Computational Astrophysics I: Introduction and basic concepts

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

SoSe 2025, 30.4.2025

C/C++ Programming

Programming languages I

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

	scripting language	compiler-level language
examples	shell (bash, tcsh), Perl, Mathematica, MATLAB, ...	C/C++, Fortran, Pascal, ...
source code	directly executable	translated to machine code, e.g., `0x90` $\rightarrow$ no operation (NOP)
runtime behavior	interpreter runs as a program $\rightarrow$ full control over execution $\rightarrow$ error messages, argument testing	error handling difficult $\rightarrow$ task of the programmer, often only crash
speed	usually slow $\rightarrow$ analysis tools	very fast by optimization $\rightarrow$ simulations, number crunching

$\rightarrow$ moreover, also bytecode compiler (JAVA) for virtual machine,
Just-in-time (JIT) compiler (JavaScript, Perl)

C/C++ I

- 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) $\gg$ C

Basic structure of a C++ program

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

every instruction must be finished with a $\boxed{;}$ (semicolon) !

Compiling a C++ program:

source file

`.cpp, .C`

$\Downarrow$

compiler + linker

`.o, .so, .a`

$\Downarrow$

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

Task 2.1 Compiling

Use a text editor to create a file `nothing.cpp`, which contains *only* the empty function `int main(){}`, compile it and execute the resulting program.

Simple program for output on screen I

Example: C++ output via streams

```
#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

Simple program for output on screen III

Task 2.2 Hello world!

Use a text editor to create a file `hello.cpp`, which prints out "Hello World!" in the terminal, compile it and execute the resulting program.

Functions I

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 ; }`

Functions II

Example

```
#include <iostream>
using namespace std ;

float cube(float x) ;

int main() {
  float x = 4. ;
  cout  << "The cube of x is: " << cube(x) << endl ;
  return 0 ;
}

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

Functions III

Task 2.3 Calling a function

Use a text editor to create a file `cubemain.cpp`, which contains the source code from the previous slide (copy & paste).

1. Compile it and execute the resulting program.
2. Modify the source code so that the program reads in a number from the user with the help of `cin`:

```
float x  ;
cout << "type in a number: " ;
cin >> x ;
```

Functions IV

`inline` functions

- usually for compiled program: functions as code sections with own address; calling a function = jump to this address, pass arguments → overhead for argument passing, address for jumping back from function (`return`) must be stored:

Example

```
nm cubemain | grep " T "
00000000004008b7 T main
00000000004007de T _start
000000000040090d T _Z4cubef
```

→ calling many *small* functions is expensive

Functions V

- solution: use keyword `inline` → compiler replaces function *call* by function *code*, each time the function is called → increases size of compiled code

Example

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

→ definition must be in the same source text file where function is called
→ not all functions can be inlined by the compiler

- methods *defined* in class headers are automatically `inline`

The cmath-library I

In C/C++ only basic mathematical operations `+`,`-`,`*`,`/`,`%` available.

By including the `cmath`-library in the beginning:

```
#include <cmath>
```

many mathematical functions become available:

`cos();`	`sin();`	`tan();`
`asin();`	`atan();`	`acos();`
`cosh();`	`sinh();`	`tanh();`
`exp();`	`fabs();`	`abs();`

`log();` ... natural logarithm (base e)

`log10();` ... decadic logarithm (base 10)

`pow(x,y);` ... x^y †

`sqrt();` $\sqrt{}$

The `cmath`-library II

Moreover, there are also predefined mathematical constants:

`M_E`	...	e
`M_PI`	...	π
`M_PI_2`	...	$\pi/2$
`M_PI_4`	...	$\pi/4$
`M_2_PI`	...	$2/\pi$
`M_SQRT2`	...	$+\sqrt{2}$

Variables

- 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 data types I

Integer numbers are represented *exactly* in the memory with help of the binary number system (base 2), e.g.

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \mathrel{\hat{=}} \boxed{1}\boxed{1}\boxed{0}\boxed{1}^{\,1} \quad \text{(binary)}$$

In the assignment

```
a = 3
```

3 is an integer literal (literal constant). Its bit pattern ($3 = 1 \cdot 2^0 + 1 \cdot 2^1 \mathrel{\hat{=}} \boxed{1}\boxed{1}$) is inserted at the corresponding positions by the *compiler*.

[1] doesn't correspond necessarily to the sequential order used by the computer → "Little Endian": store least significant bit first, so actually: 1011

Integer data types II

on 64-bit systems

`int`	compiler reserves 32 bit (= 4 byte) of memory "1 bit for sign" (see below) and $2^{31} = 2\,147\,483\,648$ values (incl. 0): $\rightarrow$ range: $\texttt{int} = -2\,147\,483\,648 \ldots +2\,147\,483\,647$
`unsigned int`	32 bit, no bit for sign $\rightarrow 2^{32}$ values (incl. 0) $\texttt{unsigned int} = 0 \ldots 4\,294\,967\,295$
`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 (1/2 byte)

binary	unsigned	signed	2'S
0000	0	0	0
0001	1	1	1
...			
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
...			
1111	15	-7	-1

Disadvantages of representation as value and sign:
$\exists$ 0 *and* -0; Which bit is sign? ($\rightarrow$ const number of digits, fill up with 0s);

Advantage of 2'S:
negative numbers† always with highest bit=1
$\rightarrow$ cf. $+1 + -1$ bitwise for value & sign vs. 2'S

Binary arithmetic: $1 + 1 = 2$

$$\begin{array}{cc} & 0001 \\ + & 0001 \\ \hline = & 0010 \end{array}$$

†How to write negative numbers in 2'S? $\rightarrow$ start with corresponding positive number, invert all bits, and add 1 ignoring any overflow

Floating point data types I

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

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

Valid assignments are, e.g.,:

```
x = 3.0 ;
y = 1.1E-3 ;
z = x / y ;
```

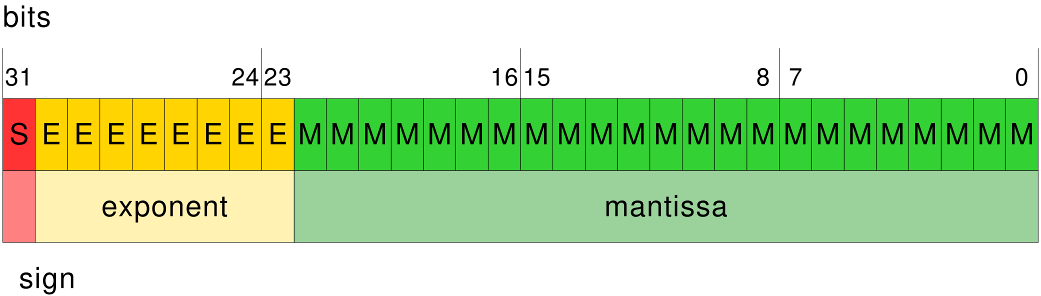
Floating point data types II

- representation (normalization) of floating point numbers are described by standard IEEE 754 :

$$x = s \cdot m \cdot b^{e} \tag{1}$$

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

- So for 32 Bit (Little Endian†), 8 bit exponent, 23 bit mantissa:



(† least significant bit at start address, read each part: $\rightarrow$)

- 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
10101100.101×2^0	base 2 $(0.625 = 1 \cdot 1/2 + 0 \cdot 1/4 + 1 \cdot 1/8)$
1.0101100101×2^7	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 01011001010000000000000

Floating point data types IV

- single precision (32 bit) `float`: exponent 8 bit, significand 23 bit

$$-126 \leq e \leq 127 \text{ (basis 2)}$$

$$\rightarrow \approx 10^{-45} \ldots 10^{38}$$

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

- for 64 bit (double precision) – `double`: exponent 11 bit, significand 52 bit

$$-1022 \leq e \leq 1023 \text{ (basis 2)}$$

$$\rightarrow \approx 10^{-324} \ldots 10^{308}$$

$$\text{digits: 15-16 } (= \log 2^{52+1})$$

Floating point data types V

some real numbers cannot be presented exactly in the binary numeral system, e.g.:

$$0.1 \approx 1.10011001100110011001101 \times 2^{-4} \tag{2}$$

$\rightarrow$ cf. 1/3 in decimal: all fractions with denominator not product of prime factors (2,5) of the base 10, e.g., 1/3, 1/6, ...
In binary numeral system only one prime factor: 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:
`abs( x - y ) <= eps`, better: relative error
`abs(1-y/x) <= eps`

Floating point arithmetic

Subtraction of floating point numbers

consider $1.000 \times 2^5 - 1.001 \times 2^1$ (only 3 bit mantissa)
$\rightarrow$ *bitwise subtraction*, requires same exponent

$$
\begin{array}{rll}
 & 1.000\,0000 & \times 2^5 \\
- & 0.000\,1001 & \times 2^5 \\
\hline
 & 0.111\,0111 & \times 2^5 \text{ infinite precision} \\
 & 1.11{\color{blue}0}\,111 & \times 2^4 \text{ shifted left to normalize} \\
 & 1.111 & \times 2^4 \text{ rounded up, as last digits} > 1/2 \text{ {\color{blue}ULP}}^\dagger
\end{array}
$$

†unit in the last place = spacing between subsequent floating point numbers

Floating point data types VII

Properties of floating point arithmetic (limited precision):

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

Example for loss of significance

$\pi - 3.141 = 3.14159265\ldots - 3.141$ with 4-digit mantissa;
maybe expected: $= 0.00059265\ldots \approx 5.927 \times 10^{-4}$;
in fact: 1.0000×10^{-3}, because π 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: $100 + 0.001 = 100$:
$1.000 \times 10^2 + 1.000 \times 10^{-3} = 1.000 \times 10^2 + 0.000\,01 \times 10^2 = 1.000 \times 10^2 + 0.000 \times 10^2 = 1.000 \times 10^2$, same for subtraction

Floating point data types VIII

- distributive and associative law usually not fulfilled, i.e. in general

$$(x+y)+z \neq x+(y+z) \tag{3}$$
$$(x\cdot y)\cdot z \neq x\cdot(y\cdot z) \tag{4}$$
$$x\cdot(y+z) \neq (x\cdot y)+(x\cdot z) \tag{5}$$
$$(x+y)\cdot z \neq (x\cdot z)+(y\cdot z) \tag{6}$$

- solution of equations, e.g., $(1+x)=1$ for 4-bit mantissa solved by any $x<10^{-4}$ (see absorption) $\rightarrow$ *smallest* float number ϵ 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

Floating point data types IX

Guard bit, round bit, sticky bit (GRS)

- in floating point arithmetic: if mantissa shifted right $\rightarrow$ 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(!)
  $\rightarrow$ sticky
- example

```
                                                   G R S
Before 1st shift: 1.11000000000000000000100 0 0 0
After 1 shift:    0.11100000000000000000010 0 0 0
After 2 shifts:   0.01110000000000000000001 0 0 0
After 3 shifts:   0.00111000000000000000000 1 0 0
After 4 shifts:   0.00011100000000000000000 0 1 0
After 5 shifts:   0.00001110000000000000000 0 0 1
After 6 shifts:   0.00000111000000000000000 0 0 1
After 7 shifts:   0.00000011100000000000000 0 0 1
After 8 shifts:   0.00000001110000000000000 0 0 1
```

GRS bits – possible values and stored values

extended sum	stored value	why
1.0100 000	1.0100	truncated because of GR bits
1.0100 001	1.0100	truncated because of GR bits
1.0100 010	1.0100	rounded down because of GR bits
1.0100 011	1.0100	rounded down because of GR bits
1.0100 100	1.0100	rounded down because of S bit
1.0100 101	1.0101	rounded up because of S bit
1.0100 110	1.0101	rounded up because of GR bits
1.0100 111	1.0101	rounded up because of GR bits

Floating point data types XI

IEEE representation of 32 bit floats:

Number name	sign, exp., f	value
normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
subnormal	$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
signed zero (± 0)	$e = 0, f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, e = 255, f = 0$	`+INF`
$-\infty$	$s = 1, e = 255, f = 0$	`-INF`
Not a number	$e = 255, f \neq 0$	`NaN`

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

If not default by compiler: enable floating-point exception handling (e.g., `-fpe-all0` for `ifort`)

Automatic type conversion

In C/C++ many data type conversions are already predefined, which will be invoked automatically:

```
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
}
```

Explicit type conversions (casts) I

Moreover, a type conversion/casting can be done explicitly:

C cast

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

- obviously possible: integer $\leftrightarrow$ floating point
- but also : pointer (see below) $\leftrightarrow$ data types
- Caution: For such C casts there is no type checking during runtime!

Explicit type conversions (casts) II

The better way: use the functions of the same name for type conversion

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

Task 2.4 Integer conversion

What is the result for `i` and `y` in this example above?

Logical variables

```
bool b ;
```

intrinsic data type, has effectively only two different values:

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

but also:

```
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 `f` and `t` for output.

Note: minimum addressable piece of memory is 1 byte → `bool` needs more memory than necessary

Character variables I

```
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';
```

Character variables II

Character input

```
char character ;
int  number ;
cout << "Character input: " ;
cin >> character ;
cout << "character is: " << character
     << " corresponds to " << int(character) << endl;
cout << "Number input: " ;
cin >> number ;
cout << "Number " << number
     << " corresponds to " << char(number) << endl;
```

Task 2.5 Characters

Complete this code example to a C++ program, compile and execute it. Which (decimal) ASCII code have }, Y and 1 ? Which character has the code 97 ?

Execution control - `for`-loops I

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

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

// also possible: non-integer loop variable -> not recommended
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.:
   `int k = 0 ;`†
   → is executed *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 *before each* iteration
3. expression: incrementing/decrementing of the loop variable, e.g.,
   `++k` or `--k` or `k += 3`
   is executed *after each* iteration

†interestingly also: `int k = 0, j = 1;`, i.e. multiple loop variables of same type

Increment operators

`sum += a`

$\longrightarrow$ `sum = sum + a`

`++x`

$\longrightarrow$ `x = x + 1` (increment operator)

`--x`

$\longrightarrow$ `x = x - 1` (decrement operator)

Note that there is also a *post* increment/decrement operator: `x++`, `x--`, i.e. incrementing/decrementing is done *after* any assignment, e.g., `y = x++`.

Logical operators I - Comparisons/inequalities

→ return either(!) `true` or `false`:

`a > b`	greater than
`a >= b`	greater than or equal
`a == b`	equal
`a != b`	not equal
`a <= b`	less than or equal
`a < b`	less than

Caution!

The exact equality == should not be used for float-type variables because of the limited precision in the representation.

$$\texttt{!(a < b)} \qquad \text{not} \qquad (2)$$

$$\texttt{(a < b) \&\& (c != a)} \qquad \text{and} \qquad (14)$$

$$\texttt{(a < b) || (c != a)} \qquad \text{or} \qquad (15)$$

It is recommend to use parentheses ( ) for combination of operations for unambiguousness.

Otherwise: Operator Precedence (incomplete list)

Precedence	Operator
5	$* \; / \; \%$
6	$+ \; -$
9	$< \; <= \; > \; >=$
10	$== \; !=$
14	&&
15	||

Loops I

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"

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

Recommendation: *local* loop variables

$\rightarrow$ declaration in loop header
$\rightarrow$ scope limited to loop body

Loops III

Our example with the `float` loop variable

```
for (float x = 0.7 ; x < 17.2 ; x = x + 0.3) { // = 55 iterations
    y = a * x + b ;
    cout << x << " " << y << endl;
}
```

can be rewritten with integer loop variables (number of iterations clear)

```
float x = 0.7 , x_inc = 0.3, x_max = 17.2 ;
int it_max = ((x_max - x) / x_inc) + 0.5 ;  // +0.5 for correct rounding
for (int i = 0 ; i < it_max ; ++i) { // it_max = 54
    y = a * x + b ;
    cout << x << " " << y << endl;
    x+= x_inc ;
}
```

→ note that when converting `float` → `int`, digits after decimal point just cut off → add `+0.5` before conversion for correct rounding

Conditional execution via `if`:

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

Conditional/branching

```
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`:

Branching II

```
switch (epxression) {
        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

```
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 ;
}
```

Arrays in C/C++

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

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

Access to individual elements:

```
total = a[0] + a[1] + a[2] + a[3] + a[4] ;
```

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).

Two-dimensional arrays I

an $m \times n$ matrix (rows $\times$ columns) :

$$\begin{array}{cc} & n \text{ columns} \rightarrow \\ \begin{array}{c} m \\ \text{rows} \\ \downarrow \\ \end{array} & \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \dots & & \\ \dots & & & \\ a_{m1} & & & a_{mn} \end{pmatrix} \end{array}$$

`int a[m][n]` ... 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

$$\text{e.g.,} \quad a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

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 :

`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[1][0]`	`a[1][1]`	`a[1][2]`
1	2	3	4	5	6

(row-major order $\rightarrow$ stored row by row)

Task 2.6 Internal order of arrays

The cache, which is the memory closest to the CPU and usually on the same chip, is limited ($\sim$ MB). Therefore it is important to design programs in a way that for a specific task data that must be read into the cache are in a subsequent order.

Let's assume for a cosmological simulation with 10^6 particles, for each particle the coordinates and velocities (3D) should be saved in an array `particle[][]`. A function loops over all particles and needs to access for each particle all $\vec{x}, \vec{v}$-data.

How should this array be dimensioned in C/C++: `particle[6][1000000]` or `particle[1000000][6]` ?

Initialization of arrays

An array can be initialized by curly braces:

```
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
```

Strings I

There are no string variables in C. Therefore strings are written to one-dimensional character arrays:

```
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

```
char text[80] ;
cout << endl << "Please enter a string:" ;
cin  >> text ;
cout << "You have entered " << text << " ." << endl ;
```

Strings II

Task 2.7

1. What is the difference between `'Y'` and `"Y"`?
2. Which of these two literals is correct: `'Yes'` oder `"Yes"`?
3. What's wrong here: `char text[2] = "No" ;`?

String comparison

C-Strings (character arrays) cannot be compared directly with ==, in this case the operator would compare the start addresses of the arrays.
Instead: Use `strcomp(string1,string2)` from library `string.h`, this will return 0 if strings are equal (arrays can have different lengths).

Declarations of variables should be at the beginning of a **block**, exception: loop variables

```
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 → **scope**
- are **local** regarding this block, can only be accessed within this block
- are unknown outside of this block, i.e., they don't exist there
- are automatically deallocated when leaving the scope,
  except those with modifier `static`

Declaration and visibility of variables II

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 ;`

Declaration and visibility of variables III

Global and local variables

```
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 outside loop
  return j ;
}
```

Defining constants

Values (e.g., numbers) that do not change during the program execution, should be *defined* as constants:

```
const float e = 2.71828 ;
const int prime[] = {2,3,5,7} ;
```

Constants must be initialized during declaration.

After initialization their value cannot be changed.

Use `const` whenever possible!
(The compiler will replace any occurrence of the constant name by the value before "translation" $\rightarrow$ no memory addressing necessary as for variables.)

Pointer I

Pointer variables – or pointer for short – allow a direct access (i.e. not via the name) to a variable.

Declaration of pointers

```
int     *pa   ; // pointer to int
float   *px   ; // pointer to float

int    **ppb  ; // pointer to pointer to int
int ***pppb  ; // pointer to pointer to pointer to int
  ...
```

C++ standard : at least 255 (static) ; in C: at least 12 (static)
but: infinite dynamic (linked lists)

Pointer II

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.

address	value	variable
1000	0.5	x
1004	42	n
1008	3.141...	d
1012	...5926	
1016	H E Y !	salutation
1020	1000	px
1024	1008	pd
1028	1004	pn
1032	1016	psalutation
1036	1028	pp

Pointer III

Pointers must be always initialized before usage!

Initialization of pointers

```
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

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

→ content of `pa` = address of `b`

Pointer IV

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

```
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.

```
cout << b << endl ; // yields 5
cout << pa << endl ; // e.g., 0x7fff5fbff75c
// and with pointer to int-pointer:
int **ppa ; ppa = &pa ; cout << **ppa << endl ; // yields also 5
```

Pointer V

Once again:

Pointer declaration:

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

Pointer initialization:

```
pz = &a ;
```

Result – output:

```
cout << "address of variable a (content of pz): "
     << pz << endl ;
cout << "content of variable a: "
     << *pz << endl ;
*pz = 5.2 ; // change value of a
```

References

```
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:

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

- This initialization cannot be changed any more within the program!

(At this stage a reference seems to be rather useless.)

Passing variables to functions I

Structure of functions – definition

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

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

- in parentheses (): 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*

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

Passing variables to functions II

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*
  $\rightarrow$ extra overhead (time)
  (exception: if parameter is an array, only *start address* is passed $\rightarrow$ pointer)

Passing variables to functions II

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.

```
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:
`sum (int const &a, int const &b) ;`

Passing variables to functions

Call by pointer

A function for swapping two `int` variables can also be written by using pointers:

```
void swap(int *a, int *b) { // pointers as formal parameters
    int t = *a ; *a = *b ; *b = t ; // remember: *a -> value at address of a
}
```

Call in `main()`:

```
  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 (see below), e.g.,
`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
- (actually only useful as function arguments or result)

Passing arrays to functions in C++ I

Declaration of a 1d-array:

```
int m[6] ; // statically dimensioned†
```

Declaration of a function with an array type argument:

```
int sumsort (int m[], int n) ; // n = length of m
```

Calling a function with an array type argument:

```
sum = sumsort (m, 6) ;
```

→ passing the array is implicitly done by a pointer, i.e. only the *start address* of the array will be passed to the function

† an array can also be declared dynamically, so with size fixed at runtime, but only *locally* and arrays with more than 1 dimension must have fixed sizes at compile time if they are passed to functions (see below)

Passing arrays to functions in C++ II

Correspondence of pointers and arrays

→ see exercise

- the assignment

```
a[i] = 1 ;
```

is equivalent to

```
*(a + i) = 1 ;
```

- when passing 1d-arrays to functions the start address and the data type (size of the entries) is sufficient

Passing arrays to functions in C++ III

Problem:

When using multi-dimensional arrays, passing of the start address alone is not sufficient. Every dimensioning after the first one must be explicitly (integer constant!) written.

Therefore:

```
float absv  (float vector[], int n) ;  // 1d-array
float trace (float matrix[][10]) ;     // 2d-array
float maxel (float tensor[][13][13]) ; // 3d-array
```

→ more flexibility by using pointers as arguments, e.g., for an array `a[3][4]`:

`float *a[3] ; ... ; a[i] = new float[4] ; float function (float **a, ...)`

→ special matrix-*classes* simplify the passing to functions

→ in Fortran, passing arrays to functions is much easier (i.e. only start address is passed)

Structs and classes – defining new data types I

Besides the intrinsic (/basic) data types there are many other data types, which can be defined by the programmer

struct

```
struct complex {
    float re ;
    float im ;
} ; [a]
```

[a] Note the necessary semicolon after the } for structs

In this example the data type `complex` is defined, it contains the *member variables* for real and imaginary part.

struct vs. class

The constructs `struct` and `class` are identical in C++ with the exception that access to `struct` is public by default and for `class` it is private. They can be defined outside or inside a function (e.g., `main`).

Structs can be imagined as collections of variables.

struct

```
struct star {
     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

```
complex z, c ;
star sun ;
```

Structs and classes – defining new data types III

Concrete structs which are declared in this way are called *instances* or *objects* (→ object-oriented programming) of a class (struct).

Declaration and initialization

```
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

```
real_part = z.re ;
sun.luminosity_lsun = 1.0 ;
```

Structs and classes – defining new data types IV

It is also possible to define functions (so-called *methods*) within structs:

Member functions

```
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.

And even operators:

Operator overloading

```
complex operator+ (const complex & c) {
  complex z ;
  // calling object is referenced with  this->
  z.re = this->re + c.re ;
  z.im = this->im + c.im ;
  return z ;
}
  ...
complex w, z, c ;
  ...
w = z + c ;
// object on left side (z) of operator calls +
// object on the right side (c) is "argument" for call
```

Structs and classes – defining new data types VI

In our example for the absolute of a complex number, the call is `c.absolute()` instead of the common `absolute(c)`

The latter call can be achieved with help of a static member function, that is shared by all objects and exists independently of them

Static member functions

```
static double abs (const complex & c)
  return ( sqrt(c.re * c.re + c.im * c.im) ) ;


...
complex::abs(c) ;
```

Static functions must be called with the class name (here: complex) and the scope operator ::

Static functions have no `this->` pointer

Classes – Example: writing/reading files I

Output to a file by using library `fstream`:

1. `#include <fstream>`
2. create an object of the class `ofstream`:

   `ofstream fileout ;`
3. method open of the class `ofstream`:

   `fileout.open("graphic.ps") ;`
4. writing data: e.g.

   `fileout << x ;`
5. close file via method `close`:

   `fileout.close() ;`

Simple alternative (Unix): Use `cout` and redirection operator > or >> of the shell:

`./program > output.txt`

Classes – Example: writing/reading files II

By including the `<fstream>` library, one can also read from a file

Input from a file

```
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;
                               // use buffer size (132)
    x[i] = atof(line) ;        // read into float array
}
```

The method `good()` checks, whether the end of file (EOF) is reached or an error occurred.

Private and public

- `class` : by default all members are private → accessible elements must be declared as public

```
class complex {
  float real, imag ; // implicitly private
  public : getreal () { return this->real ; }
};
```

- member variables usually set private, access to them via public methods (e.g., get. . . , set. . .)
- keywords `public` and `private` (with `:`) valid until next of those occurs

Constructors

- each `class` has a default constructor with empty argument list if no constructor is explicitly defined:

```
struct complex {
  ...
};
...
complex z ; // default constructor
z = {x , 1.} ; // initialization (only if constructor is public)
```

- one may define more constructors, e.g.:

```
struct complex {
 public : complex (double x, double y) {real = x ; imag = y ;}
  ...
};
complex z (x, y) ; // constructor initializes real and imaginary part
```

Templates I

Templates allow to create universal definitions of certain structures. The final realization for a specific data type is done by the compiler.

Function templates

```
template <class T> // instead of keyword 'class' also 'typename' allowed
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.,

```
double w = 3.34 ; int k = 2 ;
cout << sqr(w) << " " << sqr(k) ;
```

→ for full convenience, templates must be already defined before the call, e.g., already in the header file (i.e. the compiler needs to know which concrete versions must be created)

Templates II

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:

Class templates

```
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.

Templates III

We can also have function templates of different types

Function template for multiple types

```
template <class T, class U>
 auto max (const T &x, const U &y) {
 return (x > y) ? x : y ; // return maximum of both arguments
}
  ...
cout max(2, 1) << " " << max(3.3, 4.4) << " " << max(1, 2.) << endl ;
  ...
```

→ `max( , )` can now be called with mixed arguments, e.g., `int` and `double`: `max(1, 2.)`
→ keyword `auto` instructs compiler to select return type automatically, e.g., `double` if arguments are `double` and `int`

In C++20 the function header above can be shorter written as

```
auto max (const auto &x, const auto &y)
```

? is the ternary conditional operator, meaning *condition* ? result_if_true : result_if_false

Typ definitions via typedef

By using `typedef` *datatype alias name* one can declare new names for data types:

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

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

```
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.

Exception handling – exceptions I

A major strength of C++ is the ability to handle runtime errors, so called exceptions:

Throwing exceptions: `try` – `throw` – `catch`

```
try {
    cin >> x ;
    if ( x < 0.) throw string("Negative value!") ;
    y  = g(x) ;
}
catch (string info) { // catch exception from try block
    cout << "Program stops, because of: " << info << endl ;
    exit (1) ;
}
...
double g (double x) {
  if (x > 1000.) throw string("x too large!") ; ... }
```

Exception handling – exceptions II

`try { ...}`

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

`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

Exception handling – exceptions III

```
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`

Exception handling – exceptions IV

Data types for exception throwing

In contrast to the simple example above, it is recommended to use specific (not built-in) data types *e* for `throw`, e.g., from class `exception`.

```
#include <exception>
 ...
 try {
    cin >> x ;
    if ( x < 0.) throw runtime_error("Negative Number!");
    y = g(x) ;
 }
 catch (const runtime_error& ex) { // catch exception from try block
    cout << "Program stops, because of: " << ex.what()  << endl ;
    exit (1) ;
  }
```

Reading arguments from program call

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

```
#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`

Summary

Common mistakes in C/C++:

- forgotten semicolon `;`
- wrong dimensioning/access of arrays
  `int m[4] ; imax = m[4] ;` → `imax = m[3] ;`
- wrong data type in instructions / function calls
  `float x ; ... switch (x)` → `int i ; ... switch (i)`
  `void swap (int *i, int *j) ; ... int m, n ; ... swap(n, m) ;`
  → `swap(&n, &m) ;`
- confusing assignment operator `=` with the equality operator `==`
  `if(i = j)` → `if(i == j)`
- forgotten function parenthesis for functions without parameters
  `clear ;` → `clear();`
- ambiguous expressions
  `if (i == 0 && ++j == 1)`
  no increment of `j`, if `i` $\neq 0$

Some recommendations I

- use always(!) the `.` for floating point literals: `x = 1./3.` instead of `x = 1/3`
- white space 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., `Teff` 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

Some recommendations II

Tips for High Performance Computing / Number Crunching

- The more flexible your program is, the harder it is for the compiler to optimize it.
  Hence:
- Use `const` whenever possible (values, arguments).
- Avoid pointers (except for argument passing).
- (Avoid dynamic allocations.)
- Use keyword `inline` (see Sect. 1) for *small* functions (vs. code size see below).
  Avoid many (nested) function calls.
- Keep loops simple, avoid too many branchings and jumps. Use matrix classes/functions instead of looping over elements.

Execution speed vs. flexibility:

$\rightarrow$ flexibility increases $\rightarrow$

Assembler   Fortran   C   C++   Python

$\leftarrow$ speed increases $\leftarrow$

Some recommendations III

Table: Latencies of memory operations in relation to each other, see github

operation	real time	scaled time ($\times 10^9$)
Level 1 cache access	0.5 ns	0.5 s ($\sim$ heart beat)
Level 2 cache access	7 ns	7 s
Multiply two floats	10 ns	10 s (estimated)
Devide two floats	40 ns	40 s (estimated)
RAM access	100 ns	1.5 min
Send 2kB over Gigabit network	20 000 ns	5.5 h
Read 1MB from RAM	250 000 ns	2.9 d
Read 1MB from SSD	1 000 000 ns	11.6 d
Read 1MB from HDD	20 000 000 ns	7.8 months
Send packet DE$\rightarrow$ US$\rightarrow$ DE	150 000 000 ns	4.8 years