CH-230-A

Programming in C and C++

C/C++

Lecture 2

Dr. Kinga Lipskoch

Fall 2019

Type Conversions

- When data of different types are combined (via operators) some rules are applied
- Types are converted to a common type
  - Usually, to the larger one (called promotion)
  - **Example:** while summing an `int` and a `float`, the `int` is converted into a `float` and then the sum is performed
- A demotion is performed when a type is converted to a smaller one
  - **Example:** a function takes an `int` parameter and you provide a `float`
- A demotion implies possible loss of information
- Therefore, be careful with what to expect
  - In the above example, the fractional part will be lost

Casting

- It is possible to overcome standard conversions (casting)
- To force to a different data type, put the desired data type before the expression to be converted
  (type name) expression
- Casting is a unary operator with high precedence

Casting: An Example

```
1       int a;
2       float f1 = 3.456;
3       float f2 = 1.22;
4       /* these operations imply demotions */
5       a = (int) f1 * f2;     /* a is now 3 */
6       a = (int) (f1 * f2);   /* a is now 4 */
```

Incrementing and Decrementing

- The unary operators ++ and -- can be applied to increase or decrease a variable by 1

```
1  int a, b;
2  a = b = 0;
3  a++; b-- ; ++a ; --b;
```

- Note that they can be both prefix and postfix operators
  - The two versions are different

Prefix and Postfix Modes

- Prefix means that first you modify and then you use the value
- Postfix means that first you use and then you modify the value
- `int a = 10, b;`

Expression	New value of `a`	New value of `b`
`b = ++a;`	11	11
`b = a++;`	11	10
`b = --a;`	9	9
`b = a--;`	9	10

The `sizeof()` Operator

- `sizeof()` returns the number of bytes needed to store a specific object
- Useful for determining the sizes of the different data types on your system

```
1 int a;
2 printf("size int %lu\n", sizeof(a));
3 printf("size float %lu\n", sizeof(float));
4 printf("size double %lu\n", sizeof(double));
```

- For strings do not confuse `sizeof()` with `strlen()`
- Compile-time operator, will not work for dynamically allocated memory

Boolean Variables

- A boolean variable can assume only two logic values: **true** or **false**
- Boolean variables and expressions are widely used in computer languages to control branching and looping
- Some operators return boolean values
- A boolean expression is an expression whose value is **true** or **false**

Boolean Operators

- Boolean operators can be applied to boolean variables
  - AND, OR, NOT

A	NOT A	A	B	A AND B	A	B	A OR B
false	true	false	false	false	false	false	false
true	false	false	true	false	false	true	true
		true	false	false	true	false	true
		true	true	true	true	true	true

Booleans in C

- Originally, C did not provide an ad-hoc boolean type but uses rather the `int` type
- 0 is false, everything different from 0 is true
- In C99 the type `_Bool` was introduced, example: `_Bool b = 0;`
- Additionally, the library `stdbool.h` defines the type `bool`, example: `bool b = false;`
- C also provides the three Boolean operators
  - `&&` for AND,
  - `||` for OR,
  - `!` for NOT
- Applied to booleans they return booleans

Boolean Operators: Example

```
1   int main() {
2     int a, b, c;
3     a = 0;              /* a is false */
4     b = 57;             /* b is true */
5     c = a || b;         /* c is true */
6     c = a && b;         /* c is false */
7     a = !a;             /* a is now true */
8     c = a && b;         /* c is now true */
9     c = (a && !b) && (a || b);
10    return 0;
11  }
```

Relational Operators

- Relational operators are applied to other data types (numeric, character, etc.) and produce boolean values

  `(b > 5) --> true`
- Relational operators with boolean operators produce boolean expressions

  `(b > 5) && (a < 1) --> true && false --> false`

Relational operator	Meaning
==	Equality test
!=	Inequality test
>	Greater
<	Smaller
>=	Greater or equal
<=	Smaller or equal

Relational Operators: Example

```
1   int main() {
2     int a = 2, b, c;
3     float f1 = 1.34;
4     float f2 = 3.56;
5     char ch = 'D';
6     b = f1 >= f2;
7     c = !b;
8     b = c == b;
9     b = b != c;
10    c = f2 > a;
11    c = ch > a;
12    return 0;
13  }
```

Branching

- Up to now programs seem to execute all the instructions in sequence, from the first to the last (a linear program)
- Change the control flow of a program with branching statements
- Branching allows to execute (or not to execute) certain parts of a program depending on boolean expressions or conditions

Selection: `if ... else`

- In general selection constructs allow to choose a way in a binary bifurcation
- De facto you can use it in three ways
  - `if ()` single selection
  - `if ()`
    `else` double selection
  - `if ()`
    `else if ()`
    `else if ()`
    `...`
    `else` multiple selection

The `if` Syntax (1)

- General syntax:

```
1 if (condition)
2   statement 1;
3 else
4   statement 2;
5 other_statement; /* always executed */
```

- The else part can be omitted
- Statement: single statement or multiple statements
- Multiple statements need to be surrounded by braces { }

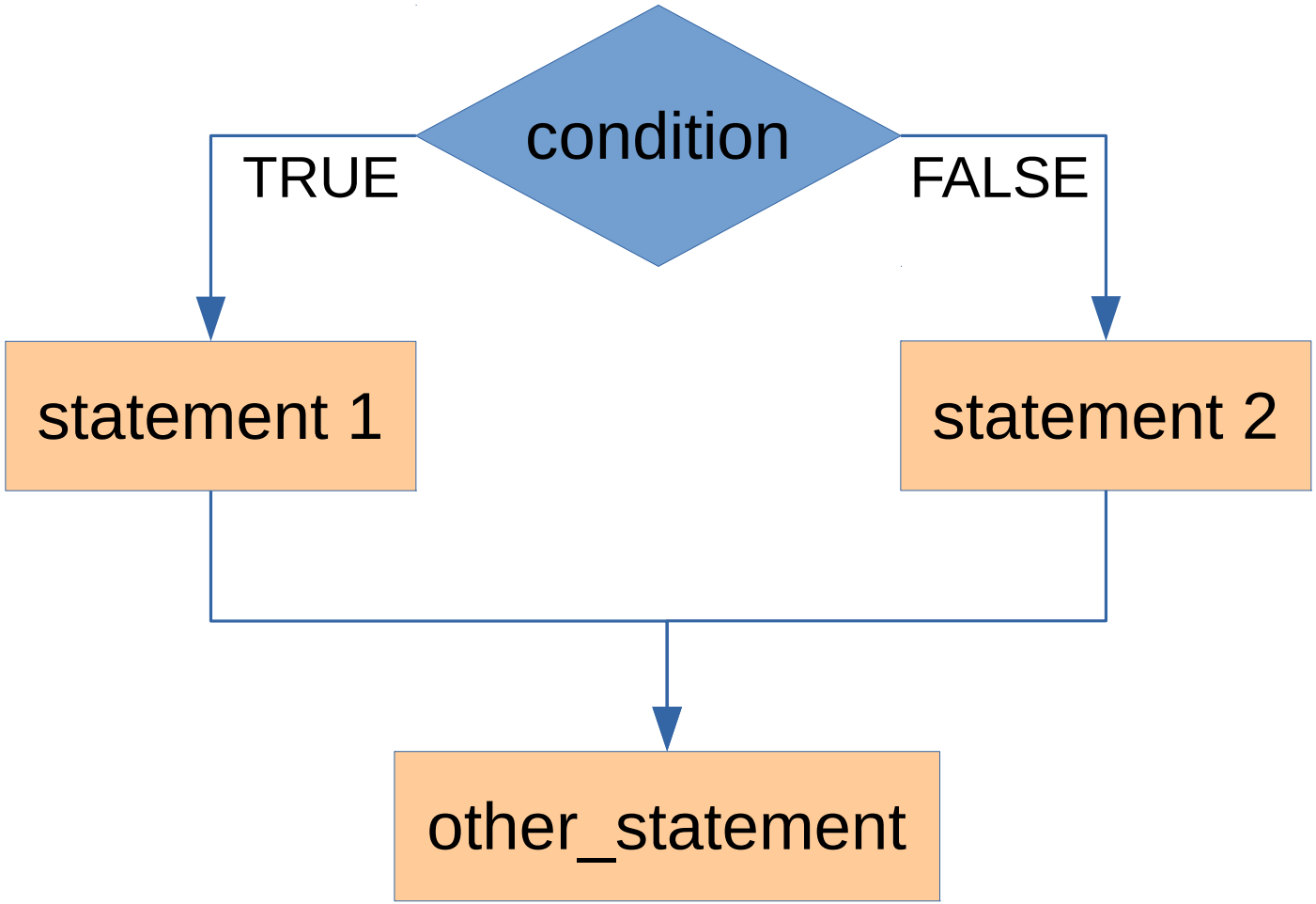
The `if` Syntax (2)

- Preferred syntax (always use braces)

```
1 if (condition) {
2   statements;
3 }
4 else {
5   statements;
6 }
```

- If you add statements, program flow is not changed (less errors)
- Using indentation, you can easily see where block starts and ends

`if`: Flow Chart



if: Example

```
1  #include <stdio.h>
2  int main() {
3    int first, second;
4    printf("Type the first number:\n");
5    scanf("%d", &first);
6    printf("Type the second number:\n");
7    scanf("%d", &second);
8    if (first > second) {
9      printf("The larger one is %d\n", first);
10   }
11   else {
12     printf("The larger one is %d\n", second);
13   }
14   printf("Can you see the logical error?\n");
15   return 0;
16 }
```

Statements and Compound Statements

- Statements can be grouped together to form compound statements
- A compound statement is a set of statements surrounded by braces

```
1 int a = 3;
2 if (a > 0) {
3   printf("a is positive %d\n", a);
4   a = a - 2 * a;
5   printf("now a is negative %d\n", a)
6 }
```

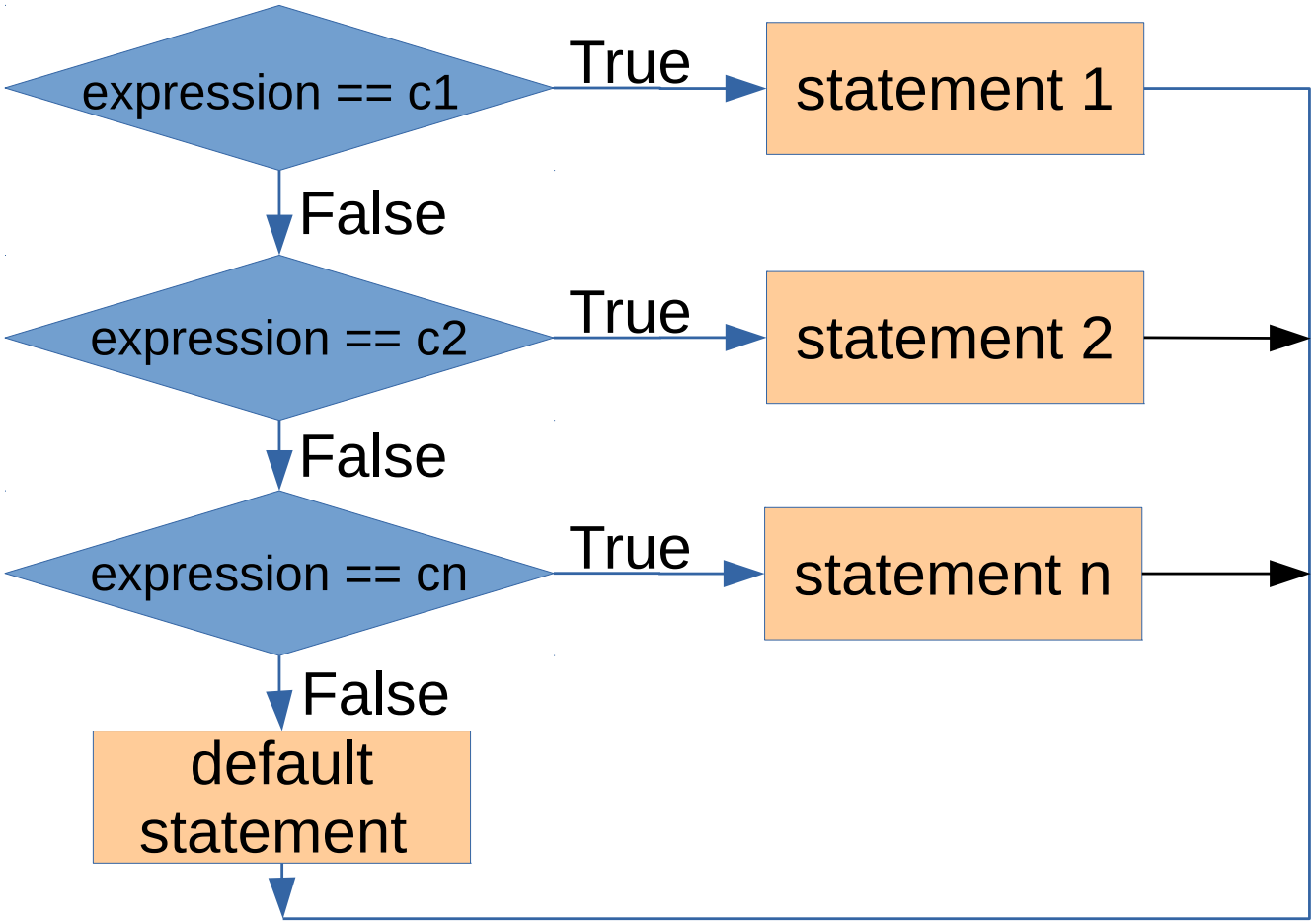
Multiple Choices: `switch`

- `switch` can be used when an expression should be compared with many values
- The same goal can be obtained with multiple `if`'s
- The expression must return an integer value

switch: The Syntax

```
1  switch (expression)  {
2    case c1:
3      statement1;
4      break;
5
6    case c2:
7      statement2;
8      break;
9
10   ...
11
12   default:
13     default_statement;
14 }
```

`switch`: Flow Chart



switch: Example

```
1  #include <stdio.h>
2  int main() {
3    int c;
4    for (c = 0; c <= 3; c++) {
5      printf("c: %d\n", c);
6
7      switch (c) {
8        case 1:
9          printf("Here is 1\n");
10         break;
11       case 2:
12         printf("Here is 2\n");
13         /* Fall through */
14       case 3:
15       case 4:
16         printf("Here is 3, 4\n");
17         break;
18       default:
19         printf("Here is default\n");
20     }
21   }
22   return 0;
23 }
```

Iterations

- In many cases it is necessary to repeat a set of operations many times
- Example: compute the average grade of the exam
  - Read all the grades, and sum them
  - Divide the sum by the number of grades
- C provides three constructs

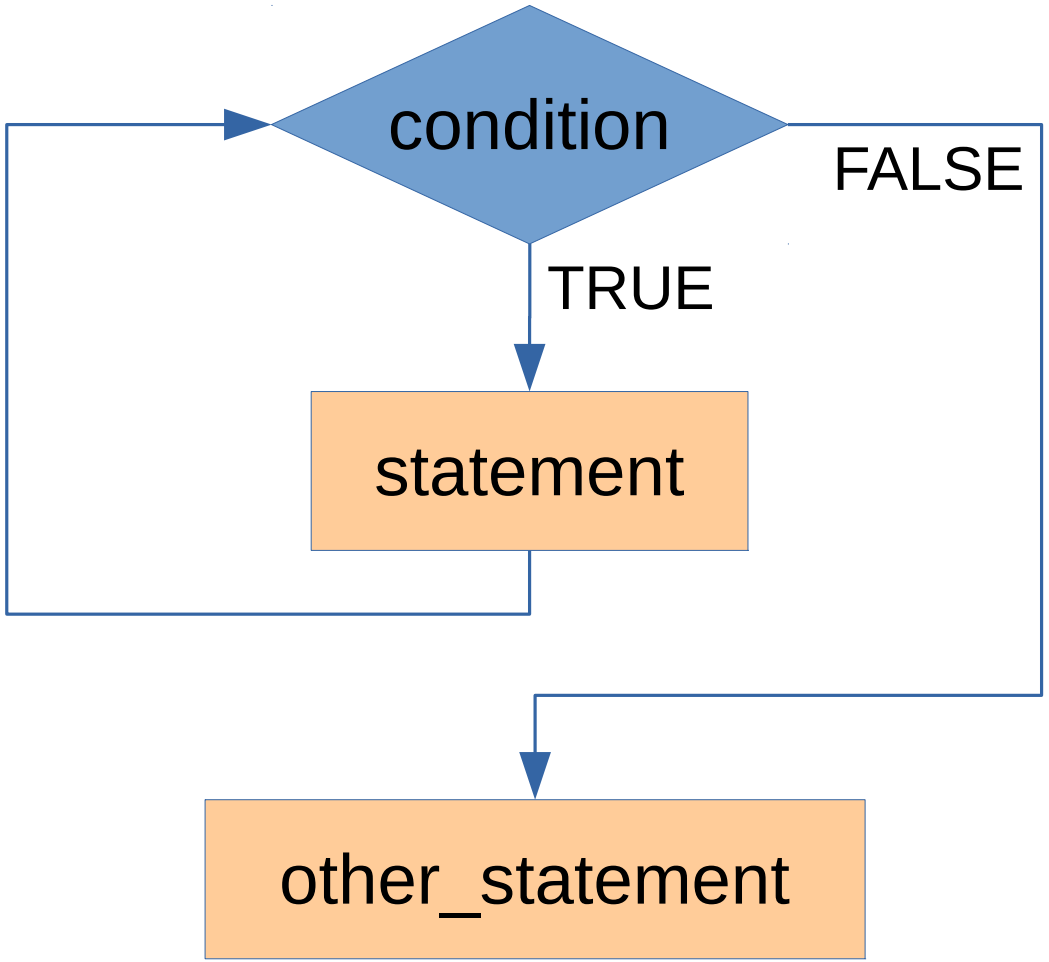
Iterations: `while`

- General syntax:

```
1 while (condition) {
2   statement;
3 }
```

- Keep executing the statement as long as the condition is true

`while`: Flow Chart



Example:
Compute the Sum of the First n Natural Numbers

```
1  #include <stdio.h>
2  int main() {
3     int idx, n, sum = 0;
4     printf("Enter a positive number ");
5     scanf("%d", &n);
6     idx = 1;
7     while (idx <= n) {
8        sum += idx;
9        idx++;
10    }
11    printf("The sum is %d\n",sum);
12    return 0;
13 }
```

Iterations: `for`

- General syntax:

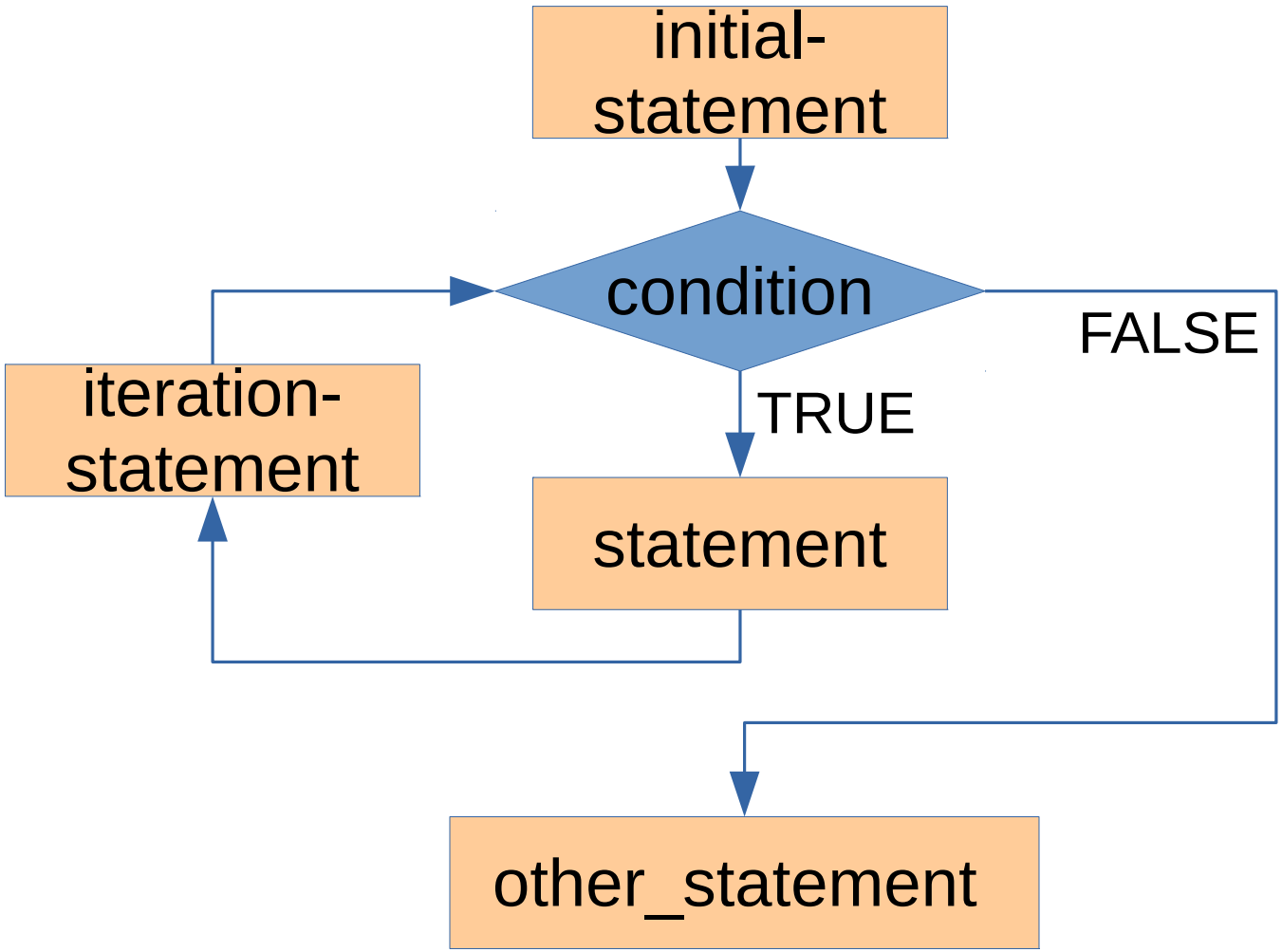
```
for (initial-statement; condition; iteration-
    statement)
  statement;
```

- Example:

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

- The `for` and `while` loops can be made interchangeable

`for`: Flow Chart



for: Example Revised

```
1  #include <stdio.h>
2  int main() {
3    int idx, n, sum = 0;
4    printf("Type a positive number ");
5    scanf("%d", &n);
6    for (idx = 1; idx <= n; idx++) {
7      printf("Processing %d..\n", idx);
8      sum += idx;
9    }
10   printf("The sum is %d\n", sum);
11   return 0;
12 }
```

Boolean Operators and if

```
1  for (n = 0; n < 3; n++) {
2    for (i = 0; i < 10; i++) {
3      if (n < 1 && i == 0) {
4        printf("n is < 1, i is 0\n");
5      }
6      if (n == 2 || i == 5) {
7        printf("HERE n: %d i:%d\n", n, i);
8      }
9      else {
10       printf("n:%d, i:%d\n", n, i);
11     }
12   }
13 }
```

Easier or Harder to Read?

```
1 for (n = 0; n < 3; n++)
2   for (i = 0; i < 10; i++) {
3     if (n < 1 && i == 0) {
4       printf("n is < 1, i is 0\n"); }
5       if (n == 2 || i == 5) {
6        printf("HERE n: %d i:%d\n", n, i); }
7       else {
8         printf("n:%d, i:%d\n", n, i); }}}
```

Iterations: do ... while

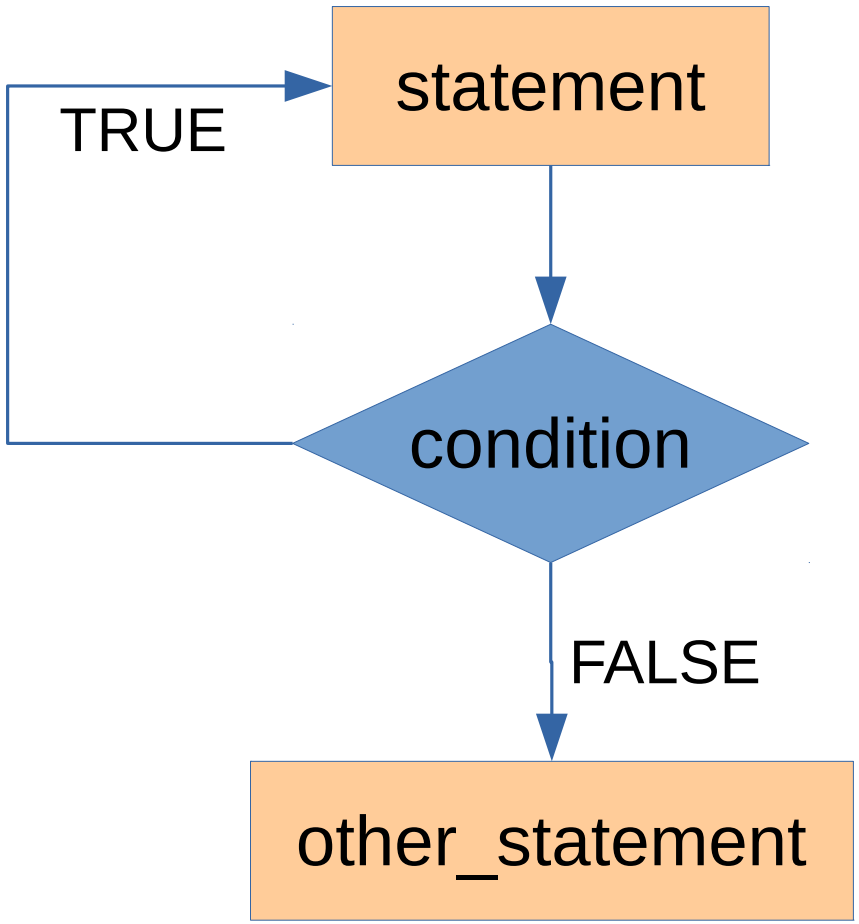
- General syntax:

```
1  do
2     statement;
3  while (condition);
```

```
1 do {
2    statement1;
3    statement2;
4 } while (condition);
```

- In this case the end condition is evaluated at the end
- The body is always executed at least once

`do ... while`: Flow Chart



do ... while: Example

```c
#include <stdio.h>
int main() {
  int n, sum = 0;
  do {
    printf("Enter number (<0 ends)");
    scanf("%d", &n);
    sum += n;
  } while (n >= 0);
  sum -= n; /* Remove last negative value */
  printf("The sum is %d\n", sum);
  return 0;
}
```

Jumping Out of a Cycle: `break`

- The keyword `break` allows to jump out of a cycle when executed
- We have already seen this while discussing `switch`

```
1 int num, i = 0;
2 scanf("%d", &num);
3 while (i < 50) {
4   printf("%d\n", i);
5   i++;
6   if (i == num)
7     break;
8 }
```

Jumping Out of a Cycle: `continue`

- `continue` jumps to the expression governing the cycle
- The expression is evaluated again and so on

```
1  char c;
2  /* code assumes that the input is
3     provided in one line like:
4     "abf23cdef" followed by enter */
5  while ((c = getchar()) != '\n') {
6    // ignore the letter b
7    if (c == 'b')
8      continue;
9    printf("%c", c);
10 }
```

Jumping Out of a Cycle

- Do not abuse `break` and `continue`
- You can always obtain the same result without using them
  - This at the price of longer coding
- By using them your code gets more difficult to read
- When you are experienced you will master their use
  - Meanwhile, learn the basics

Iterations: General Comments

- Inside the body of the loop you must insert an instruction that can cause the condition to become false
- If you do not do that, your program will fall into an infinite loop and will be unable to stop (Press `Ctrl-C` to stop such a program)
- `do ... while` is far less used than `while` and `for`
- The same constructs are provided in the majority of other programming languages