## 2.4 Control statements

*We are what we repeatedly do. Excellence, then, is not an act but a habit.*

*Will Durant in a summary of Aristotle's ideas,*
*The Story of Philosophy: The Lives and Opinions*
*of the World's Greatest Philosophers (1926)*

*This section introduces four concepts to control the execution of a program:* selection, iteration, blocks, *and* jumps. *These concepts enable us to deviate from the default linear control flow which executes statement by statement from top to bottom. You will learn how these concepts are implemented in C++, and how to apply them to create interesting programs.*

The programs we have seen so far are all pretty simple. They consist of a sequence of statements that are executed one by one from the first to the last. Such a program is said to have a *linear control flow*. This type of control flow is quite restrictive, as each statement in the source code is executed at most once during the execution of the program. Suppose you want to implement an algorithm that performs 10,000 steps for some input. Then you would have to write a program with at least 10,000 lines of code. Obviously this is undesirable. Therefore, in order to implement non-trivial algorithms, more powerful mechanisms to control the flow of a program are needed.

### 2.4.1 Selection: if– and if-else statements

One particularly simple way to deviate from linear control flow is to select whether or not a particular statement is executed. In C++ this can be done via an `if` *statement*. The syntax is

`if` ( *condition* )
  *statement*

where *condition* is an expression or variable declaration of a type whose values can be converted to `bool`, and *statement* — as the name suggests — is a statement.[11] The semantics is the following: *condition* is evaluated; if and only if its value is *true*, *statement* is executed afterwards. In other words, an `if` statement splits the control flow into two branches. The value of *condition* selects which of these branches is executed. For example, the following lines of code

[11] In case you are missing a semicolon after *statement*: recall that this semicolon is part of the statement.



```
int a;
std::cin >> a;
if (a % 2 == 0) std::cout << "even";
```

read a number from standard input into the variable `a` and write "`even`" to standard output if and only if `a` is even.

Optionally, an `if` statement can be complemented by an `else`-*branch*. The syntax is

`if` ( *condition* )
  *statement1*
`else`
  *statement2*

and the semantics is as follows: *condition* is evaluated; if its value is *true*, *statement1* is executed afterwards; otherwise, *statement2* is executed afterwards. For example, the following lines of code

```
int a;
std::cin >> a;
if (a % 2 == 0)
  std::cout << "even";
else
  std::cout << "odd";
```

read a number from standard input into the variable `a`. Then if `a` is even, "`even`" is written to standard output; otherwise, "`odd`" is written to standard output.

When formatting an `if` statement, it is common to insert a line break before *statement1*, before `else`, and before *statement2*. Moreover, *statement1* and *statement2* are indented and `else` is aligned with `if`, as shown in the example above. If the whole statement fits on a single line then it can also be typeset as a single line.

Collectively, `if`- and `if-else` statements are known as *selection statements*.

### 2.4.2 Iteration: for statements

A much more powerful way of manipulating the control flow is provided by *iteration statements*. Iteration allows to execute a statement many times, possibly with different parameters each time. Iteration statements are also called *loops*, as they "loop through" a statement (potentially) several times. Selection and iteration statements are collectively referred to as *control statements*.

Consider the problem of computing the sum $S_n = \sum_{i=1}^{n} i$ of the first $n$ natural numbers, for a given $n \in \mathbb{N}$. Program 6 reads in a variable `n` from standard input, defines another variable `s` to contain the result, computes the result and finally outputs it. In order to understand why the program `sum_n.C` indeed behaves as claimed, we have to explain the different parts of a `for` *statement*.

```
1  // Program: sum_n.C
2  // Compute the sum of the first n natural numbers.
3
4  #include <iostream>
5
6  int main()
7  {
8    // input
9    std::cout << "Compute the sum 1+...+n for n =? ";
10   unsigned int n;
11   std::cin >> n;
12
13   // computation of sum_{i=1}^n i
14   unsigned int s = 0;
15   for (unsigned int i = 1; i <= n; ++i) s += i;
16
17   // output
18   std::cout << "1+...+" << n << " = " << s << ".\n";
19   return 0;
20 }
```

**Program 6**: *progs/sum_n.C*

**for statement.** The `for` statement is a very compact form of an iteration statement, as it combines three statements or expressions into one. In most cases, the `for` statement serves as a "counting loop" as in Program 6. Its syntax is defined by

`for` ( *init-statement condition*; *expression* )
*statement*

where *init-statement* is an expression statement, a declaration statement, or the null statement, see Section 2.1.14. In all of these cases, *init-statement* ends with a semicolon such that there are always two semicolons in between the parentheses after a `for`. Usually *init-statement* defines and initializes a variable that is used to control and eventually end the iteration statement's execution. In `sum_n.C`, *init-statement* is a declaration statement that defines the variable `i`.

As in an `if` statement, *condition* is an expression or variable declaration whose type can be converted to `bool`. It defines how long the iteration goes on, namely as long as *condition* returns *true*. It is allowed that *condition* is empty in which case its value is interpreted as *true*. As the name suggests, *expression* is an arbitrary expression that may also be empty (in which case it has no effect). *statement* is an arbitrary statement. It is referred to as the *body* of the `for` statement.



Typically, *expression* has the effect of changing a value that appears in *condition*. Such an effect is said to "make progress towards termination". The goal is that *condition* is *false* after *expression* has been evaluated a finite number of times. In that sense, every evaluation of *expression* makes a step towards the end of the `for` statement. In `sum_n.C`, *expression* increments the variable `i` which is bounded from above in *condition*. In cases like this where the value of a single variable is accessed and changed by *condition* and *expression*, we call this variable the *control variable* of the `for` statement.

We are now ready to precisely define the semantics of a `for` statement. First, *init-statement* is executed once. Thereafter *condition* is evaluated. If it returns *true*, an *iteration* of the loop starts. If *condition* returns *false*, the `for` statement *terminates*, that is, its processing ends immediately.

Each single iteration of a `for` statement consists of first executing *statement* and then evaluating *expression*. After each iteration, *condition* is evaluated again. If it returns *true*, another iteration follows. If *condition* returns *false*, the `for` statement terminates. The execution order is therefore *init-statement*, *condition*, *statement*, *expression*, *condition*, *statement*, *expression*, ... until *condition* returns *false*.

Let's see this in action: Consider the `for` statement

```
for (unsigned int i = 1; i <= n; ++i) s += i;
```

in `sum_n.C` and suppose `n == 2`. First, the variable `i` is defined and initialized to 1. Then it is tested whether `i <= n`. As `1 <= 2` is *true*, the first iteration starts. The statement `s += i` is executed, setting `s` to 1, and thereafter `i` is incremented by one such that `i == 2`. One iteration is now complete. As a next step, the condition `i <= n` is evaluated again. As `2 <= 2` is *true*, another iteration follows. First `s += i` is executed, setting `s` to 3. Thereafter, `i` is incremented by one such that `i == 3`. The second iteration is now complete. The subsequent evaluation of `i <= n` entails `3 <= 2` which is *false*. Thus, no further iteration takes place and the processing of the `for` statement ends. The value of `s` is now 3, the sum of the first `n == 2` natural numbers.

**Infinite loops.** It is easily possible to create loops that do not terminate. For example, recall that both *condition* and *expression* may be empty. Moreover, both *init-statement* and *statement* can be the null statement. In this case we get the `for` statement

```
for (;;);
```

As the empty *condition* has value *true*, executing this statement runs through iteration after iteration without actually doing anything. Therefore, `for (;;)` may be read as "forever". In general, a statement which does not terminate is called an *infinite loop*.

Clearly, infinite loops are extremely undesirable and programmers try hard to avoid them. Nevertheless, sometimes such loops occur even in real life software. If you regularly use a computer, you have probably experienced this kind of phenomenon: a program "hangs".

You may ask: Why doesn't the compiler simply detect infinite loops and warns me about them just as it complains about syntax errors? Indeed, this would be a great thing

to have and it would solve many problems in software development. The problem is that infinite loops are not always as easy to spot as in the above example. Loops can be pretty complicated, and possibly they loop infinitely when executed in certain program states only.

In fact, the situation is hopeless: It can be shown that the problem of detecting infinite loops (commonly referred to as the *halting problem*) cannot be solved by a computer, as we have and understand it today (see the Details). Therefore, some care is needed when designing loops. We have to check "by hand" that the iteration statement terminates for all possible program states that can occur.

**Gauss.** You may know or have realized that our program `sum_n.C` is actually a bad example. It is bad in the sense that it does not convincingly demonstrate the power of control statements.

In his primary school days, the German mathematician Carl Friedrich Gauss (1777–1855) was told to sum up the numbers $1, 2, 3, \ldots, 100$. The teacher had planned to keep his students busy for a while, but Gauss came up with the correct result 5050 very quickly. He had imagined writing down the numbers in increasing order, and one line below once again in decreasing order. Clearly, the two numbers in each column sum up to 101; hence, the overall sum is $100 \cdot 101 = 10100$, half of which is the number that was asked for.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | ... | 98 | 99 | 100 |
| 100 | 99 | 98 | ... | 3 | 2 | 1 |
| 101 | 101 | 101 | ... | 101 | 101 | 101 |

In this way, Gauss discovered the formula

$$\sum_{i=1}^{n} i = n(n+1)/2,$$

for any $n \in \mathbb{N}$. The `for` statement in `sum_n.C` can therefore be replaced by the much more elegant and efficient statement [12]

```
s = n * (n + 1) / 2;
```

We next get to a real application of selection and iteration statements.

**Prime numbers.** In the introductory Section 1.1, we have talked a lot about prime numbers. How would a program look like that tests whether or not a given number is prime? According to the usual definition, a number $n \in \mathbb{N}, n \geq 2$ is prime if and only if it is not divisible by any number $d \in \{2, \ldots, n-1\}$. The strategy for our program is therefore clear: Write a loop that runs through all these numbers, and test each of them for being a divisor of $n$. If a divisor is found, we can stop and output a factorization of $n$ into two

[12] Note that in this statement, the integer division coincides with the real division, since for all $n$, the product $n(n+1)$ is even.



numbers, proving that $n$ is not prime. Otherwise, we output that $n$ is prime. Program 7 implements this strategy in C++, using one `for` statement, and one `if` statement. Remarkably, the `for` statement has an empty body, since we have put the divisibility test into the *condition*. The important observation is that the *condition* `n % d != 0` definitely returns *false* for `d == n`, so that the loop is guaranteed to terminate; if (and only if) *condition* returns *false* earlier, we have found a divisor of $n$ in the range $\{2, \ldots, n-1\}$.

```
1  // Program: prime.C
2  // Test if a given natural number is prime.
3
4  #include <iostream>
5
6  int main ()
7  {
8    // Input
9    unsigned int n;
10   std::cout << "Test if n>1 is prime for n =? ";
11   std::cin >> n;
12
13   // Computation: test possible divisors d
14   unsigned int d;
15   for (d = 2; n % d != 0; ++d);
16
17   // Output
18   if (d < n)
19     // d is a divisor of n in {2,...,n-1}
20     std::cout << n << " = " << d << " * " << n / d << ".\n";
21   else
22     // no proper divisor found
23     std::cout << n << " is prime.\n";
24
25   return 0;
26 }
```

**Program 7:** *progs/prime.C*

### 2.4.3 Blocks and scope

In C++ it is possible to group a sequence of one or more statements into one single statement that is then called a *compound statement*, or simply a *block*. This mechanism does not manipulate the control flow directly. Blocks allow to structure a program by grouping statements that logically belong together. In particular, they are a tool to design powerful and at the same time readable control statements.

Syntactically, a block is simply a sequence of zero or more statements that are enclosed

in curly braces.

{ *statement1 statement2 ... statementN* }

Each of the statements may in particular be a block, so it is possible to have nested blocks. The simplest block is the empty block `{}`.

You have already seen blocks. Each program contains a special block, the so-called *function body* of the `main` function. This block encloses the sequence of statements that is executed when the `main` function is called by the operating system.

Using blocks, one can create selection and iteration statements whose body contains a sequence of two or more statements. For example, suppose that for testing purposes we would like to write out all partial sums during the computation in `sum_n.C`:

```
for (unsigned int i = 1; i <= n; ++i) {
  s += i;
  std::cerr << i << "-th partial sum is " << s << "\n";
}
```

Here two statements are executed in each iteration of the loop. First, the next summand is added to `s`, then the current value of `s` is written to standard error output.

Blocks should in general be formatted as shown above. That is, a line break appears after the opening and before the closing brace, and all lines in between are indented one level. Only if the block consists of just one single statement and it all fits on one line, the block can be formatted as one single line.

The type of test output we have created in the previous example is called *debugging output*. A *bug* is a commonly used term to denote a programming error, hence "debugging" is the process of finding and eliminating such errors. It is good practice to write debugging output to standard error output since it can then more easily be separated from the "real" program output that usually goes to standard output.

**Visibility.** Blocks do not only structure a program visually but they also provide a logical boundary around declarations (of variables, for example). Any declaration that appears inside a block is called *local* to that block. A local declaration extends only until the end of the block in which it appears. A name that is introduced by a local declaration is not "visible" outside of the block where it is declared. For example, in

```
1 int main()
2 {
3   {
4     int i = 2;
5   }
6   std::cout << i; // error, undeclared identifier
7   return 0;
8 }
```



the variable `i` declared inside the block in line 3–5 is not visible in the output statament in line 6. Thus, if you confront the compiler with this code, it issues an error message.

**Control statements and blocks.** Control statements act like blocks themselves. Therefore any declaration appearing in a control statement is local to that control statement. In particular, this applies to a variable defined in the *init-statement* of a `for` statement. For example, in

```
1 int main()
2 {
3   for (unsigned int i = 0; i < 10; ++i) s += i;
4   std::cout << i; // error, undeclared identifier
5   return 0;
6 }
```

the expression `i` in line 4 does *not* refer to the variable `i` defined in line 3.

**Declarative region.** After having seen these first examples, we will now introduce the precise terminology that allows us to deduce which names can be used where in the program. Each declaration has an associated *declarative region*. This region is the part of the program in which the declaration appears. Such a region can be a block, a function definition, or a control statement. In all these cases the declaration is said to have *local scope*. A declaration can also have *namespace scope*, if it appears inside a namespace, see Section 2.1.3. Finally, a declaration that is outside of any particular other structure has *global scope*.

**Scope.** A name introduced by a declaration D is *valid* or *visible* in a part of its declaration's declarative region, called the *scope* of the declaration. Within the scope of D, the name introduced by D may be used and actually refers to the declaration D. In most cases, the scope of a declaration is equal to its *potential scope*.

The *potential scope* of a declaration starts at the point where the declaration appears. For the name to be declared this is called its *point of declaration*. The potential scope extends until the end of the declarative region.

To get the scope of a declaration, we start from its potential scope but we possibly have to remove some parts of it. This happens when the potential scope contains one or more declarations of the *same* name. As an example, consider Program 8.

```
1 #include <iostream>
2
3 int main()
4 {
5   int i = 2;
6   for (int i = 0; i < 5; ++i)
7     std::cout << i;   // outputs 0, 1, 2, 3, 4
```

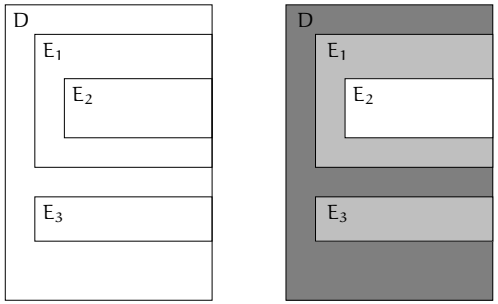


**Figure 5**: *Potential scopes of declarations* $D, E_1, E_2, E_3$ *of the same name, drawn as rectangles with the corresponding declaration in the upper left corner (left); on the right, we see the resulting scopes of* $D$ *(dark gray),* $E_1, E_3$ *(light gray) and* $E_2$ *(white).*

```
 8   std::cout << i;      // outputs 2
 9   return 0;
10 }
```

**Program 8**: *progs/scope.C*

The `i` in line 7 refers to the declaration from line 6, whereas the `i` in line 8 refers to the declaration from line 5. Therefore, the program outputs first `0`, `1`, `2`, `3`, `4`, and then `2`. In some sense, the declaration in line 6 temporarily hides the previous declaration of `i` from line 5. This phenomenon is called *name hiding*. But when the declarative region of the second declaration ends in line 7, the second declaration "becomes invisible" (we say: "it runs out of scope") and the first declaration takes over again. In particular, since the name `i` in line 8 refers to the variable defined in line 5, we get the output `2` in line 8.

It is good practice to avoid name hiding since this unnecessarily obfuscates the program. On the other hand, name hiding allows us (like in Program 8) to use our favorite identifier `i` as the name of the control variable in a `for` statement, without having to check whether there is some other name `i` somewhere else in the program. This is an acceptable and even useful application of name hiding.

Now we can get to the formal definition of scope in the general case (possible presence of multiple declarations of the same name). The *scope* of a declaration $D$ is obtained from its potential scope as follows: For each declaration $E$ in the potential scope of $D$ such that both $D$ and $E$ declare the same name, the potential scope of $E$ is removed from the scope of $D$. Figure 5 gives a symbolic picture of the situation.

In Program 8, the declarative region of the declaration in line 5 is line 4–10 (a block),



its potential scope is line 5–10, and its scope is line 5 plus line 8–10. For the declaration in line 6, declarative region (a control statement), potential scope and scope are line 6–7.

Breaking down the scopes into lines is in general not possible, of course, since line breaks may (or may not) appear almost anywhere. If we want to talk about scope on a line-by-line basis, we have to format the program accordingly.

**Storage duration.** Related to the scope of a variable is its *storage duration*. This term denotes the time in which the address of the variable is valid, that is, some memory location is assigned to it.

For a variable with local scope, the storage duration is usually the time in which the program's control is in the variable's potential scope. During program execution, this means that whenever the variable declaration is reached, some memory location is assigned and the address becomes valid. And whenever the execution gets to the end of the declarative region, the associated memory is freed and the variable's address becomes invalid.[13] We therefore get a "fresh instance" of the variable everytime its declaration is executed.

This behavior is called *automatic storage duration*. For example, in

```
for (unsigned int i = 0; i < 10; ++i) {
  int k = 2;
  // do something with k
}
```

the address of the variable `k` may change in each iteration of the loop. Also the initialization to `3` takes place in each iteration.

As a more concrete example, consider the following code fragment.

```
1 int i = 5;
2 for (int j = 0; j < 5; ++j) {
3   std::cout << ++i; // outputs 6, 7, 8, 9, 10
4   int k = 2;
5   std::cout << --k; // outputs 1, 1, 1, 1, 1
6 }
```

Since line 3 belongs to the scope of the declaration in line 1, the effect of line 3 is to increment the variable defined in line 1 in every iteration of the `for` statement. Line 5, on the other hand, belongs to the scope of the declaration in line 4; the effect of line 5 is therefore to decrement the "fresh" variable `k` in every iteration, and this always results in value 1.

In contrast, a variable that is defined in namespace scope or global scope has *static storage duration*. This means that its address is determined at the beginning of the program's execution, and it does not change (hence "static") nor become invalid until the execution of the program ends. The variables named by `std::cin` and `std::cout`,

[13] Note that the address does not necessarily remain the same throughout the program's execution.

for instance, have static storage duration. Variables with static storage duration are also referred to as *static variables*.

### 2.4.4 Iteration: while statements

So far, we have seen one iteration statement, the `for` statement. The `while` *statement* is a simplified `for` statement, where both *init-statement* and *expression* are omitted. Its syntax is

`while` ( *condition* )
  *statement*

where *condition* and *statement* are as in a `for` statement. As before, *statement* is referred to as the body of the `while` statement. Semantically, a `while` statement is equivalent to the corresponding `for` statement

`for` ( ; *condition* ; )
  *statement*

The execution order is therefore *condition*, *statement*, *condition*,... until *condition* returns *false*.

Since `while` statements are so easy to rewrite as `for` statements, why do we need them? The main reason is readability. As its name suggests, a `for` statement is typically perceived as a counting loop in which the increment (or decrement) of a single variable is responsible for the progress towards termination. In this case, the progress is most conveniently made in the `for` statement's *expression*. But the situation can be more complex: the progress may depend on the values of several variables, or on some condition that we check in the loop's body. In some of these cases, a `while` statement is preferable. The next section describes an example.

**The Collatz problem.** Given a natural number $n \in \mathbb{N}$, we consider the *Collatz sequence* $n_0, n_1, n_2, \ldots$ with $n_0 = n$ and

$$n_i = \begin{cases} n_{i-1}/2, & \text{if } n_{i-1} \text{ is even} \\ 3n_{i-1}+1, & \text{if } n_{i-1} \text{ is odd} \end{cases} \qquad i \geq 1.$$

For example, if $n = 5$, we get the sequence $5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$. Since the sequence gets repetitive as soon as 1 appears, we may stop at this point. Program 9 reads in a number $n$ and outputs the elements of the sequence $(n_i)_{i \geq 1}$ until the number 1 appears.

```
1  // Program: collatz.C
2  // Compute the Collatz sequence of a number n.
3
4  #include <iostream>
```



```
5
6  int main()
7  {
8    // Input
9    std::cout << "Compute the Collatz sequence for n =? ";
10   unsigned int n;
11   std::cin >> n;
12
13   // Iteration
14   while (n > 1) {
15     if (n % 2 == 0)
16       n = n / 2;
17     else
18       n = 3 * n + 1;
19     std::cout << n << " ";
20   }
21   std::cout << "\n";
22   return 0;
23 }
```

**Program 9:** *progs/collatz.C*

The loop can of course be written as a `for` statement with empty *init-statement* and *expression*, but the resulting variant of the program is less readable since it tries to advertise the rather complicated iteration as a simple counting loop. As a rule of thumb, if there is a simple *expression* that captures the loop's progress, use a `for` statement. Otherwise, consider formulating your loop as a `while` statement.

Talking about progress: is it clear that the number 1 always appears? If not, the program `collatz.C` contains an infinite loop for certain values of `n`. If you play with the program, you will observe that 1 indeed appears for all numbers you try, although this may take a while. You will find, for example, that the Collatz sequence for $n = 27$ is

> 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

It is generally believed that 1 eventually comes up for all values of $n$, but mathematicians have not yet been able to produce a proof of this conjecture. As innocent as it looks, this problems seems to be a very hard mathematical nut to crack (see also the Details section), but you are certainly invited to give it a try!

### 2.4.5 Iteration: do statements

`Do` *statements* are similar to `while` statements, except that the condition is evaluated *after* every iteration of the loop instead of *before* every iteration. Therefore, in contrast to for– and `while` statements, the body of a `do` statement is executed at least once.
The syntax of a `do` statement is as follows.

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

where *expression* is of a type whose values can be converted to `bool`.

The semantics is defined as follows. An iteration of the loop consists of first executing *statement* and then evaluating *expression*. If *expression* returns *true* then another iteration follows. Otherwise, the `do` statement terminates. The execution order is therefore *statement*, *expression*, *statement*, *expression*, ... until *expression* returns *false*.

Alternatively, the semantics could be defined in terms of the following equivalent `for` statement.

```
for ( bool firsttime = true; firsttime || expression; firsttime = false )
  statement
```

This behaves like our "simulation" of the `while` statement, except that in the first iteration, *expression* is not evaluated (due to short circuit evaluation, see Section 2.3.3), and *statement* is executed unconditionally.

Consider a simple calculator-type application in which the user enters a sequence of numbers, and after each number the program outputs the sum of the numbers entered so far. By entering `0`, the user indicates that the program should stop. This is most naturally written using a `do` statement, since the termination condition can only be checked *after* the next number has been entered.

```
int a;      // next input value
int s = 0; // sum of values so far
do {
  std::cout << "next number =? ";
  std::cin >> a;
  s += a;
  std::cout << "sum = " << s << "\n";
} while (a != 0);
```

In this case, it is *not* possible to declare `a` where we would usually do it, namely immediately before the input statement. The reason is that `a` would then be local to the body of the `do` statement and would not be visible in the `do` statement's *expression* `a != 0`.



### 2.4.6 Jump statements

At this point, we would like to extend our arsenal of control statements with a special type of statements that are referred to as *jump statements*. These statements are not necessary in the sense that they would allow you to do something which is not possible otherwise. Instead, just like `while`– and `do` statements (which are also unnecessary in that sense), jump statements provide additional flexibility in designing iteration statements. You should use this flexibility wherever it allows you to improve your code. However, be also warned that jump statements should be used with care since they tend to complicate the control flow. The complication of the control flow has to be balanced by a significant gain in one of the other categories. Therefore, think carefully before introducing a jump statement!

When a jump statement is executed, the program flow unconditionally "jumps" to a certain point. There are two different jump statements that we want to discuss here.

The first jump statement is called a `break` *statement*; its syntax is rather simple.

```
break;
```

When a `break` statement is executed within an iteration statement, [14] the smallest enclosing iteration statement terminates immediately. The execution continues at the statement after the iteration statement (if any). For example,

```
for (;;) break;
```

is not an infinite loop but rather a complicated way of writing a null statement. Here is a more useful appearance of `break`. In our calculator example from Page 83, it would be more elegant to suppress the irrelevant addition of `0` in the last iteration. This can be done with the following loop.

```
for (;;) {
  std::cout <<  "next number =? ";
  std::cin >> a;
  if (a == 0) break;
  s += a;
  std::cout << "sum = " << s << "\n";
}
```

Here, we see the typical usage of `break`, namely the termination of a loop "somewhere in the middle". Note that we could equivalently write

```
do {
  std::cout <<  "next number =? ";
  std::cin >> a;
  if (a == 0) break;
  s += a;
```

[14] otherwise, it can only occur in a `switch` statement, see the Details.

```
  std::cout << "sum = " << s << "\n";
} while (true);
```

In this case `for` is preferable, though, since it nicely reads as "forever". Of course, the same functionality is possible without `break`, but the resulting code requires an additional block and evaluates `a != 0` twice.

```
do {
  std::cout <<  "next number =? ";
  std::cin >> a;
  if (a != 0) {
    s += a;
    std::cout << "sum = " << s << "\n";
  }
} while (a != 0);
```

The second jump statement is called a `continue` *statement*; again the syntax is simple.

```
continue;
```

When a continue statement is executed, the remainder of the smallest enclosing iteration statement's body is skipped, and execution continues at the end of the body. The iteration statement itself is *not* terminated.

If the surrounding iteration statement is a `while`– or `do` statement, the execution therefore continues by evaluating its *condition*. If the surrounding iteration statement is a `for` statement, the execution continues by evaluating its *expression* and then its *condition*. Like the `break` statement, the `continue` statement can therefore be used to manipulate the control flow "in the middle" of a loop.

In our calculator example, the following variant of the loop ignores negative input. Again, it would be possible to do this without `continue`, at the expense of another nested block.

```
for (;;) {
  std::cout <<  "next number =? ";
  std::cin >> a;
  if (a < 0) continue;
  if (a == 0) break;
  s += a;
  std::cout << "sum = " << s << "\n";
}
```

### 2.4.7 Equivalence of iteration statements

In terms of pure functionality, the `while`– and `do` statements are redundant, as both of them can equivalently be expressed using a `for` statement. This may create the impression that `for` statements have more expressive power than `while`– and `do` statements.



In this section we show that this is not the case: all three iteration statements are functionally equivalent. More precisely, we show how to use

- `do` statements to express `while` statements, and
- `while` statements to express `for` statements.

If we denote "A can be used to express B" by $A \Rightarrow B$, we therefore have

`do` statement $\Rightarrow$ `while` statement $\Rightarrow$ `for` statement $\Rightarrow$ `do` statement,

where we know the last implication from the previous section. Together, this clearly "proves" the claimed equivalence.

Note that we put the word *proves* in quotes, as our reasoning cannot be considered a formal proof. In order to really prove a statement like this, we first of all would have to be more formal in defining the semantics of statements. Semantics of programming languages is a subject of its own, and the formal treatment of semantics is way beyond what we can do here. In other words: The following is as much of a "proof" as you will get here, but it is sufficient to understand the relations between the three iteration statements.

**do statement $\Rightarrow$ while statement.** Consider the `while` statement

```
while ( condition )
  statement
```

Your first idea how to simulate this using a `do` statement might look like this:

```
if ( condition )
  do
    statement
  while ( condition );
```

Indeed, this induces the execution order *condition*, *statement*, *condition*,... until *condition* returns *false* and the statement terminates. But there is a simple technical problem: if *condition* is a variable declaration, we can't use is as the *expression* in the `do` statement. Here is a reformulation that works. [15]

```
do
  if ( condition )
    statement
  else
    break;
while ( true );
```

This induces exactly the `while` statement's execution order *condition*, *statement*, *condition*,... until *condition* returns *false* and the loop is terminated using `break`.

[15] We are not saying that this should be done in practice. On the contrary, this should *never* be done in practice. This section is about *conceptual* equivalence, not about practical equivalence.

**while statement ⇒ for statement.** Simulating the `for` statement

```
for ( init-statement condition; expression )
  statement
```

by a `while` statement seems easy:

```
{
  init-statement
  while ( condition ) {
    statement
    expression;
  }
}
```

Indeed, this will work, *unless statement* contains a `continue`. In the `for` statement, execution would then proceed with the evaluation of *expression*, but in the simulating `while` statement, *expression* is skipped, and *condition* comes next. This reformulation is therefore wrong. Here is a version that works:

```
{
  init-statement
  while ( condition ) {
    bool b = false;
    while ( b = !b )
      statement
    if ( b ) break;
    expression;
  }
}
```

This looks somewhat more complicated, so let us explain what is going on.

We may suppose that the identifier `b` does not appear in the given `for` statement (otherwise we choose a different name). Note that the whole statement forms a separate block, as does a `for` statement. A potential declaration in *init-statement* as well as the scope of `b` is thus limited to this block.

Consider an execution of the outer `while` statement. First, *condition* is evaluated, and if it returns *false* the statement terminates. Otherwise, the variable `b` is set to *true* in the inner `while` statement's condition, meaning that *statement* is executed next. [16] If *statement* does not contain a `break`, the inner loop evaluates its condition for the second time. In doing so, `b` is set to *false*, and the condition returns *false*. Therefore, the inner loop terminates. Since `b` is now *false*, *expression* is evaluated next, followed by *condition*. This induces the `for` statement's execution order *condition*, *statement*, *expression*, *condition*,... until *condition* returns *false* and the outer loop terminates.

[16] Recall that the assignment operator returns the new value of its left operand.



In the case where *statement* contains a `break`, the inner loop terminates immediately, and `b` remains *true*. In this case, we also terminate the outer loop that represents our original `for` statement.

In retrospect, we should now check that jump statements cause no harm in our previous reformulation of the `while` statement in terms of the `do` statement. We leave this as an exercise.

### 2.4.8 Choosing the "right" iteration statements

We have seen that from a functional point of view, the `for` statement, the `while` statement and the `do` statement are equivalent. Moreover, the `break` and `continue` statements are redundant. Still, C++ offers all of these statements, and this gives you the freedom (but also the burden) of choosing the appropriate control statements for your particular program.

Writing programs is a dynamic process. Even though the program may do what you want at some point, the requirements change, and you will keep changing the program in the future. Even if there is currently no need to change the functionality of the program, you may want to replace a complicated iteration statement by an equivalent simpler formulation. The general theme here is *refactoring*: the process of rewriting a program to improve its readability or structure, while keeping its functionality unchanged.

Here is a simple guideline for writing "good" loops. Choose the loop that leads to the most *readable* and *concise* formulation. This means

- few statements,
- few lines of code,
- simple control flow, and
- simple expressions.

Almost never there is *the* one and only best formulation; however, there are always arguably bad choices which you should try to avoid. Usually, there are some tradeoffs, like fewer lines of code versus more complicated expressions, and there is also some amount of personal taste involved. You should experience and find out what suits you best.

Let us look at some examples to show what we mean. Suppose that you want to output the odd numbers between 0 and 100. Having just learned about the `continue` statement, you may write the following loop.

```
for (unsigned int i = 0; i < 100; ++i) {
  if (i % 2 == 0) continue;
  std::cout << i << "\n";
}
```

This is perfectly correct, but the following version is preferable since it has **fewer statements and fewer lines of code.**

```
for (unsigned int i = 0; i < 100; ++i)
  if (i % 2 != 0) std::cout << i << "\n";
```

This variant still contains nested control statements; but you can get rid of the `if` statement and obtain code with **simpler control flow.**

```
for (unsigned int i = 1; i < 100; i += 2)
  std::cout << i << "\n";
```

The same output can be produced with a `while` statement and equally simple control flow.

```
int i = -1;
while ((i += 2) < 100)
  std::cout << i << "\n";
```

But here, the condition is more complicated, since it combines assignment and comparison operators. Such expressions are comparatively difficult to understand due to the effect of the assignment operation. Also, the initialization of `i` to $-1$ is counter-intuitive, given that we deal with natural numbers.

You can solve the latter problem and at the same time get **simpler expressions** by writing

```
unsigned int i = 1;
while (i < 100) {
  std::cout << i << "\n";
  i += 2;
}
```

The price to pay is that you get less concise code; there are now five lines instead of the two lines that the `for` statement needs. It seems that for the simple problem of writing out odd numbers, a `for` statement with *expression* `i += 2` is the loop of choice.

### 2.4.9 Details

**Nested if-else statements.** Consider the statement

```
if(true) if (false); else std::cout << "Where do I belong?";
```

It is not a priori clear what its effect is: if the `else` branch belongs to the outer `if`, there will be no output (since the condition has value *true*), but if the `else` branch belongs to the inner `if`, we get the output `Where do I belong?`

The intuitive rule is that the `else` branch belongs to the `if` immediately preceding it, in our case to the inner `if`. Therefore, the output is `Where do I belong?`, and we should actually format the statement like this:

```
if(true)
  if (false)
    ; // null statement
  else
    std::cout << "Where do I belong?";
```



Whenever you are unsure about rules like this, you can make the structure clear through explicit blocks:

```
if(true) {
  if (false) {
    ; // null statement
  }
  else {
    std::cout << "Where do I belong?";
  }
}
```

**The switch statement.** Besides `if...else` there exists a second selection statement in C++: the `switch` *statement*. It is useful to select between many alternative statements, using the following syntax.

`switch` ( *condition* )
*statement*

The value of *condition* must be convertible to an integral type. This is in contrast to the other control statements where *condition* has to be convertible to `bool`.

*statement* is usually a block that contains several *labels* of the form

`case` *literal*:

where *literal* is a literal of integral type. For no two labels shall these literals have the same value. There can also be a label `default:`.

The semantics of a `switch` statement is the following. *condition* is evaluated and the result is compared to each of the literals which appear in a label in *statement*. If for any of them the values agree, the execution continues at the statement immediately following the label. If there is no agreement but a `default:` label, the execution continues at the statement immediately following the `default:` label. Otherwise, *statement* is ignored and the execution continues after the `switch` statement.

Note that `switch` only selects an entry point for the processing of *statement*, it does not exit when the execution reaches another label. If one wants to separate the different alternatives, one has to use `break` (and this is the only legal use of `break` outside of an iteration statement). Consider for example the following piece of code, and let us suppose that `x` is a variable of type `int`.

```
switch (x) {
  case 0: std::cout << "0";
  case 1: std::cout << "1"; break;
```

```
  default: std::cout << "whatever";
}
```

For `x==0` the output is `01`; for `x==1` the output is `1`; otherwise we get the output `whatever`.

The `switch` statement is powerful in the sense that it allows the different alternatives to share code. However, this power also makes `switch` statements hard to read and error prone. A frequent problem is that one forgets to put a `break` where there should be one. Therefore, we mention `switch` here for completeness only. Whenever there are only a few alternatives to be distinguished, play it safe and use `if...else` rather than `switch`.

**The Halting Problem, Decidability, and Computability.** The halting problem is one of the fundamental problems in the theory of computation. Informally speaking, the problem is to decide (using an algorithm) whether a given program halts (terminates) when executed on a given input (program state). The term "program" may refer to a C++-program, but also to a program in any other common programming language.

To attack the problem formally, the British mathematician Alan Turing (1912-1954) defined in a seminal paper a "minimal" programming language; a program in this language is known as a *Turing machine*.

Turing proved that the halting problem is undecidable for Turing machines, but the same arguments can also be used to prove the same statement for C++ programs.

What does "undecidable" mean? We have seen a simple loop for which it was painfully evident that it is an infinite loop, haven't we? Yes, indeed one *can* decide the halting problem for many concrete programs. Undecidable means that (in a particular model of computation) there cannot be an algorithm that decides the halting problem for *all possible* programs.

Despite their simplicity, Turing machines are a widely accepted model of computation; in fact, just like machine language, Turing machines can do everything that C++ programs can do, except that they usually need a huge number of very primitive operations for that.

At the same time as Turing, the American mathematician Alonzo Church (1903–1995) developed a computational model called $\lambda$-calculus. As it turned out, his model is equivalent to Turing machines in terms of computational power. The Church-Turing thesis states that "every function that is naturally regarded as computable can be computed by a Turing machine". As there is no rigorous definition of what is "naturally regarded as computable", this statement is not a theorem but a hypothesis that cannot be proven mathematically. As of today, the hypothesis has not been disproved. In theoretical computer science the term *computable* used without further qualification is a synonym for "computable by a Turing machine" (equivalently, a C++ program).

**Point of declaration.** Our approach of defining potential scope and scope line by line is a simplification, even if the code is suitably formatted and we only have one declaration per line. The truth is that the point of declaration of `i` in

```
int i = 5;
```



is in the *middle* of the declaration, after the name `i` has appeared. The potential scope therefore does not include the full line, but only the part starting from `=`. This explains what happens in the following code fragment, but fortunately this is consistent with our line-by-line approach. In

```
1 int i = 5;
2 {
3   int i = i;
4 }
```

the name `i` after the `=` in line 3 refers to the declaration in line 3. Consequently, `i` is initialized with itself in this line, meaning that its value will be undefined, and not 5.

In other situations it may happen, though, that the appearance of a name in the declaration of the same name refers to a *previous* declaration of this name. For now, we can easily avoid such subtleties by the following rule: any declaration should contain the name to be declared only once.

**The Collatz problem and the ?-operator.** The Collatz sequence goes back to the German mathematician Lothar Collatz (1910–1990) who studied it in the 1930's. Several prizes have been offered to anyone who proves or disproves the conjecture that the number 1 appears in the Collatz sequence of every number $n \geq 1$. The famous Hungarian mathematician Paul Erdős (1913–1996) offered \$500, which is much by his standards (he used to offer much lower amounts for very difficult problems). Erdős said that "Mathematics is not yet ready for such problems". Indeed, the conjecture is still unsolved.

We have presented the computation of the Collatz sequence as an application of the `while` statement, pointing out that the conditional change of `n` is too complicated to put it into a `for` statement's *expression*. Well, that's not exactly true: the designers of C, the precursor to C++, had a weakness for very compact code and came up with the *conditional operator* that allows us to simulate `if` statements by expressions. The syntax of this *ternary* operator (arity 3) is

*condition* ? *expression1* : *expression2*

Here, *condition* is an expression of a type whose values can be converted to `bool`, and *expression1* and *expression2* are expressions. The semantics is as follows. First, *condition* is evaluated. If it returns *true*, *expression1* is evaluated, and its value is returned as the value of the composite expression. Otherwise (if *condition* returns *false*), *expression2* is evaluated, and its value is returned. The token `?` is a sequence point (see Section 2.2.10), meaning that all effects of *condition* are processed before either *expression1* or *expression2* are evaluated.

Using the conditional operator, the loop of Program 9 could quite compactly be written as follows.

```
for ( ; n > 1; std::cout << (n % 2 == 0 ? n=n/2 : n=3*n+1) << " ");
```

We leave it up to you to decide whether you like this variant better.

**Static variables.** The discussion about storage duration above does not tell the whole story: it is also possible to define variables with local scope that have static storage duration.

This is done by prepending the keyword `static` to the variable declaration. For example, in

```
for (int i = 0; i < 5; ++i) {
  static int k = i;
  k += i;
  std::cout << k << "\n";
}
```

the address of `k` remains the same during all iterations, and `k` is initialized to `i` *once only*, in the first iteration. The above piece of code will therefore output the sequence of values $0, 1, 3, 6, 10$ (remember Gauss). Without the `static` keyword, the result would simply be the sequence of even numbers $0, 2, 4, 6, 8$.

Static variables have been quite useful in C, for example to count how often a specific piece of code is executed; in C++, they are less important.

For variables of fundamental type the initial value may be undefined, as in the definition `int x;`. However, the value is undefined only if `x` has automatic storage duration. In contrast, variables with static storage duration are always *zero-initialized*, that is, filled with a "zero" of the appropriate type.

**Jump statements.** There are two more jump statements in C++ that we haven't discussed in this section. One of them is the `return` statement that you already know (Section 2.1.14): ist may occur only in a function, and its execution lets the program flow jump to the end of the corresponding function body. The other jump statement is the `goto` statement, but since this one is rarely needed (and somewhat difficult to use), we omit it.

### 2.4.10 Goals

**Dispositional.** At this point, you should ...

1) know the syntax and semantics of `if...else`–, `for`–, `while`–, and `do` statements;
2) understand the concepts block, selection, iteration, declarative region, scope, and storage duration;
3) understand the concept of an infinite loop and be aware of the difficulty of detecting such loops;
4) understand the conceptual equivalence of for–, while–, and do statements;
5) know the syntax and semantics of `continue`– and `break` statements;
6) know at least four criteria to judge the code quality of iteration statements.



**Operational.** In particular, you should be able to ...

(G1) check a given simple program (as defined below) for syntactical correctness and point out possible errors;

(G2) read and understand a given simple program and explain what happens during its execution;

(G3) find (potential) infinite loops in a given simple program;

(G4) find the matching declaration for a given identifier;

(G5) determine declarative region and scope of a given declaration;

(G6) reformulate a given `for`–, `while`–, or `do` statement equivalently using any of the other two statements;

(G7) compare the code quality of two given iteration statements and pick the one that is preferable (if any);

(G8) design simple programs for given tasks.

The term *simple program* refers to a program that consists of a main function in which up to three possibly nested control statements appear. Naturally, only the fundamental types and operations discussed in the preceding sections are used.

### 2.4.11 Exercises

**Exercise 28** *Correct all syntax errors in the program below. What does the resulting program output for the following inputs?*
*(a)* -4 *(b)* 0 *(c)* 1 *(d)* 3 (G1)(G2)

```
1  #include <iostraem>
2  int main()
3  {
4    unsinged int x = +1;
5    { std::cin >> x; }
6    for (int y = 0u; y < x) {
7      std:cout << ++y;
8    return 0;
9  }
```

**Exercise 29** *What is the problem with the code below? Fix it and explain what the resulting code computes.* (G2)(G3)

```
1  unsigned int s = 0;
2  do {
3    int i = 1;
4    if (i % 2 == 1) s *= i;
5  } while (++i < 10);
```

**Exercise 30** *For each variable declaration in the following program give its declarative region and its scope in the form "line x–y". What is the output of the program?* (G2)(G5)

```
1   #include <iostream>
2   int main()
3   {
4     int s = 0;
5     {
6       int i = 0;
7       while (i < 4)
8       {
9         ++i;
10        int f = i + 1;
11        s += f;
12        int s = 3;
13        i += s;
14      }
15      unsigned int t = 2;
16      std::cout << s + t << "\n";
17    }
18    int k = 1;
19    return 0;
20  }
```

**Exercise 31** *Consider the program given below for each of the listed input numbers. Determine the values of* `x`, `s`, *and* `i` *at begin of the first five iterations of the for-loop, before the condition is evaluated. What does the program output for these inputs? (a)* -1 *(b)* 1 *(c)* 2 *(d)* 3 (G2)(G3)

```
1   #include <iostream>
2   int main()
3   {
4     int x;
5     std::cin >> x;
6     int s = 0;
7     for (int i = 0; i < x; ++i) {
8       s += i;
9       x += s / 2;
10    }
11    std::cout << s << "\n";
12    return 0;
13  }
```

**Exercise 32** *Find at least four problems in the code given below.* (G3)(G4)(G5)



```
1   #include <iostream>
2   int main()
3   {
4     { unsigned int x; }
5     std::cin << x;
6     unsigned int y = x;
7     for (unsigned int s = 0; y >= 0; --y)
8       s += y;
9     std::cout << "s=" << s << "\n";
10    return 0;
11  }
```

**Exercise 33** *For which input numbers is the output of the program given below well defined? List those input/output pairs and argue why your list is complete.* (G3)(G4)(G5)

```
1   #include <iostream>
2   int main()
3   {
4     unsigned int x;
5     std::cin >> x;
6     int s = 0;
7     for (unsigned int y = 1 + x; y > 0; y -= x)
8       s += y;
9     std::cout << "s=" << s << "\n";
10    return 0;
11  }
```

**Exercise 34** *Reformulate the code below equivalently in order to improve its readability. Describe the program's output as a function of its input* `n`. (G2)(G6)(G7)

```
1   unsigned int n;
2   std::cin >> n;
3   int x = 1;
4   if (n > 0) {
5     int k = 0;
6     bool e = true;
7     do {
8       if (++k == n) e = false;
9       x *= 2;
10    } while (e);
11  }
12  std::cout << x;
```

**Exercise 35** *Reformulate the program below equivalently in order to improve its readability and efficiency. Describe the program's output as a function of its input* `x`. (G2)(G6)(G7)

```
1   #include <iostream>
2   int main()
3   {
4     int x;
5     std::cin >> x;
6     int s = 0;
7     int i = -10;
8     do
9       for (int j = 1;;)
10        if (j++ < i) s += j - 1; else break;
11    while (++i <= x);
12    std::cout << s << "\n";
13    return 0;
14  }
```

**Exercise 36** *Write a program* `fak-1.C` *to compute the factorial* $n!$ *of a given input number* $n$. (G8)

**Exercise 37** *Write a program* `dec2bin.C` *that inputs a natural number* $n$ *and outputs the binary digits of* $n$ *in reverse order. For example, for* `n==2` *the output is* `01` *and for* `n==11` *the output is* `1101` *(see also Exercise 40).* (G8)

**Exercise 38** *Write a program* `cross_sum.C` *that inputs a natural number* $n$ *and outputs the sum of the (decimal) digits of* $n$. *For example, for* `n==10` *the output is* `1` *and for* `n==112` *the output is* `4`. (G8)

**Exercise 39** *Write a program* `perfect.C` *to test whether a given natural number* $n$ *is perfect. A number* $n \in \mathbb{N}$ *is called* perfect *if and only if it is equal to the sum of its proper divisors, that is,* $n = \sum_{k \in \mathbb{N},\, s.t.\, k<n \,\wedge\, k|n} k$. *For example,* $28 = 1+2+4+7+14$ *is perfect, while* $12 < 1+2+3+4+6$ *is not.*

*Extend the program to find all perfect numbers between* $1$ *and* $n$. *How many perfect numbers exist in the range* $[1, 50000]$*?* (G8)

**Exercise 40** *Write a program* `dec2bin2.C` *that inputs a natural number* $n$ *and outputs the binary digits of* $n$ *in the correct order. For example, for* `n==2` *the output is* `10` *and for* `n==11` *the output is* `1011` *(see also Exercise 37).*

### 2.4.12 Challenges

**Exercise 41** *The* $n$-queens problem *is to place* $n$ *queens on an* $n \times n$ *chessboard such that no two queens threaten each other. Formally, this means that there is no horizontal, vertical, or diagonal with more than one queen in it. Write a program that outputs the number of different solutions to the* $n$*-queens problem for a given input* $n$. *Assuming a 32 bit system, the program should work up to* $n = 9$ *at least.*



*Check through a web search whether the numbers that your program computes are correct.*