COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Getting started

Università degli Studi di Padova

Agenda

- Variables and arithmetic expressions
- Output formatting
- Loops
    - Counter cycle
    - Cycle at initial condition
    - Cycle with final condition

Variables and Arithmetic Expressions

Using the formula °C=(5/9)(°F-32) print a table of Fahrenheit temperatures and their centigrade or Celsius equivalents, for values from 0 to 100 °F

The program can still consist of the definition of a single function named main.

Several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output are introduced

```
#include <stdio.h>
main()
{
            int fahr, celsius;
            fahr = 0;
            celsius = 5 * (fahr-32) / 9;
            printf("%df\t%dc\n", fahr, celsius);
            fahr = fahr + 1;
            celsius = 5 * (fahr-32) / 9;
            printf("%df\t%dc\n", fahr, celsius);
            …
}
```

In C, **all variables must be declared before they are used**, usually at the beginning of the function before any executable statements.

A declaration announces the properties of variables; it consists of a type and a list of variables

Variables and arithmetic expressions

The type int means that the variables listed are integers; by contrast with float, which means floating point, i.e., numbers that may have a fractional part.

The range of both int and float is machine-dependant.

Computation in the temperature conversion program begins with the **assignment** statements which set the variables to their initial values.

Individual statements are terminated by semicolons

```
        fahr = 0;
```

Other data types

C provides several other data types besides int and float, including:

char character - a single byte
short short integer
long long integer
double double-precision floating point

The size of these objects is also machine-dependent. There are also *arrays*, *structures* and *unions* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in the course.

Variables and Arithmetic Expressions

```
#include <stdio.h>
main()
{
        int fahr, celsius;
        fahr = 0;
        while (fahr <= 100) {
                celsius = 5 * (fahr-32) / 9;
                printf("%df\t%dc\n", fahr, celsius);
                fahr = fahr + 1;
        }
}
```

```
#include <stdio.h>
main()
{
        int fahr, celsius;
        int lower, upper;
        lower = 0;
        upper = 100;
        fahr = lower;
        while (fahr <= upper) {
                celsius = 5 * (fahr-32) / 9;
                printf("%df\t%dc\n", fahr, celsius);
                fahr = fahr + 1;
        }
}
```

Output

```
0f      -17c
1f      -17c
2f      -16c
3f      -16c
4f      -15c
5f      -15c
6f      -14c
7f      -13c
8f      -13c
9f      -12c
10f     -12c
………………..
96f     35c
97f     36c
98f     36c
99f     37c
100f    37c
```

Some problems

- The output isn't pretty because the numbers are not justified. We can augment each %d directive in the printf statement with a width, therefore the printed numbers will be right-justified. For example

    printf("%3d %6d\n", fahr, celsius);

So, the first number of each line will be printed in a field three digits wide, and the second in a field six digits wide

```
 0f   -17c
 8f   -13c
 9f   -12c
……………
10f   -12c
```

Output

```
0 f    -17.7778 c
1 f    -17.2222 c
2 f    -16.6667 c
3 f    -16.1111 c
4 f    -15.5556 c
5 f    -15.0000 c
6 f    -14.4444 c
7 f    -13.8889 c
8 f    -13.3333 c
9 f    -12.7778 c
10 f   -12.2222 c
………………………..
96 f   35.5556 c
97 f   36.1111 c
98 f   36.6667 c
99 f   37.2222 c
100 f  37.7778 c
```

Another (more serious) problem is that because we have used integer arithmetic, the Celsius temperatures are not accurate; for instance, 0°F is actually about -17.8°C, not -17. We shall use floating-point arithmetic instead of integer in order to produce more accurate outputs. Here the ouput produced by a second version using floating arithmetic

Output

```
0 f     -17 c
1 f     -17 c
2 f     -16 c
3 f     -16 c
4 f     -15 c
5 f     -15 c
6 f     -14 c
7 f     -13 c
8 f     -13 c
9 f     -12 c
10 f    -12 c
....................
96 f    35 c
97 f    36 c
98 f    36 c
99 f    37 c
100 f   37 c
```

```
 0f    -17.7778c
 1f    -17.2222c
 2f    -16.6667c
 3f    -16.1111c
 4f    -15.5556c
 5f    -15.0000c
 6f    -14.4444c
 7f    -13.8889c
 8f    -13.3333c
 9f    -12.7778c
10f    -12.2222c
..............................
96f    35.5556c
97f    36.1111c
98f    36.6667c
99f    37.2222c
100f   37.7778c
```

```c
#include <stdio.h>

int main()
{
        float celsius, fahr;
        float lower, upper;
        lower = 0;
        upper = 100;
        fahr = lower;
        while (fahr <= upper) {
                celsius = 5 * (fahr - 32) / 9;
                printf("%3.0ff\t%6.4fc\n", fahr, celsius);
                fahr = fahr + 1;
        }
}
```

With comments

```
#include <stdio.h>

int main()
{
        /* print Fahrenheit-Celsius table
        for fahr = 0, 1, ..., 100 */
        float celsius, fahr;
        float lower, upper;
        lower = 0;
        upper = 100;
        fahr = lower;
        while (fahr <= upper) {
                celsius = 5 * (fahr - 32) / 9;
                printf("%3.0ff\t%6.4fc\n", fahr, celsius);
                fahr = fahr + 1;
        }
}
```

Arithmetic operations

- If an arithmetic operator has integer operands, an integer operation is performed.
- If an arithmetic operator has one floating-point operand and one integer operand, the integer will be converted to floating point before the operation is done.

Writing floating-point constants with explicit decimal points emphasizes their floating-point nature for human readers.
The assignment ***fahr = lower;*** and the test ***while (fahr <= upper)*** work in the natural way - the int is converted to float before the operation is done.

Printf conversion specifications

The printf conversion specification %3.0f says that a floating-point number (here fahr) is to be printed at least 3 characters wide, with no decimal point and no fraction digits. %6.4f describes another number (celsius) that is to be printed at least 6 characters wide, with 4 digits after the decimal point. The output looks like this:

```
 96f   35.5556c
 97f   36.1111c
 98f   36.6667c
 99f   37.2222c
100f    37.7778c
```

Width and precision may be omitted from a specification.

Printf conversion specifications

SPECIFICATION	RESULT
%d	print as decimal integer
%4d	print as decimal integer, at least 4 characters wide
%f	print as floating point
%6f	print as floating point, at least 6 characters wide
%.4f	print as floating point, 4 characters after decimal point
%6.4f	print as floating point, at least 6 wide and 4 after decimal point

printf also recognizes %o for octal, %x for hexadecimal, %c for character, %s for character string and %% for itself.

Exercises

- *Modify the temperature conversion program to print a heading above the table.*
- *Write a program to print the corresponding Celsius to Fahrenheit table*.

print a heading above the table

```c
#include <stdio.h>

int main()
{
        float celsius, fahr;
        float lower, upper;
        lower = 0;
        upper = 100;
        fahr = lower;
        printf("Fahrenheit\tCelsius\n");
        while (fahr <= upper) {
                celsius = 5 * (fahr - 32) / 9;
                printf("%3.0ff\t\t%6.4fc\n", fahr, celsius);
                fahr = fahr + 1;
        }
}
```

```
Fahrenheit      Celsius
  0f            -17.7778c
  1f            -17.2222c
  2f            -16.6667c
  3f            -16.1111c
  4f            -15.5556c
  5f            -15.0000c
  6f            -14.4444c
  7f            -13.8889c
  8f            -13.3333c
  9f            -12.7778c
 10f            -12.2222c
```

print the Celsius to Fahrenheit table

```
#include <stdio.h>

int main()
{
        float celsius, fahr;
        float lower, upper;
        lower = 0;
        upper = 100;
        celsius = lower;
        printf("Celsius\tFahrenheit\n");
        while (celsius <= upper) {
                fahr = ((celsius * 9) / 5) + 32 ;
                printf("%3.0fc\t\t%6.4ff\n", celsius, fahr);
                celsius = celsius + 1;
        }
}
```

```
Celsius      Fahrenheit
  0c            32.0000f
  1c            33.8000f
  2c            35.6000f
  3c            37.4000f
  4c            39.2000f
  5c            41.0000f
  6c            42.8000f
  7c            44.6000f
  8c            46.4000f
  9c            48.2000f
 10c            50.0000f
```

print the Celsius to Fahrenheit table

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
        float celsius, fahr;
        float lower, upper;
        lower = 0;
        upper = 100;
        celsius = lower;
        while (celsius <= upper) {

                // Convert Celsius to Fahrenheit
                fahr = (celsius * 9.0 / 5.0) + 32;

                // Print the result
                cout << fixed << setprecision(2);
                cout << setw(6) << celsius << "c: " << setw(6) << fahr << "f" << endl;
                celsius++;
        }
        return 0;
}
```

```
  0.00c:  32.00f
  1.00c:  33.80f
  2.00c:  35.60f
  3.00c:  37.40f
  4.00c:  39.20f
  5.00c:  41.00f
  6.00c:  42.80f
  7.00c:  44.60f
  8.00c:  46.40f
  9.00c:  48.20f
 10.00c:  50.00f
 11.00c:  51.80f
 12.00c:  53.60f
 13.00c:  55.40f
 14.00c:  57.20f
 15.00c:  59.00f
 16.00c:  60.80f
 17.00c:  62.60f
 18.00c:  64.40f
 19.00c:  66.20f
 20.00c:  68.00f
```

Loops

Each line of the temperature table is computed in the same way, so we use a loop that repeats once per output line; this is the purpose of the while loop

```
while (fahr <= upper) {
...
}
```

The while loop operates as follows: The condition in parentheses is tested. If it is true, the body of the loop is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false the loop ends, and execution continues at the statement that follows the loop.

The body of a while loop may be one or more statements enclosed in braces, or a single statement without braces.

Loops

A cycle executes a block of instructions, called a cycle block (composed of the instructions instructionLoop1, instructionLoop2, ..), 0, 1 or more times, depending on the value of a **cond** condition.

The program evaluates a condition. As long as the condition is true, the program executes the loop block and checks the condition again.

When the condition becomes false, the program ends the execution of the block.

A single execution of the loop block is called an **iteration**.

For example, when the loop block is executed three times, it is said that three iterations have been executed.

A function (or algorithm) containing a loop is said to be iterative.

Loops

Of course, in the loop block we shall insert an instruction that shall make the condition cond false sooner or later.

If this instruction was not there, the condition would always be true and therefore the cycle would be repeated an infinite number of times.

There are two particular situations regarding the value of the cond condition:

1. The cond condition is false immediately, that is, when the program checks it for the first time. In this case, the block is executed 0 or 1 time, depending on the type of loop (as we will see later in the slides).
2. The cond condition is always true. In this case, there is an **infinite** loop, because the program always executes the block and, therefore, will never execute instructions placed after the end of the block.

Loops

Sometimes we could purposely design an infinite loop. But, quite often, there is an infinite loop because we have made a logical error.

An iterative function requires more complicated tracing than a generic function because it modifies the same variables multiple times. For this reason, we need to set up its trace table by considering the following rules:

- We need to insert a column into the table containing the value of the loop condition.
- We must insert a row into the table for each iteration, where we insert the values of the variables or expressions modified by the iteration.

We will see some examples later in the course.

Loops

In C programming, loops are control flow statements that repeatedly execute a block of code based on a condition. C provides three primary loop constructs: for, while, and do-while loops corresponding to the following three types of cycles.

- **Counter cycle**.

- **Cycle at initial condition**.

- **Cycle with final condition**.

Loops

Counter cycle.

- We use it when we know the number of iterations before executing the loop.
- The **for** loop is used when the number of iterations is known before entering the loop. It consists of three parts: initialization, condition, and update.

```
for (initialization; condition; update) {
    // Code to execute in each iteration
}
```

Example

```
int i;
for (i = 0; i < 5; i++) {
    printf("%d ", i);
}
```

Loops

- **Counter cycle**.
  - Initialization: typically sets a counter variable, which executes only once.
  - Condition: checked before every iteration. If it evaluates to true, the loop continues. If it’s false, the loop terminates.
  - Update: executed after each iteration, often used to modify the loop variable.

Loops

- **Cycle at initial condition**.
  - We must use it when:
    - we don't know the number of iterations, and
    - there is at least one input that the loop does not have to process even once
    - the **while** loop executes as long as a specified condition is true.

```
while (condition) {
  // Code to execute as long as condition is true
}
```

Example:

```
int i = 0;
while (i < 5) {
    printf("%d ", i);
    i++;
}
```

This loop will print numbers from 0 to 4.

Loops

- **Cycle with final condition**.
    - We must use it when:
        - we don't know the number of iterations, and
        - the loop must process each input to the problem at least once
        - the **do-while** loop is similar to the while loop, but it guarantees that the loop body is executed at least once, even if the condition is false.

```
do {
   // Code to execute
} while (condition);
```

Example:

```
int i = 0;
do {
    printf("%d ", i);
    i++;
} while (i < 5);
```

The for statement

There are many different ways to write a program for a particular task. Let's try a variation about the temperature converter.

```
#include <stdio.h>
/* print Fahrenheit-Celsius table */
main()
{
        float fahr;
        for (fahr = 0; fahr <= 100; fahr = fahr + 1)
                printf("%3.0ff\t%6.4fc\n",fahr, (5.0/9.0)*(fahr-32));
}
```

This produces the same answers, but it certainly looks different

The for statement

The syntax of the for statement is

```
for (statement1; cond2; statement3) {
        /* This is where the for block begins. */
        instructionLoop1;
        instructionLoop2;
        ..
        } /* for */
```

cond2 is a condition. statement1, cond2 and statement3 are optional.

The effect of the for loop is:

The program executes statement1.

As long as cond2 is true, the program executes the loop block and then executes statement3. When the loop block has only one statement, we can eliminate its curly braces.

The for statement: some hints

Use the for loop as follows:

Use statement1 to initialize a counter i to an initial value;

Use cond2 to check if counter i is less than the final value;

Use statement3 to increment (or decrement) the counter i.

```
for (i = ini; i < end; i++) {
        /* The for block begins here. */
        instructionLoop1;
        instructionLoop2;
        ..
        } /* for */
```

The program loops for every i from ini to end - 1 inclusive.

The for statement: some hints

So, if ini < end, the program executes the loop end – ini times.

If, ini >= end, the program does not execute the loop even once.

The loop defined using the for statement is called a counter loop, because the variable i "counts" the iterations performed.

If we use the variable i only to count the iterations, it is better to initialize it with the value 0, for a reason that you will see when studying arrays.

So, to execute a loop n times, we should use the loop:

```
for (i = 0; i < n; i++) {
        ..
        } /* for */
```

The do – while loop

We have already encountered the while and for loops.

```
        while (expression)
                statement
```

the expression is evaluated. If it is non-zero, statement is executed and expression is reevaluated.

This cycle continues until expression becomes false, at which point execution resumes after statement.

```
        for (expr1; expr2; expr3)
                statement
```

the while and for loops test the termination condition at the top.

By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

The do – while loop

The syntax of the do – while loop is

```
    do
        statement
    while (expression);
```

The statement is executed, then expression is evaluated. If it is true, statement is executed again, and so on. When the expression becomes false, the loop terminates.

Experience shows that do-while is much less used than while and for. Nonetheless, from time to time it is valuable

break and continue statements

break: exits the loop immediately, regardless of the condition.

Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i equals 5
    }
    printf("%d ", i);
}
```

This loop will print numbers from 0 to 4 and then stop.

break and continue statements

continue: Skips the remaining code in the current iteration and moves to the next iteration of the loop.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skip the rest of the loop body when i equals 5
    }
    printf("%d ", i);
}
```

This will print numbers from 0 to 9 but skip 5.

Summary Table

Loop Type	Condition Check	Execution if False
for	Before iteration	Never executes
while	Before iteration	Never executes
do-while	After iteration	Executes once

These loops allow flexible control over how many times a code block runs, and combined with **break** and **continue**, they give you powerful tools for managing the flow of your programs.

C++ loops

C++ offers similar loop constructs to those in C, with some additional features that enhance flexibility and ease of use. In C++, loops include for, while, and do-while, just like in C, but C++ also introduces the **range-based for loop** for easier iteration over collections.

```
for (int i = 0; i < 5; i++) {
    std::cout << i << " ";
}
----------------------------------------------
int i = 0;
while (i < 5) {
    std::cout << i << " ";
    i++;
}
```

```
int i = 0;
do {
    std::cout << i << " ";
    i++;
} while (i < 5);
```

Range-based for loop (C++11 and later)

The **range-based for loop** simplifies iteration over collections such as arrays, vectors, or other iterable containers. It's especially useful for iterating over elements without manually managing the index.

```
for (element_type element : collection) {
   // Code to execute on each element
}
```

Example

```
std::vector<int> nums = {1, 2, 3, 4, 5};
for (int num : nums) {
   std::cout << num << " ";
}
```

This loop will print the elements of the vector: 1 2 3 4 5.

Range-based for loop (C++11 and later)

- Element type: Automatically inferred using auto, or explicitly declared.
- Collection: The range or container that is being iterated over.

Example with auto:

```
for (auto num : nums) {
    std::cout << num << " ";
}
```

As in C, C++ uses break and continue to control the flow within loops.

- **break**: exits the loop immediately, regardless of the loop's condition.
- **continue**: skips the remaining code in the current iteration and moves to the next iteration of the loop.

break statement

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i equals 5
    }
    std::cout << i << " ";
}
```

This loop will print numbers from 0 to 4 and then stop.

continue statement

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skip the rest of the loop body when i equals 5
    }
    std::cout << i << " ";
}
```

This will print numbers from 0 to 9, skipping 5.

Summary Table

Loop Type	Condition Check	Execution if False
for	Before iteration	Never executes
while	Before iteration	Never executes
do-while	After iteration	Executes once
rNpotange-based for	Not applicable	Iterates over a collection

The range-based for loop is a major improvement in C++ for working with containers or collections, making it easier to write clean and expressive code. Otherwise, the loop constructs in C++ are quite similar to C.