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;
}
```

main block

block

„Blickrichtung“ ←

Potential Scope

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; ... }
```

Potential Scope

in the block

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

scope

in function body

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

scope

in control statement

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

scope

Scope

```
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;
}
```

Potential Scope

```
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;
}
```

Real Scope

```
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;
}
```

Local Variables

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

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

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

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

`while` Statement

```
while (condition)
  statement
```

`while` Statement

```
while (condition)
  statement
```

is equivalent to

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

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

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

The Collatz-Sequence

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

n=5: 5

The Collatz-Sequence

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

n=5: 5, 16

The Collatz-Sequence

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

n=5: 5, 16, 8

The Collatz-Sequence

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

n=5: 5, 16, 8, 4

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4, 2

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1

The Collatz-Sequence

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

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

`do` Statement

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

`do` Statement

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

is equivalent to

```
statement
while (condition)
  statement
```

`break` and `continue` in practice

- Advantage: Can avoid nested **`if-else`** blocks (or complex disjunctions)

`break` and `continue` in practice

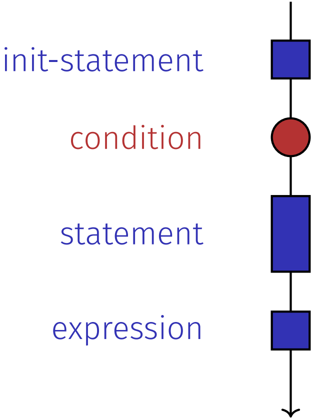
- Advantage: Can avoid nested **`if-else`**blocks (or complex disjunctions)
- But they result in additional jumps and thus potentially complicate the control flow

`break` and `continue` in practice

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

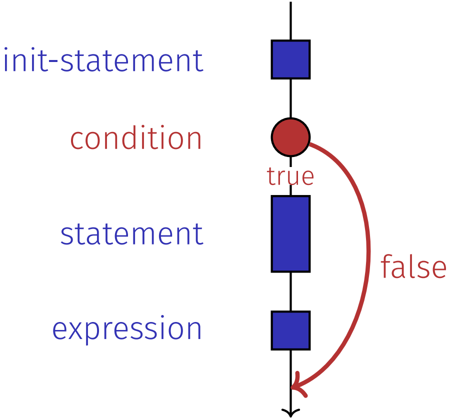
Control Flow `for`

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



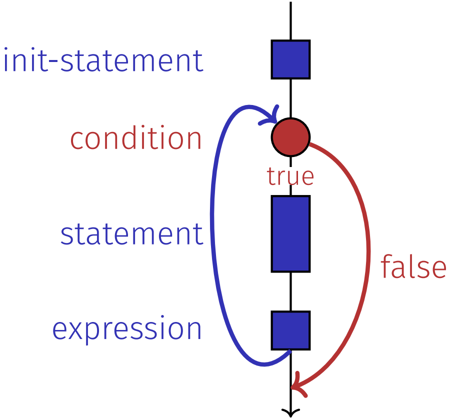
Control Flow `for`

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

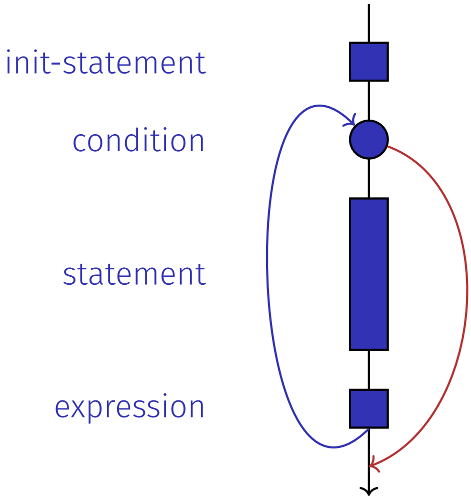


Control Flow `for`

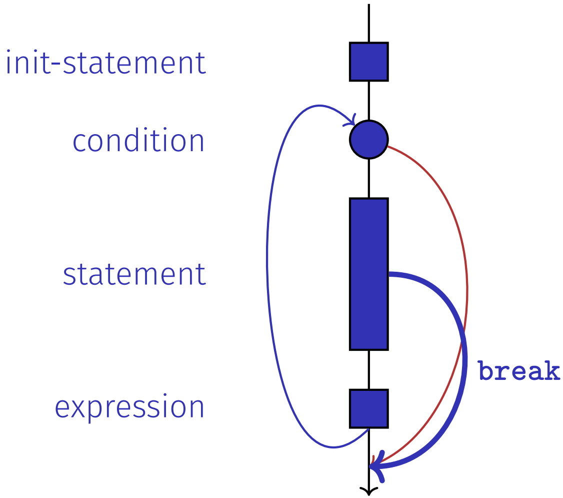
`for` (*init statement* *condition* **;** *expression*)
    statement



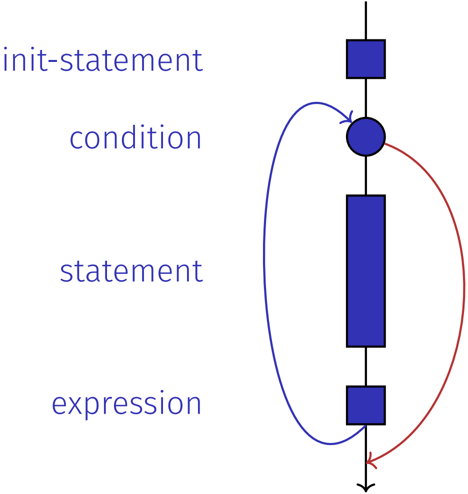
Control Flow `break` and `continue` in for



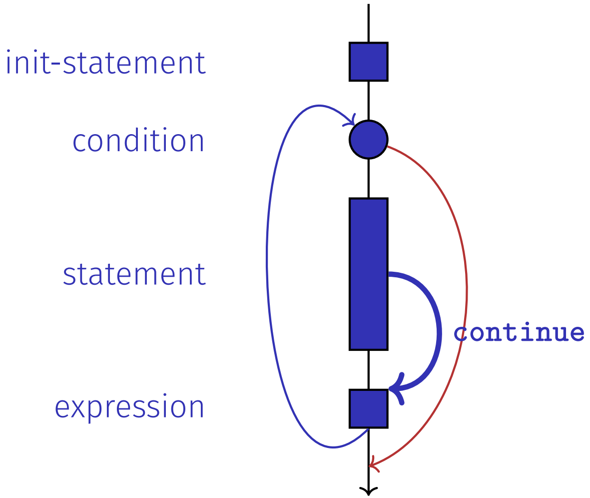
Control Flow *break* and continue in for



Control Flow `break` and `continue` in `for`



Control Flow **`break`** and ***`continue`*** in for



Control Flow: the Good old Times?

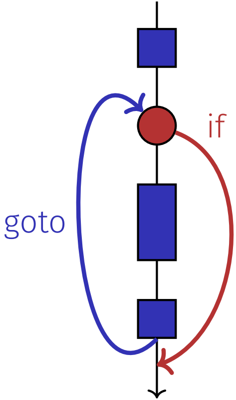
Observation

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

Control Flow: the Good old Times?

Observation

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



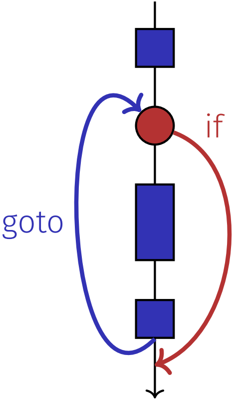
Control Flow: the Good old Times?

Observation

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

Languages based on them:

- Machine Language



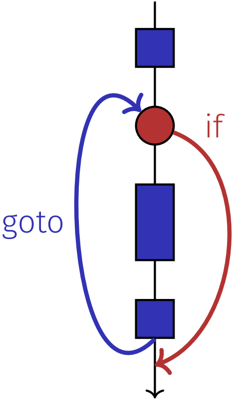
Control Flow: the Good old Times?

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)



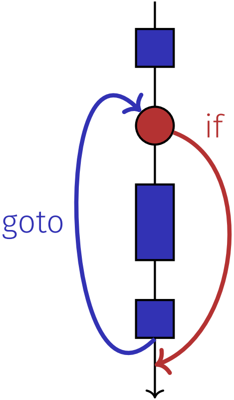
Control Flow: the Good old Times?

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 programming language for the general public (1964)



BASIC and home computers...

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

Home-Computer Commodore C64 (1982)

Spaghetti-Code with `goto`

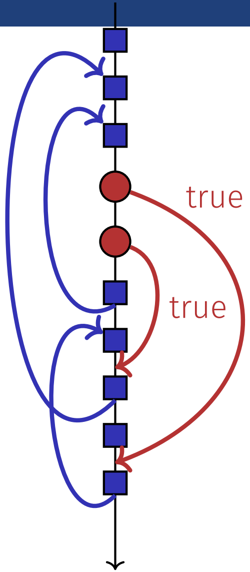
Output of of ???????????
using the programming language BASIC:

```
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
```

Spaghetti-Code with `goto`

Output 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

The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements

The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements
- few lines of code

The “right” Iteration Statement

Goals: readability, conciseness, in particular

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

The “right” Iteration Statement

Goals: readability, conciseness, in particular

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

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 $\{0, \ldots, 100\}$

First (correct) attempt:

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

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

Less statements, **less** lines:

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

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

Less statements, **simpler** control flow:

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

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

Less statements, **simpler** control flow:

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

This is the “right” iteration statement

Outputting Grades

1. Functional requirement:

$$
\begin{aligned}
6 &\rightarrow \texttt{"Excellent ... You passed!"} \\
5, 4 &\rightarrow \texttt{"You passed!"} \\
3 &\rightarrow \texttt{"Close, but ... You failed!"} \\
2, 1 &\rightarrow \texttt{"You failed!"} \\
\textit{otherwise} &\rightarrow \texttt{"Error!"}
\end{aligned}
$$

Outputting Grades

1. Functional requirement:

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

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!";
```

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) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

← Jump to matching `case`

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Fall-through

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Fall-through

Exit `switch`

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Fall-through

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Fall-through

Exit `switch`

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

← In all other cases

Outputting Grades with `switch` Statement

```
switch (grade) {
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Advantage: Control flow clearly recognisable

The `switch`-Statement

```
switch (expression)
  statement
```

- *expression*: Expression, convertible to integral type
- *statement* : arbitrary statemet, in which **`case`** and **`default`**-lables are permitted, **`break`** has a special meaning.

The `switch`-Statement

```
switch (expression)
  statement
```

- *expression*: 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)

7. Floating-point Numbers I

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

“Proper” Calculation

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
int celsius;
std::cin >> celsius;

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

28 degrees Celsius are 82 degrees Fahrenheit.

“Proper” Calculation

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
int celsius;
std::cin >> celsius;

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

28 degrees Celsius are 82 degrees Fahrenheit.

richtig wäre 82.4

“Proper” Calculation

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
float celsius; // Enable fractional numbers
std::cin >> celsius;

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

28 degrees Celsius are 82.4 degrees Fahrenheit.

Fixed-point numbers

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

Fixed-point numbers

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

```
82.4 = 0000082.400
```

Fixed-point numbers

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

```
82.4 = 0000082.400
```

Disadvantages

- Value range is getting *even* smaller than for integers.

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

- 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 numbers

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

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

  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 $\mathit{Mantissa} \times 10^{\mathit{Exponent}}$

Types `float` and `double`

- are the fundamental C++ types for floating point numbers
- approximate the field of real numbers $(\mathbb{R}, +, \times)$ from mathematics

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

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)

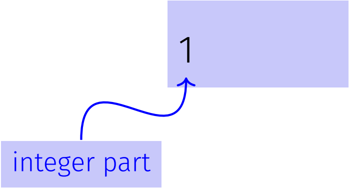
Arithmetic Operators

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

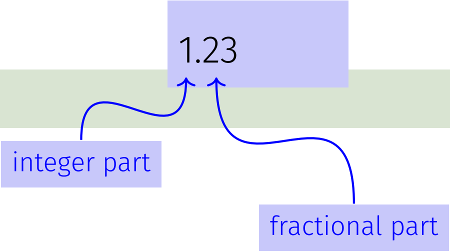


Literals

are different from integers by providing

- decimal point

  `1.0` : type **`double`**, value 1

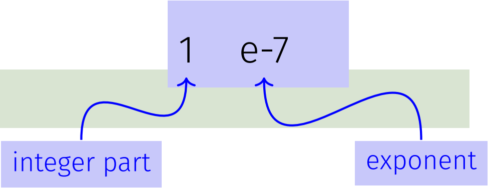


Literals

are different from integers by providing

- decimal point

  `1.0` : type **`double`**, value 1



- or exponent.

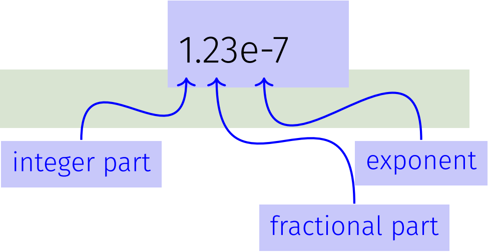
  `1e3` : type **`double`**, value 1000

Literals

are different from integers by providing

- decimal point

  `1.0` : type **`double`**, value 1



- and / or exponent.

  `1e3` : type **`double`**, value 1000

  `1.23e-7` : type **`double`**, value $1.23 \cdot 10^{-7}$

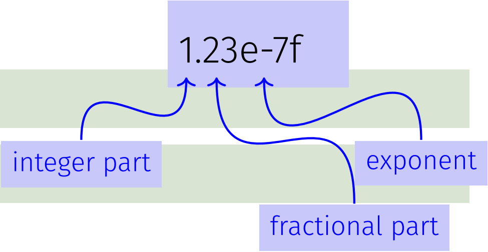
Literals

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.

Mixed Expressions, Conversion

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

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
```

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
```

Typ **`float`**, value 28

Mixed Expressions, Conversion

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

```
9 * 28.0f / 5  + 32
```

Mixed Expressions, Conversion

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

```
9 * 28.0f / 5  + 32
```

is converted to **`float`** : 9.0f

Mixed Expressions, Conversion

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

```
252.0f / 5  + 32
```

is converted to **`float`** : 5.0f

Mixed Expressions, Conversion

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

```
50.4f + 32
```

is converted to **`float`** : 32.0f

Mixed Expressions, Conversion

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

```
                    82.4f
```

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";
```

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 1.5

input 1.0

input 0.5

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 1.5

input 1.0

input 0.5

output 0

Holes in the value range

```
float n1;
std::cout << "First number  =? ";        input 1.1
std::cin >> n1;

float n2;
std::cout << "Second number =? ";        input 1.0
std::cin >> n2;

float d;
std::cout << "Their difference =? ";     input 0.1
std::cin >> d;

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

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

Holes in the value range

```
float n1;
std::cout << "First number  =? ";          input 1.1
std::cin >> n1;

float n2;
std::cout << "Second number =? ";          input 1.0
std::cin >> n2;

float d;
std::cout << "Their difference =? ";       input 0.1
std::cin >> d;

std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << "\n";          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”.

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”.