One can, e.g., distinguish:

scripting languages
- bash, csh $\rightarrow$ Unix shell
- Perl, Python
- IRAF, IDL, Midas $\rightarrow$ especially for data reduction in astrophysics

compiler-level languages
- C/C++ $\rightarrow$ very common, therefore our favorite language
- Fortran $\rightarrow$ very common in astrophysics, especially in radiative transfer
Programming languages II

<table>
<thead>
<tr>
<th>scripting language</th>
<th>compiler-level language</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples</td>
<td></td>
</tr>
<tr>
<td>shell (bash, tcsh), Perl, Mathematica, MATLAB, ...</td>
<td>C/C++, Fortran, Pascal, ...</td>
</tr>
<tr>
<td>source code</td>
<td></td>
</tr>
<tr>
<td>directly executable</td>
<td>translated to</td>
</tr>
<tr>
<td></td>
<td>machine code, e.g.,</td>
</tr>
<tr>
<td></td>
<td>0x90  $\rightarrow$ no operation (NOP)</td>
</tr>
<tr>
<td>runtime behavior</td>
<td></td>
</tr>
<tr>
<td>interpreter runs as a program $\rightarrow$ full control over</td>
<td></td>
</tr>
<tr>
<td>execution $\rightarrow$ error messages, argument testing</td>
<td></td>
</tr>
<tr>
<td>speed</td>
<td></td>
</tr>
<tr>
<td>usually slow</td>
<td>very fast by optimization</td>
</tr>
<tr>
<td>$\rightarrow$ analysis tools</td>
<td>$\rightarrow$ simulations, number crunching</td>
</tr>
</tbody>
</table>

$\rightarrow$ moreover, also bytecode compiler (JAVA) for virtual machine, Just-in-time (JIT) compiler (JavaScript, Perl)
C is a *procedural* (imperative) language

C++ is an *object oriented* extension of C with the same syntax

C++ is because of its additional structures (template, class) "\(\Rightarrow\)" C

**Basic structure of a C++ program**

```cpp
#include <iostream>
using namespace std;
int main () {
    instructions of the program ;
    // comment
    return 0 ;
}
```

every instruction must be finished with a `;` (semicolon) !
Compiling a C++ program:

- Source file: `.cpp, .C`
- Compiler + Linker: `.o, .so, .a`
- Executable program: `a.out, program`
Command for compiling + linking:

```
g++ -o program program.cpp
```

(GNU compiler for C++)

- only compiling, do not link:
  ```
g++ -c program.cpp
  ```
  creates `program.o` (object file, not executable)

- option `-o name` defines a name for a file that contains the executable program, otherwise program file is called: `a.out`
  the name of the executable program can be arbitrarily chosen
Example: C++ output via streams

```cpp
#include <iostream>

using namespace ::std ;

int main () {
    cout << endl << "Hello world!" << endl ;
    return 0 ; // all correct
}
```
Simple program for output on screen II

- `<iostream>` ... is a C++ library (input/output)
- `main()` ... program (function)
- `return 0` ... returns the return value 0 to main (all ok)
- source code can be freely formatted, i.e., it can contain an arbitrary number of spaces and empty lines (white space) → useful for visual structuring
- comments are started with `//` - everything after it (in the same line) is ignored,
  C has only `/* comment */` for comment blocks
- `cout` ... output on screen/terminal (C++)
- `<<` ... output/concatenate operator (C++)
- `string "Hello world!"` must be set in quotation marks
- `endl` ... manipulator: new line and stream flush (C++)
- a block several instructions which are hold together by curly braces
C/C++ is a procedural language
The procedures of C/C++ are functions.

- Main program: function with specific name `main()`{}
- every function has a type (for return), e.g.: `int main (){}`
- functions can get arguments by call, e.g.:
  `int main (int argc, char *argv[]){}`
- functions must be *declared before* they can be called in the main program,
  e.g., `void swap(int &a, int &b) ;`
  or included via a header file:
  `#include <cmath>`
- within the curly braces `{ }`, the so-called function body, is the *definition* of the function (what shall be done how), e.g.:
  `int main () { return 0 ; }`
```cpp
#include <iostream>
using namespace std;

float cube(float x) {
    return x*x*x;
}

int main() {
    float x = 4.;
    cout << cube(x) << endl;
    return 0;
}
```

A variable is a piece of memory.

in C/C++ data types are explicit and static

We distinguish regarding visibility ("scope"):

- global variables → declared outside of any function, before main
- local variables → declared in a function or in a block `{ }`, only there visible

...regarding data types → intrinsic data types:

- int → integer, e.g., `int n = 3 ;`
- float → floats (floating point numbers),
  e.g., `float x = 3.14, y = 1.2E-4 ;`
- char → characters, e.g., `char a_character ;`
- bool → logical (boolean) variables, e.g., `bool btest = true ;`
Integer numbers are represented exactly in the memory with help of the binary number system (base 2), e.g.

\[ 13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \equiv 1101 \quad \text{(binary)} \]

In the assignment

\[ a = 3 \]

3 is an integer literal (literal constant). Its bit pattern \( 3 = 1 \cdot 2^0 + 1 \cdot 2^1 \equiv 11 \) is inserted at the corresponding positions by the compiler.

\(^1\text{doesn’t correspond necessarily to the sequential order used by the computer → “Little Endian”: store least significant bit first, so actually: 1011}\)
on 64-bit systems

int
  compiler reserves 32 bit (= 4 byte) of memory
  1 bit for sign and
  \(2^{31} = 2\,147\,483\,648\) values (incl. 0): \(\rightarrow\) range:
  int = \(-2\,147\,483\,648 \ldots + 2\,147\,483\,647\)

unsigned int
  32 bit, no bit for sign \(\rightarrow\) \(2^{32}\) values (incl. 0)
  unsigned int = 0 \ldots 4\,294\,967\,295

long
  on 64 bit systems: 64 bit (= 8 byte),
  1 bit for sign: \(-9.2 \times 10^{18} \ldots 9.2 \times 10^{18}\) (quintillions)

unsigned long
  64 bit without sign: 0 \ldots 1.8 \times 10^{19}

and also: char (1 byte), smallest addressable (!); short (2 byte); long long (8 bytes)
Two’s complement

Table: Representation: unsigned value (0s), value and sign (sig), two’s complement (2’S) for a nibble (½ byte)

<table>
<thead>
<tr>
<th>binary</th>
<th>0s</th>
<th>sig</th>
<th>2’S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-0</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-1</td>
<td>-7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-7</td>
<td>-1</td>
</tr>
</tbody>
</table>

Disadvantages of representation as value and sign:
∃ 0 and -0; Which bit is sign?
(→ const number of digits, fill up with 0s);

Advantage of 2’S:
negative numbers always with highest bit=1

→ cf. +1 + −1 bitwise for value & sign vs. 2’S
Floating point data types

Floating point numbers are an **approximate** representation of real numbers. Floating point numbers can be declared via, e.g.,:

```c
float radius, pi, euler, x, y;
double radius, z;
```

Valid assignments are, e.g.,:

```c
x = 3.0;
y = 1.1E-3;
z = x / y;
```
representation (normalization) of floating point numbers are described by standard IEEE 754:

\[ x = s \cdot m \cdot b^e \]  

(1)

with base \( b = 2 \) (IBM Power6: also \( b = 10 \)), sign \( s \), and normalized significand (mantissa) \( m \), bias

So for 32 Bit (Little Endian\(^\dagger\)), 8 bit exponent, 23 bit mantissa:

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)

\( M \)
• mantissa is *normalized* to the form (e.g.)

\[ 1,0100100 \times 2^4 \]

i.e. with a 1 before the decimal point. This 1 is not stored, so \( m = 1.f \)

Moreover, a bias (127 for 32 bit, 1023 for 64 bit) is added to the exponent (results in non-negative integer)

---

**Example: Conversion of a decimal number to IEEE-32-Bit**

172.625 base 10

\[ 10101100.101 \times 2^0 \] base 2

\[ 1.0101100101 \times 2^7 \] base 2 normalized

add bias of 127 to exponent = 134 = \[ 1 \cdot 2^7 + \ldots + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \]

0 10000110 0101100101000000000000000000000000
Floating point data types IV

- single precision (32 bit) float: exponent 8 bit, significand 23 bit
  
  \[-126 \leq e \leq 127 \text{ (basis 2)}\]
  
  \[\rightarrow \approx 10^{-45} \ldots 10^{38}\]

  digits: 7-8 (\(= \log 2^{23+1} = 24 \log 2\))

- for 64 bit (double precision) – double: exponent 11 bit, significand 52 bit
  
  \[-1022 \leq e \leq 1023 \text{ (basis 2)}\]
  
  \[\rightarrow \approx 10^{-324} \ldots 10^{308}\]

  digits: 15-16 (\(= \log 2^{52+1}\))
some real numbers cannot be presented exactly in the binary numeral system (cf. 1/3 in decimal):

\[ 0.1 \approx 1.10011001100110011001101 \times 2^{-4} \]  

(2)

**Warning**

Do not compare two floating point numbers blindly for equality (e.g., 0.362 * 100.0 == 36.2), but rather use an accuracy limit:

\[ \text{abs}(x - y) \leq \text{eps}, \text{ better: relative error } \]

\[ \text{abs}(1-y/x) \leq \text{eps} \]
Floating point arithmetics

Subtraction of floating point numbers

Consider $1.000 \times 2^5 - 1.001 \times 2^1$ (only 3 bit mantissa)

→ *bitwise subtraction*, requires same exponent

\[
\begin{array}{rcl}
1.0000000 \times 2^5 & - & 0.0001001 \times 2^5 \\
\hline 
0.1110111 \times 2^5 & \text{infinite precision} & \\
1.110111 \times 2^4 & \text{shifted left to normalize} & \\
1.111 \times 2^4 & \text{rounded up, as last digits} > 1/2 \text{ ULP}^\dagger & \\
\end{array}
\]

\dagger \text{unit in the last place} = \text{spacing between subsequent floating point numbers}
Floating point data types VII

Properties of floating point arithmetics (limited precision):

- loss of significance / catastrophic cancellation: occurs for subtraction of almost equal numbers

Example for loss of significance

\[ \pi - 3.141 = 3.14159265 \ldots - 3.141 \text{ with 4-digit mantissa}; \text{ maybe expected: } = 0.00059265 \ldots \approx 5.927 \times 10^{-4}; \text{ in fact: } 1.0000 \times 10^{-3}, \text{ because } \pi \text{ is already rounded to } 3.142 \]

- absorption (numbers of different order of magnitude): addition of subtraction of a very small number does not change the larger number

Example for absorption

for 4-digit mantissa: \[ 0.001 + 100 = 100: 1.000 \times 10^2 + 1.000 \times 10^{-3} = 1.000 \times 10^2 + 0.000 \ 01 \times 10^2 = 1.000 \times 10^2 + 0.000 \times 10^2 = 1.000 \times 10^2, \] same for subtraction
distributive and associative law usually not fulfilled, i.e. in general

\[(x + y) + z \neq x + (y + z)\] \hspace{1cm} (3)
\[(x \cdot y) \cdot z \neq x \cdot (y \cdot z)\] \hspace{1cm} (4)
\[x \cdot (y + z) \neq (x \cdot y) + (x \cdot z)\] \hspace{1cm} (5)
\[(x + y) \cdot z \neq (x \cdot z) + (y \cdot z)\] \hspace{1cm} (6)

solution of equations, e.g., \((1 + x) = 1\) for 4-bit mantissa solved by any \(x < 10^{-4}\) (see absorption) \(\rightarrow\) smallest float number \(\epsilon\) with \(1 + \epsilon > 1\) called machine precision

Multiplication and division of floating point numbers:
mantissas multiplied/divided, exponents added/subtracted
\(\rightarrow\) no cancellation or absorption problem
Guard bit, round bit, sticky bit (GRS)

- in floating point arithmetics: if mantissa shifted right → loss of digits
- therefore: during calculation 3 extra bits (GRS)
  - Guard bit: 1st bit, just extended precision
  - Round bit: 2nd (Guard) bit, just extended precision (same as G)
  - Sticky bit: 3rd bit, set to 1, if any bit beyond the Guard bits non-zero, stays then 1(!) → sticky

example

<table>
<thead>
<tr>
<th>G</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
</table>
Before 1st shift: 1.110000000000000000000000000100 0 0 0
After 1 shift: 0.111000000000000000000000000010 0 0 0
After 2 shifts: 0.011100000000000000000000000001 0 0 0
After 3 shifts: 0.001110000000000000000000000000 1 0 0
After 4 shifts: 0.000111000000000000000000000000 0 1 0
After 5 shifts: 0.000011100000000000000000000000 0 0 1
After 6 shifts: 0.000001110000000000000000000000 0 0 1
After 7 shifts: 0.000000111000000000000000000000 0 0 1
After 8 shifts: 0.000000011100000000000000000000 0 0 1
### GRS bits – possible values and stored values

<table>
<thead>
<tr>
<th>extended sum</th>
<th>stored value</th>
<th>why</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0100 000</td>
<td>1.0100</td>
<td>truncated because of GR bits</td>
</tr>
<tr>
<td>1.0100 001</td>
<td>1.0100</td>
<td>truncated because of GR bits</td>
</tr>
<tr>
<td>1.0100 010</td>
<td>1.0100</td>
<td>rounded down because of GR bits</td>
</tr>
<tr>
<td>1.0100 011</td>
<td>1.0100</td>
<td>rounded down because of GR bits</td>
</tr>
<tr>
<td>1.0100 100</td>
<td>1.0100</td>
<td>rounded down because of S bit</td>
</tr>
<tr>
<td>1.0100 101</td>
<td>1.0101</td>
<td>rounded up because of S bit</td>
</tr>
<tr>
<td>1.0100 110</td>
<td>1.0101</td>
<td>rounded up because of GR bits</td>
</tr>
<tr>
<td>1.0100 111</td>
<td>1.0101</td>
<td>rounded up because of GR bits</td>
</tr>
</tbody>
</table>
IEEE representation of 32 bit floats:

<table>
<thead>
<tr>
<th>Number name</th>
<th>sign, exp., f</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>$0 &lt; e &lt; 255$</td>
<td>$(-1)^s \times 2^{e-127} \times 1.f$</td>
</tr>
<tr>
<td>subnormal</td>
<td>$e = 0, f \neq 0$</td>
<td>$(-1)^s \times 2^{-(127)} \times 0.f$</td>
</tr>
<tr>
<td>signed zero (±0)</td>
<td>$e = 0, f = 0$</td>
<td>$(-1)^s \times 0.0$</td>
</tr>
<tr>
<td>+∞</td>
<td>$s = 0, e = 255, f = 0$</td>
<td>+INF</td>
</tr>
<tr>
<td>−∞</td>
<td>$s = 1, e = 255, f = 0$</td>
<td>−INF</td>
</tr>
<tr>
<td>Not a number</td>
<td>$e = 255, f \neq 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- if float $> 2^{128}$ → overflow, result may be NaN or unpredictable
- if float $< 2^{-128}$ → underflow, result is set to 0

If not default by compiler: enable floating-point exception handling (e.g., `-fpe-all0` for ifort)
In C/C++ many data type conversions are already predefined, which will be invoked automatically:

```c
int main () {
    int a = 3 ;
    double b ;
    b = a ;       // implicit conversion of a to double
    b = 1. / 3 ;  // implicit conversion of 3 to double
    return 0.2 ;  // implicit conversion of 0.2 to integer 0
}
```
Moreover, a type conversion/casting can be done explicitly:

```c
int main () {
    int a = 3;
    double b;
    b = (double) a; // type cast
    return 0;
}
```

- obviously possible: integer ↔ floating point
- but also: pointer (see below) ↔ data types
- Caution: For such C casts there is no type checking during runtime!
The better way: use the functions of the same name for type conversion

```c
int i, k = 3;
float x = 1.5, y;
i = int(x) + k;
y = float(i) + x;
```
Logical variables

```cpp
bool b;
```

Intrinsic data type, has effectively only two different values:

```cpp
bool btest, bdo;
btest = true; // = 1
bdo = false; // = 0
```

But also:

```cpp
btest = 0.; // = false
btest = -1.3E-5; // = true
```

Output via `cout` yields 0 or 1 respectively. By using `cout << boolalpha << b;` is also possible to obtain `t` and `f` for output.

Note: Minimum addressable piece of memory is 1 byte $\rightarrow$ `bool` needs more memory than necessary.
Executable control constructs modify the program execution by selecting a block for repetition (loops, e.g., for) or branching to another statement (conditional, e.g., if/unconditional, e.g., goto).

Repeated execution of an instruction/block:

**for loop**

```cpp
for (int k = 0 ; k < 6 ; ++k ) sum = sum + 7 ;

for (float x = 0.7 ; x < 17.2 ; x = x + 0.3) {
    y = a * x + b ;
    cout << x << " " << y << endl;
}
```
Execution control - for-loops II

Structure of the loop control (header) of the for loop:

There are (up to) three arguments, separated by semicolons:

1. initialization of the loop variable (loop counter), if necessary with declaration, e.g.:
   \[
   \text{int } k = 0 ;
   \]
   → is executed \textit{before the first} iteration

2. condition for termination of the loop, usually via arithmetic comparison of the loop variable, e.g.,
   \[
   k < 10 ;
   \]
   is tested \textit{before each} iteration

3. expression: incrementing/decrementing of the loop variable, e.g.,
   \[
   ++k \text{ or } --k \text{ or } k += 3
   \]
   is executed \textit{after each} iteration

\[\dagger\] interestingly also: \text{int } k = 0, \text{ j } = 1;
Increment operators

\[
\begin{align*}
\text{sum} & \; += \; a \\
\rightarrow \quad \text{sum} & \; = \; \text{sum} \; + \; a \\
++x & \\
\rightarrow \quad \text{x} & \; = \; \text{x} \; + \; 1 \quad \text{(increment operator)} \\
--x & \\
\rightarrow \quad \text{x} & \; = \; \text{x} \; - \; 1 \quad \text{(decrement operator)}
\end{align*}
\]

Note that there is also a \textit{post} increment/decrement operator: \texttt{x++}, \texttt{x--}, i.e. incrementing/decrementing is done \textit{after} any assignment, e.g., \texttt{y = x++}. 
Logical operators I - Comparisons/inequalities

→ return either(!) true or false:

\[ a > b \quad \text{greater than} \]
\[ a >= b \quad \text{greater than or equal} \]
\[ a == b \quad \text{equal} \]
\[ a != b \quad \text{not equal} \]
\[ a <= b \quad \text{less than or equal} \]
\[ a < b \quad \text{less than} \]

Caution!
The exact equality == should not be used for float-type variables because of the limited precision in the representation.
Moreover, there exist also:

**while loops**

```
while (x < 0.) x = x + 2. ;
```

```
do x = x + 2. ; // do loop is executed
while (x < 0.) ; // at least once!
```

**Instructions for loop control**

```
break ; // stop loop execution / exit current loop
continue ; // jump to next iteration
```
Loops II

In C/C++: no real “for loops”

→ loop variable (counter, limits) can be changed in loop body
slow, harder to optimize for compiler/processor

Recommendation: *local* loop variables

→ declaration in loop header
→ scope limited to loop body
Conditional execution via if:

```cpp
if (z != 1.0) k = k + 1 ;
```

Conditional/branching

```cpp
if (a == 0) cout << "result" ; // one-liner

if (a == 0) a = x2 ; // branching
else if (a > 1) {
    a = x1 ;
}
else a = x3 ;
```
If the variable used for branching has only discrete values (e.g., int, char, but not floats!), it is possible to formulate conditional statements via switch/case:

```java
switch (Ausdruck) {
    case value1 : instruction ; break ;
    case value2 : instruction1 ;
                 instruction2 ; break ;
    default    : instruction ;
}
```

Heads up!

Every case instruction section should be finished with a break, otherwise the next case instruction section will be executed automatically.
Example: switch

```c++
int k;
cout << "Please enter number, 0 or 1: " ;
cin >> k;
switch(k) {
    case 0 : cout << "pessimist" << endl ; break ;
    case 1 : cout << "optimist" << endl ; break ;
    default : cout << "neutral" << endl ;
}
```
Declarations of variables should be at the beginning of a block, exception: loop variables

```c
float x, y; // declaration of x and y
int n = 3; // declaration and initialization of n
```

Local variables / variables in general

- are only visible within the block (e.g., in `int main() { }`), where they have been declared
- are **local** regarding this block, their value can only be changed within this block
- are unknown outside of this block, i.e., they don’t exist there
Global variables

- must be declared outside of any function, e.g., before `main()`
- are visible/known to all following functions within the same program
- have file wide visibility (i.e., if you split your source code into different files, you have to put the declaration into every file)
- are only removed from memory when execution of the program is ended

A locally declared variable will hide a global variable of the same name. The global variable can be still accessed with help of the scope operator `::`, e.g., `cout << ::m ;`
Global and local variables

```c
int m = 0; // global variable

void calc() {
    int k = 0; // local variable
    m = 1; // ok, global variable
    ++j; // error, as j only known in main
}

int main() {
    int j = 3;
    ++j; // ok
    for (int i = 1; i < 10; ++i) {
        j = m + i; // ok, all visible
    }
    m = j - i; // error: i not visible
    return j;
}
```
Values (e.g., numbers) that do not change during the program execution, should be *defined* as constants:

```c
const float e = 2.71828;
```

Constants must be initialized during declaration.

After initialization their value cannot be changed.

Use `const` whenever possible!
char character;

are encoded as integer numbers:

char character = 'A';
char character = 65;

mean the same character (ASCII code)

Assignments of character literals to character variables require single quotation marks ‘:

char yes = 'Y';
Arrays in C/C++

Static array declaration for a one-dimensional array of type `double`:

```c
double a[5] ;
```

one-dimensional array with 5 elements of type `double` (e.g., vectors)

Access to individual elements:

```c
```

Heads up!

In C/C++ the index for arrays starts always at 0 and runs in this example until 4, so the last element is `a[4]`.

A common source of errors in C/C++ !!!

Note: While the size of the array can be set during runtime, the size cannot be changed after declaration (static declaration).
an $m \times n$ matrix (rows $\times$ columns):

$$n \text{ columns } \rightarrow \\
\begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & \ldots \\
\vdots \\
a_{m1} & \ldots & a_{mn}
\end{pmatrix}$$

int a[m][n] \ldots static allocation of two-dimensional array, e.g., for matrices ($m$, $n$ must be constants)

access via, e.g., a[i][j]

i is the index for the rows,
j for the columns.
Two-dimensional arrays II

e.g., \( a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \)

Note that in C/C++ the second (last) index runs first, i.e. the entries of \( a[2][3] \) are in this order in the memory:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
a[0][0] & a[0][1] & a[0][2] & a[1][0] & a[1][1] & a[1][2]
\end{array}
\]

(row-major order \( \rightarrow \) stored row by row)
Initialization of arrays

An array can be initialized by curly braces:

```c
int array[5] = {0, 1, 2, 3, 4} ;

short field[] = {0, 1} ; // array field is automatically dimensioned

float x[77] = {0} ; // set all values to 0
```
There are no string variables in C. Therefore strings are written to one-dimensional character arrays:

```c
char text[6] = "Hello" ;
```

The string literal constant "Hello" consists of 5 printable characters and is terminated automatically by the compiler with the null character \0, i.e. the array must have a length of 6 characters! Note the double quotation marks!

Example

```c
char text[80] ;
cout << endl << "Please enter a string:" ;
cin >> text ;
cout << "You have entered " << text << "." << endl ;
```
Pointer variables – or pointer for short – allow a direct access (i.e. not via the name) to a variable.

**Declaration of pointers**

```c
int *pa ; // pointer to int
float *px ; // pointer to float
int **ppb ; // pointer to pointer to int
```
A pointer is a variable that contains an address, i.e. it points to a specific part of the memory. As every variable in C/C++ a pointer variable must have a data type. The value at address (memory) to which the pointer points, must be of the declared data type.

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5</td>
<td>x</td>
</tr>
<tr>
<td>1004</td>
<td>42</td>
<td>n</td>
</tr>
<tr>
<td>1008</td>
<td>3.141...</td>
<td>d</td>
</tr>
<tr>
<td>1012</td>
<td>...5926</td>
<td></td>
</tr>
<tr>
<td>1016</td>
<td>H E Y !</td>
<td>salutation</td>
</tr>
<tr>
<td>1020</td>
<td>1000</td>
<td>px</td>
</tr>
<tr>
<td>1024</td>
<td>1008</td>
<td>pd</td>
</tr>
<tr>
<td>1028</td>
<td>1004</td>
<td>pn</td>
</tr>
<tr>
<td>1032</td>
<td>1016</td>
<td>psalutation</td>
</tr>
<tr>
<td>1036</td>
<td>1028</td>
<td>pp</td>
</tr>
</tbody>
</table>
Pointers must be always **initialized** before usage!

**Initialization of pointers**

```c
int *pa ; // pointer to int
int b ;   // int
pa = &b ; // assigning the address of b to a
```

The character `&` is called the address operator ("address of") (not to be confused with the reference `int &i = b ;`).

**Declaration and initialization**

```c
int b ;
int *pa = &b ;
```

→ **content of pa = address of b**
With help of the **dereference operator** `*` it is possible to get access to the value of the variable `b`, one says, pointer `pa` is dereferenced:

**Dereferencing a pointer**

```cpp
int b, *pa = &b ;
*pa = 5 ;
```

Here, `*` is the **dereference operator** and means “value at address of ...”. The part of the memory to which `pa` points, contains the value `5`, that is now also the value of the variable `b`.

```cpp
cout << b << endl ; // yields 5
cout << pa << endl ; // e.g., 0x7fff5fbff75c
```
Once again:

Pointer declaration:

```c
float *pz, a = 2.1;
```

Pointer initialization:

```c
pz = &a;
```

Result – output:

```c
cout << "address of variable a (content of pz): " << pz << endl;
cout << "content of variable a: " << *pz << endl;
pz = 5.2; // change value of a
```
int &n = m ;
m2 = n + m ;

- A **reference** is a new name, an **alias for a variable**. So, it is possible to address the same part of the memory (variable) by different names within the program. Every modification of the reference is a modification of the variable itself - and vice versa.

- References are declared via the **& character (reference operator)** and **must** be initialized instantaneously:

  ```c
  int a ;
  int &b = a ;
  ```

- This initialization cannot be changed any more within the program!
Structure of functions – definition

```c
    type name (arg1, ...) { ... }
```  

example: `int main (int argc, char *argv[]) { }`

- in parenthesis: arguments of the function / formal parameters
- when function is called: copy arguments (values of the given variables) to function context → call by value / pass by value

```c
    setzero (float x) { x = 0.; }
    int main () {
        float y = 3.;
        setzero (y);
        cout << y; // prints 3. }
```
Call by value

Pros:
- the value of a passed variable cannot be changed unintentionally within the function

Cons:
- the value of a passed variable can also not be changed on purpose
- for every function call all value must be copied → extra overhead (time)
  (exception: if parameter is an array, only start address is passed → pointer)
Structure of functions: Call by reference

```c
void swap(int &a, int &b) ;
```

Passing arguments as references:

The variables passed to the function `swap` are changed in the function and keep these values after returning from `swap`.

```c
void swap (int &a, int &b) {
    int t = a ; a = b ; b = t ; }
```

→ and called via: `swap (n, m) ;`

Thereby we can pass an arbitrary number of values back from a function.

**Hint:** The keyword `const` prevents that a passed argument can be changed within the function:

```c
sum (int const &a, int const &b) ;
```
A function for swapping two int variables can also be written by using pointers:

```c
void swap(int *a, int *b) { // pointers as formal parameters
    int t = *a; *a = *b; *b = t;
}
```

Call in `main()`:

```c
swap (&x, &y); // Passing addresses (!)
// of x and y
```

Passing arrays to functions

In contrast to (scalar) variables, arrays are automatically passed by address (pointer) to functions, e.g.,

```c
myfunc ( float x[], )
```
Pointers and references

Pointer variables
- store addresses
- must be dereferenced (to use the value of the spotted variable)
- can be assigned as often as desired to different variables (of the same, correct type) within the program

References
- are alias names for variables,
- can be used by directly using their names (without dereferencing)
- the (necessary!) initialization at declaration cannot be changed later
Besides the intrinsic (/basic) data types there are many other data types, which can be defined by the programmer.

```c
struct complex {
    float re ;
    float im ;
} ;
```

\[ ^a \text{Note the necessary semicolon after the } \} \text{ for structs} \]

In this example the data type `complex` is defined, it contains the *member variables* for real and imaginary part.
Structs can be imagined as collections of variables.

```c
struct
{
    char full_name[30] ;
    unsigned short binarity ;
    float luminosity_lsun ;
} ;
```

These (self defined) data types can be used in the same way as intrinsic data types:

**Declaration of struct objects**

```c
complex z, c ;
star sun ;
```
Concrete structs which are declared in this way are called *instances* or *objects* (→ object-oriented programming) of a class (struct).

**Declaration and initialization**

```c
complex z = {1.1 , 2.2} ;
star sun = {"Sun", 1, 1.0 } ;
```

The access to *member variables* is done by the *member selection operator* . (dot):

**Access to members**

```c
real_part = z.re ;
sun.luminosity_lsun = 1.0 ;
```
It is also possible to define functions (so-called *methods*) within structs:

```cpp
struct complex {
    ...  
    float absolute () {
        return (sqrt(re*re + im*im)) ;
    }
}
complex c = {2., 4.} ;
cout << c.absolute() << endl ;
```

The call of the *member function* is also done with the ., the function (method) is associated with the object.
Output to a file by using library fstream:

1. `#include <fstream>
2. create an object of the class ofstream:
   ```cpp
   ofstream fileout;
   ```
3. method open of the class ofstream:
   ```cpp
   fileout.open("graphic.ps");
   ```
4. writing data: e.g.
   ```cpp
   fileout << x;
   ```
5. close file via method close:
   ```cpp
   fileout.close();
   ```

Alternatively (Unix): Use cout and redirection operator `>` or `>>` of the shell:
```
./program > output.txt
```
By including the `<fstream>` library, one can also read from a file.

### Input from a file

```cpp
char line[132];
ifstream filein; // create ifstream object
filein.open("data.txt"); // open the file
while ( filein.good() ) {
    filein.getline(line,132); // read in line;
    x[i] = atof(line); // read into float array
}
```

The method `good()` checks, whether the end of file (EOF) is reached or an error occurred.
Templates allow to create universal definitions of certain structures. The final realization for a specific data type is done by the compiler.

**Function templates**

```cpp
template <class T> // instead of class also typename
T sqr (const T &x) {
    return x * x ;
}
```

The keyword `template` and the angle brackets `< >` signalize the compiler that `T` is a template parameter. The compiler will process this function if a specific data type is invoked by a function call, e.g.,

```cpp
double w = 3.34 ; int k = 2 ;
cout << sqr(w) << " " << sqr(k) ;
```
Moreover, templates can be used to create structs/classes. For example, the class `complex` of the standard C++ library (`#include <complex>`) is realized as template class:

```cpp
template <class T>
class std::complex {
  T re, im ;
public:
  ...
  T real() const return re ;
}  
```

Therefore, the member variables `re` and `im` can be arbitrary (numerical) data types.
By using typedef `datatype aliasname` one can declare new names for data types:

```cpp
typedef unsigned long large;
typedef char* pchar;
typedef std::complex<double> complex_d;
```

These new type names can then be used for variable declarations:

```cpp
large mmm;
pchar Bpoint;
complex_d z = complex_d(1.2, 3.4);
```

In the last example, the constructor for the class template `complex` gets the same name as the variable through the typedef command.
A major strength of C++ is the ability to handle runtime errors, so called exceptions:

**Throwing exceptions: try – throw – catch**

```cpp
try {
    cin >> x;
    if (x < 0.) throw "Negative value!";
    y = g(x);
}
catch (char* info) { // catch exception from try block
    cout << "Program stops, because of: " << info << endl;
    exit (1);
}

double g(double x) {
    if (x > 1000.) throw "x too large!"; ... }
```
Exception handling – exceptions II

```java
try {
    // ...}

- within a `try` block an arbitrary exception can be thrown

```java
throw e;
```

- throw an exception `e`

- the data type of `e` is used to identify to the corresponding `catch` block to which the program will jump

- exceptions can be intrinsic or self defined data types
catch ( type e ) { ... }

- after a try one or more catch blocks can be defined
- from the data type of e the first matching catch block will be selected
- any exception can be caught by catch (...)
- if after a try no matching catch block is found, the search is continued in the next higher call level
- if no matching block at all is found, the terminate function is called; its default is to call abort
Sometimes it is more convenient to pass the parameters the program needs directly at the call of the program, e.g.,

```
./rstarcalc 3.5 35.3
```

this can be realized with help of the library `stdlib.h`

---

**Read an integer number from command line call**

```c
#include "stdlib.h"
int main (int narg, char *args[]) {
    int k;
    // convert char array to integer
    if (narg > 1) k = atoi(args[1]) ;
}
```

- if the string cannot be converted to `int`, the returned value is 0
- there exist also `atol` and `atof` for conversion to `long` and `float`
Common mistakes in C/C++:

- forgotten semicolon ;

- wrong dimensioning/access to arrays
  
  \[
  \text{int } m[4] ; \text{ imax } = m[4] ; \rightarrow \text{ imax } = m[3] ;
  \]

- wrong data type in instructions / function calls
  
  \[
  \text{float } x ; \ldots \text{ switch (x)}
  \text{ void swap (int } *i, \text{ int } *j) ; \ldots \text{ swap(n,m)} ;
  \]

- confusing assignment operator = with the equality operator ==
  
  \[
  \text{if}(i = j) \rightarrow \text{if}(i == j)
  \]

- forgotten function parenthesis for functions without parameters
  
  \[
  \text{clear} ; \rightarrow \text{clear()};
  \]

- ambiguous expressions
  
  \[
  \text{if (i } == 0 \&\& \text{ ++j } == 1)
  \]
  
  no increment of \( j \), if \( i \neq 0 \)
Some recommendations I

- use always(!) the . for floating point literals: \( x = 1. \div 3. \) instead of \( x = 1 \div 3 \)

- whitespace is for free \( \rightarrow \) use it extensively for structuring your source code (indentation, blank lines)

- comment so that you(!) understand your source code in a year

- use self-explaining variable names, e.g., \( T_{\text{eff}} \) instead of \( T \) (think about searching for this variable in the editor)

- use integer loop variables:
  
  ```
  for (int i = 1; i < n ; ++i) {
    x = x + 0.1 ; ... }
  ```
  instead of
  
  ```
  for (float x = 0.; x < 100. ; x = x + 0.1) {... }
  ```

- take special care of user input, usually: \( t_{\text{input}} \ll t_{\text{calc}} \), so exception catching for input is never wasted computing time