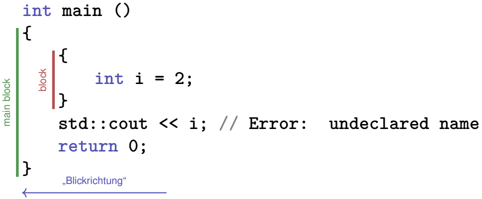
6. Control Statements II

Visibility, Local Variables, While Statement, Do Statement, Jump Statements

---

Visibility

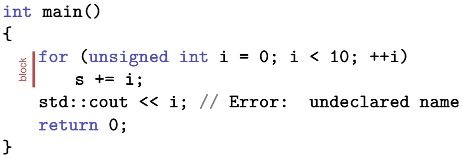
Declaration in a block is not *visible* outside of the block.



```
int main ()
{
    {
        int i = 2;
    }
    std::cout << i; // Error:  undeclared name
    return 0;
}
```

Control Statement defines Block

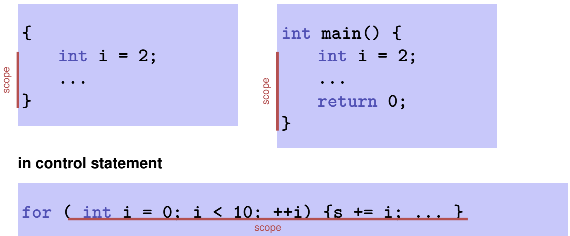
In this respect, statements behave like blocks.



```
int main()
{
    for (unsigned int i = 0; i < 10; ++i)
        s += i;
    std::cout << i; // Error:  undeclared name
    return 0;
}
```

Scope of a Declaration

Potential scope: from declaration until end of the part that contains the declaration.



in the block

```
{
    int i = 2;
    ...
}
```

in function body

```
int main() {
    int i = 2;
    ...
    return 0;
}
```

in control statement

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

Scope of a Declaration

Real scope = potential scope minus potential scopes of declarations of symbols with the same name

```
int main()
{
  int i = 2;
  for (int i = 0; i < 5; ++i)
     // outputs 0,1,2,3,4
     std::cout << i;
   // outputs 2
   std::cout << i;
  return 0;
}
```

in main

i_2 in for

scope of i

Automatic Storage Duration

Local Variables (declaration in block)

- are (re-)created each time their declaration is reached
  - memory address is assigned (allocation)
  - potential initialization is executed
- are deallocated at the end of their declarative region (memory is released, address becomes invalid)

Local Variables

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

Local variables (declaration in a block) have *automatic storage duration*.

`while` Statement

`while (` *condition* `)`
  statement

- *statement*: arbitrary statement, body of the `while` statement.
- *condition*: convertible to `bool`.

`while` Statement

`while` (*condition*)
  statement

is equivalent to

`for` (; *condition* ;)
  statement

`while`-Statement: Semantics

`while` (*condition*)
  statement

- *condition* is evaluated
  - `true`: iteration starts
    statement is executed
  - `false`: `while`-statement ends.

`while`-statement: why?

- In a `for`-statement, the expression often provides the progress ("counting loop")

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

- If the progress is not as simple, `while` can be more readable.

Example: The Collatz-Sequence $(n \in \mathbb{N})$

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, if } n_{i-1} \text{ even} \\ 3n_{i-1} + 1 & \text{, if } n_{i-1} \text{ odd} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (repetition at 1)

The Collatz Sequence in $\mathrm{C}{++}$

```
// Program: collatz.cpp
// Compute the Collatz sequence of a number n.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute the Collatz sequence for n =? ";
  unsigned int n;
  std::cin >> n;

  // Iteration
  while (n > 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
    std::cout << n << " ";
  }
  std::cout << "\n";
  return 0;
}
```

The Collatz Sequence in $\mathrm{C}{++}$

```
n = 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
```

The Collatz-Sequence

Does 1 occur for each n?

- It is conjectured, but nobody can prove it!
- If not, then the `while`-statement for computing the Collatz-sequence can theoretically be an endless loop for some n.

`do` Statement

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

- *statement*: arbitrary statement, body of the `do` statement.
- *expression*: convertible to `bool`.

`do` Statement

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

is equivalent to

```
statement
while ( expression )
  statement
```

`do`-Statement: Semantics

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

- Iteration starts
    - *statement* is executed.
- *expression* is evaluated
    - `true`: iteration begins
    - `false`: `do`-statement ends.

`do`-Statement: Example Calculator

Sum up integers (if 0 then stop):

```
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);
```

Conclusion

- Selection (conditional *branches*)
    - `if` and `if-else`-statement
- Iteration (conditional *jumps*)
    - `for`-statement
    - `while`-statement
    - `do`-statement
- Blocks and scope of declarations

Jump Statements

- `break;`
- `continue;`

`break`-Statement

```
break;
```

- Immediately leave the enclosing iteration statement
- useful in order to be able to break a loop “in the middle” [5]

---

[5] and indispensible for switch-statements

Calculator with `break`

Sum up integers (if 0 then stop)

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    // irrelevant in last iteration:
    s += a;
    std::cout << "sum = " << s << "\n";
}  while (a != 0);
```

Calculator with `break`

Suppress irrelevant addition of 0:

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0)
```

Calculator with `break`

Equivalent and yet more simple:

```
int a;
int s = 0;
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Calculator with `break`

Version without break evaluates `a` twice and requires an additional block.

```
int a = 1;
int s = 0;
for (;a != 0;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a != 0) {
        s += a;
        std::cout << "sum = " << s << "\n";
    }
```

`continue`-Statement

`continue;`

- Jump over the rest of the body of the enclosing iteration statement
- Iteration statement is *not* left.

`break` and `continue` in practice

- Advantage: Can avoid nested `if-else`blocks (or complex disjunctions)
- But they result in additional jumps (for- and backwards) and thus potentially complicate the control flow
- Their use is thus controversial, and should be carefully considered

Calculator with `continue`

Ignore negative input:

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

Equivalence of Iteration Statements

We have seen:

- **`while`** and **`do`** can be simulated with **`for`**

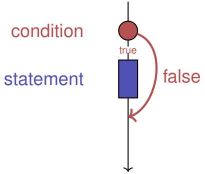
It even holds: Not so simple if a continue is used!

- The three iteration statements provide the same “expressiveness” (lecture notes)

Control Flow

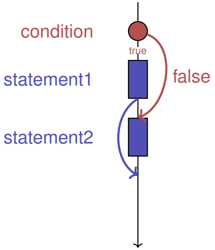
Order of the (repeated) execution of statements

- generally from top to bottom. . .
- . . . except in selection and iteration statements



if (*condition*)
statement

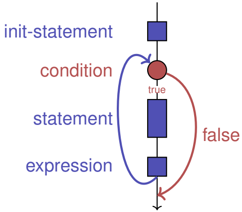
Control Flow `if else`



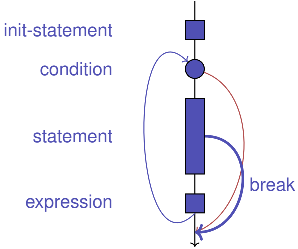
if (*condition*)
statement1
else
statement2

Control Flow `for`

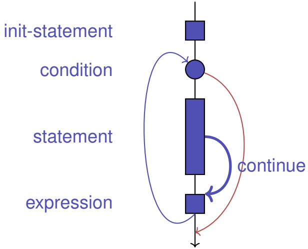
`for` (*init statement* *condition* ; *expression*)
    statement



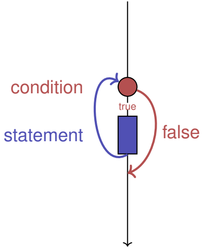
Control Flow `break` in for



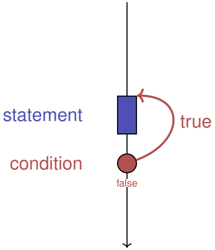
Control Flow `continue` in for



Control Flow `while`



Control Flow `do while`



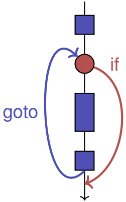
Control Flow: the Good old Times?

Observation

Actually, we only need `if` and jumps to arbitrary places in the program (`goto`).

Languages based on them:

- Machine Language
- Assembler ("higher" machine language)
- BASIC, the first prorgamming language for the general public (1964)



BASIC and home computers...

...allowed a whole generation of young adults to program.

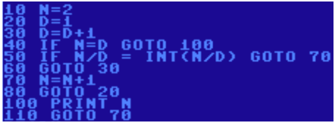
Home-Computer Commodore C64 (1982)

http://de.wikipedia.org/wiki/Commodore_64

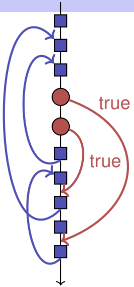
Spaghetti-Code with `goto`

Output of of ??????????all prime numbers

using the programming language BASIC:



```
10 N=2
20 D=1
30 D=D+1
40 IF N=D GOTO 100
50 IF N/D = INT(N/D) GOTO 70
60 GOTO 30
70 N=N+1
80 GOTO 20
100 PRINT N
110 GOTO 70
```



The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements
- few lines of code
- simple control flow
- simple expressions

Often not all goals can be achieved simultaneously.

Odd Numbers in $\{0, \dots, 100\}$

First (correct) attempt:

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

Odd Numbers in $\{0, \dots, 100\}$

Less statements, *less* lines:

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

Odd Numbers in $\{0, \dots, 100\}$

Less statements, *simpler* control flow:

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

This is the “right” iteration statement

Jump Statements

- implement unconditional jumps.
- are useful, such as `while` and `do` but not indispensible
- should be used with care: only where the control flow is *simplified* instead of making it *more complicated*

Outputting Grades

1. Functional requirement:

$6 \rightarrow$ `"Excellent ...  You passed!"`
$5, 4 \rightarrow$ `"You passed!"`
$3 \rightarrow$ `"Close, but ...  You failed!"`
$2, 1 \rightarrow$ `"You failed!"`
otherwise $\rightarrow$ `"Error!"`

2. Moreover: Avoid duplication of text and code

Outputting Grades with `if` Statements

```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Disadvantage: Control flow – and thus program behaviour – not quite obvious

---

Outputting Grades with `switch` Statement

```
switch (grade) {  ←——————————  Jump to matching case
  case 6: std::cout << "Excellent ... ";
  case 5:                                   Fall-through
  case 4: std::cout << "You passed!";
    break;  ←——————————————————  Exit switch
  case 3: std::cout << "Close, but ... ";
  case 2:                                   Fall-through
  case 1: std::cout << "You failed!";
    break;  ←——————————————————  Exit switch
  default: std::cout << "Error!";  ←——  In all other cases
}
```

Advantage: Control flow clearly recognisable

The `switch`-Statement

`switch` (*condition*)
statement

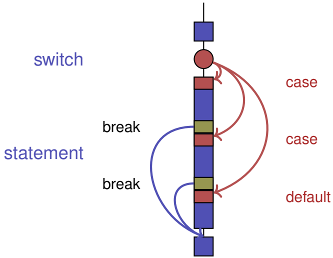
- *condition*: Expression, convertible to integral type
- *statement* : arbitrary statemet, in which `case` and `default`-lables are permitted, `break` has a special meaning.
- Use of fall-through property is controversial and should be carefully considered (corresponding compiler warning can be enabled)

Semantics of the `switch`-statement

`switch` (*condition*)
statement

- `condition` is evaluated.
- If `statement` contains a `case`-label with (constant) value of `condition`, then jump there
- otherwise jump to the `default`-lable, if available. If not, jump over `statement`.
- The `break` statement ends the `switch`-statement.

Control Flow `switch`



---

7. Floating-point Numbers I

Types `float` and `double`; Mixed Expressions and Conversion; Holes in the Value Range

"Proper" Calculation

```
// Program: fahrenheit_float.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
  // Input
  std::cout << "Temperature in degrees Celsius =? ";
  float celsius;
  std::cin >> celsius;

  // Computation and output
  std::cout << celsius << " degrees Celsius are "
            << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
  return 0;
}
```

Fixed-point numbers

- fixed number of integer places (e.g. 7)
- fixed number of decimal places (e.g. 3)

`0.0824 = 0000000.082` ← third place truncated

Disadvantages

- Value range is getting *even* smaller than for integers.
- Representability depends on the position of the decimal point.

Floating-point numbers

- Observation: same number, different representations with varying "efficiency", e.g.

$$\begin{aligned} 0.0824 &= 0.00824 \cdot 10^1 &&= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &&= 824 \cdot 10^{-4} \end{aligned}$$

  Number of *significant digits* remains constant

- Floating-point number representation thus:
  - Fixed number of significant places (e.g. 10),
  - Plus position of the decimal point via exponent
  - Number is $\textit{Mantissa} \times 10^{\textit{Exponent}}$

Types `float` and `double`

- are the fundamental C++ types for floating point numbers
- approximate the field of real numbers $(\mathbb{R}, +, \times)$ from mathematics
- have a big value range, sufficient for many applications:
  - **`float`**: approx. 7 digits, exponent up to ± 38
  - **`double`**: approx. 15 digits, exponent up to ± 308
- are fast on most computers (hardware support)

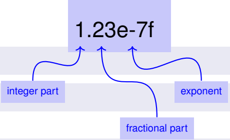
Arithmetic Operators

Analogous to **`int`**, but . . .

- Division operator / models a “proper” division (real-valued, not integer)
- No modulo operator, i.e. no **%**

Literals

are different from integers by providing



- decimal point
  - `1.0` : type **`double`**, value 1
  - `1.27f` : type **`float`**, value 1.27
- and / or exponent.
  - `1e3` : type **`double`**, value 1000
  - `1.23e-7` : type **`double`**, value $1.23 \cdot 10^{-7}$
  - `1.23e-7f` : type **`float`**, value $1.23 \cdot 10^{-7}$

Computing with `float`: Example

Approximating the Euler-Number

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx 2.71828\ldots$$

using the first 10 terms.

Computing with `float`: Euler Number

```
std::cout << "Approximating the Euler number... \n";

// values for i−th iteration, initialized for i = 0
float t = 1.0f; // term 1/i!
float e = 1.0f; // i−th approximation of e

// iteration 1, ..., n
for (unsigned int i = 1; i < 10; ++i) {
    t /= i;    // 1/(i−1)! −> 1/i!
    e += t;
    std::cout << "Value after term " << i << ": "
              << e << "\n";
}
```

Computing with `float`: Euler Number

```
Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828
```

Mixed Expressions, Conversion

- Floating point numbers are more general than integers.
- In mixed expressions integers are converted to floating point numbers.

```
9 * celsius / 5  + 32
```

Holes in the value range

```
float n1;
std::cout << "First number  =? ";
std::cin >> n1;

float n2;
std::cout << "Second number =? ";
std::cin >> n2;

float d;
std::cout << "Their difference =? ";
std::cin >> d;

std::cout << "Computed difference − input difference = "
          << n1 − n2 − d << "\n";
```

input 1.1

input 1.0

input 0.1

output 2.23517e-8

What is going on here?

Value range

Integer Types:

- Over- and Underflow relatively frequent, but ...
- the value range is contiguous (no holes): $\mathbb{Z}$ is “discrete”.

Floating point types:

- Overflow and Underflow seldom, but ...
- there are holes: $\mathbb{R}$ is “continuous”.