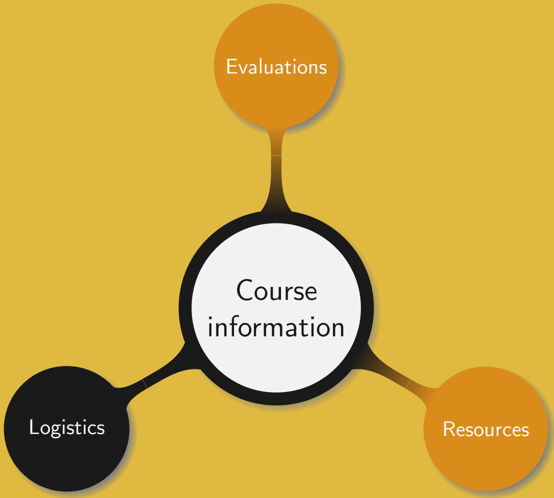
Accelerated Introduction to Computer and Programming

Manuel – Fall 2021

0. Course information


Evaluations
Course information
Logistics
Resources

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Jiache (zjc_he@sjtu.edu.cn)
  - Zhen (xuzhen1023@sjtu.edu.cn)
  - Qinzhe (ivyoung7@sjtu.edu.cn)
  - Kaiwen (kevin.zhang@sjtu.edu.cn)

Important rules:

- When contacting a TA for an important matter, `CC` the instructor
- Prepend [ENGR151] to the subject, e.g. `Subject: [ENGR151] Grades`
- Use SJTU jBox service to share large files ($>$ 2 MB)

Never send large files by email

Course arrangements:

- Lectures:
  - Tuesday 12:10 – 13:50
  - Thursday 12:10 – 13:50
- Labs:
  - Tuesday 18:20 – 20:55
  - Wednesday 18:20 – 20:55
  - Thursday 18:20 – 20:55
- Manuel's office hours: Appointment (Zoom)
- TAs' office hours: TBA

Course objectives

Primary goals:

- Understand the main concepts of computer and programming
- Design simple algorithms
- Implement clearly stated algorithms in MATLAB, C, and C++

Be able to quickly adjust to new languages and libraries

Course workflow

Learning strategy:

- Course side:
  1. Understand the basics on computers
  2. Get familiar with programming through MATLAB
  3. Understand deeper concepts with C
  4. Bridge the gap between computers and humans using C++
- Personal side:
  1. Read and write code
  2. Write more code
  3. Write even more code
  4. Do not stop writing code
  5. Relate known strategies to new problems
  6. Perform extra research

Detailed goals:

- Know how to define and work with variables of different data types
- Be familiar with stream input-output, including files and standard input-output
- Be familiar with input and output in functions
- Be proficient with arithmetic and logical operators, as well as common mathematical functions
- Be proficient at designing, implementing, and testing functions
- Be proficient with conditional statements and loops
- Be familiar with primitive data types and composite data types such a structures and classes
- Be able to work with pointers and arrays
- Be able to design simple algorithms, including recursive ones
- Know how to organise a short project using classes, inheritance, and polymorphism
- Learn basics of software management systems

ENGR151 features more advanced tasks than VG101

More advanced tasks means:

- More practice
- Slightly more content
- Some content from the slides is moved to labs
- Probably **higher** workload

ENGR151 is most suitable for students who:

- Already know programming
- Like to learn more
- Want to major in ECE **with an emphasis on programming**

Labs and recitation classes

Labs are split into three parts:

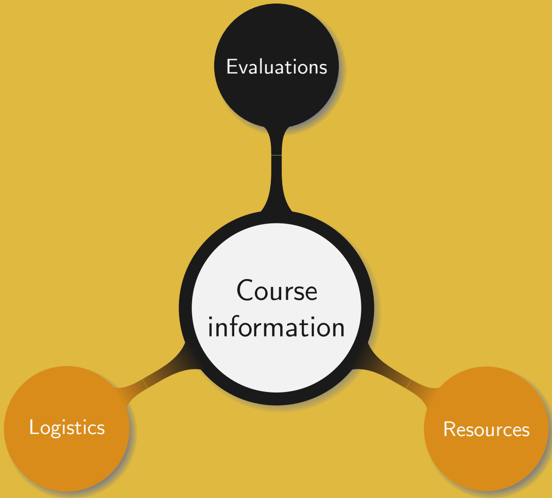
1. Analysis of the common issues in the previous homework
2. Discussion of topics related to the slides with the ᚴ mark
3. Work along the mandatory part of the lab manual:
   - Group discussions
   - Presentations
   - Class discussions

Recitation classes:

1. Review of the main points of the lectures
2. Discussion based on the worksheets

Labs are mandatory, recitation classes are optional

Chapter organisation



Assignments

Homework:

- Total: 8
- Content: basic algorithms, Matlab, C, and C++

Labs:

- Total: $9 + 3$
- Content: guided sessions in Matlab, C, and C++

Projects:

- Total: 3
- Content: advanced problems in Matlab, C, and C++

Grading policy

Grade weighting:

- Matlab midterm: 20%
- C midterm: 20%
- C++ final: 20%
- Projects: 30%
- Homework: 5%
- Labs: 5%

Assignment submissions:

- Projects: −10% per day, not accepted after three days
- Homework: one day with no penalty, rejected afterwards

Grades will be curved with the median in the range $[\![B, B+]\!]$

Homework:

- Mostly the workflow is graded, completed in groups
- Each student must complete all the mandatory exercises
- Each student must review the code of at least one teammate
- A final improved version must be submitted for each group
- Submissions should be successfully compiled or interpreted
- Group discussions must be handled using issues on the git server

Students not following guidelines will receive deductions on their homework grade

General rules:

- Not allowed:
  - Reuse the code or work from other students or groups
  - Reuse the code or work from the internet
  - Share too many details on how to complete a task
- Allowed:
  - Reuse part the course or textbooks and quoting the source
  - Share ideas and understandings on the course
  - Provide hints on where or how to find information

Honor Code

Documents allowed during the exams:

- Part A: a mono or bilingual dictionary
- Part B:
  - The lecture slides with **notes on them** (paper or electronic)
  - A mono or bilingual dictionary

Group works:

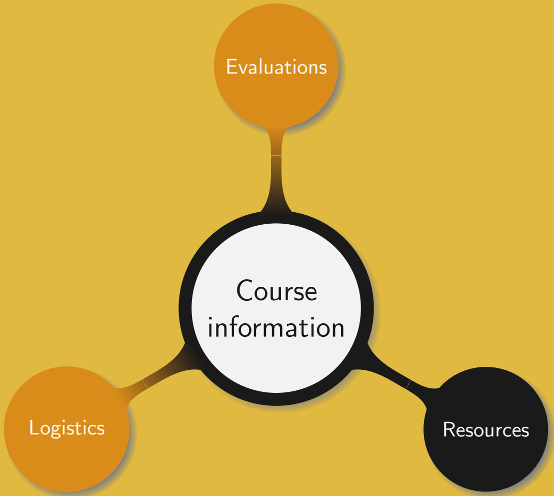
- Every student in a group is responsible for his group's submission
- If a student breaks the Honor Code, the whole group is guilty

Contact us as early as possible when:

- Facing special circumstances, e.g. full time work, illness, etc.
- Feeling late in the course
- Feeling to work hard without any result

Any late request will be rejected

Chapter organisation



Information and documents available on Canvas:

- Course materials:
  - Syllabus
  - Lecture slides
  - Homework
  - Labs
  - Projects
- Course information:
  - Announcements
  - Notifications
  - Grades
  - Surveys

Gitea will be:

- Introduced in the first labs
- Used for all the course submissions

References

Useful places where to find information:

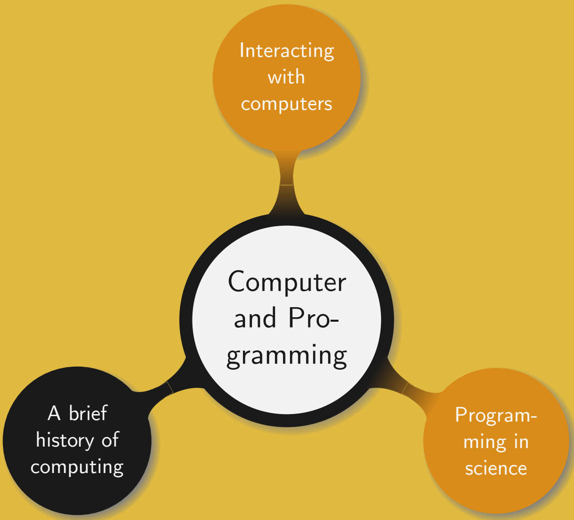
- MATLAB documentation
- *C for Engineers and Scientists*, by Harry H. Cheng
- *Thinking in C++*, by Bruce Eckel
- Piazza
- Search information online, i.e. $\{\mathit{internet} \setminus \{\mathit{non\text{-}English\ websites}\}\}$

Key points

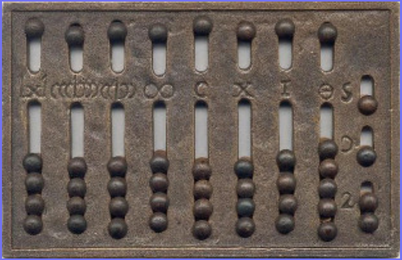
- Work regularly, do not wait the last minute/day
- Respect the Honor Code
- Go beyond what is taught
- Do not learn, understand
- Keep in touch with us
- Advice and suggestions are always much appreciated

1. Computer and Programming

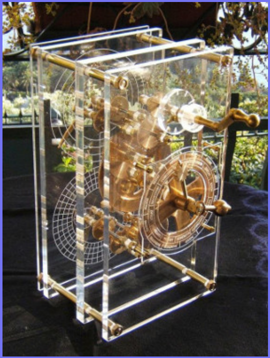
Chapter organisation



Ancient era

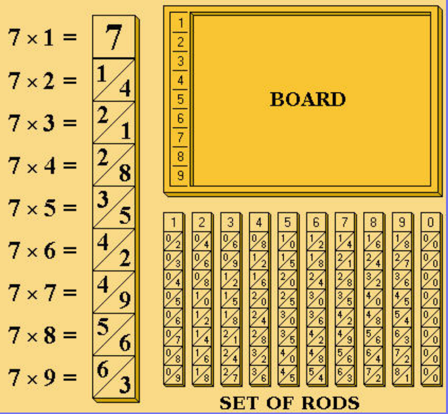


Abacus (-2700)

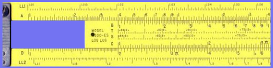


Antikythera mechanism (-100)

Calculation tools



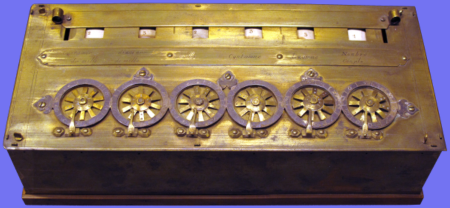
Napier's bones (1617)



Sliderule (1620)

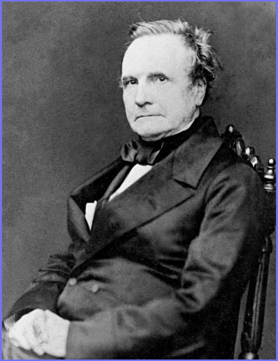
First pocket calculator introduced around 1970 in Japan

Mechanical calculators



Pascaline (1642)

Arithmomètre (1820)



Charles Babbage (1791–1871) achievements:

- Difference engine: built in the 1990es
- Analytical engine: never built

Ada Byron (1815–1852) achievements:

- Extensive notes on Babbage's engines
- Algorithm to calculate Bernoulli numbers

The birth of modern computing

First part of the 20th century:

- 1936: First freely programmable computer
- 1946: First electronic general-purpose computer
- 1948: Invention of the transistor
- 1951: First commercial computer
- 1958: Integrated circuit

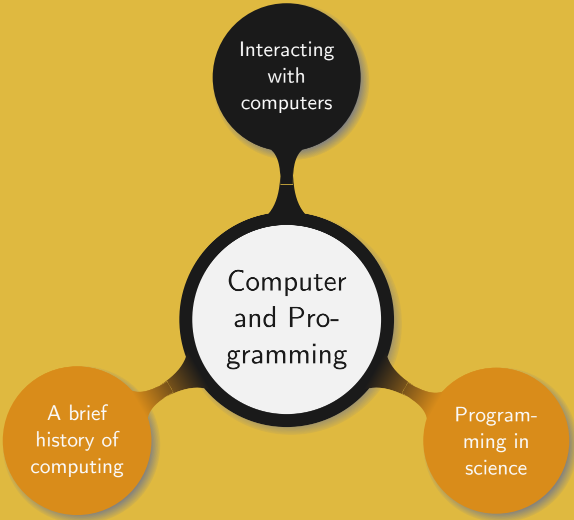
UNIVAC I (1951)

Apple I (1976)

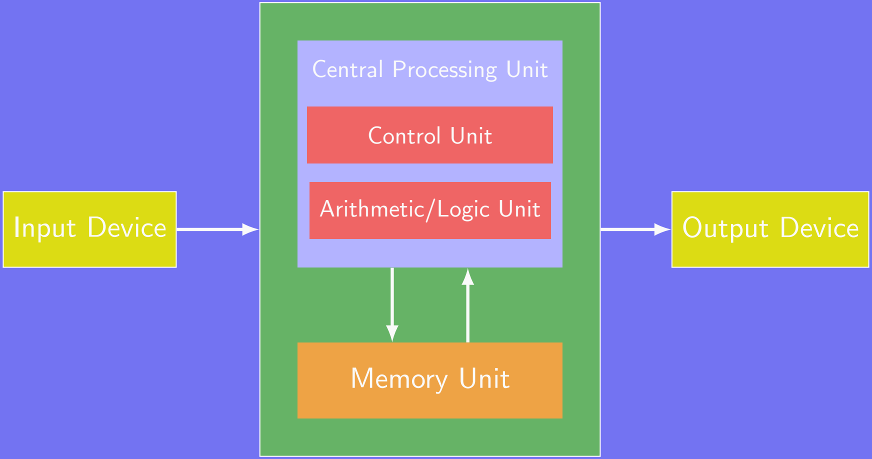
Second part of the 20th century:

- 1962: First computer game
- 1969: ARPAnet
- 1971: First microprocessor
- 1975: First consumer computers
- 1981: First PC, MS-DOS
- 1983: First home computer with a GUI
- 1985: Microsoft Windows
- 1991: Linux

Chapter organisation



Von Neumann architecture



Numbers in various bases:

- *Decimal:* $\{0, 1, \cdots, 9\}$, e.g. $(253)_{10}$
- *Binary:* $\{0, 1\}$, e.g $(11111101)_2$
- *Hexadecimal:* $\{0, 1, \cdots, 9, A, B, C, D, E, F\}$, e.g. $(\mathrm{FD})_{16}$

Base conversion:

- From base b into decimal: evaluate the polynomial
  $(11111101)_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 253$
  $(\mathrm{FD})_{16} = \mathrm{F} \cdot 16^1 + \mathrm{D} \cdot 16^0 = 15 \cdot 16^1 + 13 \cdot 16^0 = 253$
- From decimal into base b: repeatedly divide n by b until the quotient is 0. Consider the remainders from right to left
  rem(253,2)=1, rem(126,2)=0, rem(63,2)=1, rem(31,2)=1, rem(15,2)=1, rem(7,2)=1, rem(3,2)=1, rem(1,2)=1 rem(253,16)=13=D, rem(15,16)=15=F
- From base b into base b^a: group numbers into chunks of a elements
  $(11111101)_2 = 1111\ 1101 = (\mathrm{FD})_{16}$

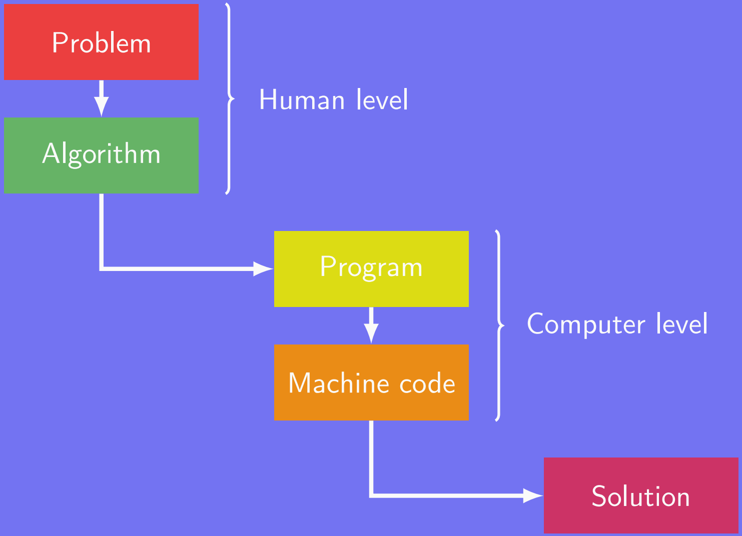
Exercise.

- Convert into hexadecimal: 1675, 321, $(100011)_2$, $(10111011)_2$
- Convert into binary: 654, 2049, ACE, 5F3EC6
- Convert into decimal: $(111110)_2$, $(10101)_2$, $(12345)_{16}$, 12C3C

Solution.

- $1675 = (68B)_{16}$, $321 = (141)_{16}$, $(100011)_2 = (23)_{16}$
- $654 = 1010001110$, $2049 = 100000000001$, $\mathrm{ACE} = 101011001110$, $\mathrm{5F3EC6} = 10111110011111011000110$
- $(111110)_2 = 62$, $(10101)_2 = 21$, $(12345)_{16} = 74565$, $\mathrm{12C3C} = 76860$

How to use a computer?



Algorithm: recipe explaining the computer how to solve a problem

Example. Detail an algorithm to prepare a jam sandwich.

Actions: `cut`, `listen`, `spread`, `sleep`, `take`, `eat`, `dip`, `assemble`
Things: `knife`, `guitar`, `bread`, `honey`, `jam jar`, `sword`, `slice`

Algorithm. (*Sandwich making*)

Input : 1 bread, 1 jam jar, 1 knife
Output: 1 jam sandwich

take the knife and cut 2 slices of bread;
dip the knife into the jam jar;
spread the jam on the bread, using the knife;
assemble the 2 slices together, jam on the inside;

From algorithm to machine code

$$
\begin{tikzcd}
\text{Algorithm} \arrow[r] & \text{Programming language} \arrow[d] \\
\text{Machine code} & \text{Interpreter or compiler} \arrow[l]
\end{tikzcd}
$$

Example. Given a square and the length of one side, what is its area?

Algorithm.

Input : side (the length of one side of a square)
Output: the area of the square

return side $\times$ side

To obtain the result in MATLAB:

1. Type the code
2. Press `Enter`

area.m

```
a=input("Side: ");
fprintf ("Area: %d",  a*a)
```

Running the program

area.c

```c
#include<stdio.h>
int main() {
  int side;
  printf("Side: ");
  scanf("%d",&side);
  printf("Area: %d", side*side);
  return 0;
}
```

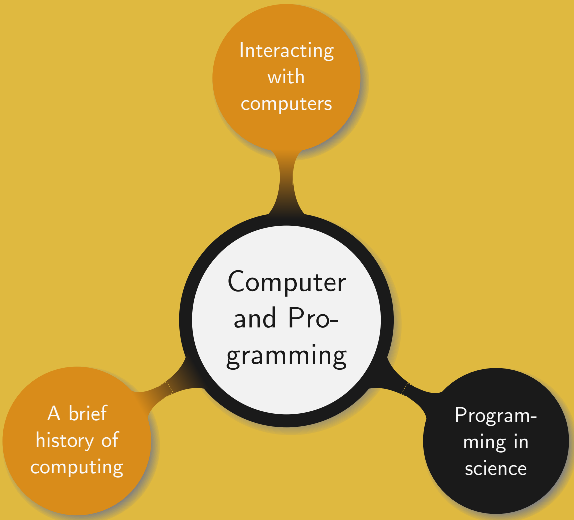
area.cpp

```cpp
#include<iostream>
using namespace std;
int main() {
int side;
  cout << "Side: ";  cin >> side;
  cout << "Area: " << side*side;
  return 0;
}
```

To obtain the result in C or C++

1. Write the source code
2. Compile the program
3. Run the program

Chapter organisation



Common mathematics software:

- Axiom
- GAP
- GP/PARI
- Magma
- Maple
- MATLAB
- Maxima
- Octave
- R
- Scilab
- Mathematica

MATrix LABoratory (MATLAB):

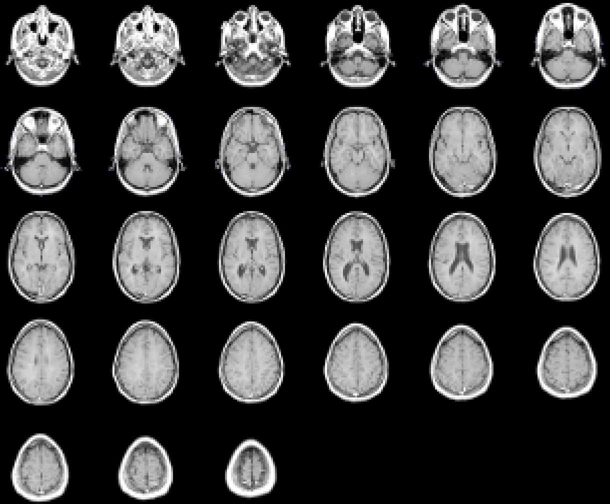
- Matrix manipulations[1]
- Implement algorithms[1]
- Plotting functions and data[1]
- User interface creation

Benefits of MATLAB:

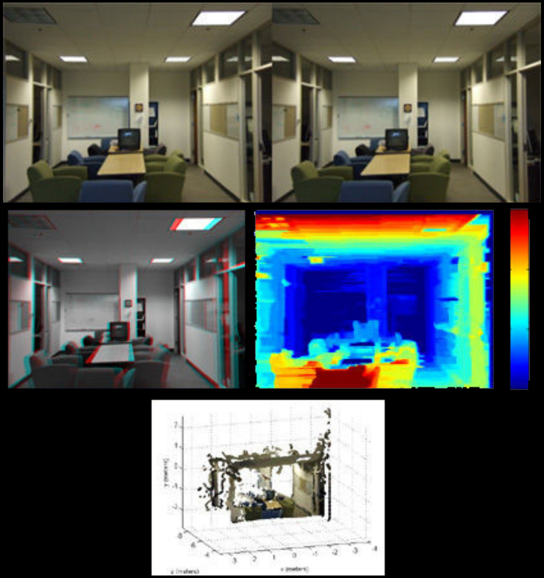
- Easy to use
- Built-in language
- Versatile
- Many toolboxes

[1] Studied in ENGR151

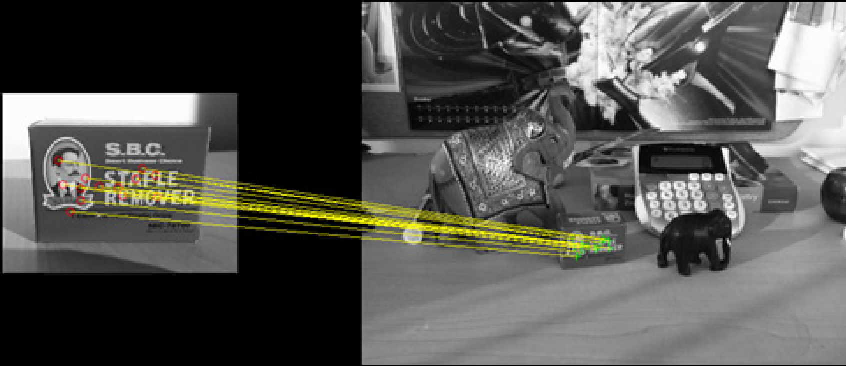
MRI slices



Stereo Vision



Object detection



Mathematics and Physics

$$
\begin{tikzcd}
\text{Physics problem} \arrow[r] & \text{Mathematics problem} \arrow[d] \\
\text{Physical solution} \arrow[u] & \text{Mathematical solution} \arrow[l]
\end{tikzcd}
$$

Before jumping on the computer and starting to code:

- Clearly state or translate the problem
- Define what is known as the *input*
- Define what is to be found as the *output*
- Develop an *algorithm*, i.e. a systematic way to solve the problem
- Verify the solution on simple input
- Implementing the algorithm

Example. Given that the sun is located $1.496 \cdot 10^8$ km away from the Earth and has a circumference of $4.379 \cdot 10^6$ km, calculate its density.

Strategy to solve the problem:

- Easy part
  - Problem: finding the density of the sun
  - Input: distance r, circumference c
  - Output: density d
- Finding the density is slightly more complicated:
  1. Approximate the Sun by a sphere and determine its volume V
  2. Think of Kepler's third law $\frac{T^2}{r^3} = \frac{4\pi^2}{GM}$
  3. Apply Kepler's third law to find the mass $M = \frac{4\pi^2 r^3}{GT^2}$

Algorithm. (*Density of the Sun*)

Input : $r = 1.496 \cdot 10^8,\ c = 4.379 \cdot 10^6,\ G = 6.674 \cdot 10^{-11},\ T = 365$

Output: Density of the Sun

$V \leftarrow \frac{4}{3}\pi(\frac{c}{2\pi})^3$;

$M \leftarrow \frac{4\pi^2 r^3}{GT^2}$;

return $\frac{M}{V}$;

After running the algorithm we find 338110866080

WRONG!

Units are inconsistent...

Algorithm. (*Density of the Sun*)

Input : $r = 1.496 \cdot 10^{11}$ m, $c = 4.379 \cdot 10^{9}$ m, $T = 365 * 24 * 3600$ s, $G = 6.674 \cdot 10^{-11}$ $\mathrm{m^3/kg/s^2}$

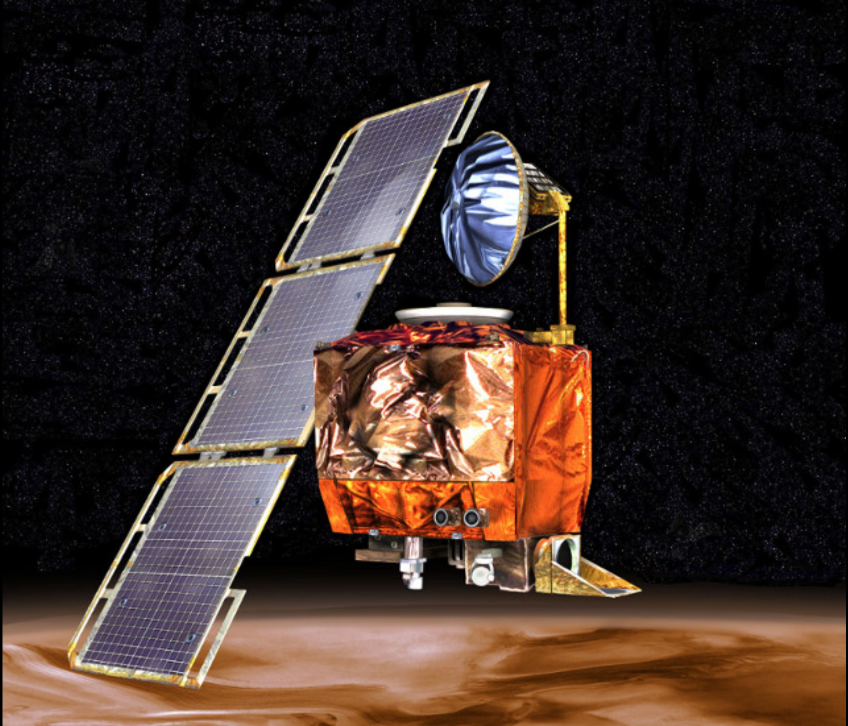
Output: Density of the Sun

$V \leftarrow \frac{4}{3}\pi\left(\frac{c}{2\pi}\right)^3$;

$M \leftarrow \frac{4\pi^2 r^3}{GT^2}$;

return $\frac{M}{V}$;

After running the algorithm we find 1404 $\mathrm{kg/m^3}$

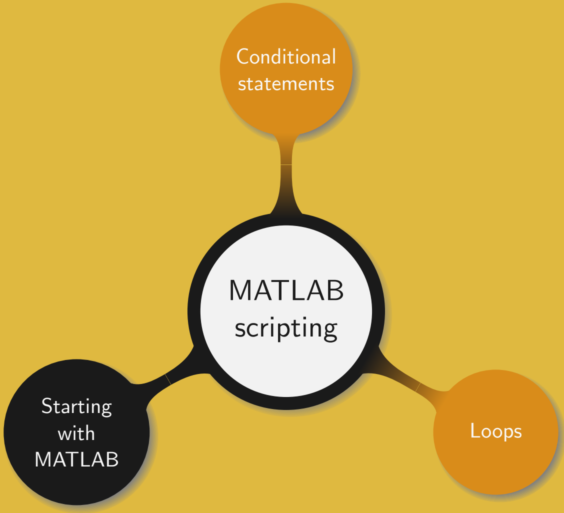


Key points

- What is a programming language?
- What are the two main types of programming language?
- What is an algorithm?
- How to tackle a problem?

2. MATLAB scripting

Chapter organisation



Two modes to start MATLAB:

- Desktop: graphical user interface
- Terminal: allows remote access, no mouse support

View in desktop mode:

1. Command history
2. Workspace
3. Command window
4. Help

Files must be in the current directory or a directory listed in the path

MATLAB as a simple calculator:

- Addition: +
- Subtraction: -
- Multiplication: *
- Power: ^
- Right division: /
- Left division: \
- Order: ()
- π: pi
- $\sqrt{-1}$: i or j
- Infinity: Inf

MATLAB as an advanced calculator:

- Hide the result: end the line with ";"
- Variables: must start with a letter, e.g. `a=1+2; A=3+2; a1_=4+5;`
- Comments: ignore everything after "%"
- Write two commands on a same line: separate them with a ","
- Split a line over several lines: end a partial line with "...", e.g.

```
very long line easier ...
  to read over two lines
```

MATLAB code to input in the workspace window:

```
r=1.496*10^11; c=4.379*10^9; G=6.674*10^-11;
T=365*24*3600;
V=4*pi/3*(c/(2*pi))^3;
M=4*pi^2*r^3/(G*T^2);
M/V
```

Understanding the code:

- How are variables named and used?
- Could the code be shorter?

MATLAB script:

- Write the code in a file and load it
- Variables are added to the workspace
- To avoid variable conflicts use: `clear`, `clearvars`, `clc`
- Add *cell breaks* to debug the code

Exercise. Write a script which prompts the user for two numbers, stores their sum in a variable, and displays the result.

```
clearvars, clc;
number1=input('Input a number: ');
number2=input('Input a number: ');
numbers=number1+number2;
disp(numbers);
```

Arrays are of a major importance in MATLAB

Generating arrays and matrices:

- Obtain a sequence of numbers: `a:b` or `a:b:c`
- Concatenate (join) elements: `[ ]`
- Define a 1-dimensional array: `[a:b]` or `[a:b:c]`
- Define a 2-dimensional array: `[a b c; d e f;]`
- Get n equidistant elements in $[a, b]$: `linspace(a, b, n)`
- Get an $n \times m$ array of 0: `zeros(n,m)`
- Get an $n \times m$ array of 1: `ones(n,m)`

Explain each of the following commands:

```
clearvars
a=magic(5)
a=[a;a+2], pause
a(:,3)=[]
a(:,3)=5
a(7,3)
```

```
a=reshape(a,5,8)
a', pause
sum(a)
sum(a(:,1))
sum(a(1,:))
```

Difference between arrays and matrices:

- Arrays:
  - Processed element by element
  - Add a "." in front of each operation, e.g. `.*`
- Matrices:
  - Default operations
  - Conjugate transpose: `'`
  - Determinant: `det`
  - Inverse: `inv`
  - Eigenvalues: `eig`

Explain each of the following commands:

```
A = [2 7 9 7 ; 3 1 5 6 ; 8 1 2 5]
A(:,[1 4]), pause
A([2 3],[3 1]), pause
reshape(A,2,6), pause
A(:), pause
flipud(A), pause
fliplr(A), pause
[A A(:,end)], pause
A(1:3,:), pause
[A ; A(1:2,:)], pause
sum(A),pause
sum(A'), pause
sum(A,2), pause
[ [ A ; sum(A) ] [ sum(A,2) ; sum(A(:)) ] ], pause
A.'
```

Accessing elements in a matrix

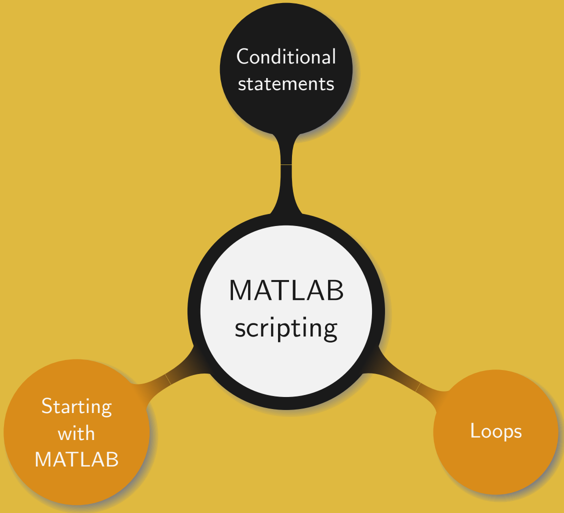
Given a matrix, elements can be accessed by:

- Coordinates: use the `(row,column)` position
- Indices:
  - Use a single number representing a position
  - The top left element has index 1
  - The bottom right "number of elements"

Example. Explain each of the following commands:

```
A=magic(5)
A(3,2)
A(6)
numel(A)
```

Chapter organisation



Run instructions based on the truth value of a given expression

Truth table for the three common operations:

A	B	$A \wedge B$	$A \vee B$	$A \oplus B$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Relational operators in MATLAB

Comparative operators:

- Less than: `<`
- Less or equal: `<=`
- Greater than: `>`
- Greater or equal: `>=`
- Equal to: `==`
- Not equal to: `~=`

Logical operators:

- And: `&`
- Or: `|`
- Not: `~`
- Xor: `xor(·,·)`

Short-circuit operators:

- Evaluate expression `B` if and only if `A` is true: `A && B`
- Evaluates expression `B` only if `A` is false: `A || B`

The `if` and `switch` statements

If it rains, then I take my umbrella

```
if expression1
  statements1
elseif expression2
  statements2
else
  statements
end
```

```
switch variable
  case value1
    statements1
  case value2
    statements2
  otherwise
    statements
end
```

When it rains, I take my umbrella, and my hat when it's sunny

Example.

```
1 exist('./file') & load('./file')
2 exist('./file') && load('./file')
3 k=input('Press a key: ','s');
4 if k>='0' && k<='9'
5   disp('Digit')
6 else
7   disp('Not a digit')
8 end
```

Understanding the code:

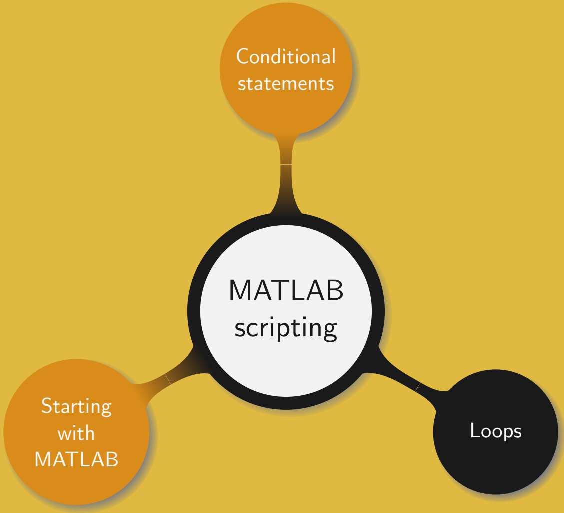
- Explain this script
- How to request a user input?
- What is `'s'` on line 3?
- What is a digit?

```
i=input('Input a digit: ');
switch i
  case 0
    disp('0')
  case {1,2,3,4}
    disp('<5')
  otherwise
    disp('>=5')
end
```

Understanding the code:

- Explain this script
- How is the code aligned?
- Why is `input` used without the parameter `'s'`?

Chapter organisation



Loops in MATLAB:

- Definition: group of statements repeatedly executed as long as a given conditional expression remains true
- Types: `while`, `for`, and vectorizing
- Vectorizing: generate a vector containing all elements
- `For` loop: clear steps and predefined end
- `While` loop: end based on a boolean expression
- Order of preference: vectorizing, `for`, and `while`

The `while` loop

```
while expression
  statements
end
```

```
i=0
while true
  i=i+1
end
```

Example.

```
i=1; o=input('Input a basic arithmetic operation: ','s');
while (o(i) >= '0' && o(i) <= '9') i = i+1; end
n1=str2num(o(1:i-1)); n=o(i); n2=str2num(o(i+1:end));
switch n
  case '+', n1+n2
  case '-', n1-n2
  case '*', n1*n2
  case '/', n1/n2
  otherwise, disp('Not a basic arithmetic operation')
end
```

Understanding the code:

- How well is the code formatted?
- Reformat the code with more spacing
- What is the user expected to input?
- What is the purpose of the `while` loop?
- How is switch used?
- What is happening if something else that an integer is input?

The for loop

```
for i=start:increment:end
  statements
end
```

```
a=[]
for i=0:2:100
  a=[a i]
end
```

Understanding the code:

- How is the code indented?
- What is the role of the increment?
- What is this code doing?
- Can you think of a faster way to obtain the same result?

Use MATLAB optimizations for vectors and array to construct lists

Example.

```
a=zeros(1,100000000); i=1;
tic; while i<=100000000; a(i)=2*(i-1); i=i+1; end; toc;
a=zeros(1,100000000);
tic; for i=1:100000000; a(i)=2*(i-1); end; toc;
tic; [0:2:199999999]; toc;
```

Understanding the code:

- Reformat and indent the code with one instruction per line
- What is this code doing?

The `continue` and `break` commands

More advanced loop commands:

- Directly jump to the next iteration: `continue`
- Exit the loop early: `break`

Example.

```
d={'1','2','3','4','5','6','7','8','9','0'}; cnt=0;
w=input('Input a word: ','s');
for i=1:length(w);
  switch w(i);
    case d;
      continue;
    case ' ';
      break;
    otherwise
      cnt=cnt+1;
  end
end
cnt
```

Understanding the code:

- What is this code doing?
- How is the code indented?
- What is the variable `d`?
- How are `continue` and `break` used?

Arrays are stored *linearly* inside memory:

- Row first: elements are read by row
- Column first: elements are read by column
- MATLAB uses the *column-major order*
- When using MATLAB the column should be in the outer loop

Exercise. Does MATLAB store $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ as 1,2,3,4,5,6 or 1,4,2,5,3,6?

Example.

```
1 N = 10000; a = zeros(N);
2 tic;
3   for j = 1:N
4     for i = 1:N
5       a(j,i) = 1;
6     end
7   end
8 toc;
```

Understanding the code:

- What is this code doing?
- Is `j` representing the rows or the columns, what about `i`?
- What is happening if `i` and `j` are switched on line 5?

Accessing specific elements in a matrix

Access elements depending on a *logical mask:*

1. Generate an logical array depending on some condition
2. Apply a transformation only on a 1 in the logical array

Example.

- For a matrix *A* set all its elements larger than 10 to 0
- Given a vector square all its even values and cube the others

```
A=magic(5); B=A >10;A(B)=0
a=input('Vector: ')
b=(mod(a,2)==0);
c=a.^2;
c(~b)=a(~b).^3
```

Understanding the code:

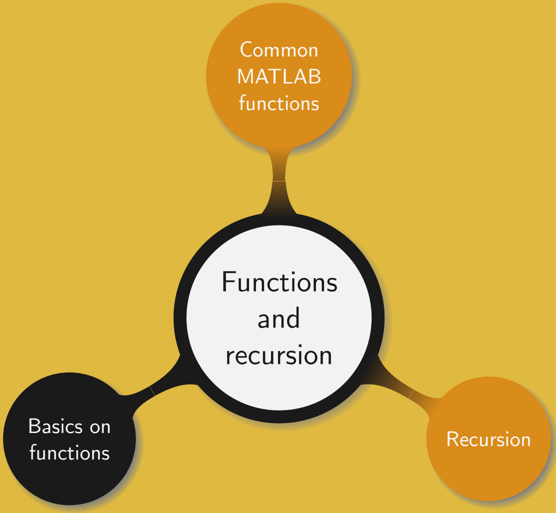
- What does `B=A > 10` mean?
- What is the goal of line 3?
- After line 4 what is in `c`?
- Why is `~b` used?
- What is better way to write lines 3 to 5?

Key points

- How to write simple scripts in MATLAB?
- What is the difference between an array and a matrix?
- What is a conditional statements?
- What loop types exist in MATLAB, which one is best used?
- What is a logical mask?

3. Functions and recursion

Chapter organisation



From script to function

Script:

- Sequence of MATLAB statements
- No input/output arguments
- Operates on data on the workspace

Function:

- Sequence of MATLAB statements
- Accepts input/output arguments
- Variable are not created on the workspace

Functions in MATLAB

Basics on MATLAB functions:

- Function saved in a .m file
- The .m file must be in the "path"
- The function name must be the same as the filename
- Prototype: `function [out1,out2,...] = Myfct(in1,in2,...)`
- Functions can be called from an .m file or from the workspace

Script

```
r=1.496*10^11; c=4.379*10^9;
G=6.674*10^-11;
T=365*24*3600;
V=4*pi/3*(c/(2*pi))^3;
M=4*pi^2*r^3/(G*T^2);
M/V
```

Function

density.m

```
function d=density(r,c,T)
G=6.674*10^-11;
  V=4*pi/3*(c/(2*pi))^3;
  M=4*pi^2*r^3/(G*T^2);
  d=M/V;
```

An .m file can contain:

- A main function: has the same name as the filename
- Sub-functions: only accessible by functions from the **same** file

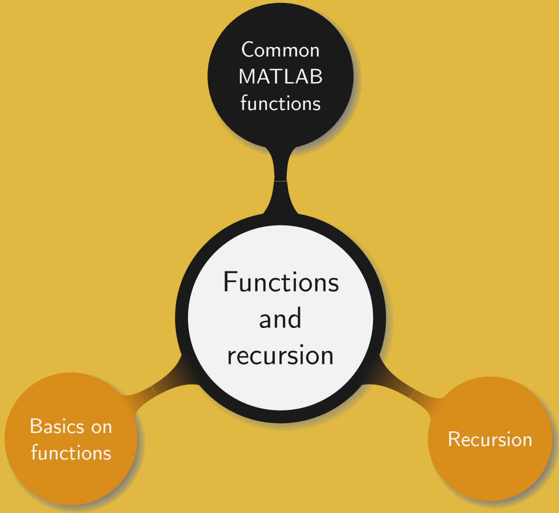
Exercise. For a vector, write a function returning the mean and the standard deviation. Calculate the mean in a sub-function

stat.m

```
function [mean,stdev] = stat(x)
    n = length(x);
    mean = avg(x,n);
    stdev = sqrt(sum((x-mean).^2)/n);

function  mean = avg(x,n)
    mean = sum(x)/n;
```

In the previous example:

- How to save both the variable mean and stdev?
- How many Input have the `avg` and `stat` functions?
- Is the function `avg` accessible from the workspace, why?
- If `mean` is changed into `m` in the first function does it need to be changed in the second function, why?


Common MATLAB functions
Functions and recursion
Basics on functions
Recursion

Basic math calculations:

- Defining a function: `f=@(x) x^2-1`
- Integral: `syms z; int(z^2+1), int(z^2+1,0,1)`
- Differentiation: `syms t; diff(sin(t^2))`
- Limit: `limit(sin(t)/t,0)`
- Finding a root of a continuous function: `fzero(f,0.5)`
- Square root: `sqrt(9)`
- Nth root: `nthroot(4, 3)`

The `save` and `load` functions:

- Save variables: `save('file','var1','var2',...,'format')`
- Load variables: `load('file','format')`

Random number generation:

- An $n \times m$ matrix of random numbers: `rand(n,m)`
- An $n \times n$ matrix of random integers between `m` and `M`: `randi([m M],n)`
- Random numbers initialized with a specific seed: `rand('state', datenum(clock))`
- A random permutation: `randperm(n)`

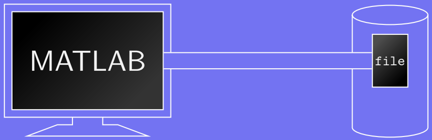
Writing formatted data into a string:

- Command: `sprintf('string', variable1, variable2,...)`
- `'string'`: text composed of
  - Words, spaces, numbers
  - "% flags", replaced by the value of variables, e.g. `'%g'`
  - Special characters, e.g `'\n\t'`

Example.

```
a=pi; b=sprintf('%g',pi)
sprintf('%d',round(pi))
sprintf('%s','pi')
a=[1 2 3;2 5 6;3 7 8];
text=sprintf('size: %d by %d', size(a))
```

Open a stream between MATLAB and a file




```
fd=fopen('file.txt', 'permission')
fclose(fd)
```

Different permissions to access a file:

- Read only: `r`
- Write in a new file: `w`
- Append to a file: `a`
- Read and write: `r+`
- Read and overwrite: `w+`
- Read and append: `a+`

Accessing a file:

- Write: `fprintf(fd, 'string', 'variables')`
- Read:
  - Following a known format: `fscanf(fd, 'format')`
    - Convert values into the specified format
    - Return an array containing the read elements
  - A whole line: `fgetl(fd)`

Any opened stream must be closed

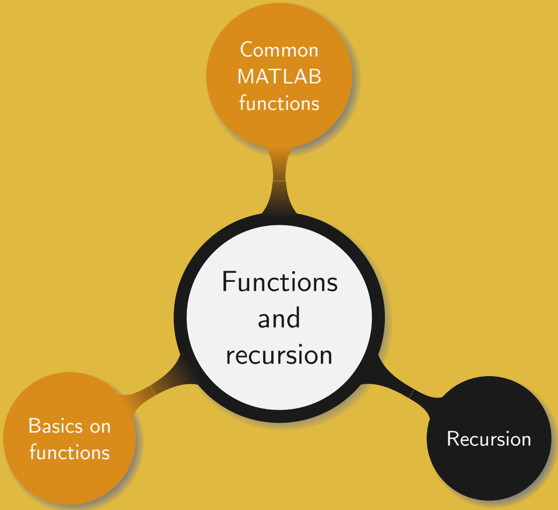
Exercise. Given a text file where each line is composed of three fields, first-name, name and email, write a MATLAB function generating a text file where (i) the order of the lines is random and (ii) each line is composed of the same fields in the following order: name, first-name, and email.

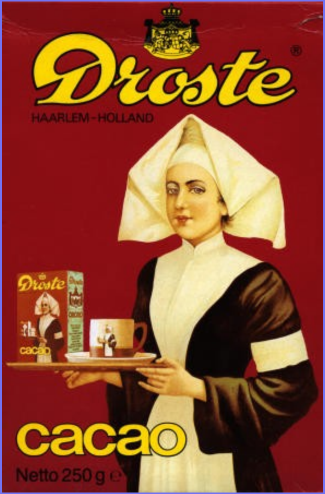
sortnames.m

```
function sortnames(finput, foutput)
  fd1=fopen(finput,'r');
  i=1;
  line=fgetl(fd1);
  while line ~= -1
    a=find(isspace(line),2);
    info{i}=sprintf('%s %s %s\n', line(a(1)+1:a(2)-1), ...
      line(1:a(1)-1), line(a(2)+1:end));
    i=i+1; line=fgetl(fd1);
  end
  fclose(fd1);

  fd2=fopen(foutput,'w');
  for j=randperm(i-1)
    fprintf(fd2,info{j});
  end
  fclose(fd2);
```

Understanding the code:

- How is the code indented?
- How to check the last line was reached, why?
- How to access the different fields?
- How to perform a random permutation?
- Each time a file is opened it **must** be ____


Common MATLAB functions
Functions and recursion
Basics on functions
Recursion



Recursive software acronyms:

- GNU: GNU's Not Unix!
- PHP: PHP Hypertext Preprocessor
- WINE: WINE Is Not an Emulator
- LAME: LAME Ain't an MP3 Encoder
- JOJ: Joint Online Judge / JOJ Online Judge

The *iterated logarithm* function, denoted $\log^*$, is defined by

$$\begin{aligned}\log^* : \mathbb{R} &\longrightarrow \mathbb{N}\\ x &\longmapsto \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log_2 x & \text{if } x > 1 \end{cases}\end{aligned}$$

The iterated logarithm of n is the number of times the logarithm function has to be applied in order to get a number smaller than 2.

Example.

$$\begin{aligned}\log^* 65536 &= 1 + \log^* 16\\ &= 1 + 1 + \log^* 4\\ &= 1 + 1 + 1 + \log^* 2\\ &= 1 + 1 + 1 + 1 + \log^* 1 = 4\end{aligned}$$

Understanding $\log^*$: can it be implemented iteratively, i.e. using loops?

Writing a recursive algorithm to evaluate the $\log^*$ function:

- What is the base case, i.e. the one allowing the recursion to stop?
- How is the function $\log^*$ calling itself?

Algorithm. ($\log^*$ *evaluation*)

Input : $x \in \mathbb{R}$
Output : $\log^* x$

Function `logstar`(x):
  if $x \leq 1$ **then return** 0;
  else return $1+$`logstar`$(\log_2 x)$;
end

logstar.m

```
function lsx=logstar(x)
  if x <=  1 ; lsx=0;
  else
    lsx=1+logstar(log2(x));
  end
```

Understanding the code:

- Why is it important to write clear algorithms before coding?
- What transformation is applied to any $x > 1$?
- Write an iterative implementation and compare the speed

A child couldn't sleep, so her mother told her a story about a little frog, who couldn't sleep, so the frog's mother told her a story about a little bear, who couldn't sleep, so the bear's mother told her a story about a little weasel who fell asleep. ...and the little bear fell asleep; ...and the little frog fell asleep; ...and the child fell asleep.

Designing a bedtime story algorithm:

- What is the base case?
- What transformation is applied to any $n > 0$?
- What is happening each time someone cannot sleep?
- What is the order for *trying* to fall asleep?
- What is the order for *effectively* falling asleep?

Algorithm. (*Bedtime story*)

```
Input  : n ∈ {0, 1, 2, 3}, with child=3, frog=2, bear=1, weasel=0
Output : Everybody asleep
Function Read(n):
    if n = 0 then sleep(n);
    else i ← n − 1; Read(i); sleep(n);
end
```

Understanding the algorithm:

- Draw a simple diagram showing how recursion is applied
- When $i = 2$, what is n?
- What is happening when `Read(2)` is encountered?
- When is `sleep(2)` run?
- Implement this algorithm in MATLAB

Numbers in words

For an automated information service a telephone company needs the digits of phone numbers to be read digit by digit. Design an algorithm allowing to rewrite an sequence of digits into words, with a space between each word, but no space at the beginning and at the end.

Understanding the problem:

- What issue arises when trying to solve it iteratively?
- When solving it recursively:
  - What is the base case?
  - Which digit should be printed first, left or right most?
  - Given an integer, how to remove its digits one by one?
  - What precise work needs to be done at each level of the recursion?
  - When going down the recursion, which digit should be printed?
  - When going up the recursion, which digit should be printed?

Algorithm. (*Numbers in words*)

Input : A large integer n

Output : n, digit by digit, using words

```
Function PrintDigit(n):
    case n do
        0: print('zero'); 1: print('one'); 2: print('two'); 3: print('three');
        4: print('four'); 5: print('five'); 6: print('six'); 7: print('seven');
        8: print('eight'); 9: print('nine'); other: error('not a digit');
    end case
end
Function PrintDigits(n):
    if n < 10 then PrintDigit (n);
    else PrintDigits (n div 10); print(' '); PrintDigit (n mod 10);
end
```

Understanding the algorithm: compare what the algorithm does to your answers on the previous slide

When to prefer recursion over iteration:

- A recursive algorithm is more obvious than an iterative one
- Each intermediate recursion level is computed only once
- Depends on the language

MATLAB, C, and C++:

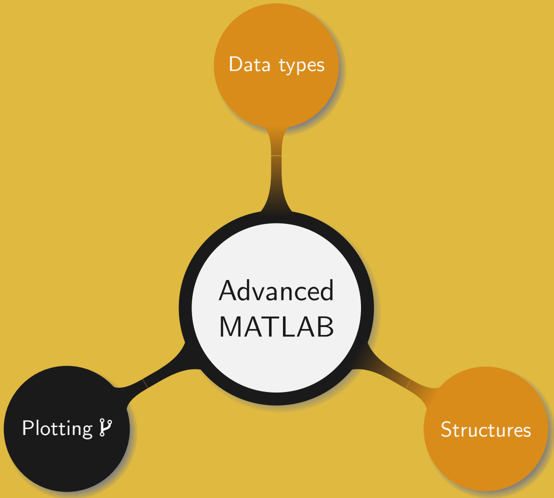
- Deal best with iterative
- Can run recursive algorithm without any problem
- Prefer iterative over recursive when facing two equivalent solutions

When using recursion pay attention to the memory usage

Key points

- Why should functions be preferred over scripts?
- How to perform mathematical calculations in MATLAB?
- How to save the state of the workspace?
- What is recursion?
- When to use recursion?

4. Advanced MATLAB


Data types
Advanced MATLAB
Plotting
Structures

Basic plotting functions:

- Plot the columns of x, versus their index: `plot(x)`
- Plot the vector x, versus the vector y: `plot(x,y)`
- Plot function between limits: `fplot(f,lim)`
- More than one graph on the figure: `hold`

Plotting properties:

- Axis properties: `axis`
- Line properties: `linespec`
- Marker properties

Explain each of the following commands:

```
y=exp(0:0.1:20);plot(y);
x=[0:0.1:20];y=exp(x);plot(x,y);
x=[-4:0.1:4];y=exp(-x.^2);plot(x,y,'-or');
hold on;
%fplot('2*exp(-x^2)',[-4 4]);
fplot(@(x)2.*exp(-x.^2))
hold off;
f=@(x) sin(1./x)
fplot(f,[0 .5])
hold;
fplot(f,[0 0.5],10000,'--r')
```

Study data in more than one dimension:

- Visualise functions of two variables
- Create a surface plot of a function
- Display the contour of a function

Example. For $t \in [0, 2\pi]$ display the curve parametrised by

$$\begin{cases} x(t) = \sin(2t) + 1 \\ y(t) = \cos(t^2) \end{cases}$$

```
t=0:.01:2*pi;
x=sin(2.*t)+1;
y=cos(t.^2);
plot3(x,y,t);
```

Example

Process 3D plotting:

1. Define the function
2. Set up a mesh
3. Display the function

Display functions:

- Contour: `contour(x,y,z)`
- Color map: `pcolor(x,y,z)`
- 3D view: `surf(x,y,z)`

Explain each of the following commands:

```
[x,y]=meshgrid(-4:0.1:4);
z=(x.^2-y.^2).*exp(-(x.^2+y.^2));
pcolor(x,y,z);
contour(x,y,z);
surf(x,y,z);
shading interp;
colormap gray;
```

2D plotting:

- Bar graph: `bar(x,y)`
- Horizontal bar graph: `barh(x,y)`
- Pie chart: `pie(x)`

3D plotting:

- 3D bar graph: `bar3(x,y)`
- 3D horizontal bar graph: `bar3h(x,y)`
- 3D pie chart: `pie3(x)`

Other useful functions:

- Polar graph: `polar(t,r)`
- More than one plot: `subplot(mnp)`

Goals of interpolation:

- Draw a curve through known data points
- Use this curve to approximate unknown values in other points

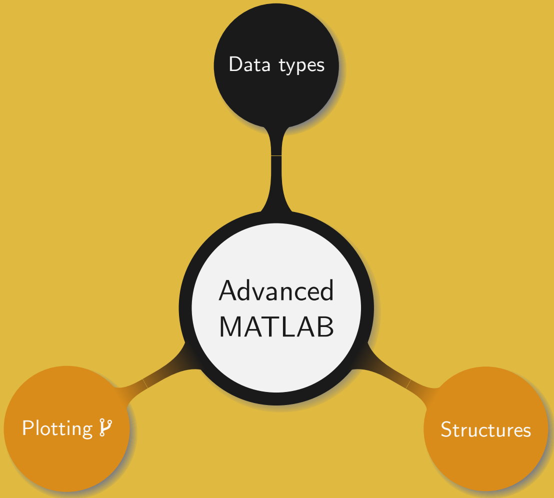
Interpolation in MATLAB:

- 2D: `interp1(X,Y,xi,m)`
- 3D: `interp2(X,Y,Z,xi,yi,m)`

Example.

```
X=[0:3:20]; Y=[12 15 14 16 19 23 24];
interp1(X,Y,4.1)
plot(X,Y,'*')
hold;
xi=[4.1 5.3 8.2 12.6];
yi=interp1(X,Y,xi);
plot(xi,yi,'or');
```

Chapter organisation



So far in MATLAB we:

- Focused on high level problems
- Did not address the internal mechanisms of the program

Not all the data is the same:

- How information is represented in the computer
- Determine the amount of storage allocated to a type of data
- Methods to encode the data
- Available operations on that data

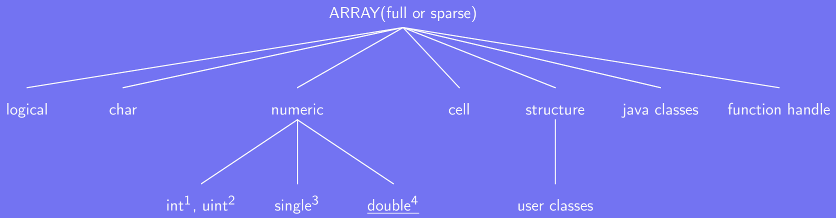
From mathematics to computer science:

- Different numbers (integer, real, complex, etc.)
- Different ranges (short, long, etc.)
- Different precisions (single, double, etc.)

Example. Representing signed integers over 8 bits:

1. Signed magnitude: 7 bits for the numbers, 1 bit for the sign
2. Two's complement: invert all the bits of a, add 1 to get $-a$
   e.g. $00101010 \to 11010101 + 1 = 11010110$
   $00101010 = -0 \cdot 2^7 + 2^5 + 2^3 + 2 = 42$
   $11010110 = -1 \cdot 2^7 + 2^6 + 2^4 + 2^2 + 2 = 86 - 128 = -42$

Data types in MATLAB



1. int: int8, int16, int32 and int64
2. uint: unit8, uint16, uint32 and uint64
3. 32bits: `realmax('single')`, `realmin('single')`
4. 64 bits: `realmax`, `realmin`

Type related functions

Type of a variable:

- `whos`
- `isnumeric`
- `isreal`
- `isnan`
- `isinf`
- `isfinite`

Numeric conversions:

- `cast(a,'type')`
- `uint8(a)`

Useful string functions:

- `isletter`
- `isspace`
- `strcmp(s1,s2)`
- `strcmpi(s1,s2)`
- `strncmp(s1,s2,n)`
- `strncmpi(s1,s2,n)`
- `strrep(s1,s2,s3)`
- `strfind(s1,s2)`
- `findstr(s1,s2)`
- `num2str(a,'format')`
- `str2num(s)`

Exercise. Input two space separated numbers as a string and calculate their sum

```
clearvars, clc;
numbers=input('Input two numbers: ', 's');
space=strfind(numbers,' ');
number1=str2num(numbers(1:space-1));
number2=str2num(numbers(space+1:end));
number1+number2
```

Understanding the code:

- What is this code doing?
- How are `strfind` and `str2num` used?
- What is `space` containing, and how is it used?

Working with a binary file:

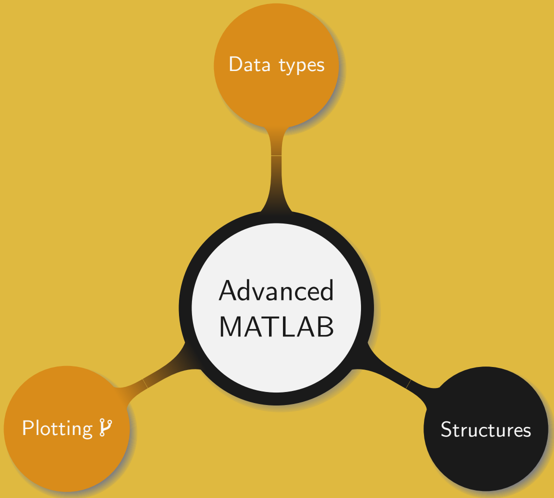
- Read: `fread(fd,count,'type')`, read `count` elements as `type`
- Write: `fwrite(fd, A, 'type')`, write `A` as `type`
- Position in a file: `ftell(fd)`
- Jump in a file: `fseek(fd,offset,'origin')`, move by `offset` bytes, starting at `origin`

Example.

```
A=3:10;
fd=fopen('test','w'); fwrite(fd,A,'int32');^^I
fclose(fd);
fd=fopen('test','r'); fseek(fd,4*4,'bof');
fread(fd,4,'int32'), ftell(fd)
fseek(fd,-8,'cof');fread(fd,4,'int32')
fclose(fd);
```

Alter the previous sample code and explain its behaviour:

- Define a different `A`
- Display the type of `A`
- Read the numbers as `int64`
- Write the numbers as `double` and read them as `int8`
- Consecutively display the first and fourth elements
- Read the file `test` using a regular text editor, what do you see?
- Read the file `test` using a tool such as `hexdump`
- Determine whether your computer uses *Big Endian* or *Little Endian*?


Data types
Advanced MATLAB
Plotting
Structures

The need for structures

Structure:

- Array with “named data containers” called fields
- A fields can contain data of any type

Example. A student is defined by a `name`, a `gender`, and some `grades`. We can represent a student in the form of a “tree” or organise many students in an array.

```
Student
 |__ Name ______ John Doe
 |__ Gender ______ Male
 |__ Marks ______ 60, 92, 71
```

Name	Gender	Marks
Iris Num	F	30 65 42
Jessica Wen	F	98 87 73
Paul Wallace	M	65 73 68

Exploiting the power of structures:

1. Initializing the structure

```
student(1)= struct('name','iris num', 'gender', 'female', ...
  'marks', [30 65 42]);
student(2)= struct('name','jessica wen', 'gender', 'female', ...
  'marks', [98 87 73]);
student(3)= struct('name','paul wallace', 'gender', 'male', ...
  'marks', [65 72 68]);
```

2. Accessing elements

```
student(3).gender
mean([student(1:3).marks])
```

3. Who got the best mark?

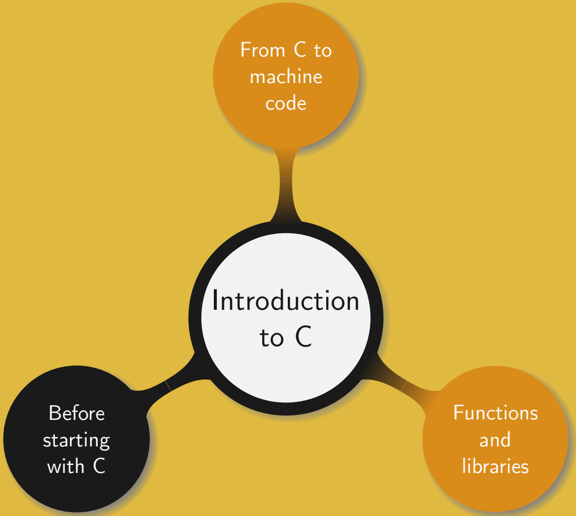
```
[m,i]=max([student(1:3).marks]);
student(ceil(i/3)).name
```

Key points

- Using `plot` draw simple geometrical shapes
- How to keep or erase previous graphs?
- In MATLAB what is the default data type? Is it a good choice?
- Cite the most common data types and their size in bytes
- What is a data structure?

5. Introduction to C

Chapter organisation



In the old time:

- Unix OS was implemented in assembly
- New hardware implied new possibilities
- New possibilities implied new code
- Much time wasted rewriting the OS for the new hardware

Development of a new language:

- Authors: Ken Thompson & Dennis Ritchie
- Location: AT&T Bell Labs
- Time frame: 1969 – 1973
- Name: C, as derived from B

Why using C?

Main characteristics:

- One of the most widely used languages
- Available for the majority of computer architectures and OS
- Many languages derived from C

Advantages of C:

- Performance
- Interface directly with hardware
- Higher level than assembly
- Low level enough
- Zero overhead principle

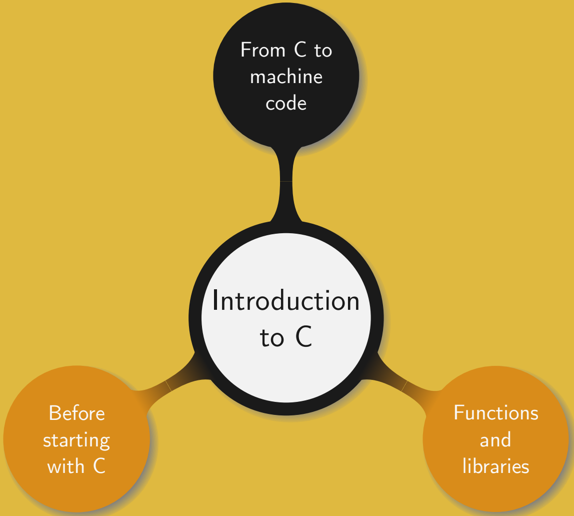
Common software to write C code:

- Text editor + compiler
- Code::Blocks, Geany, Xcode, Clion, Visual studio code
- Microsoft visual C++ ← **BAD!**

Common C compilers:

- GNU C Compiler
- Clang
- Intel C Compiler
- Tiny C Compiler

Chapter organisation



A first example

gm_base.c

```c
#include <stdio.h>
int main () {
  printf("good morning!\n");
  return 0;
}
```

Program structure:

- A unique `main` function: used only to “dispatch” the work
- Other functions: effectively doing the work

Writing a C function

```c
OType FName(IType IName,...) {
  function's body
}
```

Compiling a C program

```sh
sh $ gcc gm_base.c -o gm_base
sh $ ./gm_base
```

Explain the following code:

blocks.c

```c
#include <stdlib.h>
#include <stdio.h>
int main () {
  {
    int a=0; printf("%d ",a);
  }
  {
    double a=1.124; printf("%lf ",a);
  }
  {
    char a='a'; printf("%c ",a);
  }
//  printf("%d",a);
}
```

Shorthand operators

Questions.

- How is the code indented?
- Why is line 13 commented out?
- What happens if lines 9 and 10 are deleted?

Common shortcuts:

- Increment: e.g. `a++`
- Decrement: e.g. `a--`
- Add: e.g. `x+=y`
- Subtract: e.g. `x-=y`
- Multiply: e.g. `x*=y`
- Divide: e.g. `x/=y`

Roles of a header file:

- Define function prototypes
- Define constants, data types...
- A function used in a program must have been declared earlier

Syntax to include `header.h`:

- Known system-wide: `#include<header.h>`
- Unknown to the system: `#include "/path/to/hearder.h"`

Result of `#include<stdio.h>`:

```
sh $ gcc -E gm_base.c
```

Goal:

- Set "type-less" read-only variables
- Hard-code values in the program
- Quickly alter hard-coded values over the whole file

gm_def.c

```c
#include <stdio.h>
#define COURSE "VG101"
int main () {
  printf("good morning %s!\n",COURSE);
}
```

Result of #define:

```sh
sh $ gcc -E gm_def.c
```

Taking advantage of #define

The #ifdef and #ifndef instructions:

- Test if some “#define variable” is (un)set
- Compile different versions of a same program

gm_ifdef.c

```c
#include <stdio.h>
//#define POLITE
int main () {
#ifdef POLITE
  printf("good morning!\n");
#endif
}
```

gm_ifndef.c

```c
#include <stdio.h>
//#define RUDE
int main () {
#ifndef RUDE
  printf("good morning!\n");
#endif
}
```

Result of #if(n)def:

```sh
sh $ gcc -E gm_ifdef.c
sh $ gcc -E gm_ifndef.c
```

Writing simple macros:

- Define type-less functions
- Perform fast and simple actions
- To be used only on specific circumstances (e.g. min/max)
- Do not use for regular functions

gm_macro.c

```c
#include <stdio.h>
#define SPEAK(x) printf("good morning %s!\n",x)
int main () {
  SPEAK("VG101");
  SPEAK("VE475");
}
```

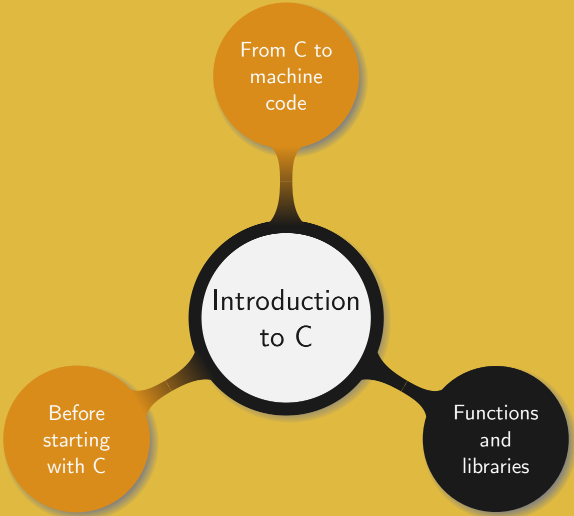
Result of macros:

```sh
sh $ gcc -E gm_macro.c
```

Common compilation errors

Often the compilation process fails because of:

- Syntax errors
- Incompatible function declarations
- Wrong Input and Output types
- Operations unavailable for a specific data types
- Missing function declarations
- Missing machine codes for some functions


From C to machine code
Introduction to C
Before starting with C
Functions and libraries

The main function:

- Never write a whole program in the main function
- Use the main function to dispatch the work to other functions
- Most of the coding must be done outside of the main function

Reminders:

- Always add comments to the code
  - A single line: start with //
  - Multiple lines: anything between /* and */
- As much as possible use a function per task or group of tasks
- If the program becomes large split it over several files

A long program

ans_orig.c

```c
#include <stdio.h>
double answer(double d);
int main () {
  double a;
  scanf("%lf",&a);
  printf("%lf\n", answer(a));
}
double answer(double d) {return d+1337;}
```

Functions and operators used:

- Display the integer contained in *a*: `printf("%d",a)`
- Read and store an integer in *a*: `scanf("%d",&a)`
- Both functions can take a variable number of parameters
- Arithmetic operators: `+`, `-`, `/`, `%`

Splitting the code over several files:

ans_main.c

```c
#include <stdio.h>
#include "ans.h"
int main () {
  double a; scanf("%lf",&a); printf("%lf\n", answer(a));
}
```

ans.c

```c
#include "ans.h"
double answer(double d) {
  return d+1337;
}
```

ans.h

```c
#ifndef ANS_H
#define ANS_H
double answer(double d);
#endif
```

Compilation:

```sh
sh $ gcc ans_main.c ans.c -o ans
```

Understanding the code:

- How was the original code split?
- What should be the content of a header file?
- What are header guards?
- Why is the program only compiling when both files are provided?
- What happens `#include` lines are removed?

A *library* is a collection of functions, macros, data types, and constants

Example. The C mathematics library:

- Mathematical functions, e.g. log, exp, trigonometric, floor, etc.
- Add the header: `#include <math.h>`
- Add the corresponding compiler flag: `-lm`

math.c

```c
#include<stdio.h>
#include<math.h>
#define PI 3.14159265358979 32
int main() {
  printf("%g\n", \
    tgamma(sqrt(atanh(PI))));
}
```

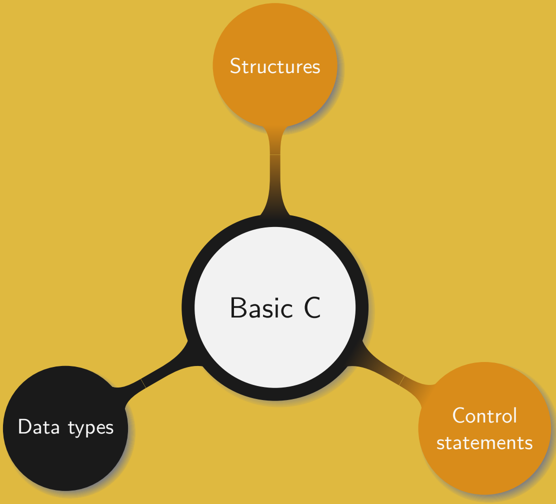
```sh
sh $ gcc math.c -lm -std=c11
     -Wall -Werror -Wextra
     -Wconversion -Wvla
     -Wpedantic
```

Key points

- Is C a compiled or interpreted language?
- How to transform a C program into machine code?
- Why are data types of a major importance?
- What are header guards, and why using them?

6. Basic C

Chapter organisation



Types of variables

Three main categories of variables:

- Constant variables: `#define PI 3.14159`
- Global variables: defined for all functions
- Local variables: defined only in the function

Never ever use global variables in ENGR151

Common use:

- Variables for `#define` are UPPERCASE
- Other variables are lowercase, or capitalised
- Variable names cannot exceed 31 characters
- Variable names can start with _ or a character
- Variables starting with _ are “hidden”

Data types in C:

- Integer: `int`
- Character: `char`
- Valueless type: `void`
- Fractional numbers:
  - Single precision: `float`
  - Double precision: `double`

The C standard only fixes the size of `char` *(1 byte)*

Different variations available:

- `char`: `signed char`, `unsigned char`
- `int`: `short int`, `signed short int`, `unsigned short int`, `signed int`, `unsigned int`, `long int`, `signed long int`, `unsigned long int`, `long long int`, `signed long long int`, `unsigned long long int`
- `double`: `long double`

Extra variations: `static`, `register`, `extern`, `volatile`

Basic number types:

- `int`: size limitation, from 0 to $2^{32}-1$
- `float`: 7 digits of precision, from $1 \cdot 10^{-38}$ to $3 \cdot 10^{38}$
- `double`: 13 digits of precision, from $2 \cdot 10^{-308}$ to $1 \cdot 10^{308}$

Example.

```
float a=1.0; int b=3; double c;
```

Characters:

- Strings are viewed as arrays of characters
- Characters are enclosed in single quotes, e.g. `char a='a';`
- Strings are enclosed in double quotes
- Character are encoded using the American Standard Codes for Information Interchange (ASCII)

What output to expect?

types1.c

```c
#include <stdio.h>
int main() {
  printf("%d %f\n",7/3,7/3);
}
```

types2.c

```c
#include <stdio.h>
int main() {
  printf("%d %f\n",7/3,7.0/3);
}
```

In the previous codes:

- What do `%f` and `%d` mean?
- What is the type of `7/3` for the compiler?
- How to ensure the compiler does not see an `int`?
- In term of datatype, what is the issue in the first code?

Changing data type:

- Float to int: `float a = 4.8; int b = (int) a;`
- Int to char: `int a = 42; char b = (char) a;`
- Try double to char, int to float

Always think of the size...

Example.

types3.c

```c
#include <stdio.h>
int main() {
  float c=4.8; printf("%d\n", (int)c);
  int f=42; printf("%c\n", (char)f);
  double a=487511234.7103;
  char b=(char) a;
  printf("%c, %c\n",b,a);
  int d=311;
  float e=(float) d;
  printf("%d %f\n",d,e);
  printf("%c\n",d);
}
```

Understanding the code:

- Which type castings work well?
- What is the length of a `char`?
- What is the length of an `int`?
- What is printed for `d`?
- What is the issue when displaying `d` as a `char`?

Exercise. Write C program featuring a function `apbp1(float a, float b)` which returns the nearest integer to `a+b+1`

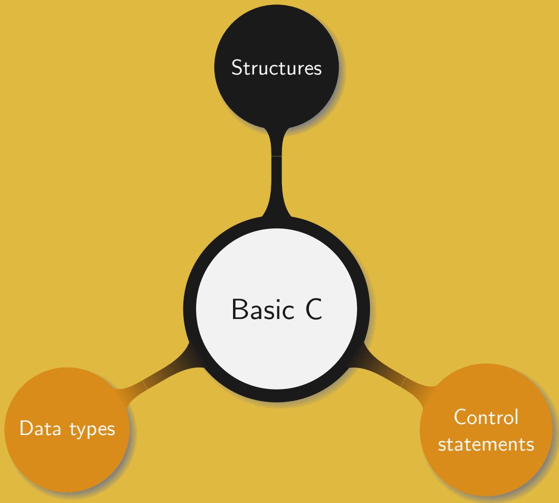
apbp1.c

```c
#include <stdio.h>
int apbp1 (float a, float b);
int main () {
  float a, b;
  scanf("%f %f", &a,&b);
  printf("%d\n", apbp1(a,b));
}
int apbp1 (float a, float b) {
  a++; a+=b;
  return((int) (a+0.5));
}
```

Understanding the code:

- Discuss the use of shorthand operators and type casting
- Why is not all the code in the main function
- How is indentation done?
- Does the code contain any global variable?

Chapter organisation



More data types in C:

- Reminder: a bit belongs to $\{0, 1\}$ and a byte is 8 bits
- Operating data at low level, e.g. shift <<, >>
- A `char` does not necessarily contains a character
- Logical operations are of a major importance
- Understanding data representation is important to be efficient
- Structures, enumerate, union

struct.c

```c
#include <stdio.h>
typedef struct _person {
  char* name;
  int age;
} person;
int main () {
  person al={"albert",32};
  person gil;
  gil.name="gilbert";
  gil.age=23;
  struct _person so={"sophie",56};
  printf("%s %d\n",al.name, al.age);
  printf("%s %d\n",gil.name, gil.age);
  printf("%s %d\n",so.name, so.age);
}
```

Understanding the code:

- How is a structure defined?
- How to define a new type?
- What are two ways to set the value of a field in a structure?
- How to access the values of the different fields in a structure?

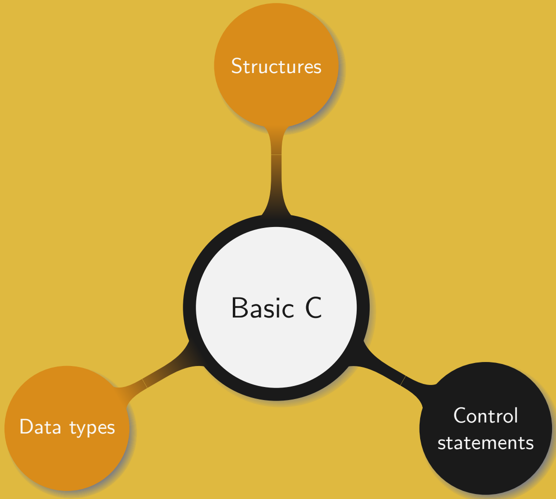
struct_fct.c

```c
#include <stdio.h>
typedef struct person {
  char* name; int age;
} person_t;
person_t older(person_t p, int a);
int main () {
  person_t al={"albert",32};
  al=older(al,10);
  printf("%s %d\n",al.name,al.age);
}
person_t older(person_t p, int a) {
  printf("%s %d\n",p.name, p.age);
  p.age+=a;
  return p;
}
```

Understanding the code:

- How is the age increased?
- How is the person's information sent to a function?
- How to return the person's information after the function?
- How many output can a C function have?

Chapter organisation



jump.c

```
#include <stdio.h>
int main() {
  int i=0;
  printf("I am at position %d\n",i);
  i++;
  goto hat;
  i++;
  printf("I am at position %d\n",i);
  hat:
    i++;
    printf("It all ends here, at position %d\n",i);
  return 0;
  i++;
  printf("Unless it's here at position %d\n",i);
}
```

Understanding the code:

- What positions are displayed?
- Why are some positions skipped?
- How to use the `goto` statement?
- Why should the `goto` statement (almost) never be used?

Basics on conditional statements:

- No boolean type, 0 means `False`, anything else `True`
- Boolean evaluation: `<`, `<=`, `>`, `>=` , `==`, `!=`
- Not: `!`, short-circuit operators: `&&`, or: `||`
- Bit operations: `&`, `|`, `^`

Conditional ternary operator: `? :`

```
condition ? expression1 : expression2
```

Example. A macro returning the max of two numbers:

```
#define MAX(a,b) a>=b ? a : b
```

The if and switch statements

```
if (condition) {
  statements;
}
else {
  statements;
}
```

```
switch(variable) {
  case value1:
    statements;
    break;
  case value2:
    statements;
    break;
  default:
    statements;
    break;
}
```

Example.

cards.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define ACE 14
#define KING 13
#define QUEEN 12
#define JACK 11
int main () {
  int c; srand(time(NULL)); c=rand()%13+2;
  switch (c) {
    case ACE: printf("Ace\n"); break;
    case KING: printf("King\n"); break;
    case QUEEN: printf("Queen\n"); break;
    case JACK: printf("Jack\n"); break;
    default: printf("%d\n",c);
  }
}
```

Understanding the code:

- How to generate random numbers in C?
- What is the use of `srand()`?
- Write this code using the `if` statement
- Adapt the code such as to display the complete card name, e.g. "Ace of spades"
- What happens if a `break` is removed?
- Explain why and compare to the behavior in MATLAB

Structure of the two types of `while` loops:

```
while (conditions) {
  statements;
}
```

```
do {
  statements;
} while (conditions);
```

Example.

```
int i=0;
while(i++<3) {
  printf("%d",i);
}
```

```
int i=0;
do {
  printf("%d",i);
} while(i++<3);
```

Understanding the code:

- What is the difference between the two outputs?
- What happens if `i++` is changed for `++i`?

Structure of a `for` loop:

```
for(init;test;step) { statements; }
```

- `init`: executed at the beginning of the loop
- `test`: tested at the beginning of each iteration
- `step`: executed at the end of each iteration

Example.

```
for(i=0; i<n; i++)
  printf("%d ", i);
i=0; for(;i<n;i++)
  printf("%d ", i);
for(i=0; i<n;)
  {printf("%d\n",i); i++;}
for(i=0;i<n;)
  printf("%d ",i++);
```

```
fct=1;
for(i=1;i<=n;i++) fct*=i;
printf("%d ", fct);
for(i=1,fct=1;i<=n;fct*=i,i++);
printf("%d ", fct);
for(i=1,fct=1;i<=n;fct*=i++);
printf("%d\n", fct);
```

The `break` and `continue` statements

Understanding the code:

- What are the loops on the right doing?
- How is the code indented?
- Which `for` loop is the clearest and best used?

Acting from within a loop:

- Early exit of a loop: `break`
- Skip to the next loop iteration: `continue`

Example.

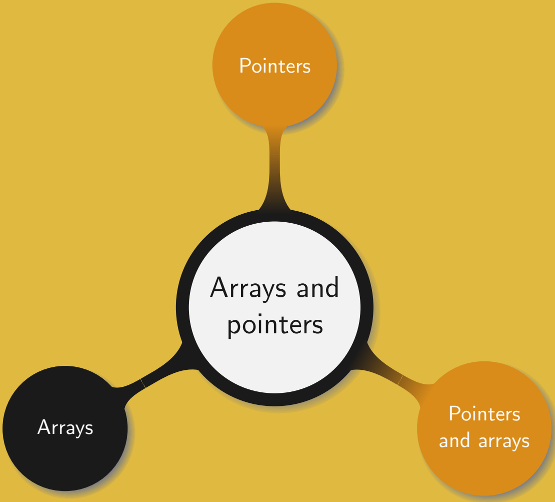
```
for(i=0;i<10;i++) {
  scanf("%d",&n);
  if(n==0) break;
  else if(n>=10) continue;
  printf("%d\n", n);
}
```

Key points

- What are the main data types in C?
- How to perform type casting?
- How to define and use structures on C?
- How to use conditional statements in C?
- How to write loops in C?

7. Arrays and pointers

Chapter organisation



In C an array is defined by three parameters:

- A name
- The data type of its elements
- A size, i.e. the number of elements compositing it

Example.

```
int a[4]={1,2,3,4};
```

Simple manipulations:

- Set the first element to 0
- Add 1 to the second element
- Set the third element to the sum of the third and fourth
- Display all the elements

```
a[0]=0;
a[1]++;
a[2]+=a[3];
for (i=0; i<4;i++)
  printf("%d\n",a[i]);
```

array_fct.c

```c
#include <stdio.h>
double average(int arr[], const size_t size);
int main () {
  int elem[5]={1000, 2, 3, 17, 50};
  printf("%lf\n",average(elem,5));
}
double average(int arr[], const size_t size) {
  unsigned long i;
  double avg, sum=0;
  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }
  avg = sum / size;
  return avg;
}
```

Understanding the code:

- Why is the prototype of the function `average` mentioned before the `main` function?
- How to pass an array to a function?
- Is the size of an array automatically passed to a function?
- When passing an array to a function how to ensure the function knows its size?
- What is `size_t`?

From one die to two dice

Understand the following code and adapt it to handle two dice

die.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIDES 6
#define ROLLS 1000
int main () {
  int i, res[SIDES];
  srand(time(NULL));
  for (i=0; i < SIDES; i++) res[i]=0;
  for (i=0; i < ROLLS; i++) res[rand()%SIDES]++;
  for (i=0; i < SIDES; i++) printf("%d (%d)\t",i+1,res[i]);
  printf("\n");
}
```

In the previous code, how is the array initialized?

More dice

dice.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define DICE 4
#define SIDES 10
#define ROLLS 100000
int main () {
  int i, j, t, res[DICE*SIDES-DICE+1]={0};
  srand(time(NULL));
  for (i=0; i < ROLLS; i++) {
    t=0;
    for(j=0;j<DICE;j++) t+=rand()%SIDES;
    res[t]++;
  }
  for (i=0;i<DICE*SIDES-DICE+1;i++) {
    printf("%d (%d)  ",i+DICE,res[i]);
  }
  printf("\n");
}
```

Understanding the code:

- How is the array initialized?
- What is `DICE*SIDES-DICE+1`?
- Why are all the elements of the table `res` initialized to 0?
- What is the variable `t` storing?

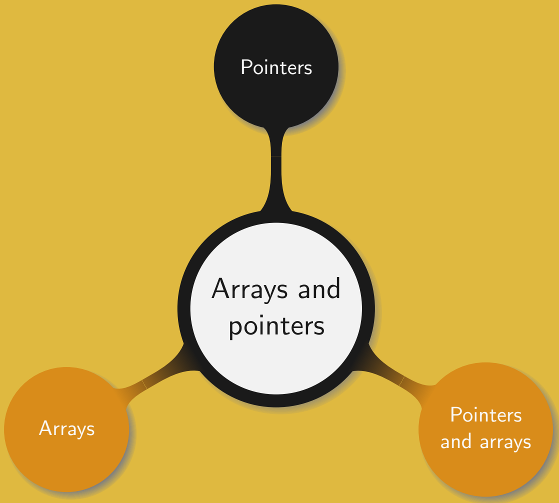
dice_m.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DICE 10
#define SIDES 6
#define ROLLS 100000
int main () {
  int i, j, t, table[DICE][ROLLS], res[DICE*SIDES-DICE+1];
  srand(time(NULL)); memset(res, 0,(DICE*SIDES-DICE+1)*sizeof(int));
  for(i=0;i<DICE;i++)
    for (j=0; j < ROLLS; j++)  table[i][j]=(rand()%SIDES)+1;
  for (i=0;i<ROLLS;i++) {
    t=0;
    for(j=0;j<DICE;j++) t+=table[j][i];
    res[t-DICE]++;
  }
  for (i=0;i<DICE*SIDES-DICE+1;i++) printf("%d (%d) ",i+DICE,res[i]);
  printf("\n");
}
```

In the previous three short programs:

- What three ways were used to initialize the arrays?
- Why is $i + 1$ in the first program and then $i + DICE$ in the two others printed, instead of i?
- In the multidimensional array program, is the order of the loops important? That is loop over DICE and then ROLLS vs. loop over ROLLS and then DICE.
- Rewrite the previous code (7.183) using a function taking dice, sides, and rolls as input
- Explain how multi-dimensional arrays are stored in the memory

Chapter organisation

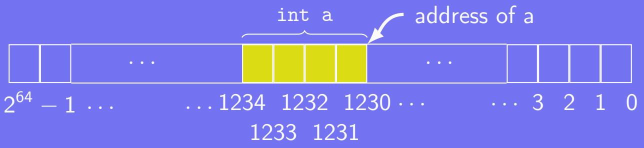


Pointer:

- Something that directs, indicates, or points
- Low level but powerful facility available in C

Pointer vs. variable:

- *Variable:* area of the memory that has been given a name
- *Pointer:* variable that stores the address of another variable



A pointer points to a variable, it is the address of the variable

Handling pointers:

- The *address* of a variable `x` is `&x`
- The value stored at address `y` is `*y`
- The operator "`*`" is called *dereferencing* operator

Type of a pointer:

- A pointer is an address represented as a `long long int`
- It is easy to define a pointer of pointer
- The type of the variable stored at an address must be provided
- Defining a pointer: `type* variable;`

Why using pointers?

swap.c

```
#include <stdio.h>
void swap(int a,int b);
int main() {
  int a=2, b=5;
  swap(a,b);
  printf("a = %d, ",a);
  printf("b = %d\n",b);
  return 0;
}
void swap(int a,int b) {
  int temp=a;
  a=b;
  b=temp;
}
```

swap_ptr.c

```
#include <stdio.h>
void swap(int *a, int *b);
int main() {
  int a=2, b=5;
  swap(&a,&b);
  printf("a = %d, ",a);
  printf("b = %d\n",b);
  return 0;
}
void swap(int* a,int* b) {
  int temp=*a;
  *a=*b;
  *b=temp;
}
```

Understanding the code:

- What is the difference between the two programs?
- Which one returns the proper result?
- Why is one of the programs not working?
- Why is the other program working?
- Why were pointers used in the second program?

Example

ptr.c

```c
#include <stdio.h>
void pointers();
int main() {pointers();}
void pointers() {
  float x=0.5; float *xp1;
  float **xp2 = &xp1; xp1 = &x;
  printf("%llu %p\n%f ",xp1,&x,**xp2);
  x=**xp2+**xp1; printf("%f\n",x);
}
```

Understanding the code:

- Without running the program guess the final value of x
- Alter the program to display `*xp2`
- Explain the result

Functions to manage memory:

- Allocate n bytes of memory, and get a pointer on the first chunk: `malloc(n)`
- Allocate n blocks of size `s` each, set the memory to 0, and get a pointer on the first chunk: `calloc(n,s)`
- Adjust the size of the memory block pointed to by `ptr` to `s` bytes, and get a pointer on the first chunk: `realloc(ptr,s)`
- Frees the memory space pointed to by `ptr`: `free(ptr)`

Any allocated memory must be released

Example.

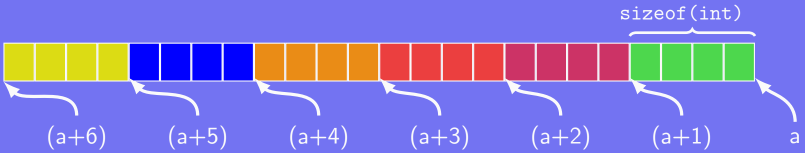
```c
int *a=malloc(6*sizeof(int));^^I
```

- Accessing first chunk

```c
printf("%d",*a);
```

- Accessing the 5th chunk

```c
printf("%d",*(a+4));
```



In this example what is `(a+6)`?

struct_ptr.c

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct person {
  char* name; int age;
} person_t;
int main () {
  struct person ya = {
    .name="Yann",
    .age=12,
  };
  person_t* group=malloc(3*sizeof(person_t));
  group->name="gilbert"; group->age=34;
  *(group+1)=(person_t){"joseph",28};
  (*(group+2)).name="emily"; (group+2)->age=42;
  printf("%s %d %lu\n",ya.name, ya.age,sizeof(struct person));
  printf("%s %d\n",(group+1)->name, (group+2)->age);
  free(group); return 0;
}
```

Understanding the code:

- How to use `malloc`?
- What are the different ways to access elements of a structure when the variable is not a pointer?
- What are the different ways to access elements of a structure when the variable is a pointer?
- Why should the pointer be freed at the end of the program?

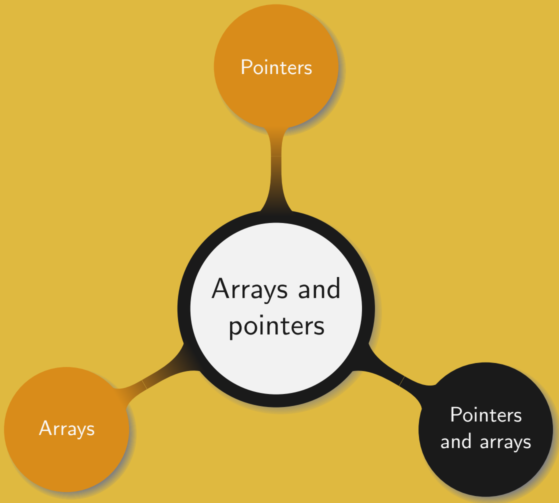
Remarks on pointers:

- Not possible to choose the address, e.g. `int *p; p=12345;`
- The `NULL` pointer "points nowhere"
- An uninitialized pointer "points anywhere", e.g. `float *a;`

A good practice consists in checking the memory allocation:

```
char* p = malloc(100);
if (p == NULL) {
  fprintf(stderr, "Error: out of memory");
  exit(1);
}
```

Chapter organisation



An array contains *elements and a pointer* points *to them*

arr_ptr.c

```c
#include <stdio.h>
#include <stdlib.h>
void ptr_vs_arr();
int main () {
  ptr_vs_arr();
}
void ptr_vs_arr(){
  int a[3]={0,1,2};
  int* p=malloc(3*sizeof(int));  if(p==NULL) exit(-1);
  *p=3; *(p+1)=4; *(p+2)=5; printf("%d %d\n",a[0], *p);
  a[0]=42;  p=a; p++ ; *p=a[2];
//a=p; p=a[0]; p=&a; a++;
  printf("%d %d %lu %lu\n",a[0], *p,sizeof(a), sizeof(p));
//free(p);
}
```

A pointer to `char` is different from an array of `char`

string_ptr.c

```c
#include <stdio.h>
#include<stdlib.h>
void str_ptr();
int main () {
  str_ptr();
}
void str_ptr(){
  char a[]="good morning!";
  char* p="Good morning!";
  printf("%c %c\n",a[0], *p);
  a[0]='t'; //*p='t';
  p=a;//a=p; p=a[0]; p=&a;
  p++; //a++;
  printf("%c %c %lu %lu\n",a[0], *p,sizeof(a), sizeof(p));
//free(p);
}
```

Exercise. Create an array `a` containing the four elements 1, 2, 3, and 4, then print `&a[i]`, `(a+i)`, `a[i]`, and `*(a+i)`

arr_ptr2.c

```c
#include <stdio.h>
void arr_as_ptr(){
  int a[4]={1, 2, 3, 4};
  for(int i=0;i<4;i++) {
    printf("&a[%d]=%p (a+%d)=%p\n"\
        "a[%d]=%d *(a+%d)=%d\n",\
        i,&a[i],i,(a+i),i,a[i],i,*(a+i));
  }
}
int main () {arr_as_ptr();}
```

In the three previous programs:

- List what can be done with a pointer but not with an array
- List what can be done with an array but not with a pointer
- Is it possible to read a pointer as an array?
- Is it possible to read an array as a pointer?
- What is the size of a pointer, why?
- Can a `char*` be changed?

Revisiting the dice

dice_mp.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void roll_dice(int dice, int sides, int rolls){
  int i, j, t;
  int *res=calloc((dice*sides-dice+1),sizeof(int));
  int *table=malloc(dice*rolls*sizeof(int));
  for(i=0;i<rolls;i++) {
    for (j=0; j < dice; j++) table[i*dice+j]=(rand()%sides)+1;
  }
  for (i=0;i<rolls;i++) {
    t=0; for(j=0;j<dice;j++) t+=table[i*dice+j]; res[t-dice]++;
  }
  for (i=0;i<dice*sides-dice+1;i++) printf("%d (%d) ",i+dice,res[i]);
  printf("\n"); free(table); free(res);
}
int main () {
  int dice=10, sides=6, rolls=1000000;
  srand(time(NULL)); roll_dice(dice,sides,rolls);
}
```

Understanding the code:

- How is the array `table` handled?
- What happened in the previous version with 1000000 rolls?
- Is the same happening now, why?
- How is the program organised?
- How are `malloc` and `calloc` used?

Limitation of C:

- No limit on the number of input
- Only one output
- Output cannot be an array

Use pointers as input (slide 7.188)

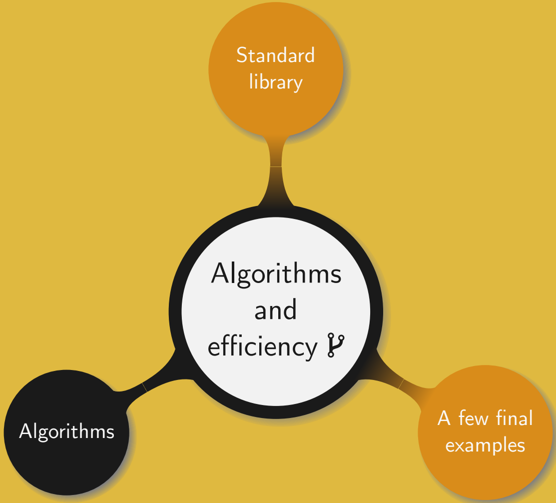
Common mistakes leading to `segmentation fault`:

- Memory has not been allocated
- Memory has been freed too early
- Memory is freed twice or more times
- Memory is accessed but does not belong to the program

Key points

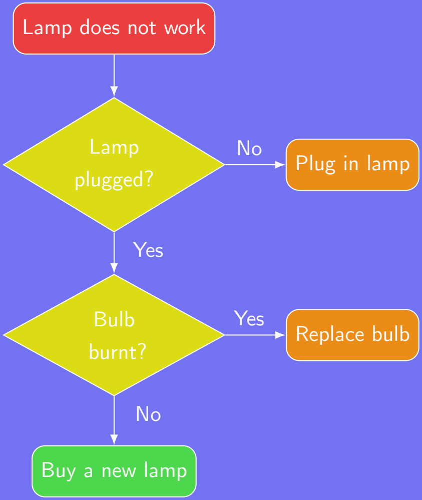
- What are the three information necessary to define an array?
- What are `&a` and `*a`?
- Given a pointer on a structure how to access a specific field?
- Are pointers and array the same?
- What to do with unused allocated memory?
- How to have more than one output in a function?

8. Algorithms and efficiency


Standard library
Algorithms and efficiency
Algorithms
A few final examples

Reminders:

- Algorithms are like recipes for computers
- An algorithm has three main components:
  - Input
  - Output
  - Instructions
- Clear algorithms are often easy to implement
- Algorithms should be adjusted to fit the language
- Algorithms can often be represented as a flowchart



Most common types of algorithms:

- Brute force: often obvious, rarely best
- Divide and conquer: often recursive
- Search and enumeration: model problem using a graph
- Randomized algorithms: feature random choices
  - Monte Carlo algorithms: return the correct answer with high probability
  - Las Vegas algorithms: always correct answer but feature random running times
- Complexity reduction: rewrite a problem into an easier one

Efficiency

When writing a program:

- How efficient does the program need to be?
- What language to choose?
- Is it possible to optimize the code?
- What size are the Input?
- Is it worth implementing a more complex algorithm?

Computational complexity:

- Evaluates how hard it is to solve a problem
- Independent of the implementation
- Considers the behavior at the infinity
- Both time and space complexity can be considered

Scanf !!!!!!

Double Float around 0 — EPS

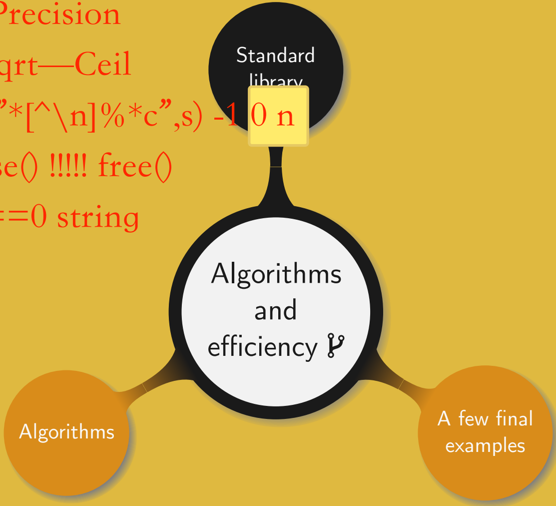
Precision

Sqrt—Ceil

fscanf(myin,"*[^\n]%*c",s) -1 0 n

fclose() !!!!! free()

I==0 string



Moving in a file:

- Open a file:
  - `FILE *fopen(const char *path, const char *mode)`
  - `mode` is one of `r`, `r+`, `w`, `w+`, `a`, `a+`
  - `NULL` returned on an error
- Close a file:
  - `int fclose(FILE *fp)`
  - `0` returned on success
- Seek in a file:
  - `int fseek(FILE *stream, long offset, int whence)`
  - `whence` is one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`
- Back to the beginning: `void rewind(FILE *stream)`

Reading and writing:

- Write in stream:
  `int fprintf(FILE *stream, const char *format, ...);`
- Write in string:
  `int sprintf(char *str, const char *format, ...);`
- Flush a stream: `int fflush(FILE *stream);`
- Read `size-1` characters from a stream:
  `char *fgets(char *s, int size, FILE *stream);`
- Read next character from `stream` and cast it to an `int`:
  `int getc(FILE *stream);`

Strings:

- Length of a string: `size_t strlen(const char *s)`
- Copy a string:
  `char *strcpy(char *dest, const char *src)`
- Copy at most *n* bytes of *src*:
  `char *strncpy(char *dest, const char *src, size_t n)`
- Compare two strings:
  - `int strcmp(const char *s1, const char *s2)`
  - Returned value is $< 0,\ 0,\ > 0$, if $s1 < s2,\ s1 = s2,\ s1 > s2$
- Compare the first *n* bytes of two strings:
  `int strncmp(const char *s1, const char *s2, size_t n);`
- Locate a character is a string:
  `char *strchr(const char *s, int c);`

Accessing memory:

- Fill memory with a constant byte:
  `void *memset(void *s, int c, size_t n);`
- Copy memory area, overlap allowed:
  `void *memmove(void *dest, const void *src, size_t n);`
- Copy memory area, overlap not allowed:
  `void *memcpy(void *dest, const void *src, size_t n);`

Useful functions for simple benchmarking:

- Getting time: `time_t time(time_t *t);`
- Calculate a time difference:
  `double difftime(time_t time1, time_t time0);`

Classifying elements:

- `int isalnum(int c);`
- `int isalpha(int c);`
- `int isspace(int c);`
- `int isdigit(int c);`
- `int islower(int c);`
- `int isupper(int c);`

Converting to uppercase or lowercase:

- `int toupper(int c);`
- `int tolower(int c);`

Common mathematical functions with `double` input and output:

- Trigonometry: `sin(x)`, `cos(x)`, `tan(x)`
- Exponential and logarithm: `exp(x)`, `log(x)`, `log2(x)`, `log10(x)`
- Power and square root: `pow(x,y)`, `sqrt(x)`
- Rounding: `ceil(x)`, `floor(x)`

Mathematics:

- Absolute value: `int abs(int j);`
- Quotient and remainder:
  - `div_t div(int num, int denom);`
  - `div_t`: structure containing two `int`, `quot` and `rem`

Pointers:

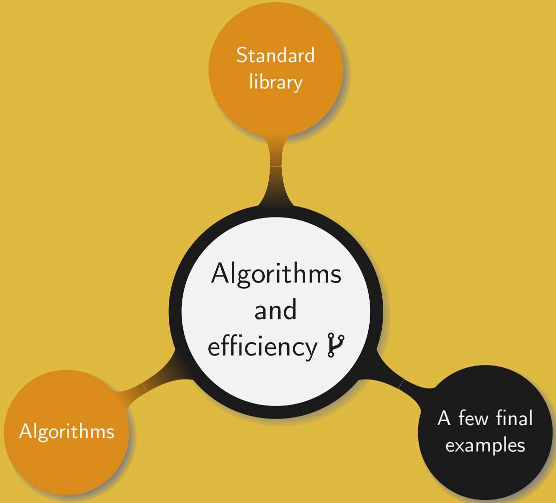
- `void *malloc(size_t size);`
- `void *calloc(size_t nobj, size_t size);`
- `void *realloc(void *p, size_t size);`
- `void free(void *ptr);`

Strings:

- String to integer: `int atoi(const char *s);`
- String to long:
```
long int strtol(const char *nptr, char **endptr,
  int base);
```

Misc:

- Execute a system command: `int system(const char *cmd);`
- Sorting:
```
void qsort(void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *));
```
- Searching:
```
void *bsearch(const void *key, const void *base, size_t nmemb,
  size_t size, int (*compar)(const void *, const void *));
```


Standard library
Algorithms and efficiency
Algorithms
A few final examples

linear_search.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
#define MAX 1000
int main () {
  int i, n, k=0;
  int data[SIZE];
  srand(time(NULL));
  for(i=0; i<SIZE; i++) data[i]=rand()%MAX;
  n=rand()%MAX;
  for(i=0; i<SIZE; i++) {
    if(data[i]==n) {
      printf("%d found at position %d\n",n,i);
      k++;
    }
  }
  if(k==0) printf("%d not found\n",n);
}
```

Adapt the previous code to:

- Read the data from a text file
- Read the value `n` for the standard input
- Exit the program when the first match is found
- Use pointers and dynamic memory allocation instead of arrays

binary_search.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
int main () {
  int i, n, k=0, low=0, high=SIZE-1, mid;
  int *data=malloc(SIZE*sizeof(int));
  srand(time(NULL));
  for(i=0;i<SIZE;i++) *(data+i)=2*i;
  n=rand()%*(data+i-1);
  while(high >= low) {
    mid=(low + high)/2;
    if(n < *(data+mid)) high = mid - 1;
    else if(n> *(data+mid)) low = mid + 1;
    else {printf("%d found at position %d\n",n,mid);
      free(data);  exit(0);}
  }
  printf("%d not found\n",n);
  free(data);
}
```

Using the previous code:

- Write a clear algorithm for binary search
- For a binary search to return a correct result what extra condition should be added on the data?
- Compare the efficiency of a binary search to a linear search; that is on the same data set compare the execution time of the two programs
- Adapt the previous code to use arrays instead of pointers

selection_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 200
#define MAX 1000
int main () {
  int data[SIZE];
  srand(time(NULL));
  for(int i=0; i<SIZE; i++) data[i]=rand()%MAX;
  for(int i=0; i<SIZE; i++) {
    int t, min = i;
    for(int j=i; j<SIZE; j++) if(data[min]>data[j]) min = j;
    t = data[i];
    data[i] = data[min];
    data[min] = t;
  }
  printf("Sorted array: ");
  for(int i=0; i<SIZE; i++) printf("%d ",data[i]);
  printf("\n");
}
```

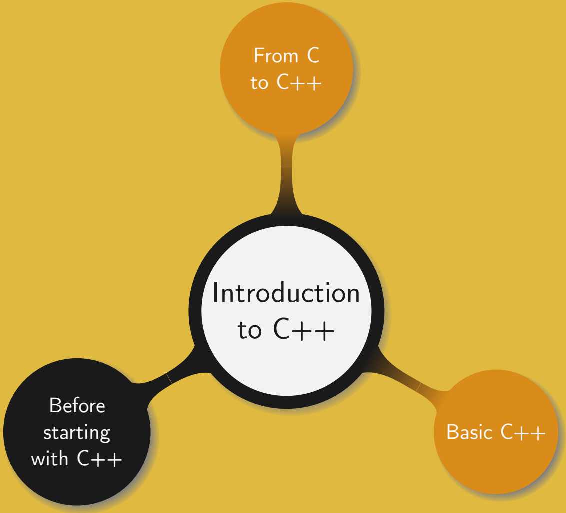
Understanding the code:

- From the previous code write a clear algorithm describing selection sorting
- How efficient is the selection sort algorithm?
- In the previous program what is the scope of the variables?
- Rewrite the previous code into an independent function
- Generate some unsorted random data and write it in a file; then read the file, sort the data and use a binary search to find a value input by the user

Key points

- Is the most important, the algorithm or the code?
- Cite two types of algorithms
- How is efficiency measured?
- Where to find C functions?

9. Introduction to C++


From C
to C++
Introduction
to C++
Before
starting
with C++
Basic C++

The birth of C++

Background information:

- Author: Bjarne Stroustrup
- Motivation: other languages are either too low level or too slow

Timeline:

- 1979: C with classes
- 1983: name changed for C++
- 1985: first commercial implementation of C++
- 1989: updated version, C++2.0
- 2011: new version, C++11, enlarged standard library
- 2014: C++14, bug fixes, minor improvements

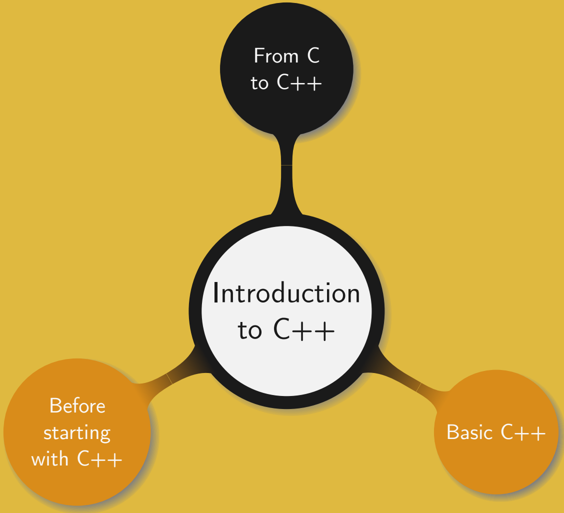
Simple description:

- Compiled programming language
- General-purpose programming language
- Intermediate level language
- Object-oriented programming language

Highlights:

- Higher level than C, but still performant
- Code often shorter and cleaner than in C
- Safer: more errors caught at compile time
- No runtime overhead

Chapter organisation



What C++ brings:

- Almost all the aspects of C are preserved
- New features are added
- Sophisticated programs are easier to code
- C++ is almost a superset of C

Is this program written in C or C++?

prg.cpp

```
#include <stdio.h>

int main () {
  int a=5;
  printf("%d\n",a);
}
```

Why easier?

A new approach:

- Easier to manage memory
- New features for generic programming
- Object oriented programming:
  - Variables are defined in term of objects
  - Objects are close from human thinking
  - An object is similar to a structure in C with more "abilities"

Programmers focus on the problem not on how to explain it

C++ syntax is similar to C's:

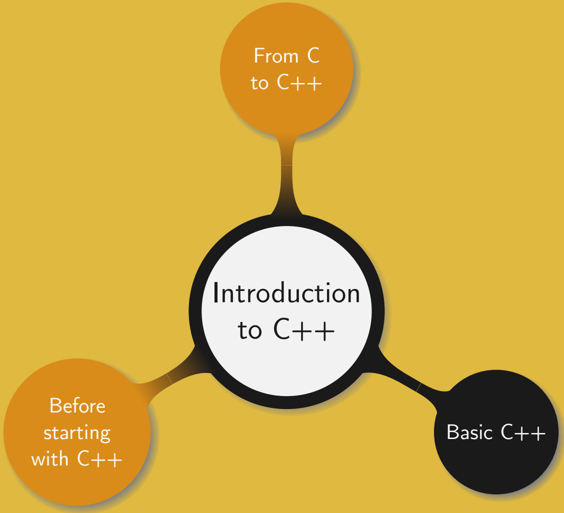
- Function declaration
- Blocks
- For loop
- While loop
- If statement
- Switch statement
- Shorthand operators
- Logical operators
- Short-circuit operators
- Conditional ternary operator

Typecasting from `void` is implicit in C and explicit in C++:

```
int *x = \
malloc(sizeof(int)*10);
```

```
int *x = \
(int *)  malloc(sizeof(int)*10);
```

Chapter organisation



New data type and headers

New in C++:

- New datatype:

```
bool a=true, b=false;
```

- New headers:

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

Namespace:

- C: function names conflicts among different libraries
- C++: introduction of *namespace*
- Each library or program has its own namespace
- Namespace for the standard library: `std`

Handling input-output (IO) without `printf` and `scanf`:

- Input: `cin >> x`
- Output: `cout << "String"`

Example.

input_pb.cpp

```cpp
#include <iostream>
using namespace std;
void TestInput(){
  int x = 0;
  do {
    cout << "Enter a number (-1 to quit): "; cin >> x;
    if(x != -1) cout << x << " was entered" << endl;
  }  while(x != -1);
  cout << "Exit" << endl;
}
int main() {TestInput(); return 0;}
```

Problem with the previous code: input a letter... and exit

input_ok1.cpp

```
#include <iostream>
using namespace std;
void TestInput(){
  int x = 0;
  do {
    cout << "Enter a number (-1 to quit): ";
    if(!(cin >> x)) {
      cout << "The input stream broke!" << endl;
      x = -1;
    }
    if(x != -1) cout << x << " was entered" << endl;
  } while(x != -1);
  cout << "Exit" << endl;
}
int main() {TestInput(); return 0;}
```

Problem with the previous code: the program exits “unexpectedly”

input_ok2.cpp

```cpp
#include <iostream>
using namespace std;
void TestInput(){
  int x=0;
  do {
    cout << "Enter a number (-1 to quit): ";
    if(!(cin >> x)) {
      cin.clear(); cin.ignore(10000,'\n');
      cout << "Wrong input, try again.\n";
    }
    else {
      if(x != -1) cout << x << " was entered" << endl;
    }
  }  while(x != -1);
  cout << "Exit" << endl;
}
int main() {TestInput(); return 0;}
```

Nicer display:

- Alignment: `setiosflags(ios::left)`
- Width: `setw(width)`
- Prefix: `setfill(z)`
- Precision: `setprecision(2)`

Example.

date.cpp

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
void showDate(int m, int d, int y) {
  cout.fill('0');
  cout << setw(2) << m << '/' << setw(2) << d << '/' << setw(4) << y << endl;
}
int main(){
  showDate(6,19,2020);
  cout << setprecision(3) << 1.2249 << endl;
  cout << setprecision(3) << 1.2259 << endl;
}
```

Note on the operators:

- What are << and >> in C?
- What about `cin >> x` or `cout << x`?
- An operator can be reused with a different meaning

Similar concept: function overloading

fo.cpp

```cpp
#include <iostream>
using namespace std;
double f(double a);
int f(int a);
int main () {cout << f(2) << endl; cout << f(2.3) << endl;}
double f(double a) {return a;}
int f(int a) {return a;}
```

No more `malloc`, `calloc` and `free`:

- Memory for a variable: `int *p = new int;`
- Memory for an array: `int *p = new int[10];`
- Array size can be a variable (not recommended in C)
- Return `NULL` on failure
- Release the memory: `delete p` or `delete[] p`

Any allocated memory must be released

Improvements on strings:

- Strings in C: array of characters
- Many limitations, low level manipulations
- New type in C++: `string`

```cpp
#include <string>
string g="good "; string m="morning";
cout << g + m + "!\n";
```

Search and learn more on how to use strings in C++

Requires header: `#include <fstream>`

- Open file for reading: `ifstream in("file.txt")`
- Read from a file: `in` used in the same way as `cin`
- Open a file for writing: `ofstream out("file.txt")`
- Write in a file: `out` used in the same way as `cout`
- Read from a file, line by line: `getline(in,s)`

Exercise. Copy the content of a text file into another text file and display each line on the console output

fio.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void FileIO() {
  string s;
  ifstream a("11.txt"); ofstream b("2.txt");
  while(getline(a,s)) {b << s << endl; cout << s;}
}
int main () {FileIO();return 0;}
```

What was wrong with the previous code?

fio_c.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void FileIO(){
  string s;
  ifstream a("1.txt"); ofstream b("2.txt",ios::app);
  if (a.is_open() && b.is_open()) {
    while(getline(a,s)) {b << s << endl; cout << s;}
    b.close(); a.close();
  }
  else cerr << "Unable to open the file(s)\n";
}
int main () {FileIO(); return 0;}
```

Constants in C style:

- Syntax: `#define PI 3.14`
- Handled early in compilation
- No record of `PI` at compile time

Constants in C++ style:

- New syntax: `static const float PI=3.14;`
- `PI` is a constant, value cannot be changed
- `PI` is known by the compiler, present in the symbol table
- Type safe

Inline functions

Short and often called functions in C:

- Macros
- Macros expanded early in the compilation
- Hard to debug
- Sides effect with complex macros

Short and often called functions in C++:

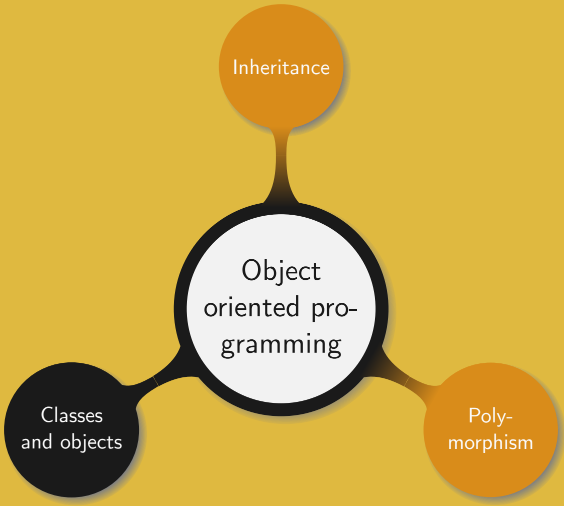
- Inline functions
- Treated by the compiler
- Similar as a regular function
- Does not call the function but write a copy of it instead
- Increase the size of the program

```cpp
inline int min(int x, int y) { return x <= y ? x : y; }
```

Key points

- What is the difference between C and C++?
- Cite a few novelties
- How to handle input-output?
- How to handle pointers?
- What are operator and function overloading?

10. Object oriented programming


Inheritance
Object oriented programming
Classes and objects
Polymorphism

Programming approach used so far:

- Program written as a sequence of procedures
- Each procedure fulfills a specific task
- All tasks together compose a whole project
- Further from human thinking
- Requires higher abstraction

A new approach:

- Everything is an object
- Objects communicate between them by sending messages
- Each object has its own type
- Object of a same type can receive the same message

Object

An object has two main components:

- Its behavior, what can be done with it, its *methods*
- The data it contains, what it knows, its *attributes*

Example. Given a simple TV:

- Methods:
  - High level actions, e.g. on-off, channel, volume
  - Low level actions, e.g. on internal electronics components
- Attributes:
  - High level elements:, e.g. button on the remote control
  - Low level elements, e.g. internal electronics components

Class:

- Defines the family, type or nature of an object
- Equivalent of the type in “traditional programming”

Instance:

- Realisation of an object from a given class
- Equivalent of a variable in “traditional programming”

Example. Two same TV models can be represented as two instances of one class

Oder of definition:

1. Define the methods
2. Define the attributes

Example. Create an object `circle`:

1. What it can do, i.e. the methods:
   - `move`
   - `zoom`
   - `area`
2. What is needed to achieve it, i.e. its attributes:
   - Position of the center (x, y)
   - Radius of the circle

The interface of a class:

- Is equivalent to `header.h` file in C
- Contains the description of the object
- Splits into two main parts
  - Public definition of the class: user methods
  - Private attributes and methods: not accessible to the user but necessary to the "good functioning"

Example. In the case of a TV:

- Public methods: on/off, change channel, change volume
- Public attributes: remote control and buttons
- Private methods: actions on the internal components
- Private attributes: internal electronics

A note on visibility

Private or public:

- Private members can only be accessed by member functions within the class
- Users can only access public members

Benefits:

- Internal implementation can be easily adjusted without affecting the user's code
- Accessing private attributes is forbidden: more secure

Only render a member public when necessary

Example.

circle_v0.h

```
class Circle {
/* user methods (and attributes)*/
  public:
    void move(float dx, float dy);
    void zoom(float scale);
    float area();
/* implementation attributes (and methods) */
  private:
    float x, y, r;
};
```

Understanding the code:

- What is defined as private and public?
- If the circle does not move, what attributes are necessary?

Using the created objects:

- Include the class using the header file
- Declare one or more instances
- Classes similar to structures in C:
  - Structure only contains attributes
  - Class also contains methods
- Calling a method on an object: `instance.method(...)`

Example.

circle_main_v0.cpp

```cpp
#include <iostream>
#include "circle_v0.h"
using namespace std;
int main () {
  float s1, s2;
  Circle circ1, circ2;
  circ1.move(12,0);
  s1=circ1.area(); s2=circ2.area();
  cout << "area: " << s1 << endl;
  cout << "area: " << s2 << endl;
  circ1.zoom(2.5); s1=circ1.area();
  cout << "area: " << s1 << endl;
}
```

Understanding the code: why is this program not compiling?

Getting things ready:

- Class interface is ready
- Instantiation is possible
- Does not compile: no implementation of the class yet
- Syntax: `classname::methodname`

Example.

circle_v0.cpp

```cpp
#include "circle_v0.h"
static const float PI=3.1415926535;
void Circle::move(float dx, float dy) {
  x += dx;
  y += dy;
}
void Circle::zoom(float scale) {
  r *= scale;
}
float Circle::area() {
  return PI * r * r;
}
```

Understanding the code: can this file be compiled alone?

Automatic construction and destruction of objects:

- Object not initialised by default (same as `int i`)
- Constructor: method that initialises an instance of an object
- Used for a proper default initialisation
- Definition: no type, name must be `classname`
- Important note: can have more than one constructor
- Destructor: called just before the object is destroyed
- Used for clean up (e.g. release memory, close a file etc...)
- Definition: no type, name must be `~classname`

Example.

circle_v1.h

```
class Circle {
/* user methods (and attributes)*/
  public:
    Circle();
    Circle(float r);
    ~Circle();
    void move(float dx, float dy);
    void zoom(float scale);
    float area();
/* implementation attributes (and methods) */
  private:
    float x, y;
    float r;
};
```

Class implementation

circle_v1.cpp

```cpp
#include "circle_v1.h"
static const float PI=3.1415926535;
Circle::Circle() {
  x=y=0.0; r=1.0;
}
Circle::Circle(float radius) {
  x=y=0.0; r=radius;
}
Circle::~Circle() {}
void Circle::move(float dx, float dy) {
  x += dx; y += dy;
}
void Circle::zoom(float scale) {
  r *= scale;
}
float Circle::area() {
  return PI * r * r;
}
```

Class instantiation

circle_main_v1.cpp

```
#include <iostream>
#include "circle_v1.h"
using namespace std;
int main () {
  float s1, s2;
  Circle circ1, circ2((float)3.1);
  circ1.move(12,0);
  s1=circ1.area(); s2=circ2.area();
  cout << "area: " << s1 << endl;
  cout << "area: " << s2 << endl;
  circ1.zoom(2.5);
//  cout << circ1.r <<endl;
  s1=circ1.area();
  cout << "area: " << s1 << endl;
}
```

Better definitions:

- Two constructor defined: `circle()` and `circle(float)`
- Proper one automatically selected

Another strategy is to set a default value in the specification.

```
Circle(float radius=1.0);
```

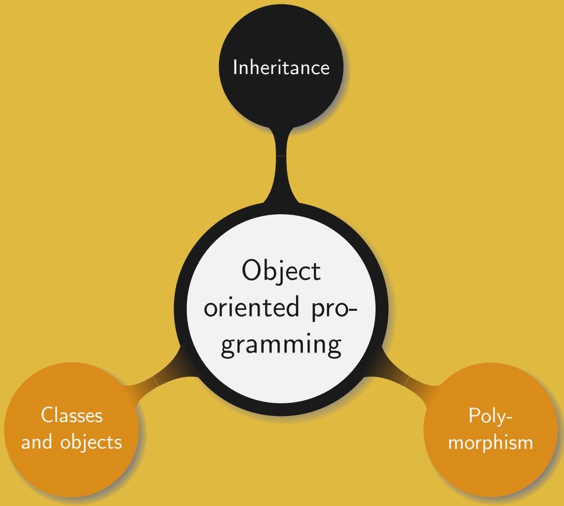
Example. A 2D geometry library is updated to support 3D. As a result the function `move` now takes three arguments: dx, dy, dz. For the old instantiations to remain valid adjust the interface (header file).

```
move(float dx, float dy, float dz=0.0);
```

Exercise. Write a new main file with two pointers: one for the two circles and one for their areas. The `main` function should not do any real work.

main_ptr.cpp

```cpp
#include <iostream>
#include "circle_v1.h"
using namespace std;
void FctCirc(Circle *circ, float *s) {
  *(circ+1)=Circle(3.1);
  *s=circ->area(); s[1]=circ[1].area();
  cout << "area: " << s[0] << endl;
  cout << "area: " << *(s+1) << endl;
  circ[0].zoom(2.5); *s=circ->area();
  cout << "area: " << s[0] << endl;
}
int main () {
  float *s=new float[2]; Circle *circ; circ=new Circle[2];
  FctCirc(circ,s);
  delete[] s; delete[] circ; return 0;
}
```


Inheritance
Object oriented programming
Classes and objects
Poly-morphism

Advantages of classes

Benefits of classes:

- Object are not too abstract
- Closer from the human point of view
- Methods only applied to object which can accept them
- Things are organised in a simple and clear way

Lets construct a zoo and work with cows...

cows_0.cpp

```cpp
#include <iostream>
using namespace std;
class Cow {
  public:
    void Speak () { cout << "Moo.\n"; }
    void Eat() {
      if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
      else cout << "I'm hungry\n";}
    Cow(int f=0){grass=f;}
  private: int grass;
};
int main () {
  Cow c1(1);
  c1.Speak(); c1.Eat(); c1.Eat();
}
```

Petting a sick cow

A sick cow does:

- Everything a cow does
- Take its medication

Two obvious strategies:

- Add a `TakeMediaction()` method to the cow
- Recopy the `cow` class, rename it and add `TakeMedication()`

What are the limitations of those strategies?

The solution consists in getting a sick cow to *inherits* the attributes and methods of a cow, while allowing it to add some more cows_1.cpp

```
#include <iostream>
using namespace std;
class Cow {
  public: Cow(int f=0){grass=f;}
    void Speak () { cout << "Moo.\n"; }
    void Eat() {
      if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
      else cout << "I'm hungry\n";}
  private: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0){grass=f; med=m;}
    void TakeMed() {
      if(med > 0) { med--; cout << "I feel better\n";}
      else cout << "I'm dying\n";}
  private: int med;
};
int main () {
  Cow c1(1); SickCow c2(1,1);
  c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
}
```

Private

Reminder on private members:

- Everything private is only available to the current class
- Derived classes cannot access or use them

Private inheritance:

- Default type of class inheritance
- Any public member from the base class becomes private
- Allows to hide “low level” details to other classes

Public

Reminder on public members:

- They are available to the current class
- They are available to any other class

Public inheritance:

- Anything public in the base class remains public
- Nothing private in the base class can be accessed

Problem:

- Private is too restrictive while public is too open
- Need a way to only allow derived classes and not others

Protected members:

- Compromise between public and private
- They are available to any derived class
- No other class can access them

Possible to bypass all this security using keyword `friend`:

- Valid for both functions and classes
- A class or function declares who are its friends
- Friends can access protected and private members

Never use `friend`

Attributes and methods:

Visibility	Classes		
	Base	Derived	Others
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Inheritance:

Base class	Derived class		
	Public	Private	Protected
Private	-	-	-
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Properly petting a sick cow

cows_2.cpp

```
#include <iostream>
using namespace std;
class Cow {
  public: Cow(int f=0){grass=f;}
    void Speak () { cout << "Moo.\n"; }
    void Eat() {
      if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
      else cout << "I'm hungry\n";}
  protected: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0){grass=f; med=m;}
    void TakeMed() {
      if(med > 0) { med--; cout << "I feel better\n";}
      else cout << "I'm dying\n";}
  private: int med;
};
int main () {
  Cow c1(1); SickCow c2(1,1);
  c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
}
```

Inheritance or not inheritance?

A cow is a *mammal, while a zoo* has *mammals and reptiles*

```cpp
class Cow : public Mammal {
  ...
}
```

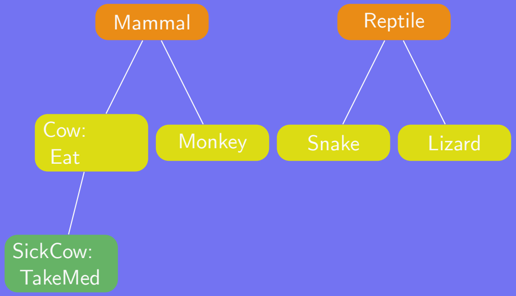
```cpp
class Zoo {
  public:
    Mammal *m; Reptile *r;
    ...
};
```

Remark. On a drawing:

- A cow *is* a figure, a cage is a figure, a zoo is a figure...
- A cow is composed of (*has*) figures, e.g. ellipsis for the body, circle for the head, rectangles for the legs and tail
- What to choose, *is a* or *has a*?

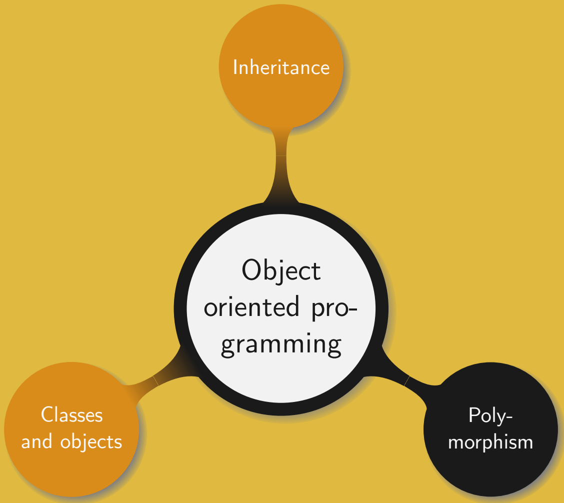
Hierarchy diagram

Representing the relationships using diagrams:



Zoo:
Reptile
Mammal
...

Chapter organisation



Polywhat????

Poly-morphism

poly: many

morphe: form/shape

Simple idea:

- Arrays cannot contain different data types
- A sick cow is *almost like* a cow
- Goal: handle sick cows as cows while preserving their specifics

Function overloading

cows_3.cpp

```cpp
#include <iostream>
using namespace std;
class Cow {
  public: Cow(int f=0){grass=f;}
    void Speak () { cout << "Moo.\n"; }
    void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
                 else cout << "I'm hungry\n";}
  protected: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0){grass=f; med=m;}
    void Speak () { cout << "Ahem... Moo.\n"; }
    void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
                     else cout << "I'm dying\n";}
  private: int med;
};
int main () {
  Cow c1; SickCow c2(1); Cow *c3=&c2;
  c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
}
```

New keyword: `virtual`

- Virtual function in the base class
- Function can be redefined in derived class
- Preserves calling properties

Drawbacks:

- Binding: connecting function call to function body
- Early binding: compilation time
- Late binding: runtime, depending on the type, more expensive
- `virtual` implies late binding

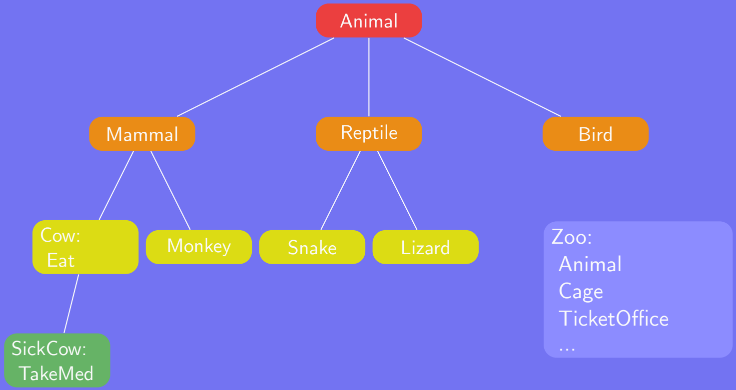
Fixing the cows

cows_4.cpp

```cpp
#include <iostream>
using namespace std;
class Cow {
  public: Cow(int f=0){grass=f;}
    virtual void Speak () { cout << "Moo.\n"; }
    void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
                 else cout << "I'm hungry\n";}
  protected: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0){grass=f; med=m;}
    void Speak () { cout << "Ahem... Moo.\n"; }
    void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
                     else cout << "I'm dying\n";}
  private: int med;
};
int main () {
  Cow c1; SickCow c2(1); Cow *c3=&c2;
  c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
}
```

Extending the idea

Applying the same idea to generalize the diagram:



Benefits:

- Feed all the animals at once
- Animals speak their own language when asked to speak

Pushing it further:

- Write a totally abstract class "at the top"
- This class has virtual member functions without any definition
- The method definition is replaced by =0

Example.

```
class Animal {
  public:
    virtual void Speak() = 0;
}
```

animals.h

```cpp
#ifndef __ANIMALS_H__
#define __ANIMALS_H__
class Animal {
  public:
    virtual void Speak() = 0; virtual void Eat() = 0; virtual ~Animal()=0;
};
class Cow : public Animal {
  public: Cow(int f=0); virtual void Speak(); void Eat();
  protected: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0); void Speak(); void TakeMed();
  private: int med;
};
class Monkey : public Animal {
  public: Monkey(int f=0); void Speak(); void Eat();
  protected: int banana;
};
#endif
```

animals.cpp

```
#include <iostream>
#include "animals.h"
using namespace std;
Animal::~Animal() {}
Cow::Cow(int f) { grass=f; }
void Cow::Speak() { cout << "Moo.\n"; }
void Cow::Eat(){
  if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
  else cout << "I'm hungry\n";
}
SickCow::SickCow(int f,int m) {grass=f; med=m;}
void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
void SickCow::TakeMed() {
  if(med > 0) { med--; cout << "I feel better\n";} else cout << "I'm dying\n";
}
Monkey::Monkey(int f) { banana=f; }
void Monkey::Speak() { cout << "Hoo hoo hoo hoo\n";}
void Monkey::Eat() {
  if(banana > 0) {banana--; cout << "Give me another banana!\n";}
  else cout << "Who took my banana?\n";
}
```

zoo.h

```cpp
#ifndef __ZOO_H__
#define __ZOO_H__
#include <iostream>
#include <string>
#include "animals.h"
using namespace std;
class Employee {
  public: string GetName(); protected: void SetName(string n); private: string name;
};
class Tamer : public Employee {
  public: Tamer(string="John Doe"); void Feed(Animal *a);
};
class Boss : public Employee {
  public: Boss(string="Jane Doe"); Tamer HireTamer(string);
          void command(Employee*, string);
};
class Zoo {
  public: Zoo(int, string); ~Zoo(); int getSize(); Boss* GetBoss();
          Tamer* GetTamer(); void ApplyTamerJob(string); Animal *GetAnimal(int i);
  private: int size; Animal **a; Tamer t; Boss b;
};
#endif
```

employees.cpp

```cpp
#include <iostream>
#include "zoo.h"
void Employee::SetName(string n) { name=n; }
string Employee::GetName() { return name; }

Tamer::Tamer(string n) { SetName(n); }
void Tamer::Feed(Animal *a) { a->Speak(); a->Eat(); }

Boss::Boss(string n) { SetName(n); }
Tamer Boss::HireTamer(string n) {
  Tamer g(n);
  cout << "Welcome to " << n << ", our new tamer.\n"; return g;
}
void Boss::command(Employee* e, string order) {
  cout << e->GetName() << order << endl;
}
```

zoo.cpp

```
#include <iostream>
#include "zoo.h"
Zoo::Zoo(int s,string n) {
  size=s; a=new Animal*[size]; b=Boss(n);
  for(int i=0; i<size; i++) {
    switch(i%4) {
      case 0: a[i]=new Cow; break; case 1: a[i]=new SickCow; break;
      case 2: a[i]=new Monkey;break; case 3: a[i]=new Monkey(1);break;
    }
  }
}
Zoo::~Zoo() {
  for(int i=0; i<size; i++) delete a[i];
  delete[] a;
}
int Zoo::getSize() { return size; }
Tamer* Zoo::GetTamer() { return &t; }
void Zoo::ApplyTamerJob(string n) { t=b.HireTamer(n); }
Boss* Zoo::GetBoss() { return &b; }
Animal *Zoo::GetAnimal(int i) { return a[i]; }
```

zoo_main.cpp

```cpp
#include <iostream>
#include "zoo.h"
int main () {
  Zoo z(10,"Bob");
  cout << "Hi I'm " << z.GetBoss()->GetName() << " the boss. ";
  z.ApplyTamerJob("Mike");
  z.GetBoss()->command(z.GetTamer(),", feed the animals!");
  for(int i=0; i<z.getSize(); i++) {
    cout << endl;
    z.GetTamer()->Feed(z.GetAnimal(i));
  }
}
```

Remark. How many lines of code are necessary to achieve the same result without inheritance and polymorphism?

Understanding the code:

- Why is the `Zoo` destructor not empty?
- Is it possible to instantiate an `Animal`?
- Why does the `Zoo` have an `Animal**` attribute?
- Why is `Animal` featuring a virtual destructor?
- What is happening when an `Animal` is destroyed?
- How to avoid memory leaks?
- How is the `Tamer` hiring process working?
- Why is the `Employee` `SetName()` method protected?

Benefits of polymorphism

Understanding the design of the project:

- Explain the benefits of polymorphism
- Draw the hierarchy diagram for the current employees
- Is it better to view `Boss` as an `Employee` or to say that a `Boss` has `Employee`?
- What other common methods and attributes could be added to `Employee`?
- Is it a good idea to use polymorphism to gather `Employee` and `Visitor` together?

Extending the project:

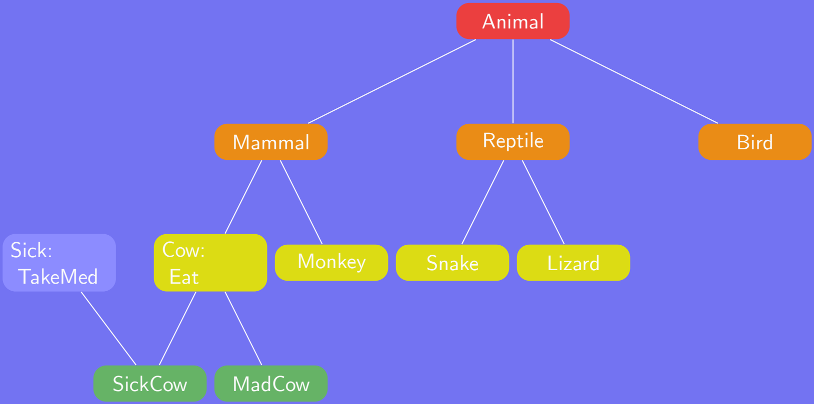
- Add more types of `Employee`, e.g. vet, guards, cashier
- Add various types of visitors, e.g. VIP, disabled
- Add some facilities, e.g. cages that can be locked and unlocked, toilets for men, women, and disabled
- Allow visitors to watch the animals, but not to feed them, i.e. they receive a fine if they do so
- If an animal escapes issue an emergency announcement and close the zoo

With multiple inheritance, a class can inherit from several classes

Example. Any sick animal should be put under medication:

- Not only cows can be sick
- Create a generic "sick class" that can be used by any animal
- A sick cow *is* a cow and *is* sick
- A sick cow inherits from sick and from cow

Multiple inheritance




```cpp
class SickCow : public Cow, public Sick {
  ...
}
```

More cows

animals_m.h

```
class Animal {
  public:
    virtual void Speak() = 0; virtual void Eat() = 0;
};
class Sick {
  public:  void TakeMed();
  protected: int med;
};
class Cow : public Animal {
  public: Cow(int f=0); void Speak(); void Eat();
  protected: int grass;
};
class SickCow : public Cow, public Sick {
  public: SickCow(int f=0,int m=0); void Speak();
};
class MadCow : public Cow {
  public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
  protected: int pills;
};
```

More cows

animals_m.cpp

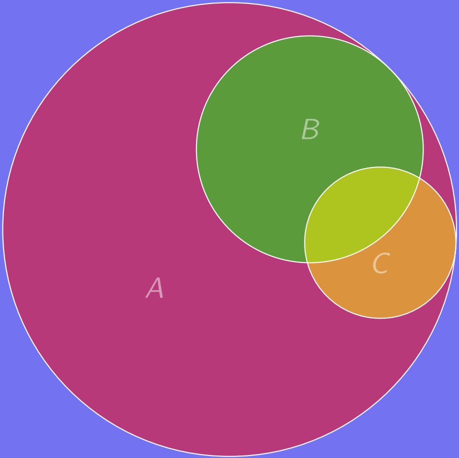
```cpp
#include <iostream>
#include "animals_m.h"
using namespace std;
void Sick::TakeMed(){
  if(med > 0) { med--; cout << "I feel better\n";}
  else cout << "I'm dying\n";
}
Cow::Cow(int f) {grass=f;}
void Cow::Speak() { cout << "Moo.\n"; }
void Cow::Eat(){
  if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
  else cout << "I'm hungry\n";
}
SickCow::SickCow(int f,int m) {grass=f; med=m;}
void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
MadCow::MadCow(int f, int p) {grass=f; pills=p;}
void MadCow::Speak() { cout << "Woof\n";}
void MadCow::TakePills() {
  if(pills > 0) {pills--; cout << "Moof, that's better\n";}
  else cout << "Woof woof woof!\n";
}
```

More cows

animals_main_m.cpp

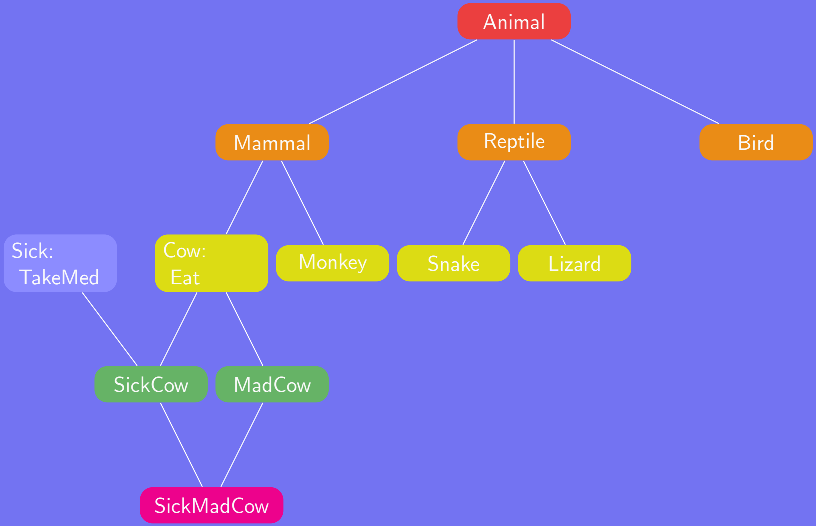
```
#include <iostream>
#include "animals_m.h"
using namespace std;
int main () {
  SickCow c1(1,1);
  c1.Speak(); c1.Eat(); c1.TakeMed();
  c1.Eat(); c1.TakeMed();
  cout << endl;
  MadCow c2(1,1);
  c2.Speak(); c2.Eat(); c2.TakePills();
  c2.Eat(); c2.TakePills();
}
```

Multiple inheritance can be tricky:



- A: Cows
- B: Sick cows
- C: Mad cows
- Sick mad cows are in $B \cap C$

The diamond problem



The diamond problem

Major issues:

- Is `Eat` inherited from `Cow` through `SickCow` or `MadCow`?
- What happens if the variable `grass` is updated?

Solutions to overcome the problem:

- Best: create a hierarchy without diamond problem
- Declare the derived classes as virtual

```
class Cow {...};
class SickCow : public virtual Cow {...};
class MadCow : public virtual Cow {...};
class SickMadCow : public SickCow, public MadCow {...};
```

Calling `Eat` *or updating* `grass` *does not generate any problem*

Never design a hierarchy diagram exhibiting a diamond problem

Sick mad cows

animals_d.h

```
class Animal {
  public: virtual void Speak() = 0; virtual void Eat() = 0;
};
class Sick {
  public:  void TakeMed();
  protected: int med;
};
class Cow : public Animal {
  public: Cow(int f=0); virtual void Speak(); void Eat();
  protected: int grass;
};
class SickCow : public virtual Cow, public Sick {
  public: SickCow(int f=0,int m=0); void Speak();
};
class MadCow : public virtual Cow {
  public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
  protected: int pills;
};
class SickMadCow : public SickCow, public MadCow {
  public: SickMadCow(int f=0, int m=0, int p=0); void Speak();
};
```

animals_d.cpp

```
#include <iostream>
#include "animals_d.h"
using namespace std;
void Sick::TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
  else cout << "I'm dying\n";
}
Cow::Cow(int f) {grass=f;}
void Cow::Speak() { cout << "Moo.\n"; }
void Cow::Eat(){ if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
  else cout << "I'm hungry\n";
}
SickCow::SickCow(int f,int m) {grass=f; med=m;}
void SickCow::Speak() { cout << "Ahem... Moo\n"; }
MadCow::MadCow(int f, int p) {grass=f; pills=p;}
void MadCow::Speak() { cout << "Woof\n";}
void MadCow::TakePills() {
  if(pills > 0) {pills--; cout << "Moof, that's better\n";}
  else cout << "Woof woof woof!\n";
}
SickMadCow::SickMadCow(int f, int m, int p) {grass=f; med=m; pills=p;}
void SickMadCow::Speak() {cout << "Ahem... Woof\n";}
```

animals_main_d.cpp

```
#include <iostream>
#include "animals_d.h"
using namespace std;
int main () {
  SickCow c1(1,1);
  c1.Speak(); c1.Eat(); c1.TakeMed();
  c1.Eat(); c1.TakeMed();
  cout << endl;
  MadCow c2(1,1);
  c2.Speak(); c2.Eat(); c2.TakePills();
  c2.Eat(); c2.TakePills();
  cout << endl;
  SickMadCow c3(1,1,1);
  c3.Speak(); c3.Eat(); c3.TakePills(); c3.TakeMed();
  c3.Eat(); c3.TakePills(); c3.TakeMed();
  SickMadCow  c4(1,1,0); Cow *c5=&c4;
  c4.Speak(); c4.Eat(); c4.TakePills(); c4.TakeMed();
  c5->Speak(); c5->Eat(); //c5->TakePills(); c5->TakeMed();
}
```

Understanding the code:

- How is polymorphism used?
- Describe the diamond problem
- How was the problem overcome?
- Draw a hierarchy diagram without the diamond problem
- What is happening if line 18 (10.308) is uncommented? Why?

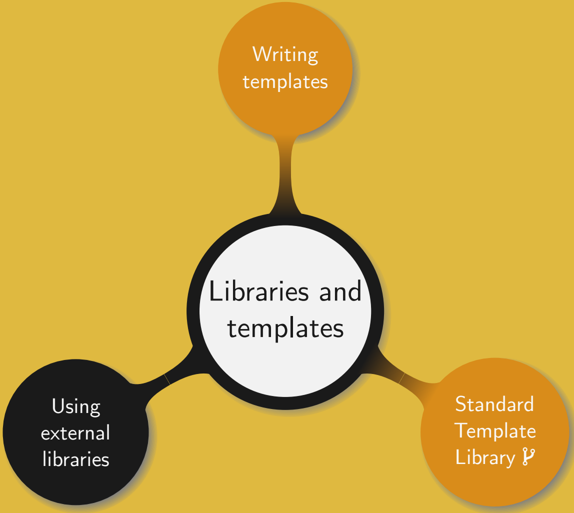
Process to organise a project:

1. Define what is needed or expected
2. Express everything in terms of objects
3. Define the relationships among the objects
4. Abstract new classes
5. Draw the hierarchy diagram
6. If there is any diamond, adjust the diagram
7. For each object define the methods
8. For each object define the attributes
9. Write the classes

Key points

- What is object oriented programming?
- In what order should the attributes and methods be defined?
- What are private and public?
- Why using inheritance?
- What is polymorphism?
- What is the best way to solve the diamond problem?

11. Libraries and templates


Writing templates
Libraries and templates
Using external libraries
Standard Template Library

Libraries

Simple overview:

- Many libraries available to define all type of objects
- Using a library:
  - Include header files
  - Possibility to use the library namespace
  - Reference the library at compilation time

To use a library the compiler must know:

- Where the header files are located
- The namespace a function belongs to
- Where the machine code is located

Overview:

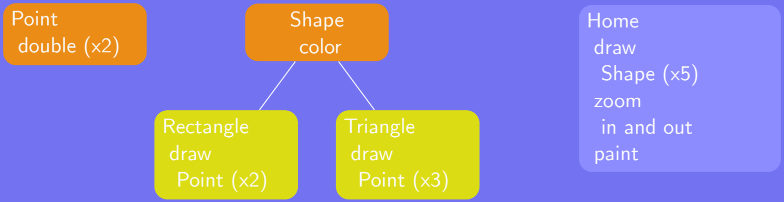
- Open Graphic Library (OpenGL)
- C library for drawing
- Cross platform
- Multi platform Application Programming Interface (API)
- API interacts with the GPU
- Widely used in games, Computer Aided Design (CAD), flight simulators, etc.

Our goal is to wrap the C functions into classes and build a home

Hierarchy diagram

First steps:

- Identify all the objects
- Organise them using a hierarchy diagram
- Identify the methods
- Define the necessary attributes



home/figures.h

```
#ifndef __FIGURES_H__
#define __FIGURES_H__
typedef struct _Point { double x,y; } Point;
class Shape {
  public: virtual void draw() = 0; virtual ~Shape()=0;
  protected: float r, g, b;
};
class Rectangle : public Shape {
  public: Rectangle(Point pt1={-.5,-.5}, Point pt2={.5,.5},
        float r=0, float g=0, float b=0);
    void draw();
  private: Point p1,p2;
};
class Triangle : public Shape {
  public: Triangle(Point pt1={-.5,-.5}, Point pt2={.5,-.5},
        Point pt3={0,.5}, float r=0, float g=0, float b=0);
    void draw();
  private:  Point p1,p2,p3;
};
#endif
```

Figures implementation

home/figures.cpp

```
#include <GL/glut.h>
#include "figures.h"
Shape::~Shape(){}
Rectangle::Rectangle(Point pt1, Point pt2,
    float red, float green, float blue) {
  p1=pt1; p2=pt2; r=red; g=green; b=blue;
}
void Rectangle::draw() {
  glColor3f(r, g, b); glBegin(GL_QUADS);
  glVertex2d(p1.x, p1.y); glVertex2d(p2.x, p1.y);
  glVertex2d(p2.x, p2.y); glVertex2d(p1.x, p2.y); glEnd();
}
Triangle::Triangle(Point pt1, Point pt2, Point pt3,
    float red, float green, float blue) {
  p1=pt1; p2=pt2; p3=pt3; r=red; g=green; b=blue;
}
void Triangle::draw() {
  glColor3f(r, g, b); glBegin(GL_TRIANGLE_STRIP);
  glVertex2d(p1.x, p1.y); glVertex2d(p2.x, p2.y);
  glVertex2d(p3.x, p3.y);  glEnd();
}
```

home/home.h

```
#ifndef __HOME_H__
#define __HOME_H__
#include "figures.h"
class Home {
  public:
    Home(Point pt1={0,-.25}, double width=1,
        double height=1.3, double owidth=.175);
    ~Home();
    void draw();
    void zoom(double *width,double *height,double *owidth);
  private:
    Point p; double w, h, o; Shape *sh[5];
    void zoomout(double *width,double *height,double *owidth);
    void zoomin(double *width,double *height,double *owidth);
    void paint(float *r, float *g, float *b);
};
#endif
```

Home implementation I

home/home.cpp

```
#include <ctime>
#include <cstdlib>
#include "home.h"

Home::Home(Point pt1, double width, double height, double owidth) {
  float r, g, b;
  Point p1, p2, p3;

/* init */
  p=pt1;
  w=width; h=height; o=owidth;
  srand((unsigned int)time(NULL));

/* body */
  p1={p.x-w/2,p.y-w/2}; p2={p.x+w/2,p.y+w/2};
  paint(&r,&g,&b); sh[0]=new Rectangle(p1,p2,r,g,b);
```

Home implementation II

```

/* door */
  p1={p.x-o,p.y-w/2}; p2={p.x+o,p.y};
  paint(&r,&g,&b); sh[1]=new Rectangle(p1,p2,r,g,b);

/* left window */
  p1={p.x-2*o,p.y+o}; p2={p.x-o,p.y+2*o};
  paint(&r,&g,&b); sh[2]=new Rectangle(p1,p2,r,g,b);

/* right window */
  p1={p.x+w/2-2*o,p.y+o}; p2={p.x+w/2-o,p.y+2*o};
  paint(&r,&g,&b); sh[3]=new Rectangle(p1,p2,r,g,b);

/* roof */
  p1={p.x,p.y+h-w/2}; p2={p.x-w/2,p.y+w/2}; p3={p.x+w/2,p.y+w/2};
  paint(&r,&g,&b); sh[4]=new Triangle(p1,p2,p3,r,g,b);
}
```

Home implementation III

```cpp

Home::~Home(){ for(int i=0;i<5;i++) delete sh[i]; }

void Home::draw() {
  for(int i=0;i<5;i++) sh[i]->draw();
}

void Home::zoom(double *width, double *height, double *owidth) {
  int static flag=0;
  if(h>=0.1 && flag==0) zoomout(width, height, owidth);
  else if (h<=2) { flag=1; zoomin(width, height, owidth); }
  else flag=0;
}

void Home::zoomout(double *width, double *height, double *owidth) {
  h/=1.01; *height=h; w/=1.01; *width=w; o/=1.01; *owidth=o;
}
```

Home implementation IV

```

void Home::zoomin(double *width, double *height, double *owidth) {
  h*=1.01; *height=h; w*=1.01; *width=w; o*=1.01; *owidth=o;
}

void Home::paint(float *r, float *g, float *b) {
  *r=(float)rand()/(float)RAND_MAX;
  *g=(float)rand()/(float)RAND_MAX;
  *b=(float)rand()/(float)RAND_MAX;
}
```

Home instantiation

home/main.cpp

```
#include <GL/glut.h>
#include "home.h"
void TimeStep(int n) {
  glutTimerFunc(n, TimeStep, n); glutPostRedisplay();
}
void glDraw() {
  double static width=1, height=1.5, owidth=.175;
  Home zh({0,-.25},width,height,owidth);
  zh.zoom(&width, &height, &owidth);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  zh.draw(); glutSwapBuffers(); glFlush();
}
int main (int argc, char *argv[]) {
  glutInit(&argc, argv);
  //  glutInitWindowSize(500, 500);
  glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
  glutCreateWindow("Home sweet home");
  glClearColor(1.0, 1.0, 1.0, 0.0); glClear(GL_COLOR_BUFFER_BIT);
  glutDisplayFunc(glDraw); glutTimerFunc(25, TimeStep, 25);
  glutMainLoop();
}
```

Basic process when using OpenGL:

1. Initialise the library: `glutInit(&argc, argv);`
2. Initialise the display: `glutInitDisplay(GLUT_RGB|GLUT_SINGLE);`
3. Create window: `glutCreateWindow(windowname);`
4. Set the clear color: `glClearColor(r,g,b);` $(r, g, b \in [0, 1])$
5. Clear the screen: `glClear(GL_COLOR_BUFFER_BIT);`
6. Register display callback function: `glutDisplayFunc(drawfct);`
7. Redraw the screen: recursive call to a timer function
8. Start the loop: `glutMainLoop();`
9. Draw the objects

Understanding the code:

- Why is the `static` keyword used in both the `glDraw` and `zoom` functions?
- Why were pointers used in he `zoom`, `zoomin` and `zoomout` functions?
- How were inheritance and polymorphism used?
- Comment the choices of `public` or `private` attributes and methods
- How is the keyword `#ifndef` used?

Compiling and running the home:

```
sh $ g++ -std=c++11 -o home main.cpp home.cpp figures.cpp -lglut \
    -lGL
sh $ ./home
```

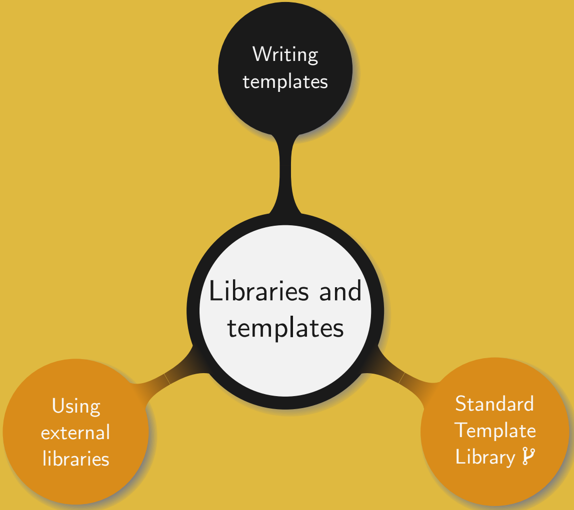
Better strategy is to use a `Makefile`:

- Simple text file explaining how to compile a program
- Useful for complex programs
- Easily handles libraries and compiler options

```
sh $ make
```

home/Makefile

```
CC = g++ # compiler
CFLAGS = -Wconversion -Wall -Wextra -Werror -std=c++14 -pedantic # co
LIBS = -lglut -lGL # libraries to use
SRCS = main.cpp home.cpp figures.cpp
MAIN = home
OBJS = $(SRCS:.cpp=.o)
.PHONY: clean # target not corresponding to real files
all:    $(MAIN) # target all constructs the home
	@echo Home successfully constructed
$(MAIN):
	$(CC) $(CFLAGS) -o $(MAIN) $(SRCS) $(LIBS)
.cpp.o: # for each .cpp build a corresponding .o file
	$(CC) $(CFLAGS) -c $<  -o $@
clean:
	$(RM) *.o *~ $(MAIN)
```


Writing templates
Libraries and templates
Using external libraries
Standard Template Library

Limitations of inheritance and polymorphism:

- High level classes, e.g. boat, company, car, etc.
- Low level classes used to define high level ones
- Still need to use function overloading to apply a function to more than one data type

This results in duplicated code, and programs harder to debug

Defining a template

A templates *is a "special class" where the data type is a parameter*

Example.

complex.h

```
#include <iostream>
using namespace std;
template<class TYPE>
class Complex {
  public:
    Complex(){ R = I = (TYPE)0; }
    Complex(TYPE real, TYPE img) {R=real;I=img;}
    void PrintComplex() {cout<<R<<'+'<<I<<"i\n";}
  private:
    TYPE R, I;
};
```

To use a template add the data type to the class name:

```
Complex<float> c1; complex<int> c2;
typedef Complex<double> dcplx; dcplx c3;
```

Exercise. Using the previous complex template, display `Complex` numbers composed of the types: `int`, `double` and `char`

complex.cpp

```
#include "complex.h"
typedef Complex<char> CComplex ;
int main () {
  Complex<double> a(3.123,4.9876); a.PrintComplex();
  Complex<int> b; b = Complex<int>(3,4);
  b.PrintComplex();
  CComplex c('a','b'); c.PrintComplex();
}
```

A bit of history

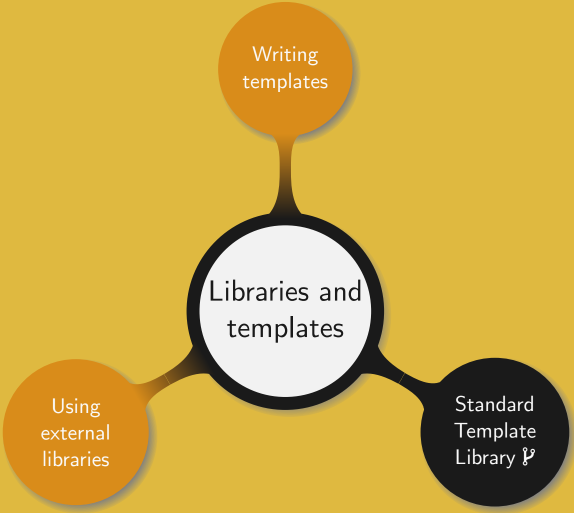
A few dates:

- 1983: C++
- 1994: templates accepted in C++
- 2011: many fixes/improvements on templates

Notes on templates:

- They are very powerful, complex and new
- They are not always handled nicely
- They might lead to long and unclear error messages
- They are not always fully optimized
- They require much work from the compiler

Chapter organisation



Basics on the Standard Template Library

C++ is shipped with a set of templates:

- Standard Template Library (STL)
- STL goals: abstractness, generic programming, no loss of efficiency
- Basic idea: use templates to achieve compile time polymorphism
- Components:
  - Containers
  - Iterators
  - Algorithms
  - Functional

Common sequence containers:

- Vector: automatically resizes, fast to access any element and to add/remove elements at the end
- Deque: vector with reasonably fast insertion deletion at beginning and end, potential issues with the iterator
- List: slow lookup, once found very fast to add/remove elements

A few other available containers:

- Set
- Multiset
- Multimap
- Bitset
- Valarray

A vector is similar to an array whose size can be changed:

- Size: automatically adjusted
- Template: no specific initial type
- A few useful functions: `push_back`, `pop_back`, `swap`

Example.

```
#include <vector>
vector<int> vint;
vector<float> vfloat;
```

vect.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> v1(4,100); vector<int> v2;
  vector<int>::iterator it;
  v1[3]=5;
  cout << v1[3] << " " << v1[0] << " " << v1[2] << endl;
  v2.push_back(2); v2.push_back(8); v2.push_back(18);
  cout << v2[0] << " " << v2[1] << " " << v2[2] << endl;
  v2.swap(v1);
  cout << v2[1] << " " << v1[1] << " " << v1.size() << endl;
  v1.erase(v1.begin()+1,v1.begin()+3);
  cout << v1[0] << " " << v1[1] << " " << v1.size() << endl;
  v1.pop_back();
  cout << v1[0] << " " << v1[1] << " " << v1.size() << endl;
  for(it=v2.begin(); it!=v2.end();it++) cout << *it << endl;
}
```

Common containers adaptors:

- Queue: First In First Out (FIFO) queue → list, deque
  Main methods: `size`, `front/back` (access next/last element), `push` (insert element) and `pop` (remove next element)

- Priority queue: elements must support comparison (determining priority) → vector, deque

- Stack: Last In First Out (LIFO) stack → vector, list, deque
  Main methods: `size`, `top` (access next element), `push` and `pop` (remove top element)

queue.cpp

```cpp
#include <iostream>
#include <queue>
using namespace std;
int main () {
  int i,j=0;
  queue<int> line;
  for(i=0;i<200;i++) line.push (i+1);
  while(line.empty() == 0) {
    cout << line.size () << " persons in the line\n"
      << "first in the line: " << line.front() << endl
      << "last in the line: " << line.back() << endl;
    line.pop ();
    if(j++%3==0) {
      line.push (++i);
      cout << "new in the line: " << line.back() <<endl;
    }
  }
}
```

A new object:

- Object that can iterate over a container class
- Iterators are pointing to elements in a range
- Their use is independent from the implementation of the container class

```
for(i=0;i<vct.size();i++) {
  ...
}
```

```
for(it=vct.begin(); \
  it !=vct.end();++it) {
    ...
}
```

Efficiency of `vct.size()`: fast operation for vectors, slow for lists

Example.

iterator.cpp

```
#include <iostream>
#include <set>
using namespace std;
int main() {
  set<int> s;
  set<int>::const_iterator it;
  s.insert(7);s.insert(2);s.insert(-6);
  s.insert(8);s.insert(1);s.insert(-4);
  for(it = s.begin(); it != s.end(); ++it)  {
    cout << *it << " ";
  }
  cout << endl;
}
```

Common algorithms implemented in templates:

- Manipulate data stored in the containers
- Mainly targeting range of elements
- Many "high low-level" functions such as:
  - Sort
  - Shuffle
  - Find with conditions
  - Partition

In a given range returns how many element are equal to some value

Example.

count.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","red","green","red"};
  vector<string> colorvect(colors, colors+8);
  int  nbcolors = count (colorvect.begin(),
      colorvect.end(), "red");
  cout << "red appears " << nbcolors  << " times.\n";
}
```

In a given range, returns an iterator to the first element that is equal to some value, or the last element in the range if no match is found

Example.

find.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","red","green","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  it=find(colorvect.begin(), colorvect.end(), "blue"); ++it;
  cout << "following blue is " << *it << endl;
}
```

Remove consecutive duplicates

Example.

unique1.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
bool cmp(string s1, string s2) { return(s1.compare(s2)==0);}
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","green","red","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  it=unique(colorvect.begin(), colorvect.end(),cmp);
  colorvect.resize(distance(colorvect.begin(),it));
  for(it=colorvect.begin(); it!=colorvect.end();++it)
    cout << ' ' << *it;
  cout << endl;
}
```

Sort elements in ascending order

Example.

sort.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
bool cmp(string s1, string s2) { return(s1.compare(s2)<0);}
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","green","red","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  sort(colorvect.begin(), colorvect.end(),cmp);
  for(it=colorvect.begin(); it!=colorvect.end();++it)
    cout << ' ' << *it;
  cout << endl;
}
```

Exercise. Remove all duplicate elements from the color vector.

unique2.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
bool cmp1(string s1, string s2) {return(s1.compare(s2)<0);}
bool cmp2(string s1, string s2) {return(s1.compare(s2)==0);}
int main () {
  string colors[8]={"red","blue","yellow","black","green","green","red","red"};
  vector<string> colorvect(colors, colors+8); vector<string>::iterator it;
  sort(colorvect.begin(), colorvect.end(),cmp1);
  it=unique(colorvect.begin(), colorvect.end(),cmp2);
  colorvect.resize(distance(colorvect.begin(),it));
  for(it=colorvect.begin(); it!=colorvect.end();++it)  cout << ' ' << *it;
  cout << endl;
}
```

Reverse the order of the elements

Example.

reverse.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","green","red","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  reverse(colorvect.begin(), colorvect.end());
  for(it=colorvect.begin(); it!=colorvect.end();++it)
    cout << ' ' << *it;
  cout << endl;
}
```

Remove elements and returns an iterator to the new end

Example.

remove.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
bool bstart(string s) { return(s[0]!='b'); }
int main () {
  string colors[8] = {"red","blue","yellow","black",
    "green","green","red","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  it=remove_if(colorvect.begin(),colorvect.end(),bstart);
  colorvect.resize(distance(colorvect.begin(),it));
  for(it=colorvect.begin(); it!=colorvect.end();++it)
    cout << ' ' << *it;
  cout << endl;
}
```

Randomly rearrange elements

Example.

random.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
int main () {
  srand (unsigned(time(0)));
  string colors[8] = {"red","blue","yellow","black",
    "green","green","red","red"};
  vector<string> colorvect(colors, colors+8);
  vector<string>::iterator it;
  random_shuffle(colorvect.begin(),colorvect.end());
  for(it=colorvect.begin(); it!=colorvect.end();++it)
    cout << ' ' << *it;
  cout << endl;
}
```

Returns min and max of two elements or the min and max in a list

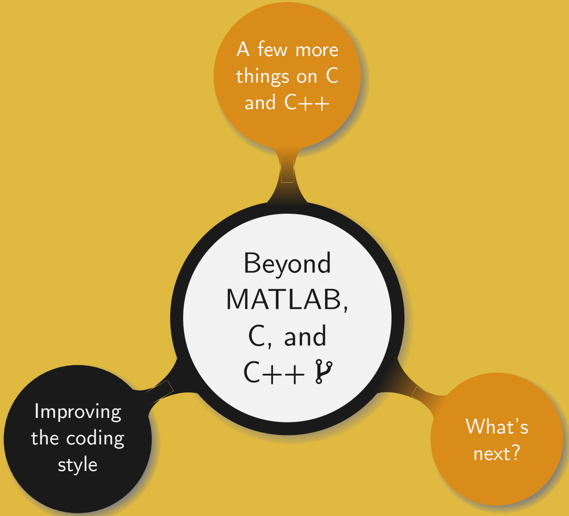
minmax.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
bool cmp(string s1, string s2) {return(s1.compare(s2)<0);}
int main () {
  srand (unsigned(time(0)));
  auto mm=minmax({"red","blue","yellow","black"},cmp);
  cout <<  mm.first << ' ' << mm.second;
  cout << endl;
}
```

Key points

- How to use external libraries?
- How to write a Makefile?
- What is the Standard Template Library?
- Why using STL?

12. Beyond MATLAB, C, and C++


A few more things on C and C++
Beyond MATLAB, C, and C++
Improving the coding style
What's next?

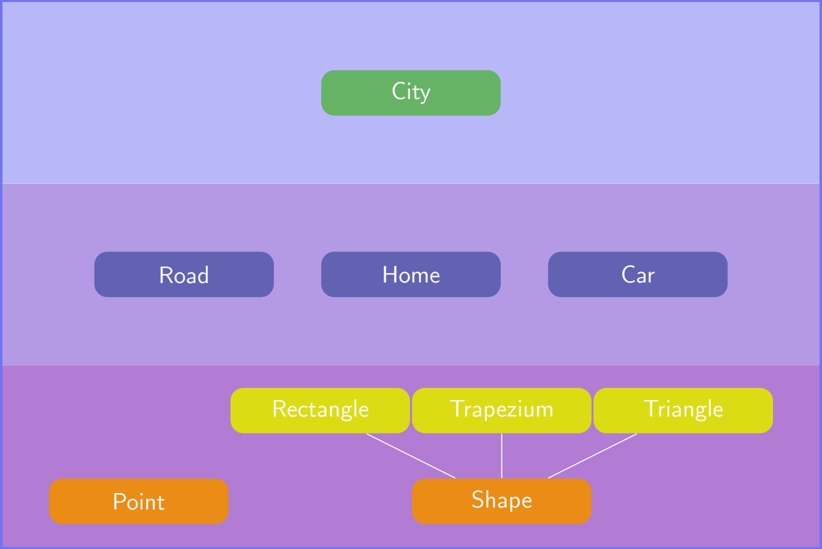
Clean coding strategy:

- Split the code into functions
- Organise the functions in different files
- Functions are organised by layers
- Functions of lower layers do not call functions of higher layers
- A function can only call functions of same or lower levels

Example. In the implementation of the home:

- Lowest layer: definition of the figures (points, rectangle, and triangle)
- Middle layer: definition of the home (home and actions on the home)
- Top layer: instantiation of the home (more actions such as construction of a compound)

Layer programming



Makefile

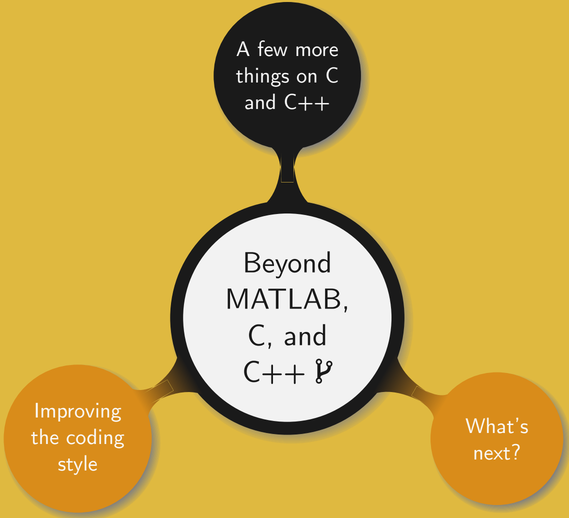
```
CCC = g++
CCFLAGS = -std=c++11 -Wall -Wextra -Werror -pedantic
LIBS = -lglut -lGL
LLIBS = -L. -lhome -lfig
LFIG_SRC = figures.cpp
LFIG_OBJ = $(LFIG_SRC:.cpp=.o)
LFIG = libfig.a
LHOME_SRC = home.cpp
LHOME_OBJ = $(LHOME_SRC:.cpp=.o)
LHOME = libhome.a
MAIN_SRC = main.cpp
MAIN = home
.PHONY: clean hlibs

all: $(LFIG_OBJ) $(LHOME_OBJ) hlibs $(MAIN)
	@echo Home successfully constructed

$(MAIN): $(MAIN_SRC)
	$(CCC) $(CCFLAGS) -o $(MAIN) $(MAIN_SRC) $(LIBS) $(LLIBS)

.cpp.o:
	$(CCC) $(CCFLAGS) -c $< -o $@

hlibs :
	ar rcs $(LFIG) $(LFIG_OBJ);   ar rcs $(LHOME) $(LHOME_OBJ)

clean:
	$(RM) *.o *.a *~ $(MAIN)
```

Clean code respecting standards

```
sh $ gcc -Wall -Wextra -Werror -pedantic file.c
sh $ g++ -Wall -Wextra -Werror -pedantic file.cpp
```

When coding:

- Ensure compatibility over various platforms
- Use tools such as *valgrind* to assess the quality of the code (e.g. spot memory leaks)
- For more complex program use a debugger such as *gdb*


A few more things on C and C++
Beyond MATLAB, C, and C++
Improving the coding style
What's next?

Constant variable:

- Creates a read-only variable
- Use and abuse `const` if a variable is not supposed to be modified
- In the case of a `const vector` use a `const` iterator:

```
vector<T>::const_iterator
```

Constant pointers vs. pointer to constant

Constant pointer

```
int const *p;
```

- The value p is pointing to can be changed
- The address p is pointing to cannot be changed

Pointer to constant

```
const int *p;
```

- The pointer p can point to anything
- What p points to cannot be changed

```
int a=0, b=1; const int *p1; int * const p2=&a;
p1=&a; cout << *p1 << *p2 << endl;
p1=&b; *p2=b; //p2=&b; *p1=b;
cout << *p1 << *p2 << endl;
```

Basics on references:

- Alias for another variable
- Changes on a reference are applied to the original variable
- Similar to a pointer that is automatically dereferenced
- Syntax: `int b=3; int &a=b`

Remarks:

- Reference variable must be initialised
- The variable it refers to cannot be changed

Example.

ref.cpp

```cpp
#include <iostream>
using namespace std;
int square0(int x) {return x*x;}
void square1(int x, int &res) { res=x*x; }
//int& square2a(int x) { int b=x*x; return b; }
int& square2b(int x) { int b=x*x; int &res=b; return res; }
int& square2c(int x) { static int b=x*x; return b; }
int main () {
  int a=2;
  cout << square0(a) << ' ' << a << endl;
  square1(a,a); cout << a << endl;
  cout << square2b(a) << endl;
  cout << square2c(a) << endl;
}
```

The `this` keyword:

- Address of the object on which the member function is called
- Mainly used for disambiguation

boat.cpp

```cpp
#include <iostream>
using namespace std;
class Boat {
  public:
    Boat(string name, int tonnage, bool IsDocked) {
      this->name=name; this->tonnage=tonnage; this->IsDocked=IsDocked;
    }
    void dock() { IsDocked=1;  cout<<"Docked!\n"; }
    void undock() { IsDocked=0; cout<<"Undocked!\n"; }
  private: bool IsDocked; string name; int tonnage;
};
int main () {
  Boat b("abc",1234,1); b.undock();
}
```

Pointer to function

Similar to pointer to variables:

- Variable storing the address of a function
- Useful to give a function as argument to another function
- Useful for callback functions (e.g. GUI)

fctptr.c

```c
#include <stdio.h>
#include <string.h>
int gm(char *n) {
  printf("good morning %s\n",n);
  return strlen(n);
}
int main () {
  int (*gm_ptr)(char *)=gm;
  printf("%d\n",(*gm_ptr)("john"));
}
```

enum_union.c

```
#include<stdio.h>
typedef struct _activity {
  enum { BOOK, MOVIE, SPORT } type;
  union {
    int pages;
    double length;
    int freq;
  } prop;
} activity;
int main() {
  activity a[5];
  a[0].type=BOOK; a[0].prop.pages=192;
  a[1].type=SPORT; a[1].prop.freq=4;
  a[2].type=MOVIE; a[2].prop.pages=123;
  a[2].prop.length=92.5;
  printf("%f",a[2].prop.length);
}
```

The argc and *argv[] parameters

arg.c

```c
#include <stdio.h>
int main (int argc, char *argv[]) {
  printf ("program: %s\n",argv[0]);

  if (argc > 1) {
    for (int i=1; i<argc; i++)
      printf("argv[%d] = %s\n", i, argv[i]);
  }
  else printf("no argument provided\n");
  return 0;
}
```

Compilation is performed in three steps:

1. Pre-processing

```
sh $ gcc -E file.c
```

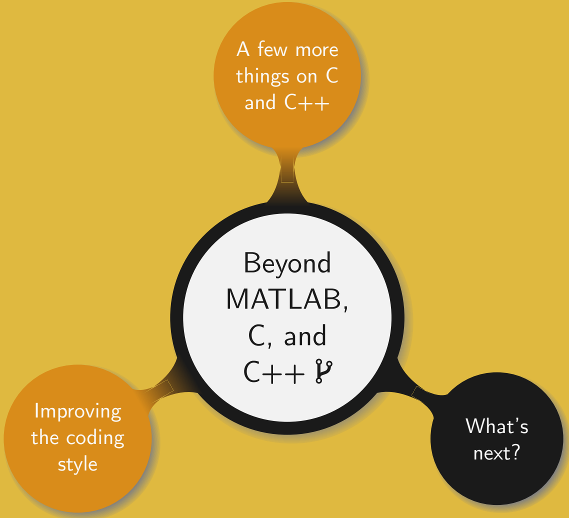
2. Assembling

```
sh $ gcc -c file.c
```

3. Linking

```
sh $ gcc file.c
```

Commands at stage *i* performs stage 1 to *i*


A few more things on C and C++
Beyond MATLAB, C, and C++
Improving the coding style
What's next?

Present

- MATLAB:
  - Testing new algorithms
  - Getting quick results
- C:
  - Lower level
  - More complex, flexible
  - Faster, less base functions
- C++:
  - New programming strategy
  - Higher level
  - Convenient for big projects

Important points that remain to be considered:

- More to learn on programming
- Languages of interest: C, Java, SQL, C++, PHP, CSS
- Other useful languages: Python, Perl, Ruby
- Designing a software: who is going to use it, where, how?
- More details on how computers are working
  - Data structures
  - Optimizations
  - How to improve efficiency

Key points

- Many things are left to learn
- Before coding always write an algorithm
- There no better way to learn than coding
- Do not reinvent the wheel, use libraries
- Each language has its own strengths, use them
- Extend your knowledge by building on what you already know

Thank you, enjoy the Winter break!

References I

1.25 https://upload.wikimedia.org/wikipedia/commons/b/b5/RomanAbacusRecon.jpg

1.25 https://upload.wikimedia.org/wikipedia/commons/7/76/Antikythera_model_front_panel_Mogi_Vicentini_2007.JPG

1.26 https://upload.wikimedia.org/wikipedia/commons/5/54/Batons_de_Napier.png

1.26 https://upload.wikimedia.org/wikipedia/commons/a/a0/Pocket_slide_rule.jpg

1.27 https://upload.wikimedia.org/wikipedia/commons/7/78/Pascaline-CnAM_823-1-IMG_1506-black.jpg

1.27 https://upload.wikimedia.org/wikipedia/commons/8/83/Thomas_Arithmometer_1975.png

1.28 https://upload.wikimedia.org/wikipedia/commons/6/6b/Charles_Babbage_-_1860.jpg

1.28 https://upload.wikimedia.org/wikipedia/commons/b/b1/Ada_Byron_aged_seventeen_%281832%29.jpg

1.29 http://www.cftea.com/c/2011/02/WPRMMSW80E5HFKI7/97GZAM05GBMCV7P9.jpg

1.30 https://upload.wikimedia.org/wikipedia/commons/a/a1/Apple_I_Computer.jpg

1.42 MATLAB documentation

1.43 MATLAB documentation

References II

1.44 MATLAB documentation

1.51 https://upload.wikimedia.org/wikipedia/commons/thumb/1/19/Mars_Climate_Orbiter_2.jpg/660px-Mars_Climate_Orbiter_2.jpg

3.97 https://upload.wikimedia.org/wikipedia/commons/0/0f/Droste_cacao_100gr_blikje%2C_foto_02.JPG

3.97 https://upload.wikimedia.org/wikipedia/en/2/22/Heckert_GNU_white.svg

5.128 https://upload.wikimedia.org/wikipedia/commons/1/1b/Ken_Thompson_and_Dennis_Ritchie--1973.jpg

9.228 https://upload.wikimedia.org/wikipedia/commons/d/da/BjarneStroustrup.jpg