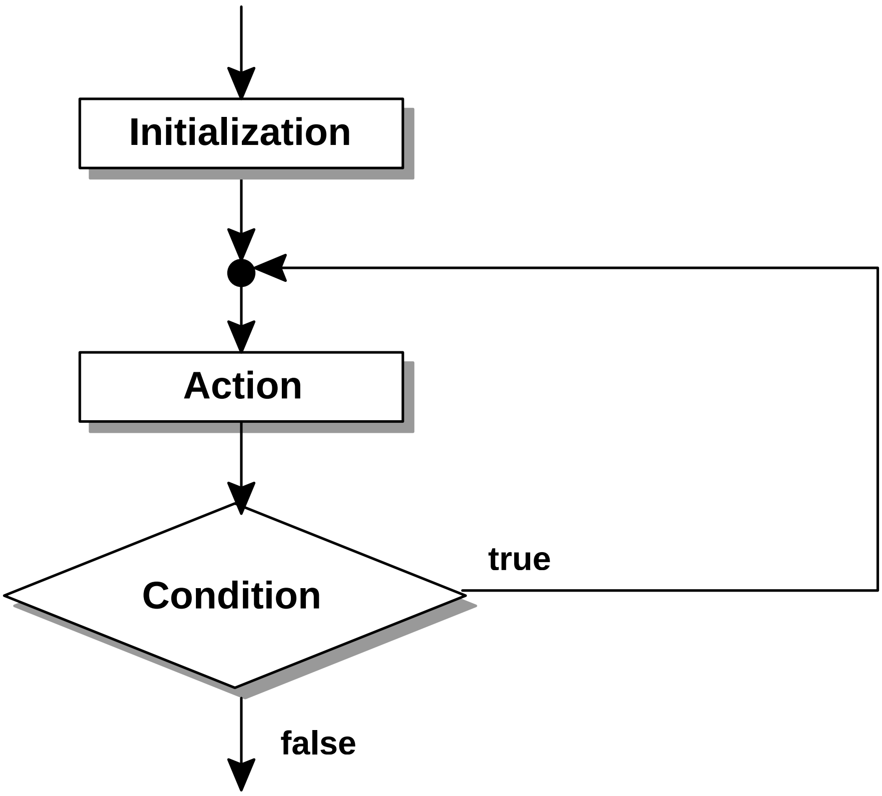
```
Initialization
while (Condition)
        Action
```




```
e.g.  i = 0;
      while (i < 10)
      {
              print("%d\n", i);
              i = i + 1;
      }

e.g.  i = 0;
      while (i < 10)
              i = i + 1;
      print("%d\n", i);
```

- **Test-last iteration**

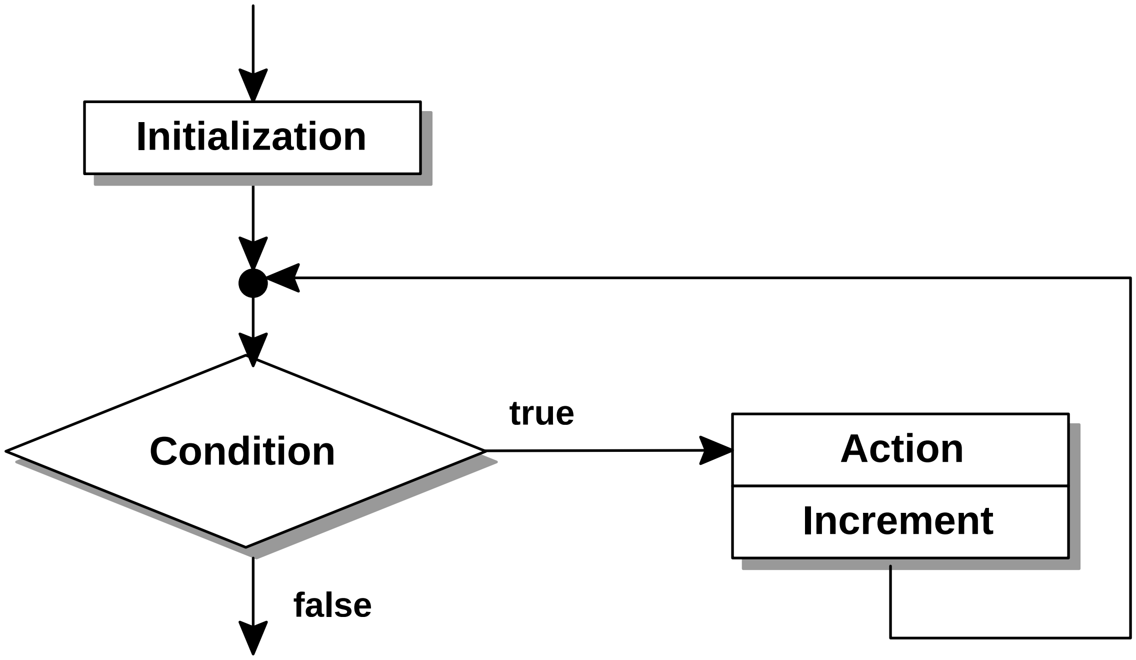
```
Initialization
do
      Action
while (Condition)
```




```
e.g.  i = 0;
      do
      {
            print("%d\n", i);
            i = i + 1;
      } while (i < 10);

e.g.  i = 10;
      do
      {
            print("%d\n", i);
            i = i + 1;
      } while (i < 10);
```

- **Complex iteration**

```
for (Initialization; Condition; Increment)
    Action
```



e.g.

```
for (i = 0; i < 10; i = i + 1)
    print("%d\n", i);
```

e.g.

```
for (i = 0; i < 10; i++)
{
    print("%d\n", i);
}
```

- ## 7. 5. 7. Statements

  - **Assignment**

    - **<u>variable = expression</u>**
      - The expression is evaluated and its value is stored in the location of memory associated with the variable

    - **<u>variable operator = expression</u>**
      - First the expression is evaluated
      - Then the operator is applied between the former value of the variable and the resulting value of the expression evaluation
      - Last the result obtained is stored in the location of memory associated with the variable

    - **<u>variable ++</u>**
      - First the value of the variable is incremented
      - Then this new value is stored in the location of memory associated with the variable

    - **<u>variable --</u>**
      - First the value of the variable is decremented
      - Then this new value is stored in the location of memory associated with the variable

- **Iteration (Loop)**

  - **<u>while</u>**

```
Initialization;
while(Condition)
{
      Statement 1;
      Statement 2;
      …
      Statement N;
      Update;
}
```

- First the **Initialization** is performed
- Then the **Condition** is evaluated
- In case of true, the **Body** is executed, otherwise is ignored
- The statements within the body are executed sequentially
- In general the last statement is an **Update** of one of the variables involved in the Condition such as the loop will end at some stage

  - **<u>do while</u>**

```
Initialization;
do
{
      Statement 1;
      Statement 2;
```

```
        …
        Statement N;
        Update;
}
while(Condition);
```

- o First the **Initialization** is performed
- o Next the statements within the **Body** are executed sequentially
- o In general the last statement is an **Update** of one of the variables involved in the Condition such as the loop will end at some stage
- o Then **Condition** is evaluated
- o In case of true, the body will be executed again, otherwise the loop will end

▪ **for**

```
for (Initialization; Condition; Update)
{
        Statement 1;
        Statement 2;
        …
        Statement N;
}
```

- o First the **Initialization** is performed
- o Then the **Condition** is evaluated
- o In case of true, the **Body** is executed, otherwise is ignored

- The statements within the body are executed sequentially
- Last the **Update** of one of the variables involved in the Condition is performed

- **Decision**
  - **<u>if</u>**

```
if (Condition)
{
      Statement_if_true 1;
      Statement_if_true 2;
      …
      Statement_if_true N;
}
else
{
      Statement_if_false 1;
      Statement_if_false 2;
      …
      Statement_if_false N;
}
```

- First the **Condition** is evaluated
- In case of true, the **Body_if_true** is executed, otherwise the **Body_if_false** is executed
- The statements within the selected body are executed sequentially

```
if (Condition)
{
    Statement_if_true 1;
    Statement_if_true 2;
    …
    Statement_if_true N;
}
```

- o First the **Condition** is evaluated
- o In case of true, the **Body_if_true** is executed, otherwise nothing is performed
- o The statements within the selected body are executed sequentially

- **switch**

```
switch (Expression)
{
    case value1: body_if_value1;
    case value2: body_if_value2;
    …
    default: body_default;
}
```

- o First the **Expression** is evaluated
- o The result is compared with **value1** and in case of match **body_if_value1** is executed
- o The result is then compared with **value2** and in case of match **body_if_value2** is executed, s.o.
- o At the end **body_default** is executed

```
switch (Expression)
{
      case value1: body_if_value1; break;
      case value2: body_if_value2; break;
      …
      default: body_default;
}
```

- First the **Expression** is evaluated
- The result is compared with **value1**
- In case of match **body_if_value1** is executed and **break** ends the switch statement
- If the values differ the execution continues with a comparison of the **Expression** evaluation result with **value2**
- In case of match **body_if_value2** is executed and **break** ends the switch statement
- Otherwise the execution of the switch statement continues, s.o.
- If non of the comparisons were successful, the **body_default** is executed

- **Other statements**
  - **break**
    - Ends the execution of any block of statements
  - **continue**
    - Ends the execution of the current loop and causes the flow to start another loop

- **Function calls**
  - **I/O functions**

    scanf("%flag", &variable);

    - Reads a value from the user and stores it in the location of memory associated with the **variable**

    printf("Formatted text %flag", variable);

    - Retrieves the value stored in the location of memory associated with the **variable** and displays it as part of the formatted text