Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Programming
  2. Variables, Types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. Structures
  6. Input and Output
- Basic system programming in Linux (10h)
Overview

1. Variables, Datatypes and Operators
   - Variables
   - Primitive data types
   - Operators
   - Data type conversion
   - Booleans

2. Control Flow
   - Loops
   - Hybrid
Motivation

- Most, if not all C programs contain **variables** that can be declared locally or globally.
- Their values are stored in a digital computer with certain accuracy, determined by their **type**.
- C has rich variety of **math operators** including +, −, ×, /, %, ++, and logical operators such as ==, !, >, <, ||, &&, to manipulate variables.
- **Control flow** determines the order in which statements and function calls are executed.
A **variable** is a name given to a storage area in the system’s memory that can be manipulated.

- For example, `int x=0, y=0; y=x+1`
- Variables `x, y;`
- Operator `+`.

**Rules for naming variables**
- can contain letters, digits and underscore
- first element must be either letter or underscore
- case sensitive
- **cannot** contain keywords.
The syntax to declare a variable is as follows:

```plaintext
<type> <name>_<variable> [=init value];
```

- type of the variable;
- name of the variable: name can have characters and digits; always start with a letter. Always keep in mind the general rules for naming variables and functions;
- you can define an init value for the variable. It is strongly suggested to always init variables.
Examples

- `int while`
Examples

- `int` while
- `int` my$number
Examples

- `int while`
- `int my$number`
- `int 2do`
Examples

- `int` while
- `int` my$number
- `int` 2do
- `int` you2
Examples

- int while
- int my$number
- int 2do
- int you2
- int my_number
Examples

- `int while` (incorrect due to keyword)
- `int my$number` (ok)
- `int 2do` (incorrect due to initial digit)
- `int you2` (ok)
- `int my_number` (ok)
There are four primitive data types

- **Integer** `int ∈ ℤ` and its derivative types.
- **Floating-point types** `double, float ∈ ℝ`.
- **Single characters** `char`.
**Primitive data types:** `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.
int: an integer (placeholder %d)

- size of values represented by int depends on the machine where your code is running.
- the predefined function sizeof() gives the length in bytes of any type of variable in C. For instance:

```c
#include <stdio.h>
int main()
{
    printf( "%d\n" , sizeof(int) );
}
```
**Primitive data types:** `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.
- the predefined function `sizeof()` gives the length in bytes of any type of variable in C. For instance:

```c
#include <stdio.h>
int main()
{
    printf("%d\n", sizeof(int));
}
```

Result

4
**Primitive data types: int**

**int**: an integer (placeholder `%d`)
- size of values represented by `int` depends on the machine where your code is running.
- the predefined function `sizeof()` gives the length in bytes of any type of variable in C. For instance:

```c
#include <stdio.h>
int main()
{
    printf("%d\n", sizeof(int));
}
```

Result

4

- the modifiers `short`, `long` and `long long` handle integers of different length.
signed int vs. unsigned int

signed and unsigned are used for numbers with and with no sign, respectively. According to ISO C docs:

- The int data type is signed and has a minimum range of at least $-32767$ through $32767$ (on a 16-bit machine). The actual values are given in limits.h as INT_MIN and INT_MAX respectively.

- An unsigned int has a minimal range of 0 through 65535 with the actual maximum value being UINT_MAX from that same header file.
Binary representation

- ISO C uses **two’s bit complement**.
- A 3-bit illustration:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Unsigned integer</th>
<th>Signed integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>-3</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>-2</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>-0</td>
</tr>
</tbody>
</table>
ISO C uses **two’s bit complement**.

A 3-bit illustration:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Unsigned integer</th>
<th>Signed integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>-3</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>-2</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>-0</td>
</tr>
</tbody>
</table>

**Note**

Be aware of underflow and overflow!
Primitive data types: reals

float, double: used to represent real numbers (single and double precision)

```c
float x=123.34;
double y=100.1e5;  // scientific notation
```
**Primitive data types: reals**

*float, double*: used to represent real numbers (single and double precision)

```c
float x = 123.34;
double y = 100.1e5; // scientific notation
```

- placeholders `%f` and `%lf`;
- `sizeof(float)` gives 4 bytes.
- `sizeof(double)` gives 8 bytes.

![32-bit representation according to IEEE 754](Source:wikipedia)

**Figure:** 32-bit representation according to IEEE 754 (Source:wikipedia)
**Primitive data types char**

**char**: a single byte representing a character (**ASCII** code). Placeholder `%c`.

```c
char a = 'a'; // chars are single quoted
```
**Primitive data types char**

**char**: a single byte representing a character (ASCII code). Placeholder `\%c`.

```c
char a = 'a'; // chars are single quoted
```

- **chars are integers in C.**
  ```c
  int a = 'a';
  printf("\%c\n", a);
  printf("\%d\n", a);
  ```

- Indeed, 97 corresponds to the ASCII code of 'a'
## Primitive data types: char (2)

Some char constants and their integer values:

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'a'</th>
<th>'b'</th>
<th>...</th>
<th>'z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>97</td>
<td>98</td>
<td>...</td>
<td>122</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'A'</th>
<th>'B'</th>
<th>...</th>
<th>'Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>65</td>
<td>66</td>
<td>...</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'0'</th>
<th>'1'</th>
<th>...</th>
<th>'9'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>48</td>
<td>49</td>
<td>...</td>
<td>57</td>
</tr>
</tbody>
</table>
### Primitive data types: char (2)

Some char constants and their integer values:

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'a'</th>
<th>'b'</th>
<th>...</th>
<th>'z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>97</td>
<td>98</td>
<td>...</td>
<td>122</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'A'</th>
<th>'B'</th>
<th>...</th>
<th>'Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>65</td>
<td>66</td>
<td>...</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char constant</th>
<th>'0'</th>
<th>'1'</th>
<th>...</th>
<th>'9'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>48</td>
<td>49</td>
<td>...</td>
<td>57</td>
</tr>
</tbody>
</table>

**Note**

There is no relationship between a char constant and its digit counterpart: '2' is not 2.
Data type size

On a typical 32-bit machine:

- `int` is 32 bits
- `long` is 32 bits
- `long long` is 64 bits

On a typical 64-bit architecture:

- `int` is 32 bits
- `long` is 32 or 64 bits
- `long long` is 64 bits

On both:

- `float` is 32 bits
- `double` is 64 bits (always!)
- `char` is 8 bits
- `signed char` is 8 bits
In other words,

- `sizeof(char) < sizeof(short) ≤ sizeof(int) ≤ sizeof(long)`
- `sizeof(char) < sizeof(short) ≤ sizeof(float) ≤ sizeof(double)`
- **Numerical data types span multiple bytes. Their order is relevant.**
LittleEndian: The least significant byte is stored in the lowest memory address, and increases address for each more significant byte. Typical representation in all x86 (intel) compatible processors.

Big endian: The most significant byte occupies the lowest memory address. Typical representation in ARM architectures.

(Source: wikipedia)
the attribute `const` can be applied to the declaration of any variable, with the effect of stating that its value will not change.

```cpp
const double pi = 3.141592;
const int five = 5;
```

**Note**

An attempt to modify constants typically ends up in a compiling error!

**Difference between `#define` and `const`??**
the attribute `const` can be applied to the declaration of any variable, with the effect of stating that its value will not change.

```plaintext
const double pi = 3.141592;
const int five = 5;
```

**Note**

An attempt to modify constants typically ends up in a compiling error!

**Difference between `#define` and `const`??**

- `#define` is a directive of the pre-processor and replaced in the source code before compilation;
- A variable defined as `const` is manipulated from the compiler: it has a type and an address.
Arithmetic operators

In C there are the following operators:

\[
+ \quad - \quad * \quad / \quad \% 
\]

representing the usual arithmetic operations. They are used to modify variables’ values.

The value of \( a \) modulo \( b \) is the remainder of the division between \( a \) and \( b \): for instance, \( 5 \% 3 = 2 \).

Modulo operator cannot be applied to float and double variables.
Operators obey to **precedence** and **associativity** rules, to establish how to evaluate an expression.
Operators obey to **precedence** and **associativity** rules, to establish how to evaluate an expression.

As usual, \(+\) and \(\) have the same precedence, lower than \(*\), \(/\) and \(\%\).
Operators obey to *precedence* and *associativity* rules, to establish how to evaluate an expression.

As usual, + and − have the same precedence, lower than *, / and %.

Moreover, addition and multiplication are both left and right associative, e.g. \((a \times b) \times c = a \times (b \times c)\) while subtraction and division, as used in conventional math notation, are inherently left-associative.
Arithmetic operators (3)

Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression `var op = expr`
  - is equivalent to `var = var op expr`
**Arithmetic operators (3)**

Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression \( var \ op = expr \)
  - is equivalent to \( var = var \ op \ expr \)
  - for instance: \( j*=i+2 \Leftrightarrow j=j*(i+2) \)
Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression `var op = expr`
  - is equivalent to `var = var op expr`
  - for instance: `j*=i+2 ⇔ j=j*(i+2)`

- **unitary increment/decrement operators**: comprising the operators `++` and `--`, respectively. They can be used either as prefix (before the variable: `++n`) or as suffix (after the variable: `n++`). The effect is the same, however:
  - `++n` execute the increment **before** using the value of `n`;
  - `n++` increments **after** using the value.
Data type conversion

- Type conversion occurs when the expression has data of mixed types.
- Common problem:

```java
double a = 1.2;
int b = 2;
double c = b/a;  // what is the precision of c? */
```
Data type conversion

- Type conversion occurs when the expression has data of mixed types.
- Common problem:

```java
double a = 1.2;
int b = 2;
double c = b/a;  // what is the precision of c?
```

- When an operator is applied to values having different types, they are converted to the same type using some automatic rules.
- Data type is promoted **from lower to higher** accuracy.
Type conversion rules

- \( f(\text{int}, \text{float}) \rightarrow f(\text{float}) \)
- \( f(\text{double}, \text{other}) \rightarrow f(\text{double}) \)
- If either operand is unsigned, the other shall be converted to unsigned i.e., \( f(\text{unsigned int}, \text{long}) \rightarrow f(\text{unsigned long}) \)
- **Promotion**: \( f(\text{unsigned char}, \text{unsigned short}) \rightarrow f(\text{unsigned int}) \)
Example

```c
short i = 1;
char ch = 'a';
printf("%zu,%zu,%zu\n", sizeof(i), sizeof(ch), sizeof(ch+i));
```
Example

```c
short i = 1;
char ch = 'a';
printf( "%zu,%zu,%zu\n", sizeof(i), sizeof(ch), sizeof(ch+i) );
```

2,1,4

Note

The type of `sizeof()` is `size_t` having format `%zu`. 
Forced Type Conversion

- occurs when the value of the **larger** data type is converted to the value of the **smaller** data type
- The result may have **lower** precision.
- Type casting is the preferred method of forced conversation.
Examples

Example 1:

```c
int e = 2.4;
pprintf("e = %d\n", e);
```

Examples

Example 1:

```c
int e = 2.4;
printf("e = %d\n", e);
```

e = 2
Examples

Example 1:

```c
int e = 2.4;
printf("e = %d\n", e);
```

\[ e = 2 \]

Example 2:

```c
float x = 12.4, y = 8.3, z = 4.7;
int result = x * y * z / 100;
printf("result = %d\n", result);
```

\[ result = 4 \]

During evaluation the integers would be \textbf{first promoted} to float and so would be the result, but \textbf{then occurs a truncation} to int.
Type Casting

Beside automatic conversions, it is possible to enforce conversions, by using casting, as follows:

\[(\text{type}) \; \text{expression};\]

Example:

```java
int sum, n;
float avg;
...
avg = sum/n; // integer division */
avg = (float)sum/n; // real numbers division */
```

The cast operator in parentheses has higher precedence, and it associates from right to left. Thus \((\text{float})\text{sum}/\text{n}\) is equivalent to \(((\text{float})\text{sum})/\text{n}\)
Booleans and relational operators

In C there does not exist Boolean type. It is represented through an `int`:

- 0 represents `FALSE`;
- A value different from 0 (typically 1) represents `TRUE`.
Booleans and relational operators

In C there does not exist Boolean type. It is represented through an int:

- 0 represents FALSE;
- A value different from 0 (typically 1) represents TRUE.

Logical operators:

- !: NOT (unary operator). Example: !a;
- &&: AND (binary operator). Example: a && b;
- ||: OR (binary operator). Example: a || b;

Returns an integer value: either 0 or 1, depending on the value (false/true) of the expression.
Booleans and relational operators

In C there does not exist Boolean type. It is represented through an `int`:

- 0 represents **FALSE**;
- A value **different from 0** (typically 1) represents **TRUE**.

Logical operators:

- `!`: NOT (unary operator). Example: `!a`;
- `&&`: AND (binary operator). Example: `a && b`;
- `||`: OR (binary operator). Example: `a || b`;

Returns an integer value: either 0 or 1, depending on the value (false/true) of the expression.

Other operators on **single bits**: shift operators (`<<, >>`), AND (`&`), OR (`|`), XOR (`^`) ...
Example

Differences between bitwise and logical AND operators in C

```c
int main() {
    int x = 3; // ...0011
    int y = 7; // ...0111

    if (y > 1 && y > x)
        printf("y is greater than 1 AND x\n");

    int z = x & y; // 0011
    printf("z = %d", z);
    return 0;
}
```

Output

```
y is greater than 1 AND x
z = 3
```
Relational Operators (2)

Checking for equality is essential in C

- The equality operator `==` compares primitive types such as char, int, float, etc.
  - e.g. `1==1` results in 1
  - e.g. `’A’==’a’` results in 0
- The inequality operator `!=` returns true if its operands are not equal, false otherwise.
  - e.g. `1!=1` results in 0
  - e.g. `’A’ !=’a’` results in 1
  - e.g. `0.999!=1` results in 1

Note

C cannot compare floating-point values due to rounding errors
Other relational operators are:

\[
< \quad > \quad <= \quad >=
\]

they are all binary: they take two expressions, and return a result of type int that can be either 0 or 1.

For instance, the expression \texttt{a<b}:
- if \texttt{a} is less than \texttt{b}, the value 1 (true);
- otherwise, the value is 0 (false).
Other relational operators are:

\[ < \quad > \quad <= \quad >= \]

they are all binary: they take two expressions, and return a result of type \texttt{int} that can be either \texttt{0} or \texttt{1}.

For instance, the expression \( a < b \):

- if \( a \) is less than \( b \), the value \( 1 \) (true);
- otherwise, the value is \( 0 \) (false).
What is the output of the following code?

```c
#include <stdio.h>
int main() {
    int const a = 5;
    a++;
    printf("a = %d", a);
}
```

1. a = 5
2. a = 6
3. Runtime error
4. Compile error
Control flow describes the order in which individual statements, instructions or function calls of our C program are executed.

- For example, $\min_u \sum_{i=1}^{10} x_i(u) \neq \sum_{i=1}^{10} \min u x_i(u)$.
- C provides two styles of flow control
Branching and Looping

- **Branching**: if, else and else if, switch, break and continue
- **Looping**: while, for, do-while
- **Hybrid**: goto (branching or looping).
The `if` Statement

```
if (test_condition_is_TRUE) {
    /* Do some stuff */
}
```

- test the condition
- if `TRUE`, evaluate body
- Otherwise, do nothing,
The **If** Statement

```c
if (test_condition_is_TRUE) {
    /* Do some stuff */
}
```

- test the condition
- if **TRUE**, evaluate body
- Otherwise, do nothing,

**Example:**

```c
int x = 3;
if (x%2) /* if condition is true */
    printf("The number is odd.");
```

The number is odd.
The `else` Keyword

```c
if (test_condition_is_TRUE) {
    /* Do some stuff */
}
else {
    /* test condition is FALSE */
}
```

- Optional
- `test expression is FALSE`
  - statement inside `if` body is `skipped`
  - statement inside `else` body is `executed`
The **Else** Keyword

```c
if (test_condition_is_TRUE) {
    /* Do some stuff */
}
else {
    /* test condition is FALSE */
}
```

- **Optional**
- **test expression is** `FALSE`
  - statement inside `if` body is **skipped**
  - statement inside `else` body is **executed**

**Example:**

```c
int x = 2;
if (x%2) printf("The number is odd.");
else printf("The number is even.");
```

The number is even.
The **Else if** Keyword

- Additional alternative control path

```c
if (test_condition_1_is_TRUE)  /* Do some stuff */
else if (test_condition_2_is_TRUE) /* Do sthg else */
else                            /* Do something else if all above false */
}
```

The number is zero.
The **Else if** Keyword

- Additional alternative control path

```c
if (test_condition_1_is_TRUE)  /* Do some stuff */
else if (test_condition_2_is_TRUE)  /* Do sthg else */
else  /* Do something else if all above false */
}
```

**Example:**

```c
int i=0;
if (i==0) printf("The number is zero.\n");
else if (i%2) printf("The number is odd.\n");
else printf("The number is non-zero and even.\n");
```

The number is zero.
The **switch** statement is alternative conditional.
The `switch` statement is alternative conditional. Syntax:

```java
switch (argument) {
    case label_1: instructions_1
        break;
    ...
    case label_n: instructions_n
        break;
    default : instructions_default
}
```
The `switch` statement (cont’d)

Semantic:

- **Input must be** `int` or `char`
- **The argument is evaluated** and compared against the different (constant) case labels;
- **when argument corresponds** to some case label, the respective instructions are executed, followed by a `break` to the next line following the `switch` statement;
- **otherwise,** (optional) `default` is executed.
The `switch` statement (cont’d)

Example:

```c
int day;
...
switch (day) {
    case 1: printf("Monday\n");
        break;  /* exit statement */
    case 2: printf("Tuesday\n");
        break;
    case 3: printf("Wednesday\n");
        break;
    case 4: printf("Thursday\n");
        break;
    case 5: printf("Friday\n");
        break;
    default: printf("Weekend\n");
}
```
The `switch` statement (cont’d)

Multiple cases:

```c
int day = 5;
...
switch (day) {
    case 1: /* break removed otherwise! */
    case 3:
    case 5:
    case 7: printf("Odd day\n");
        break;
    case 2:
    case 4:
    case 6: printf("Even day\n");
        break:
    default: printf("Invalid day\n");
}
```

Odd day
The **break** and **continue** **keywords**

- **break** keyword provides an early exit from `for`, `while` and `do`, just as from `switch`

```c
#include <stdio.h>
int main () {
    char c;
    while(1) /* infinite loop */
    {
        printf( "Shall we make a break? (y/n) " );
        c = getchar();
        if ( c == 'y' ) break;
    }
    return 0;
}
```

**Note**

Break works fine but *Shall we make a break? (y/n)* will be printed *2x*. Why?
The **break** and **continue** **keywords**

- The **continue** keyword skips rest of **for**, **while** and **do-while** loop.

```c
#include <stdio.h>
int main () {
    char c = 'n';
    while (1) {  /* infinite loop */
        puts("Shall we make a break? (y/n) ");
        scanf(" %c", &c);
        if (c == 'n') continue;
        if (c == 'y') break;
        printf("Your answer is unclear. ");
    }
    return 0;
}
```
Loops: The **while** loop

- A loop that executes a block of statements over and over again until a given condition returns **FALSE**.

```plaintext
while (test_condition_is_TRUE) {
/* sequence of statements */
}
```
#include <stdio.h>

int main() {
    char c = 'y'; /* Initialize to a value as true in while */
    while (c == 'y') {
        printf("Keep going ? (y/n) ");
        scanf("%c", &c);
    }
    return 0;
}

Output
Keep going ? (y/n) y
Keep going ? (y/n) n
$
Example (2)

```c
#include <stdio.h>
int main () {
    char c;
    while (1) { /* condition always true */
        printf("Gimme a char: ");
        scanf(" %c", &c);
    } /* do forever */
    return 0;
}
```

Output

Gimme a char: g
Gimme a char: f
...
Gimme a char: z
^C
The **for** loop

- A **counting** loop that executes a block of statements over and over again until a given condition returns `FALSE`.
- Internal counter in contrast to **while**-loop.

```python
for (initialization; test_condition; increment_or_decrement_counter)
{
    /* sequence of statements */
}
```

- Internal counter is only updated **after** the block of statements
  - true for both pre/post counter (`++counter/counter++`).
The *for* loop

- Some arguments of *for* function can be **empty**.

```c
for (int i = 0;; i++) { /* infinite loop */
}
```

```c
for (int i = 3;;) { /* keeps at i=3 */
}
```

- Multiple declarations are separated by **comma**.
- Expressions are evaluated left-to-right
### Example

```plaintext
for ( i=1, j=1; i<10; j*=i , i ++) {
    /* first j=j*i then i=i+1 */
}
```

```plaintext
for ( i=1, j=1; i<10; i ++ , j*=i ) {
    /* first i=i+1 then j=j*i */
}
```

<table>
<thead>
<tr>
<th>Counter variables</th>
<th>Counter variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>j= 1, i= 1</td>
<td>i= 1, j= 1</td>
</tr>
<tr>
<td>j= 1, i= 2</td>
<td>i= 2, j= 2</td>
</tr>
<tr>
<td>j= 2, i= 3</td>
<td>i= 3, j= 6</td>
</tr>
<tr>
<td>j= 6, i= 4</td>
<td>i= 4, j= 24</td>
</tr>
<tr>
<td>j= 24, i= 5</td>
<td>i= 5, j= 120</td>
</tr>
<tr>
<td>j= 120, i= 6</td>
<td>i= 6, j= 720</td>
</tr>
<tr>
<td>j= 720, i= 7</td>
<td>i= 7, j= 5040</td>
</tr>
<tr>
<td>j= 5040, i= 8</td>
<td>i= 8, j= 40320</td>
</tr>
<tr>
<td>j= 40320, i= 9</td>
<td>i= 9, j= 362880</td>
</tr>
</tbody>
</table>
```
The do-while loop

A do-while loop executes the body of the loop and only then tests some condition.

- will be executed at least once, even if the condition is FALSE.

```plaintext
do { /* execute statements */
} while (test_condition_is_TRUE);
```
#include <stdio.h>

int main () {
    int i = 4;
    do { /* in any case */
        printf("My integer: %d \n", i);
        i++;
    } while (i < 5);
    return 0;
}

Output
My integer: 4
$
Hybrid: The *goto* statement

- *goto* jumps unconditionally to a named location in the code, i.e.
- a **label** followed by a colon ":", that
- can be placed anywhere (within the same function).

```c
#include <stdio.h>
int main() {
    int a = 1;
    LOOP: do {
        if (a == 3) {
            a = a + 1; /* skip iterating */
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    } while (a < 5);
    return 0;
}
```
The `goto` statement

Compiling and executing the program, we obtain

```
$ gcc -Wall -o myprogram *.c
$ ./myprogram
value of a: 1
value of a: 2
value of a: 4
```
What is the output of the code?

```c
#include <stdio.h>
int main() {
    int a=0, i=0;
    for (i=0; i<3; i++) {
        a++
        continue;
    }
    printf("a = %d\n", a);
}
```
The keyword getting out of recursion is:

1. `break`
2. `return`
3. `exit`
4. Both 1) and 2)