

Introduction to Computer and Programming

Manuel

Fall 2018

Table of contents

Chapter 0:	Course information
Chapter 1:	Computer and Programming
Chapter 2:	MATLAB scripting
Chapter 3:	Functions and recursion
Chapter 4:	Advanced MATLAB
Chapter 5:	Introduction to C
Chapter 6:	Basic C
Chapter 7:	Arrays and pointers
Chapter 8:	Algorithms and efficiency
Chapter 9:	Introduction to C++
Chapter 10:	Object oriented programming
Chapter 11:	Libraries and templates
Chapter 12:	Beyond MATLAB, C, and C++

Chapter 0

Course information

Outline

- 1 Logistics
- 2 Evaluations
- 3 Resources

Who?

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
 - Jiayi (jane_chen@sjtu.edu.cn)
 - Zihao (shenzihao@sjtu.edu.cn)
 - Zhi (linzhilynn@gmail.com)
 - Shuhan (harrywsh@sjtu.edu.cn)

Who?

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
 - Jiayi (jane_chen@sjtu.edu.cn)
 - Zihao (shenzihao@sjtu.edu.cn)
 - Zhi (linzhilynn@gmail.com)
 - Shuhan (harrywsh@sjtu.edu.cn)

Important notes:

- When contacting a TA for an important matter such as updating a grade cc the message to the instructor
- Add the tag [vg101] to the email subject
e.g. Subject: [vg101] important issue
- Do not send large files (> 2 MB) by email, instead use [SJTU jBox service](#)

When?

Course organisation:

- Lectures:
 - Tuesday 10:00 – 11:40
 - Thursday 10:00 – 11:40
 - Friday 10:00 – 11:40 (even weeks)
- Lab sessions:
 - Monday 18:20 – 20:00
 - Wednesday 12:10 – 13:50
 - Thursday 18:20 – 20:00
- Recitation classes: will be announced on **Canvas**
- Office hours:
 - Tuesday 15:40 – 17:50
 - Other times available on appointment

What?

Main goals of this course:

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

What?

Main goals of this course:

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

Ultimate goal: understand programming and be able to quickly adjust to new languages/libraries

How?

Learning strategy:

- Course side:
 - ① Getting familiar with programming (MATLAB)
 - ② Understand deeper concepts (C)
 - ③ Bridge the gap between computer and human views (C++)

How?

Learning strategy:

- Course side:
 - ① Getting familiar with programming (MATLAB)
 - ② Understand deeper concepts (C)
 - ③ Bridge the gap between computer and human views (C++)
- Personal side:
 - ① Read and write code
 - ② Relate known strategies to new problems
 - ③ Perform extra research

Course outcomes

Detailed goals:

- Proficiency with data representation and naming
- Proficiency with data input and output
- Proficiency with programming with math and logical operators and functions
- Proficiency with designing, testing, and implementing functions and procedures
- Proficiency with control flow using selection and iteration
- Proficiency with use of pre-defined data structures
- Proficiency with primitive and complex data types
- Proficiency with visualization of data
- Proficiency with algorithm design for engineering analysis

Outline

- 1 Logistics
- 2 Evaluations
- 3 Resources

Assignments, labs, and projects

Assignments:

- Total: 8
- Content: basic algorithms, Matlab, C, C++
- Not graded, completed in groups or individually

Labs:

- Total: 8
- 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: 35%
- Labs: 5%

Grading policy

Grade weighting:

- Matlab midterm: 20%
- C midterm: 20%
- C++ final: 20%
- Projects: 35%
- Labs: 5%

Late submission: -10% per day, not accepted after 3 days

Final letter grade: curved to balance the three sections

Grading policy

Assignments:

- Not graded
- Each student must complete all the mandatory exercises
- Each student must review the code of each of his teammates
- A final improved version is submitted for each group
- Submissions should be successfully compiled or interpreted

Grading policy

Assignments:

- Not graded
- Each student must complete all the mandatory exercises
- Each student must review the code of each of his teammates
- A final improved version is submitted for each group
- Submissions should be successfully compiled or interpreted

Students not following these rules will receive a
–5% deduction on their final course grade

Honor Code

General rules:

- Not allowed:
 - Reuse the code/work from other students or groups
 - Reuse the code/work from the internet
 - Give too many details on how to solve an exercise

Honor Code

General rules:

- Not allowed:
 - Reuse the code/work from other students or groups
 - Reuse the code/work from the internet
 - Give too many details on how to solve an exercise
- Allowed:
 - Share ideas and understandings on the course
 - Provide general directions 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 submission
- If a student breaks the Honor Code, the whole group is sent to Honour Council

Special circumstances

Contact us as early as possible when:

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

Any late request will be rejected

Outline

- 1 Logistics
- 2 Evaluations
- 3 Resources

Canvas

On **Canvas** platform:

- Course materials and assignments
- Announcements and notifications
- Polls

References

Places to find information:

- MATLAB documentation
- *C for Engineers and Scientists* by Harry H. Cheng
- *Thinking in C++* by Bruce Eckel
- Search the web

References

Places to find information:

- MATLAB documentation
- *C for Engineers and Scientists* by Harry H. Cheng
- *Thinking in C++* by Bruce Eckel
- Search the web
- Do not use baidu

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
- Any advice/suggestions will be much appreciated

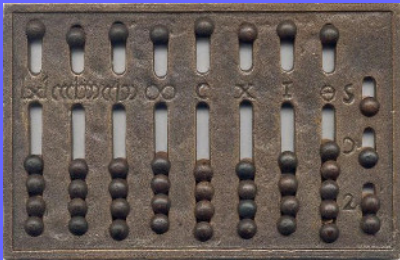
Chapter 1

Computer and Programming

Outline

- ① A brief history of computing
- ② Interacting with computers
- ③ Programming in science

Ancient Era

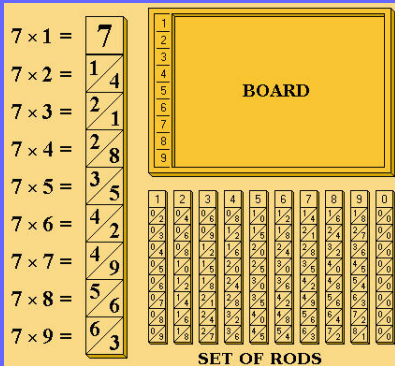


Abacus (-2700)



Antikythera mechanism (-100)

Calculating Tools



Napier's bones (1617)



Sliderule (1620)

Calculating Tools



Napier's bones (1617)



Sliderule (1620)

Note: first pocket calculator around 1970 in Japan.

Mechanical Calculators

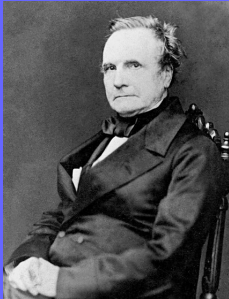


Pascaline (1642)



Arithmomètre (1820)

The 19th Century



Charles Babbage (1791–1871)

- Difference Engine (Built in the 1990es)
- Analytical Engine (Never built)

Ada Byron (1815–1852)

- Extensive notes on Babbage work
- Algorithm to calculate a sequence of Bernoulli numbers using the Analytical Engine



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)

Toward Modern Computing

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

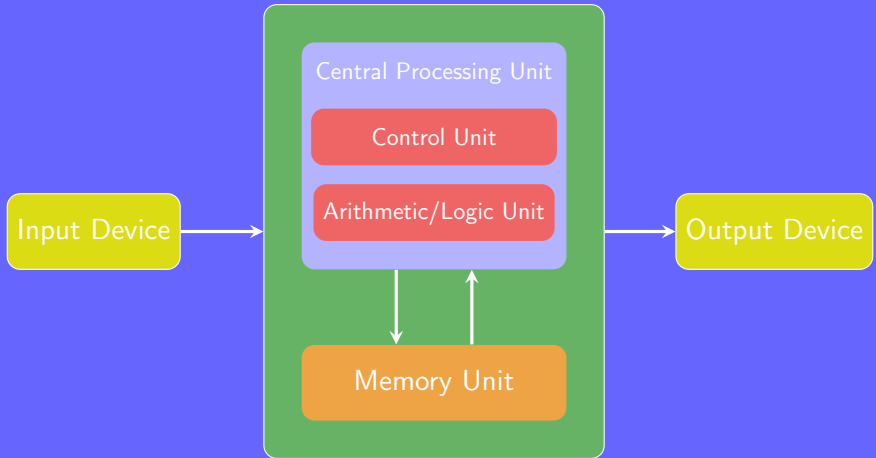


Apple I (1976)

Outline

- ① A brief history of computing
- ② Interacting with computers
- ③ Programming in science

Von Neumann architecture



What does a computer understand?

Numbers in various bases:

- Humans use *decimal* (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), e.g. $(253)_{10}$
- Computers work internally using *binary* (0,1), e.g. $(11111101)_2$
- Human-friendly way to represent binary: *hexadecimal* (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F), e.g. $(FD)_{16}$

Number base conversion

Base conversion:

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

Quick examples

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

Quick examples

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.

$1675 = 68B$, $321 = (141)_{16}$, $(100011)_2 = (23)_{16}$,

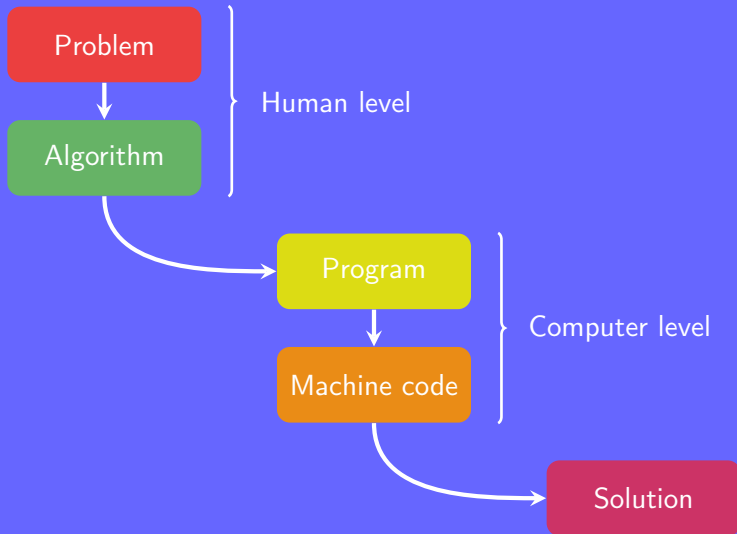
$654 = (1010001110)_2$, $2049 = (10000000001)_2$,

$ACE = 101011001110$, $5F3EC6 = (10111110011111011000110)_2$

$(111110)_2 = 62$, $(10101)_2 = 21$, $(12345)_{16} = 74565$,

$12C3C = 76860$

How to use a computer?



Algorithm

Algorithm: recipe telling the computer how to solve a problem.

Algorithm

Algorithm: recipe telling the computer how to solve a problem.

Example.

I am the “computer”, detail an algorithm such that I can prepare a jam sandwich.

Actions: cut, listen, spread, sleep, read, take, eat, dip, assemble

Things: knife, guitar, bread, honey, jamjar, sword, slice

Algorithm

Algorithm: recipe telling the computer how to solve a problem.

Example.

I am the “computer”, detail an algorithm such that I can prepare a jam sandwich.

Actions: cut, listen, spread, sleep, read, take, eat, dip, assemble

Things: knife, guitar, bread, honey, jamjar, sword, slice

Algorithm. (*Sandwich making*)

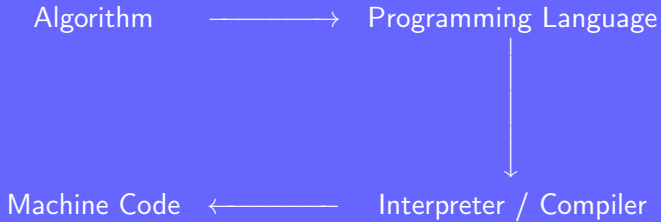
Input : 1 bread, 1 jamjar, 1 knife

Output: 1 jam sandwich

- 1 take the knife and cut 2 slices of bread;
 - 2 dip the knife into the jamjar;
 - 3 spread the jam on the bread, **using the knife**;
 - 4 assemble the 2 slices together, **jam on the inside**;
-

Program

Algorithm vs Machine code



Example

Problem: 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

1 **return** side * side

area.c

```
1  #include<stdio.h>
2  int main() {
3      int side;
4      printf("Side: ");
5      scanf("%d",&side);
6      printf("Area: %d",\
7          side*side);
8  }
```

area.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int side;
5      cout << "Side: ";
6      cin >> side;
7      cout << "Area: "\
8          << side*side;
9      return 0;
10 }
```

area.m

```
1  a=input("Side: ");
2  printf ("Area: %d",...
3      a*a)
```


Running the program

To see the result of a program:

- C or C++
 - ① Write the source code
 - ② Compile the program
 - ③ Run the program
- MATLAB
 - ① Type the code
 - ② Press Return

Outline

- ① A brief history of computing
- ② Interacting with computers
- ③ Programming in science

Mathematical software

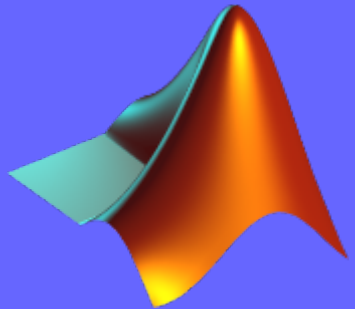
Common math software:

- Axiom
- GAP
- gp
- Magma
- Maple
- Mathematica
- MATLAB
- Maxima
- Octave
- R
- Scilab

MATLAB

MATLAB=MATrix LABoratory

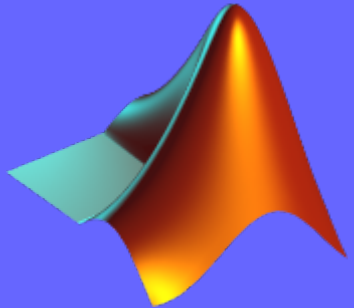
- **Matrix manipulations**
- **Implement algorithms**
- **Plotting functions/data**
- Create user interfaces
- Interfaced with other programming languages



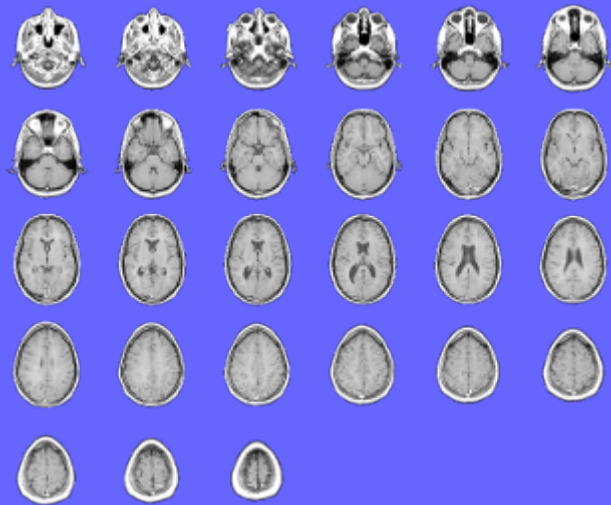
Why MATLAB?

Engineers like MATLAB:

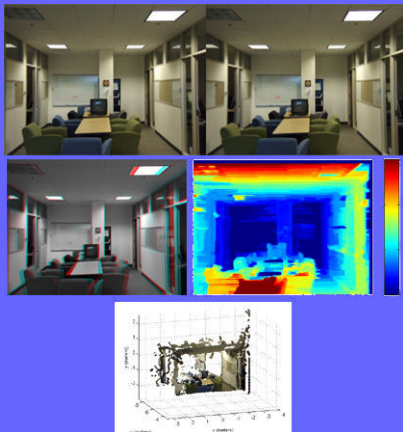
- Easy to use
- Versatile
- Built-in programming languages
- Many toolboxes
- Widely used in academia and industry



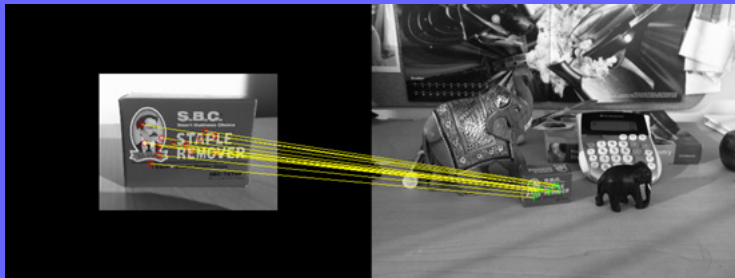
MRI slices



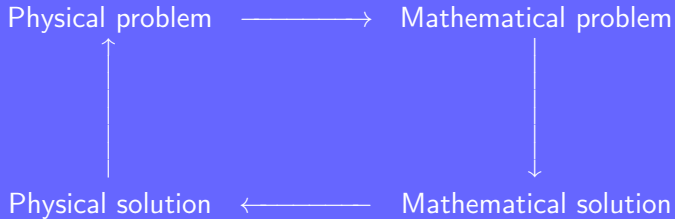
Stereo Vision



Object detection



Mathematics and Physics



What to do?

Before jumping on the computer and start coding:

- Clearly state/translate the problem
- What is known \longrightarrow INPUT
- What is to be found \longrightarrow OUTPUT
- Find a systematic way to solve the problem \longrightarrow Algorithm
- Check the solution
- Start implementing

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.

Studying the problem

Problem: 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.

- Easy part
 - Problem: finding the density of the sun
 - Initial input: distance r , circumference c
 - Output: density d

Studying the problem

Problem: 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.

- Easy part
 - Problem: finding the density of the sun
 - Initial input: distance r , circumference c
 - Output: density d
- Potentially more complicated part
 - ① Density
 - ② Sun \sim sphere, $radius = \frac{circumference}{2\pi} \Rightarrow$ volume V
 - ③ Mass of the sun:

Studying the problem

Problem: 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.

- Easy part
 - Problem: finding the density of the sun
 - Initial input: distance r , circumference c
 - Output: density d
- Potentially more complicated part
 - ① Density
 - ② Sun \sim sphere, $radius = \frac{circumference}{2\pi} \Rightarrow$ volume V
 - ③ Mass of the sun: Kepler's 3rd law: $\frac{T^2}{r^3} = \frac{4\pi^2}{GM}$
 - ④ $M = \frac{4\pi^2 r^3}{GT^2}$

The Algorithm

Problem: 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.

Algorithm.

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

Output : Density of the Sun

- 1 $V \leftarrow \frac{4}{3}\pi\left(\frac{c}{2\pi}\right)^3;$
 - 2 $M \leftarrow \frac{4\pi^2 r^3}{GT^2};$
 - 3 **return** $\frac{M}{V};$
-

The Algorithm

Problem: 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.

Algorithm.

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

Output : Density of the Sun

- 1 $V \leftarrow \frac{4}{3}\pi\left(\frac{c}{2\pi}\right)^3;$
 - 2 $M \leftarrow \frac{4\pi^2 r^3}{GT^2};$
 - 3 **return** $\frac{M}{V};$
-

Run the algorithm: 338110866080

WRONG!

WRONG!

UNITS...

The Algorithm

Problem: 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.

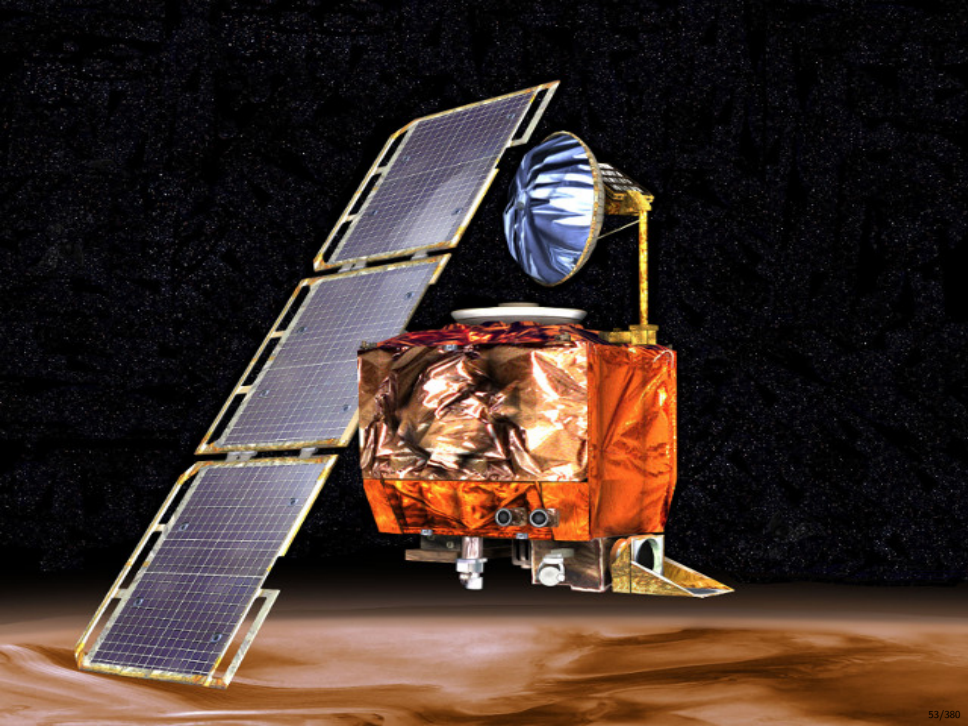
Algorithm.

Input : $r = 1.496 \cdot 10^{11}$ m, $c = 4.379 \cdot 10^9$ m,
 $G = 6.674 \cdot 10^{-11}$ m³/kg/s², $T = 365 * 24 * 3600$ s

Output : Density of the Sun

- 1 $V \leftarrow \frac{4}{3}\pi\left(\frac{c}{2\pi}\right)^3;$
 - 2 $M \leftarrow \frac{4\pi^2 r^3}{GT^2};$
 - 3 **return** $\frac{M}{V};$
-

Run the algorithm: 1404 kg/m³



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?

Chapter 2

MATLAB scripting

Outline

- ① Starting with MATLAB
- ② Conditional statements
- ③ Loops

Running MATLAB

Two modes:

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

View in desktop mode:

- Command history
- Command window
- Workspace
- Help

File location: current directory or a directory listed in the path

Basic use

MATLAB as a calculator:

- Operation: $1+2$ vs. `1+2;`
- Variables: start with a letter, case sensitive
e.g. `a=1+2; A=3+2; a123_=4+5;`
- Comments: ignore everything after a `%`
- Separate two commands on a same line: `cmd1, cmd2`
- Split a line over two lines: keep reading on next line after `...`
e.g. `long ...`
`line`

Simple operations

More MATLAB operations:

- Addition: $+$
- Subtraction: $-$
- Multiplication: $*$
- Power: $^$
- (Right) division: $/$
- Left division: \backslash
- Order of evaluation: $()$
- $\text{pi} = \pi$
- $i = \sqrt{-1}$
- $j = \sqrt{-1}$
- $\text{Inf} = \text{Infinity}$
- NaN : Not a Number

Density of the Sun

MATLAB code to input in the workspace window:

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

Density of the Sun

MATLAB code to input in the workspace window:

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

Questions.

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

M-File

MATLAB script:

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

M-File

MATLAB script:

- Write the code in a file and load it
- Variables are added to the workspace
- To avoid variable conflicts use: `clear`, `clear all`, `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.

M-File

MATLAB script:

- Write the code in a file and load it
- Variables are added to the workspace
- To avoid variable conflicts use: `clear`, `clear all`, `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.

```
1 clear all, clc;
2 number1=input('Input a number: ');
3 number2=input('Input a number: ');
4 numbers=number1+number2;
5 disp(numbers);
```

Arrays and MATLAB

Array

Arrangement of quantities in rows and columns

Arrays and MATLAB

Array

Arrangement of quantities in rows and columns



Matrix

Two-dimensional numeric array

Arrays and MATLAB

Array

Arrangement of quantities in rows and columns



Matrix

Two-dimensional numeric array



MATLAB

MATrix LABoratory

Arrays and MATLAB

Array

Arrangement of quantities in rows and columns



Matrix

Two-dimensional numeric array



MATLAB

MATrix LABoratory



Arrays are the **most important** concept to understand

Generating arrays and matrices

Creating arrays and matrices:

- Sequence of numbers: `a:b` or `a:b:c`
- Concatenate (join) elements: `[]`

Generating arrays and matrices

Creating arrays and matrices:

- Sequence of numbers: `a:b` or `a:b:c`
- Concatenate (join) elements: `[]`
- 1-dimensional array: `[a:b]` or `[a:b:c]`
- 2-dimensional array: `[a b c; d e f;]`

Generating arrays and matrices

Creating arrays and matrices:

- Sequence of numbers: `a:b` or `a:b:c`
- Concatenate (join) elements: `[]`
- 1-dimensional array: `[a:b]` or `[a:b:c]`
- 2-dimensional array: `[a b c; d e f;]`
- n elements from $[a, b]$: `linspace(a, b, n)`
- `zeros(a,b)`
- `ones(a,b)`

Dealing with matrices

Explain the result of the following commands:

```
1 clear all
2 a=magic(5)
3 a=[a;a+2], pause
4 a(:,3)=[]
5 a(:,3)=5
6 a(7,3), pause
7 whos a
8 a=reshape(a,5,8)
9 a', pause
10 sum(a)
11 sum(a(:,1))
12 sum(a(1,:))
```

Array vs. Matrix

Arrays

- Element by element
- `.*`
- `./`
- `.\`
- `.^`

Matrices

- Complex conjugate transpose: `'`
- Nonconjugate transpose: `*`
- `det`
- `inv`
- `eig`

Basic operations

Explain the result of the following commands:

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

Accessing elements in a matrix

Given a matrix, elements can be accessed by:

- Coordinates: using their (row,column) position
- Indices: using a single number representing their position; the top left element has index 1 and the bottom right “number of elements”

Accessing elements in a matrix

Given a matrix, elements can be accessed by:

- Coordinates: using their (row,column) position
- Indices: using a single number representing their position; the top left element has index 1 and the bottom right “number of elements”

Example.

Explain the following commands:

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

Outline

- ① Starting with MATLAB
- ② Conditional statements
- ③ Loops

The if statement

If it rains, then I take an umbrella

The if statement

If it rains, then I take an umbrella

Structure in MATLAB:

```
1  if expression1
2      statements1
3  elseif expression2
4      statements2
5  else
6      statements
7  end
```

Boolean logic

Boolean logic: introduced by George Boole around mid 1800s

Truth table for the 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

Boolean logic

Boolean logic: introduced by George Boole around mid 1800s

Truth table for the 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

Run instructions based on the truth value of a given expression

Relational operators

Comparative operators:

- $<$ less than
- $<=$ less than or equal to
- $>$ greater than
- $>=$ greater than or equal to
- $==$ equal to
- $\sim=$ not equal to

Relational operators

Comparative operators:

- $<$ less than
- $<=$ less than or equal to
- $>$ greater than
- $>=$ greater than or equal to
- $==$ equal to
- $\sim =$ not equal to

Logical operators:

- $\&$ and
- $|$ or
- \sim not
- $\text{xor}(\cdot, \cdot)$ exclusive or

Relational operators

Comparative operators:

- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to
- `==` equal to
- `~=` not equal to

Logical operators:

- `&` and
- `|` or
- `~` not
- `xor(·, ·)` exclusive or

Short-circuit operators:

- `A && B` evaluates expression B only if A is True
- `A || B` evaluates expression B only if A is False

Simple application script

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
```

Simple application script

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
```

Questions.

- What are those commands doing?
- How to request some input from the user?
- What is 's' on line 1?

The switch statement

When it rains, I take an umbrella; When it's sunny I take a hat.

The switch statement

When it rains, I take an umbrella; When it's sunny I take a hat.

Structure in MATLAB:

```
1  switch variable
2      case value1
3          statements1
4      case value2
5          statements2
6      otherwise
7          statements
8  end
```

Note: the variable is expected to be a scalar or a string

Example

Write a script which prompts the user for a digit, displays 0 on a 0, < 5 if it is between 1 and 4, and ≥ 5 if it is larger or equal to 5.

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

Questions.

- How is the code aligned?
- Why is input used without the 's' flag?

Outline

- ① Starting with MATLAB
- ② Conditional statements
- ③ Loops

The while loop

Loops in MATLAB:

- Definition: group of statements repeatedly executed as long as a given conditional expression evaluates to True
- Types: while, for, and vectorizing

The while loop

Loops in MATLAB:

- Definition: group of statements repeatedly executed as long as a given conditional expression evaluates to True
- Types: while, for, and vectorizing

Structure in MATLAB:

```
1 while expression
2     statements
3 end
```

Example.

```
1 i=0
2 while true
3     i=i+1
4 end
```

Example

```
1  o=input('Input a basic arithmetic operation: ','s');
2  i=1;
3  while (o(i) >= '0' && o(i) <= '9')
4      i = i+1;
5  end
6  n1=str2num(o(1:i-1));
7  n=o(i);
8  n2=str2num(o(i+1:end));
9  switch n
10     case '+'
11         n1+n2
12     case '-'
13         n1-n2
14     case '*'
15         n1*n2
16     case '/'
17         n1/n2
18     otherwise
19         disp('Not a basic arithmetic operation')
20 end
```

Questions

In the previous code:

- How is the code formatted?
- 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 than an integer is input?

The for loop

Structure in MATLAB:

```
1  for i=start:increment:end  
2      statements  
3  end
```

The for loop

Structure in MATLAB:

```
1  for i=start:increment:end
2      statements
3  end
```

Example.

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

Questions.

- How is the code indented?
- What is this code doing?

Vectorizing loop

MATLAB: array/matrix language



Convert for/while loops into vector/matrix operations

Vectorizing loop

MATLAB: array/matrix language



Convert for/while loops into vector/matrix operations

Example.

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

Questions.

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

The continue and break commands

More advanced loop commands:

- `continue`: directly jump to the next iteration
- `break`: exit the loop early

The continue and break commands

More advanced loop commands:

- continue: directly jump to the next iteration
- break: exit the loop early

Example.

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

Questions

In the previous code:

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

Efficiency

Arrays are stored linearly in memory:

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

Efficiency

Arrays are stored linearly in memory:

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

Example.

To store the matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ in memory is MATLAB using
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;
```

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;
```

Questions.

- What is this code doing?
- Is j representing the rows of 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

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

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

Questions

In the previous code:

- What is the result of `whos B`?
- What does $B = A > 10$ mean?
- What is the goal of line 3?
- After line 4 what is in c ?
- Why is $\sim b$ used?

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?

Chapter 3

Functions and recursion

Outline

- ① Basics on functions
- ② Common MATLAB functions
- ③ Recursion

From script to function

Script:

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

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,inp2,...)`
- Functions can be called from an .m file or from 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,inp2,...)`
- Functions can be called from an .m file or from the workspace

Script vs. function:

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

density.m

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

Sub-functions

A .m file can contain:

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

Example.

Write a function returning the mean and the standard deviation.

Calculate the mean in a sub-function

Sub-functions

A .m file can contain:

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

Example.

Write a function returning the mean and the standard deviation.
Calculate the mean in a sub-function

stat.m

```
1 function [mean,stdev] = stat(x)
2     n = length(x);
3     mean = avg(x,n);
4     stdev = sqrt(sum((x-mean).^2)/n);
5
6 function mean = avg(x,n)
7     mean = sum(x)/n;
```

Functions and sub-functions

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?

Outline

- ① Basics on functions
- ② Common MATLAB functions
- ③ Recursion

Mathematical functions

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)`

Useful functions

The save and load functions:

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

Useful functions

The save and load functions:

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

Random number generation:

- `rand(n,m)`: $n \times m$ matrix of random numbers (uniform distribution)
- `randn(n,m)`: $n \times m$ matrix of random numbers (standard normal distribution)
- `random('dist',parameters)`: random numbers following the distribution `dist`
- `rand('state',datenum(clock))`: enforce a specific seed
- `randperm(n)`: random permutation

The sprintf function

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'`

The sprintf function

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.

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

File input/output

Basic idea: open a stream between MATLAB and a file



File input/output

Basic idea: open a stream between MATLAB and a file



```
1 fd=fopen('file.txt', 'permission')
2 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+

The fprintf and fscanf functions

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: `fgetline(fd)`

The fprintf and fscanf functions

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: `fgetline(fd)`

Any opened stream must be closed

Exercise

Given a text file where each line is composed of three fields, namely 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.

Exercise

sortnames.m

```
1 function sortnames(fininput, foutput)
2     fd1=fopen(fininput,'r');
3     i=1;
4     line=fgetl(fd1);
5     while line ~= -1
6         a=find(isspace(line),2);
7         info{i}=sprintf('%s %s %s\n', line(a(1)+1:a(2)-1), ...
8             line(1:a(1)-1), line(a(2)+1:end));
9         i=i+1; line=fgetl(fd1);
10    end
11    fclose(fd1);
12
13    fd2=fopen(foutput,'w');
14    for j=randperm(i-1)
15        fprintf(fd2,info{j});
16    end
17    fclose(fd2);
```

Questions

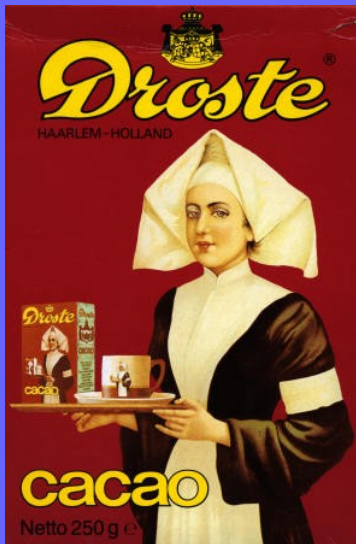
In the previous code:

- 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 _____

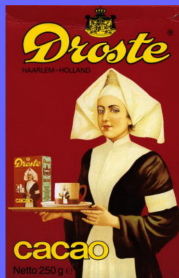
Outline

- ① Basics on functions
- ② Common MATLAB functions
- ③ Recursion

What is recursion?



What is recursion?



What is recursion?



14

10/8

17/18

Recursive acronyms

Famous acronyms:

- GNU: GNU's Not Unix
- WINE: WINE Is Not an Emulator
- PHP: PHP Hypertext Preprocessor
- LAME: LAME Ain't an MP3 Encoder



Recursion in computer science

Basic idea behind recursion:

- General case: given a process P and some data D , describe P using itself together with a simplified version D' of D
- Computer science case: inside the function $P(D)$, call the function $P(D')$

Recursion in computer science

Basic idea behind recursion:

- General case: given a process P and some data D , describe P using itself together with a simplified version D' of D
- Computer science case: inside the function $P(D)$, call the function $P(D')$

Example.

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.

A short recursive story

For the sake of simplicity we work with integers and map the child to 3, the frog to 2, the bear to 1, and the weasel to 0.

Algorithm. (*Bedtime story*)

Input : An integer n representing an animal or a child

Output: The child and all the animals asleep

```
1 Function Read( $n$ ):  
2   | if  $n = 0$  then sleep( $n$ );  
3   | else  $i \leftarrow n - 1$ ; Read( $i$ ); sleep( $n$ );  
4 end
```

Question.

Draw a simple diagram showing how the recursion is applied

Numbers in words

For an automated information service a telephone company needs the digits of phone numbers to be read digit by digit. Therefore you are asked to rewrite a sequence of digits into words, with a space between each word; no space at the beginning and at the end.

Numbers in words

Algorithm. (*Numbers in words*)

Input : A large integer n

Output : n , digit by digit, using words

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

Recursion vs. iteration

When to prefer recursion over iteration:

- Recursive algorithm more obvious than iterative one
- Depends on the language

In MATLAB, C and C++, iterative algorithms should be preferred

Recursion vs. iteration

When to prefer recursion over iteration:

- Recursive algorithm more obvious than iterative one
- Depends on the language

In MATLAB, C and C++, iterative algorithms should be preferred

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?

Chapter 4

Advanced MATLAB

Outline

- 1 Plotting
- 2 Data types
- 3 Structures

General plotting process

Simple workflow:

- ① Use plotting tools or functions to create a graph
- ② Extract data info/perform data fitting
- ③ Edit components (axes, labels...)
- ④ Add labels, arrow
- ⑤ Export, save, print...

2D plotting

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

Example

Explain the result of the following commands:

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

3D plotting

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

3D plotting

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

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

```
1 t=0:.01:2*pi;  
2 x=sin(2.*t)+1;  
3 y=cos(t.^2);  
4 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)`

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 the result of the following commands:

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

More plotting

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)`

Curve fitting process

Many problems and experiments feature several variables:

- How do they relate to each other?
- Can a variable be described by some other variables?

Performing curve fitting:

- ① Collect data, e.g. US population from 1790 to 1990
- ① Import data into MATLAB, e.g. `load census`
- ② Open curve fitting tool: `cftool`
- ③ Determine the best fit: test various types of fits
- ④ Monitor the error: display the residual plot, check the SSE
- ⑤ Extrapolate the data, e.g. check the curve's behavior in 2010

Interpolation

Goal of interpolation:

- Draw a smooth 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.

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

Outline

- 1 Plotting
- 2 Data types
- 3 Structures

Main problematic

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

Why data types?

From mathematics to computer science:

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

Why data types?

From mathematics to computer science:

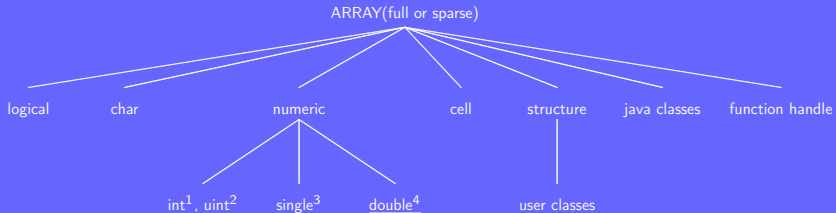
- Different numbers (integer, real, complex...)
- Different ranges (short, long...)
- Different precisions (single, double...)

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, and get $-a$
e.g. $00101010 \rightarrow 11010101 + 1 = 11010110$
 $00101010 = -0 \cdot 2^7 + 2^5 + 2^3 + 2 = 42$
 $11010110 = -1 \cdot 2^7 + 2^6 + 2^4 + 2^2 + 2 = 86 - 128 = -42$

Data types in MATLAB



1. int: int8, int16, int32 and int64
2. uint: uint8, 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')`, and e.g. `uint8(a)`

Type related functions

Type of a variable:

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

Numeric conversions: `cast(a, 'type')`, and e.g. `uint8(a)`

MATLAB string: array of characters, defined using single quotes

Useful string functions:

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

String parsing

Example.

Input two numbers as strings and calculate their sum

String parsing

Example.

Input two numbers as strings and calculate their sum

```
1 clear all, clc;  
2 numbers=input('Input two numbers: ', 's');  
3 space=strfind(numbers, ' ');  
4 number1=str2num(numbers(1:space-1));  
5 number2=str2num(numbers(space+1:end));  
6 number1+number2
```

Questions.

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

Binary file functions

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

Binary file functions

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.

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

Questions

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

Outline

- 1 Plotting
- 2 Data types
- 3 Structures

What is a structure?

Structure:

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

What is a structure?

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

Structures in MATLAB

① Initializing the structure

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

Structures in MATLAB

① Initializing the structure

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

② Using the structure

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

③ To go further: who got the best mark?

Structures in MATLAB

1 Initializing the structure

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

2 Using the structure

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

3 To go further: who got the best mark?

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

Key points

- Using `plot` draw simple geometrical shapes
- How to keep or erase previous graphs?
- How to measure the quality of a fit?
- Cite the most common data types and their size in bytes
- What is a data structure?

Chapter 5

Introduction to C

Outline

- 1 Before starting with C
- 2 From C to machine code
- 3 Functions and libraries

The birth of C

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

The birth of C

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

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

Development environment

Common software to write C code:

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

Development environment

Common software to write C code:

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

Common C compilers:

- GNU C Compiler: `gcc`
- Clang
- Intel C Compiler: `icc`

Development environment

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: `gcc`
- Clang
- Intel C Compiler: `icc`

Outline

- ① Before starting with C
- ② From C to machine code
- ③ Functions and libraries

A first example

gm-base.c

```
1  #include <stdio.h>
2  int main () {
3      printf("good morning!\n");
4      return 0;
5  }
```

Program structure:

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

A first example

gm-base.c

```
1  #include <stdio.h>
2  int main () {
3      printf("good morning!\n");
4      return 0;
5  }
```

Program structure:

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

Writing a C function:

```
1  OType FName(IType IName,...) {
2      function's body
3  }
```

Compilation: gcc gm-base.c -o gm-base

Blocks

Explain the following code:

blocks.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  int main () {
4      {
5          int a=0; printf("%d ",a);
6      }
7      {
8          double a=1.124; printf("%f ",a);
9      }
10     {
11         char a='a'; printf("%c ",a);
12     }
13     // printf("%d",a);
14 }
```

Shorthand operators

Questions.

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

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`

The `#include` instruction

Roles of a header file:

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

Syntax to include `header.h`:

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

The #include instruction

Roles of a header file:

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

Syntax to include header.h:

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

Result of `#include<stdio.h>`: `gcc -E gm-base.c`

The #define instruction

Goal:

- Set “type-less” read-only variables
- Hardcode values in the program
- Quickly alter hardcoded values over the whole file

The #define instruction

Goal:

- Set “type-less” read-only variables
- Hardcode values in the program
- Quickly alter hardcoded values over the whole file

gm-def.c

```
1  #include <stdio.h>
2  #define COURSE "VG101"
3  int main () {
4      printf("good morning %s!\n", COURSE);
5  }
```

Result of #define: 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

```
1 #include <stdio.h>
2 #define POLITE
3 int main () {
4 #ifdef POLITE
5     printf("good morning!\n");
6 #endif
7 }
```

gm-ifndef.c

```
1 #include <stdio.h>
2 int main () {
3 #ifndef RUDE
4     printf("good morning!\n");
5 #endif
6 }
```

Result of #if(n)def: gcc -E gm-if(n)def.c

More on #define

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

```
1  #include <stdio.h>
2  #define SPEAK(x) printf("good morning %s!\n",x)
3  int main () {
4      SPEAK("VG101");
5      SPEAK("VE475");
6  }
```

Result of macros: 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

Outline

- 1 Before starting with C
- 2 From C to machine code
- 3 Functions and libraries

More complex programs

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

```
1  #include <stdio.h>
2  double answer(double d);
3  int main () {
4      double a;
5      scanf("%lf",&a);
6      printf("%lf\n", answer(a));
7  }
8  double answer(double d) {return d+1337;}
```

A long program

ans-orig.c

```
1  #include <stdio.h>
2  double answer(double d);
3  int main () {
4      double a;
5      scanf("%lf",&a);
6      printf("%lf\n", answer(a));
7  }
8  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: `+`, `-`, `/`, `%`

Organising a long program

Splitting the code over several files:

ans-main.c

```
1  #include <stdio.h>
2  #include "ans.h"
3  int main () {
4      double a; scanf("%lf",&a); printf("%lf\n", answer(a));
5  }
```

ans.c

```
1  #include "ans.h"
2  double answer(double d) {
3      return d+1337;
4  }
```

ans.h

```
1  #ifndef ANS_H
2  #define ANS_H
3  double answer(double d);
4  #endif
```

Compilation: gcc ans-main.c ans.c -o ans

Libraries

Library: collection of functions, macros, data types and constants

Example.

The C mathematics library:

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

Libraries

Library: collection of functions, macros, data types and constants

Example.

The C mathematics library:

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

math.c

```
1  #include<stdio.h>
2  #include<math.h>
3  int main() {
4      printf("%g\n",gamma(sqrt(cosh(M_PI/2))));
5  }
```

Key points

- Why is C one of the most widely used programming language?
- Is C a compiled or interpreted language?
- How to transform a C program into machine code?
- Why are data types of a major importance?

Chapter 6

Basic C

Outline

① Data types

② Structures

③ Control statements

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

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 VG101

Initialising variables

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”

Basic data types

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)

Basic data types

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`

Data types

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.

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

Data types

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.

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

Characters:

- No type for strings, only for single 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)

Wrong data type

What output to expect?

types1.c

```
1  #include <stdio.h>
2  int main() {
3      printf("%d %f\n", 7/3, 7/3);
4  }
```

Wrong data type

What output to expect?

types1.c

```
1  #include <stdio.h>
2  int main() {
3      printf("%d %f\n", 7/3, 7/3);
4  }
```

Fixed version:

types2.c

```
1  #include <stdio.h>
2  int main() {
3      printf("%d %f\n", 7/3, 7.0/3);
4      int a=42; char b=(char) a;
5      printf("%c\n", b);
6  }
```

Wrong data types

Understanding the code:

- What do %f, %d and %c mean?
- What is the type of 7/3 for the compiler?
- What is displayed for b?
- What is this character corresponding to?
- Why is this character displayed?

Type casting

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

Type casting

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

```
1  #include <stdio.h>
2  int main() {
3      float c=4.8; printf("%d\n", (int)c);
4      int f=42; printf("%c\n", (char)f);
5      double a=487511234.7103;
6      char b=(char) a;
7      printf("%c, %c\n",b,a);
8      int d=311;
9      float e=(float) d;
10     printf("%d %f\n",d,e);
11     printf("%c\n",d);
12 }
```

Example

Understanding the code:

- Which type casting 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`?

Example

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

Example

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

apbp1.c

```
1  #include <stdio.h>
2  int apbp1 (float a, float b);
3  int main () {
4      float a, b;
5      scanf("%f %f", &a,&b);
6      printf("%d\n", apbp1(a,b));
7  }
8  int apbp1 (float a, float b) {
9      a++; a+=b;
10     return((int) (a+0.5));
11 }
```

Question. How are shorthand operators and type casting used?

Outline

① Data types

② Structures

③ Control statements

What, why data types?

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

Structures

st.c

```
1  #include <stdio.h>
2  typedef struct _person {
3      char* name;
4      int age;
5  } person;
6  int main () {
7      person al={"albert",32};
8      person gil;
9      gil.name="gilbert";
10     gil.age=23;
11     struct _person so={"sophie",56};
12     printf("%s %d\n",al.name, al.age);
13     printf("%s %d\n",gil.name, gil.age);
14     printf("%s %d\n",so.name, so.age);
15 }
```

Structures

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?

Functions and structures

st-fct.c

```
1  #include <stdio.h>
2  typedef struct person {
3      char* name; int age;
4  } person_t;
5  person_t older(person_t p, int a);
6  int main () {
7      person_t al={"albert",32};
8      al=older(al,10);
9      printf("%s %d\n",al.name,al.age);
10 }
11 person_t older(person_t p, int a) {
12     printf("%s %d\n",p.name, p.age);
13     p.age=p.age+a;
14     return p;
15 }
```

Functions and structures

Understanding the code:

- How is the age increased?
- How are 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?

Outline

- ① Data types
- ② Structures
- ③ Control statements

Jumping!

jump.c

```
1  #include <stdio.h>
2  int main() {
3      int i=0;
4      printf("I am at position %d\n",i);
5      i++;
6      goto end;
7      printf("I am at position %d\n",i);
8      end:
9          i++;
10         printf("It all ends here, at position %d\n",i);
11         return 0;
12         i++;
13         printf("Unless it's here at position %d\n",i);
14     }
```

Jumping!

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?

Important operators

Basics on conditional statements:

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

Important operators

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: ?:

```
1 condition ? expression1 : expression2
```

Important operators

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: ?:

```
1 condition ? expression1 : expression2
```

Example.

A macro returning the max of two numbers:

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


The if and switch statements

```
1  if (condition) {  
2      statements;  
3  }  
4  else {  
5      statements;  
6  }
```

```
1  switch(variable) {  
2      case value1:  
3          statements;  
4          break;  
5      case value2:  
6          statements;  
7          break;  
8      default:  
9          statements;  
10         break;  
11 }
```

Example

cards.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4  #define ACE 14
5  #define KING 13
6  #define QUEEN 12
7  #define JACK 11
8  int main () {
9      int c;
10     srand(time(NULL)); c=rand()%13+2;
11     switch (c) {
12         case ACE: printf("Ace\n"); break;
13         case KING: printf("King\n"); break;
14         case QUEEN: printf("Queen\n"); break;
15         case JACK: printf("Jack\n"); break;
16         default: printf("%d\n",c); break;
17     }
18 }
```

Example

Understanding the code:

- 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

The while and do... while statements

Structure of a while loop:

```
1 while (conditions) {  
2     statements;  
3 }
```

Structure of a do... while loop:

```
1 do {  
2     statements;  
3 } while (conditions);
```

The while and do... while statements

Structure of a while loop:

```
1 while (conditions) {  
2     statements;  
3 }
```

Structure of a do... while loop:

```
1 do {  
2     statements;  
3 } while (conditions);
```

Example.

```
1 int i=0;  
2 while(i++<3) {  
3     printf("%d",i);  
4 }
```

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

Questions.

- What is the difference between the two outputs?
- What happens if ++i and i++ are changed?

The for statement

Structure of a for loop:

```
1 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

The for statement

Structure of a for loop:

```
1  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.

```
1  for(i=0; i<n; i++)
2      printf("%d ", i);
3  i=0; for(;i<n;i++)
4      printf("%d ", i);
5  for(i=0; i<n;)
6      {printf("%d\n",i); i++;}
7  for(i=0;i<n;)
8      printf("%d ",i++);
```

```
1  fct=1;
2  for(i=1;i<=n;i++) fct*=i;
3  printf("%d ", fct);
4  for(i=1,fct=1;i<=n;fct*=i,i++);
5  printf("%d ", fct);
6  for(i=1,fct=1;i<=n;fct*=i++);
7  printf("%d\n", fct);
```

The break and continue statements

Questions.

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

The break and continue statements

Questions.

- 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.

```
1  for(i=0;i<10;i++) {  
2      scanf("%d",&n);  
3      if(n==0) break;  
4      else if(n>=10) continue;  
5      printf("%d\n", n);  
6  }
```

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 perform conditional statements in C?
- How to write loops in C?

Chapter 7

Arrays and pointers

Outline

① Arrays

② Pointers

③ Pointers and arrays

Array definition

In C an array is defined by three parameters: its name, the data type of its content, and its size

Example.

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

How to:

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

Array definition

In C an array is defined by three parameters: its name, the data type of its content, and its size

Example.

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

How to:

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

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

Arrays and functions

array-fct.c

```
1  #include <stdio.h>
2  double average(int arr[], size_t size);
3  int main () {
4      int elem[5]={1000, 2, 3, 17, 50};
5      printf("%lf\n",average(elem,5));
6  }
7  double average(int arr[], size_t size) {
8      unsigned long i;
9      double avg, sum=0;
10     for (i = 0; i < size; ++i) {
11         sum += arr[i];
12     }
13     avg = sum / size;
14     return avg;
15 }
```

Arrays and functions

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?

From one die to two dice

Understand the following code and adapt it to handle two dice.

die.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIDES 6
5  #define ROLLS 1000
6  int main () {
7      int i, tab[SIDES];
8      srand(time(NULL));
9      for (i=0; i < SIDES; i++) tab[i]=0;
10     for (i=0; i < ROLLS; i++) tab[rand()%SIDES]++;
11     for (i=0;i<SIDES;i++) printf ("%d (%d)\t",i+1,tab[i]);
12     printf("\n");
13 }
```

Question. How is the array initialized?

More dice

dice.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define DICE 4
5  #define SIDES 10
6  #define ROLLS 100000
7  int main () {
8      int i, j, t, res[DICE*SIDES-DICE+1]={0};
9      srand(time(NULL));
10     for (i=0; i < ROLLS; i++) {
11         t=0;
12         for(j=0;j<DICE;j++) t+=rand()%SIDES;
13         res[t]++;
14     }
15     for (i=0;i<DICE*SIDES-DICE+1;i++) {
16         printf("%d (%d)  ",i+DICE,res[i]);
17     }
18     printf("\n");
19 }
```

More dice

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?

Multidimensional arrays

dice-m.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #define DICE 10
6  #define SIDES 6
7  #define ROLLS 100000
8  int main () {
9      int i, j, t, table[DICE][ROLLS], res[DICE*SIDES-DICE+1];
10     srand(time(NULL));
11     memset(res, 0, (DICE*SIDES-DICE+1)*sizeof(int));
12     for(i=0; i<DICE; i++)
13         for (j=0; j < ROLLS; j++) table[i][j]=(rand()%SIDES)+1;
14     for (i=0; i<ROLLS; i++) {
15         t=0;
16         for(j=0; j<DICE; j++) t+=table[j][i];
17         res[t-DICE]++;
18     }
19     for (i=0; i<DICE*SIDES-DICE+1; i++) printf("%d (%d) ", i+DICE, res[i]);
20     printf("\n");
21 }
```

Summary questions

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 (7.191) using a function taking dice, sides, and rolls as input
- Explain how multi-dimensional arrays are stored in the memory

Outline

- 1 Arrays
- 2 Pointers
- 3 Pointers and arrays

What is a pointer?

Pointer:

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

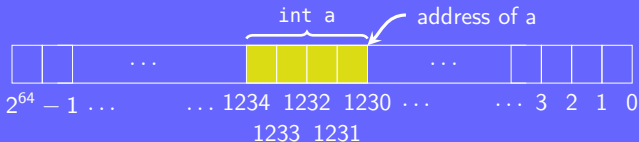
What is a pointer?

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



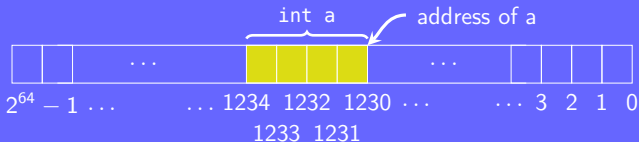
What is a pointer?

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

How to use pointers

Handling pointers:

- If a variable x is defined, then its address is $\&x$
- If the address of a variable is x , then the value stored at this address is $*x$;
- The operator “ $*$ ” is called *dereferencing* operator

How to use pointers

Handling pointers:

- If a variable `x` is defined, then its address is `&x`
- If the address of a variable is `x`, then the value stored at this address is `*x`;
- 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

```
1  #include <stdio.h>
2  void swap(int a,int b);
3  int main() {
4      int a=2, b=5;
5      swap(a,b);
6      printf("a = %d, ",a);
7      printf("b = %d\n",b);
8      return 0;
9  }
10 void swap(int a,int b) {
11     int temp=a;
12     a=b;
13     b=temp;
14 }
```

swap-p.c

```
1  #include <stdio.h>
2  void swap(int *a, int *b);
3  int main() {
4      int a=2, b=5;
5      swap(&a,&b);
6      printf("a = %d, ",a);
7      printf("b = %d\n",b);
8      return 0;
9  }
10 void swap(int* a,int* b) {
11     int temp=*a;
12     *a=*b;
13     *b=temp;
14 }
```

Why using pointers?

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

```
1  #include <stdio.h>
2  void pointers();
3  int main() {pointers();}
4  void pointers() {
5      float x=0.5; float *xp1;
6      float **xp2 = &xp1; xp1 = &x;
7      printf("%llu %p\n%f ",xp1,&x,**xp2);
8      x=**xp2+*xp1; printf("%f\n",x);
9  }
```

Questions.

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

Dynamic memory

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)`

Dynamic memory

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

Accessing memory

Example.

```
1 int *a=malloc(6*sizeof(int));
```

- Accessing first chunk
- Accessing the 5th chunk

Accessing memory

Example.

```
1 int *a=malloc(6*sizeof(int));
```

- Accessing first chunk

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

- Accessing the 5th chunk

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

Accessing memory

Example.

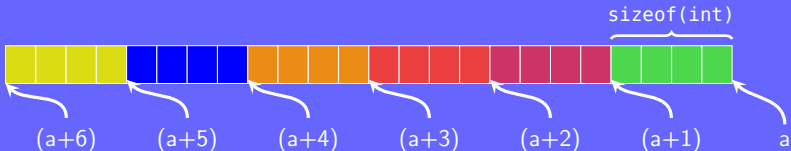
```
1 int *a=malloc(6*sizeof(int));
```

- Accessing first chunk

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

- Accessing the 5th chunk

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



Question. What is $(a+6)$?

Pointers and structures

str-p.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct person {
4      char* name; int age;
5  } person_t;
6  int main () {
7      person_t al={"albert",32};
8      person_t* group1=malloc(3*sizeof(person_t));
9      group1->name="gilbert";
10     group1->age=34;
11     *(group1+1)=(person_t){ "joseph",28};
12     (*(group1+2)).name="emily";
13     (group1+2)->age=42;
14     printf("%s %d %lu\n",al.name, al.age, sizeof(person_t));
15     printf("%s %d\n",(group1+1)->name, (group1+2)->age);
16     free(group1);
17     return 0;
18 }
```

Pointers and structures

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?

General notes

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;`)

General notes

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:

```
1 char* p = malloc(100);
2 if (p == NULL) {
3     fprintf(stderr, "Error: out of memory");
4     exit(1);
5 }
```

Outline

- ① Arrays
- ② Pointers
- ③ Pointers and arrays

Pointer vs. array

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

arr-ptr.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void ptr_vs_arr();
4  int main () {
5      ptr_vs_arr();
6  }
7  void ptr_vs_arr(){
8      int a[3]={0,1,2};
9      int* p=malloc(3*sizeof(int));
10     *p=3; *(p+1)=4; *(p+2)=5; printf("%d %d\n",a[0], *p);
11     a[0]=42; p=a; p++ ; *p=a[2];
12     //a=p; p=c; p=a[0]; p=&a; a++;
13     printf("%d %d %lu %lu\n",a[0], *p,sizeof(a), sizeof(p));
14 }
```

Pointers and strings

A pointer to char is different from an array of char

str-ptr.c

```
1  #include <stdio.h>
2  void str_ptr();
3  int main () {
4      str_ptr();
5  }
6  void str_ptr(){
7      char a[]="good morning!";
8      char* p="Good morning!";
9      printf("%c %c\n",a[0], *p);
10     a[0]='t'; /*p='t';
11     p=a;//a=p; p=c; p=a[0]; p=&a;
12     p++; //a++;
13     printf("%c %c %lu %lu\n",a[0], *p,sizeof(a), sizeof(p));
14 }
```

Arrays as pointers

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

Arrays as pointers

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

arr_ptr2.c

```
1  #include <stdio.h>
2  void arr_as_ptr(){
3      int a[4]={1, 2, 3, 4};
4      for(int i=0;i<4;i++) {
5          printf("&a[%d]=%p (a+%d)=%p\n"\  
6              "a[%d]=%d *(a+%d)=%d\n",\  
7              i,&a[i],i,(a+i),i,a[i],i,*(a+i));
8      }
9  }
10 int main () {arr_as_ptr();}
```

Arrays and pointers

Questions on 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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  void roll_dice(int dice, int sides, int rolls){
5      int i, j, t;
6      int *res=calloc((dice*sides-dice+1),sizeof(int));
7      int *table=malloc(dice*rolls*sizeof(int));
8      for(i=0;i<rolls;i++) {
9          for (j=0; j < dice; j++) table[i*dice+j]=(rand()%sides)+1;
10     }
11     for (i=0;i<rolls;i++) {
12         t=0; for(j=0;j<dice;j++) t+=table[i*dice+j]; res[t-dice]++;
13     }
14     for (i=0;i<dice*sides-dice+1;i++) printf("%d (%d) ",i+dice,res[i]);
15     printf("\n"); free(table); free(res);
16 }
17 int main () {
18     int dice=4, sides=6, rolls=10000000;
19     srand(time(NULL)); roll_dice(dice,sides,rolls);
20 }
```

Revisiting the dice

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?

Summary on pointers

Limitation of C:

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

Summary on pointers

Limitation of C:

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

Not an issue: use pointers as input (slide 7.196)

Summary on pointers

Limitation of C:

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

Not an issue: use pointers as input (slide 7.196)

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?

Chapter 8

Algorithms and efficiency

Outline

- ① Algorithms
- ② Standard library
- ③ A few final examples

What is already known

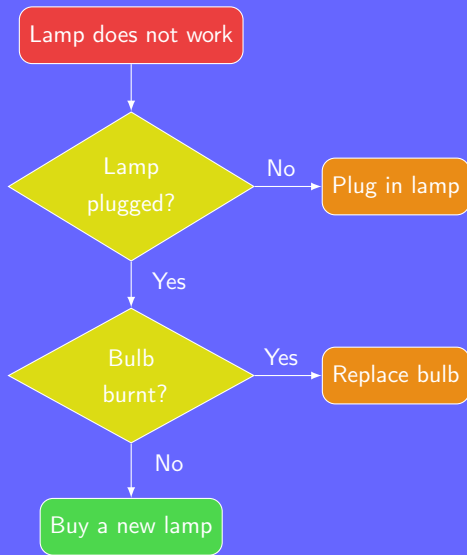
Reminders:

- 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

What is already known

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



Design paradigms

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?

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

Outline

- ① Algorithms
- ② Standard library
- ③ A few final examples

Moving in a file:

- Open a file: `FILE *fopen(const char *path, const char *mode);` where mode is one of `r`, `r+`, `w`, `w+`, `a`, `a+`; `NULL` returned on error
- Close a file: `int fclose(FILE *fp);` return 0 upon successful completion
- Seek in a file: `int fseek(FILE *stream, long offset, int whence);` where whence can be set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`
- Current position: `long ftell(FILE *stream);`
- 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 int 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 in a string:
`char *strchr(const char *s, int c);`

<string.h> and <time.h>

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);
```

<string.h> and <time.h>

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 time difference:
`double difftime(time_t time1, time_t time0);`

<ctype.h> and <math.h>

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);`

<ctype.h> and <math.h>

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);`

<stdlib.h>

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
*));`

Outline

- ① Algorithms
- ② Standard library
- ③ A few final examples

Linear search

linear-search.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIZE 200
5  #define MAX 1000
6  int main () {
7      int i, n, k=0;
8      int data[SIZE];
9      srand(time(NULL));
10     for(i=0; i<SIZE; i++) data[i]=rand()%MAX;
11     n=rand()%MAX;
12     for(i=0; i<SIZE; i++) {
13         if(data[i]==n) {
14             printf("%d found at position %d\n",n,i);
15             k++;
16         }
17     }
18     if(k==0) printf("%d not found\n",n);
19 }
```

Linear search

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

binary-search.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIZE 200
5  int main () {
6      int i, n, k=0, low=0, high=SIZE-1, mid;
7      int *data=malloc(SIZE*sizeof(int));
8      srand(time(NULL));
9      for(i=0;i<SIZE;i++) *(data+i)=2*i;
10     n=rand()%(data+i-1);
11     while(high >= low) {
12         mid=(low + high)/2;
13         if(n < *(data+mid)) high = mid - 1;
14         else if(n> *(data+mid)) low = mid + 1;
15         else {printf("%d found at position %d\n",n,mid);
16             free(data); exit(0);}
17     }
18     printf("%d not found\n",n);
19     free(data);
20 }
```


Binary search

Using the previous code:

- Write a clear algorithm for the 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

selection-sort.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIZE 200
5  #define MAX 1000
6  int main () {
7      int data[SIZE];
8      srand(time(NULL));
9      for(int i=0; i<SIZE; i++) data[i]=rand()%MAX;
10     for(int i=0; i<SIZE; i++) {
11         int t, min = i;
12         for(int j=i; j<SIZE; j++) if(data[min]>data[j]) min = j;
13         t = data[i];
14         data[i] = data[min];
15         data[min] = t;
16     }
17     printf("Sorted array: ");
18     for(int i=0; i<SIZE; i++) printf("%d ",data[i]);
19     printf("\n");
20 }
```

Selection sort

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?

Chapter 9

Introduction to C++

Outline

① Before starting with C++

② From C to C++

③ 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



C++ in a few words

Simple description:

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

C++ in a few words

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

Outline

① Before starting with C++

② From C to C++

③ Basic C++

C vs. C++

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

C vs. C++

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

```
1  #include <stdio.h>
2
3  int main () {
4      int a=5;
5      printf("%d\n",a);
6  }
```

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”

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 can focus more on the problem rather than on how to explain it to the computer

Basics

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

Basics

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

A small difference between C and C++:

C

Implicit assignment from `*void`

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

C++

No implicit assignment from `*void`

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


Outline

- ① Before starting with C++
- ② From C to C++
- ③ Basic C++

New data type and headers

New in C++:

- New datatype:
- New headers:

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

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

New data type and headers

New in C++:

- New datatype:

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

- New headers:

```
1 #include <iostream>
2 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`

New input/output style

Handling I/O without `printf` and `scanf`:

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

New input/output style

Handling I/O without printf and scanf:

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

Example.

input-pb.cpp

```
1  #include <iostream>
2  using namespace std;
3  void TestInput(){
4      int x = 0;
5      do {
6          cout << "Enter a number (-1 to quit): "; cin >> x;
7          if(x != -1) cout << x << " was entered" << endl;
8      } while(x != -1);
9      cout << "Exit" << endl;
10 }
11 int main() {TestInput(); return 0;}
```

Input

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

input-ok1.cpp

```
1  #include <iostream>
2  using namespace std;
3  void TestInput(){
4      int x = 0;
5      do {
6          cout << "Enter a number (-1 to quit): ";
7          if(!(cin >> x)) {
8              cout << "The input stream broke!" << endl;
9              x = -1;
10         }
11         if(x != -1) cout << x << " was entered" << endl;
12     } while(x != -1);
13     cout << "Exit" << endl;
14 }
15 int main() {TestInput(); return 0;}
```

Input

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

input-ok2.cpp

```
1  #include <iostream>
2  using namespace std;
3  void TestInput(){
4      int x=0;
5      do {
6          cout << "Enter a number (-1 to quit): ";
7          cin >> x;
8          cin.clear();
9          cin.ignore(10000, '\n');
10         if(x != -1) cout << x << " was entered" << endl;
11     } while(x != -1);
12     cout << "Exit" << endl;
13 }
14 int main() {TestInput(); return 0;}
```

Formatting output

Nicer display:

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

Formatting output

Nicer display:

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

Example.

date.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4  void showDate(int m, int d, int y) {
5      cout.fill('0');
6      cout << setw(2) << m << '/' << setw(2) << d << '/' << setw(4) << y << endl;
7  }
8  int main(){
9      showDate(6,19,2014);
10     cout << setprecision(3) << 1.2249 << endl;
11     cout << setprecision(3) << 1.22549 << endl;
12 }
```

Operator and function overloading

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

Operator and function overloading

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

```
1  #include <iostream>
2  using namespace std;
3  double f(double a);
4  int f(int a);
5  int main () {cout << f(2) << endl; cout << f(2.3) << endl;}
6  double f(double a) {return a;}
7  int f(int a) {return a;}
```

Pointers

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`

Pointers

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

Strings

Improvements on strings:

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

Strings

Improvements on strings:

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

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

Strings

Improvements on strings:

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

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

More possibilities: search and learn how to use strings in C++

File I/O

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)`

File I/O

Example.

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

File I/O

Example.

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

fio.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std;
5  void FileIO() {
6      string s;
7      ifstream a("1.txt"); ofstream b("2.txt");
8      while(getline(a,s)) {b << s << endl;  cout << s;}
9  }
10 int main () {FileIO();return 0;}
```

File I/O

What was wrong with the previous code?

fio-c.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std;
5  void FileIO(){
6      string s;
7      ifstream a("1.txt"); ofstream b("2.txt",ios::app);
8      if (a.is_open() && b.is_open()) {
9          while(getline(a,s)) {b << s << endl; cout << s;}
10         b.close(); a.close();
11     }
12     else cerr << "Unable to open the file(s)\n";
13 }
14 int main () {FileIO();return 0;}
```

Defining constants

C

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

C++

- `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

C

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

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 size of the program

```
1 inline int sq(int x) { return x*x; }
```

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?

Chapter 10

Object oriented programming

Outline

① Classes and objects

② Inheritance

③ Polymorphism

Procedural programming

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

Object oriented programming

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:

- The data it contains, what is known to the object, its attributes or data members
- The behavior it has, what can be done by the object, its methods or function members

Object

An object has two main components:

- The data it contains, what is known to the object, its attributes or data members
- The behavior it has, what can be done by the object, its methods or function members

Example.

Given a simple TV:

- Methods: high level actions (e.g. on/off, channel, volume) and low level actions (e.g. on internal electronics components)
- Attributes: buttons and internal electronics components

Class and instance

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 TVs (same model/manufacturer) are two instances from a same class

Class specification

Oder of definition:

- ① Define the methods
- ② Define the attributes

Class specification

Oder of definition:

- ① Define the methods
- ② Define the attributes

Example.

Create an object `circle`:

- ① What is requested (methods):
 - `move`
 - `zoom`
 - `area`
- ② How to achieve it (attributes):
 - Position of the center (x, y)
 - Radius of the circle

Class interface

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/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 code
- Accessing private attributes is forbidden: more secure

In C++ the default behavior is to set everything as private and only render a member public when necessary

Circle interface

Example.

circle-v0.h

```
1  class Circle {  
2  /* user methods (and attributes)*/  
3  public:  
4      void move(float dx, float dy);  
5      void zoom(float scale);  
6      float area();  
7  /* implementation attributes (and methods) */  
8  private:  
9      float x, y, r;  
10 };
```

Circle interface

Example.

circle-v0.h

```
1  class Circle {  
2  /* user methods (and attributes)*/  
3      public:  
4          void move(float dx, float dy);  
5          void zoom(float scale);  
6          float area();  
7  /* implementation attributes (and methods) */  
8      private:  
9          float x, y, r;  
10 };
```

Questions.

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

Class instantiation

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`

Circle instantiation

Example.

circle-main-v0.cpp

```
1  #include <iostream>
2  #include "circle_v0.h"
3  using namespace std;
4  int main () {
5      float s1, s2;
6      Circle circ1, circ2;
7      circ1.move(12,0);
8      s1=circ1.area(); s2=circ2.area();
9      cout << "area: " << s1 << endl;
10     cout << "area: " << s2 << endl;
11     circ1.zoom(2.5); s1=circ1.area();
12     cout << "area: " << s1 << endl;
13 }
```

Circle instantiation

Example.

circle-main-v0.cpp

```
1  #include <iostream>
2  #include "circle_v0.h"
3  using namespace std;
4  int main () {
5      float s1, s2;
6      Circle circ1, circ2;
7      circ1.move(12,0);
8      s1=circ1.area(); s2=circ2.area();
9      cout << "area: " << s1 << endl;
10     cout << "area: " << s2 << endl;
11     circ1.zoom(2.5); s1=circ1.area();
12     cout << "area: " << s1 << endl;
13 }
```

Question. Why is this code not compiling?

Class implementation

Getting things ready:

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

Example

Example.

circle-v0.cpp

```
1  #include "circle_v0.h"
2  static const float PI=3.1415926535;
3  void Circle::move(float dx, float dy) {
4      x += dx;
5      y += dy;
6  }
7  void Circle::zoom(float scale) {
8      r *= scale;
9  }
10 float Circle::area() {
11     return PI * r * r;
12 }
```

Example

Example.

circle-v0.cpp

```
1  #include "circle_v0.h"
2  static const float PI=3.1415926535;
3  void Circle::move(float dx, float dy) {
4      x += dx;
5      y += dy;
6  }
7  void Circle::zoom(float scale) {
8      r *= scale;
9  }
10 float Circle::area() {
11     return PI * r * r;
12 }
```

Question. Can this file be compiled alone?

Constructor and destructor

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

```
1  class Circle {
2  /* user methods (and attributes)*/
3      public:
4          Circle();
5          Circle(float r);
6          ~Circle();
7          void move(float dx, float dy);
8          void zoom(float scale);
9          float area();
10 /* implementation attributes (and methods) */
11     private:
12         float x, y;
13         float r;
14 };
```

Example

circle-v1.cpp

```
1  #include "circle_v1.h"
2  static const float PI=3.1415926535;
3  Circle::Circle() {
4      x=y=0.0; r=1.0;
5  }
6  Circle::Circle(float radius) {
7      x=y=0.0; r=radius;
8  }
9  Circle::~~Circle() {}
10 void Circle::move(float dx, float dy) {
11     x += dx; y += dy;
12 }
13 void Circle::zoom(float scale) {
14     r *= scale;
15 }
16 float Circle::area() {
17     return PI * r * r;
18 }
```

Example

circle-main-v1.cpp

```
1  #include <iostream>
2  #include "circle_v1.h"
3  using namespace std;
4  int main () {
5      float s1, s2;
6      Circle circ1, circ2((float)3.1);
7      circ1.move(12,0);
8      s1=circ1.area(); s2=circ2.area();
9      cout << "area: " << s1 << endl;
10     cout << "area: " << s2 << endl;
11     circ1.zoom(2.5);
12     // cout << circ1.r << endl;
13     s1=circ1.area();
14     cout << "area: " << s1 << endl;
15 }
```

Overloading

Better definitions:

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

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

```
1 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).

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

Problem

Rewrite the `main.cpp` file using two pointers: one for the two circles and one for their areas. The pointers should be initialised in the `main` function while all the rest of the work is performed in another function.

Solution

main-ptr.cpp

```
1  #include <iostream>
2  #include "circle_v1.h"
3  using namespace std;
4  void FctCirc(Circle *circ, float *s) {
5      *(circ+1)=Circle(3.1);
6      *s=circ->area(); s[1]=circ[1].area();
7      cout << "area: " << s[0] << endl;
8      cout << "area: " << *(s+1) << endl;
9      circ[0].zoom(2.5); *s=circ->area();
10     cout << "area: " << s[0] << endl;
11 }
12 int main () {
13     float *s=new float[2]; Circle *circ; circ=new Circle[2];
14     FctCirc(circ,s);
15     delete[] s; delete[] circ; return 0;
16 }
```

Outline

① Classes and objects

② Inheritance

③ Polymorphism

Why using 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

Managing a cow

In this section we will construct a zoo and work with cows...

cows-0.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4  public:
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() {
7          if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8          else cout << "I'm hungry\n";}
9      Cow(int f=0){grass=f;}
10 private: int grass;
11 };
12 int main () {
13     Cow c1(1);
14     c1.Speak(); c1.Eat(); c1.Eat();
15 }
```

Managing a sick cow

A sick cow does:

- Everything a cow does
- Take its medication

Managing 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()`

Question. What are the limitations of those strategies?

Managing 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()`

Question. What are the limitations of those strategies?

Inheritance: a sick cow inherits all the attributes and methods of a cow, and more can be freely added

Managing a sick cow

cows-1.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4  public: Cow(int f=0){grass=f;}
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() {
7          if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8          else cout << "I'm hungry\n";}
9  private: int grass;
10 };
11 class SickCow : public Cow {
12 public: SickCow(int f=0,int m=0){grass=f; med=m;}
13     void TakeMed() {
14         if(med > 0) { med--; cout << "I feel better\n";}
15         else cout << "I'm dying\n";}
16 private: int med;
17 };
18 int main () {
19     Cow c1(1); SickCow c2(1,1);
20     c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
21 }
```


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

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

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
- As much as possible do not use friend

Summary on visibility

Attributes and methods:

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

Summary on visibility

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

In practice mainly public inheritance is used.

Properly managing a sick cow

cows-2.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4  public: Cow(int f=0){grass=f;}
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() {
7          if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8          else cout << "I'm hungry\n";}
9      protected: int grass;
10 };
11 class SickCow : public Cow {
12 public: SickCow(int f=0,int m=0){grass=f; med=m;}
13     void TakeMed() {
14         if(med > 0) { med--; cout << "I feel better\n";}
15         else cout << "I'm dying\n";}
16     private: int med;
17 };
18 int main () {
19     Cow c1(1); SickCow c2(1,1);
20     c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
21 }
```

Inheritance or not inheritance?

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

```
1 class Cow : public Mammal {  
2     ...  
3 }
```

```
1 class Zoo {  
2     public:  
3         Mammal *m; Reptile *r;  
4         ...  
5     };
```

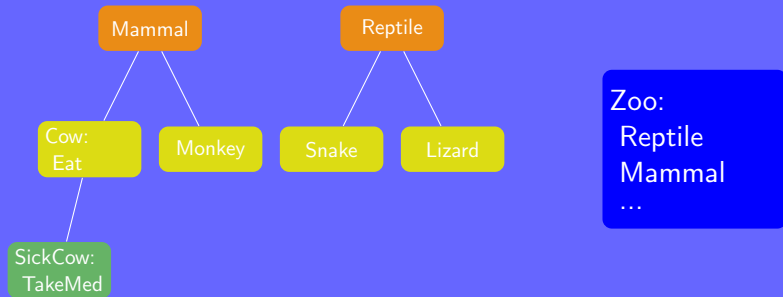
Remark.

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*?

Diagram

Representing the relationships using diagrams:



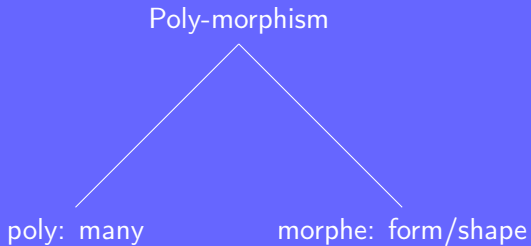
Outline

① Classes and objects

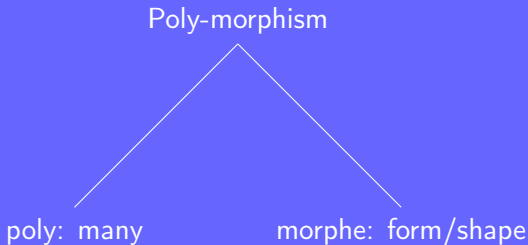
② Inheritance

③ Polymorphism

Polywhat????



Polywhat????



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

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4  public: Cow(int f=0){grass=f;}
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
7                  else cout << "I'm hungry\n";}
8  protected: int grass;
9  };
10 class SickCow : public Cow {
11 public: SickCow(int f=0,int m=0){grass=f; med=m;}
12     void Speak () { cout << "Ahem... Moo.\n"; }
13     void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
14                     else cout << "I'm dying\n";}
15 private: int med;
16 };
17 int main () {
18     Cow c1; SickCow c2(1); Cow *c3=&c2;
19     c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
20 }
```

Overcoming the limitations

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

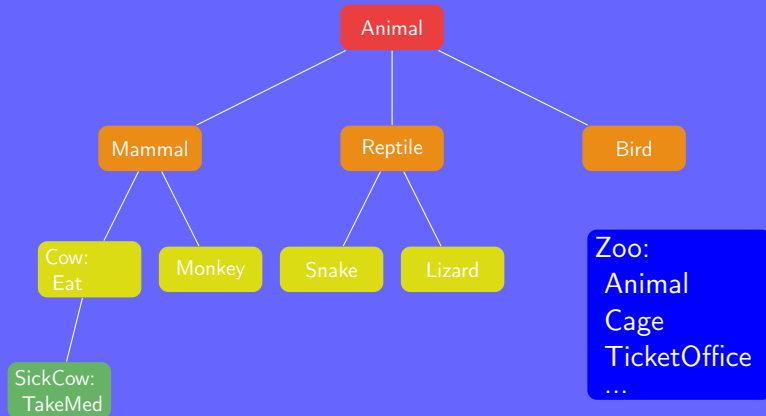
Fixing the cows

cows-4.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4  public: Cow(int f=0){grass=f;}
5      virtual void Speak () { cout << "Moo.\n"; }
6      void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
7                  else cout << "I'm hungry\n";}
8  protected: int grass;
9  };
10 class SickCow : public Cow {
11 public: SickCow(int f=0,int m=0){grass=f; med=m;}
12     void Speak () { cout << "Ahem... Moo.\n"; }
13     void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
14                     else cout << "I'm dying\n";}
15 private: int med;
16 };
17 int main () {
18     Cow c1; SickCow c2(1); Cow *c3=&c2;
19     c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
20 }
```

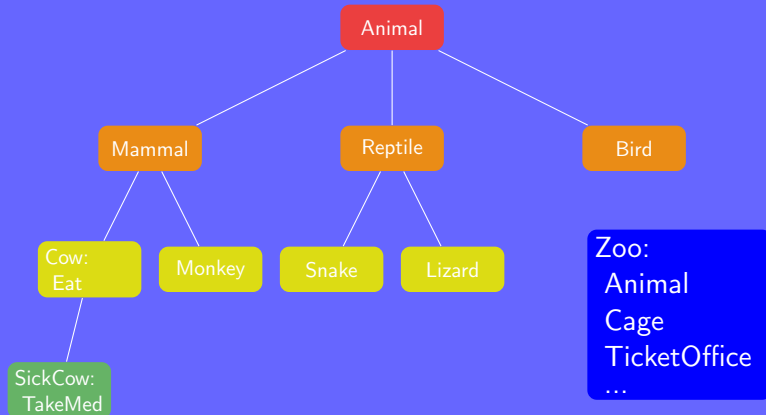
Extending the idea

Applying the same idea to generalize the diagram:



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

Pure virtual methods

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.

```
1  class Animal {  
2      public:  
3          virtual void Speak() = 0;  
4  }
```

Animals

animals.h

```
1  class Animal {
2      public:
3          virtual void Speak() = 0;
4          virtual void Eat() = 0;
5  };
6  class Cow : public Animal {
7      public:
8          Cow(int f=0); virtual void Speak(); void Eat();
9      protected: int grass;
10 };
11 class SickCow : public Cow {
12     public:
13         SickCow(int f=0,int m=0); void Speak(); void TakeMed();
14     private: int med;
15 };
16 class Monkey : public Animal {
17     public:
18         Monkey(int f=0); void Speak(); void Eat();
19     protected: int banana;
20 };
```

Animals

animals.cpp

```
1  #include <iostream>
2  #include "animals.h"
3  using namespace std;
4  Cow::Cow(int f) {grass=f;}
5  void Cow::Speak() { cout << "Moo.\n"; }
6  void Cow::Eat(){
7      if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8      else cout << "I'm hungry\n";
9  }
10 SickCow::SickCow(int f,int m) {grass=f; med=m;}
11 void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
12 void SickCow::TakeMed() {
13     if(med > 0) { med--; cout << "I feel better\n";}
14     else cout << "I'm dying\n";
15 }
16 Monkey::Monkey(int f) {banana=f;}
17 void Monkey::Speak() { cout << "Hoo hoo hoo hoo\n";}
18 void Monkey::Eat() {
19     if(banana > 0) {banana--; cout << "Give me another banana!\n";}
20     else cout << "Who took my banana?\n";
21 }
```

Zoo

zoo.h

```
1  #include <iostream>
2  #include <string>
3  #include "animals.h"
4  using namespace std;
5  class Employee {
6  public:
7      void setName(string n); string getName();
8  private:
9      string name;
10 };
11 class Tamer : public Employee {
12 public: void Feed(Animal *a);
13 };
14 class Zoo {
15 public:
16     Zoo(int s);
17     ~Zoo();
18     int getSize(); Tamer* getTamer(); Animal *getAnimal(int i);
19 private:
20     int size; Animal **a; Tamer *g;
21 };
```

zoo.cpp

```
1  #include <iostream>
2  #include "zoo.h"
3  void Employee::setName(string n) { name=n; }
4  string Employee::getName() { return name; }
5  void Tamer::Feed(Animal *a) {a->Speak(); a->Eat();}
6  Zoo::Zoo(int s) {
7      size=s; a=new Animal*[size]; g=new Tamer;
8      for(int i=0; i<size; i++) {
9          switch(i%4) {
10             case 0: a[i]=new Cow; break; case 1: a[i]=new SickCow; break;
11             case 2: a[i]=new Monkey; break; case 3: a[i]=new Monkey(1); break;
12          }
13      }
14  }
15  Zoo::~~Zoo() {
16      for(int i=0; i<size; i++) delete a[i];
17      delete[] a; delete g;
18  }
19  int Zoo::getSize() { return size; };
20  Tamer* Zoo::getTamer() { return g; }
21  Animal *Zoo::getAnimal(int i) {return a[i];}
```

Benefits of polymorphism

zoo-main.cpp

```
1  #include <iostream>
2  #include "zoo.h"
3  int main () {
4      Zoo z(10); z.getTamer()->setName("Mike");
5      cout << "Hi " << z.getTamer()->getName()
6          << ", please feed the animals.\n";
7      for(int i=0; i<z.getSize(); i++) {
8          cout << endl;
9          z.getTamer()->Feed(z.getAnimal(i));
10     }
11 }
```

Question. Think how to many lines of code are necessary to achieve the same result without inheritance and polymorphism...

Benefits of polymorphism

Understanding the code:

- Explain the benefits of polymorphism
- Why is the Zoo destructor not empty?
- Is it possible to instantiate an Animal?
- Adapt the previous classes and main function to add:
 - Cages that can be locked and unlocked
 - A vet and more guards
 - A boss, who gives orders while other employees do the real work (feed, give medication, open cages...)
 - Visitors who can watch the animals, get a fine if they feed the animals...
 - If an animal escapes there is an emergency announcement and the zoo closes

Multiple inheritance

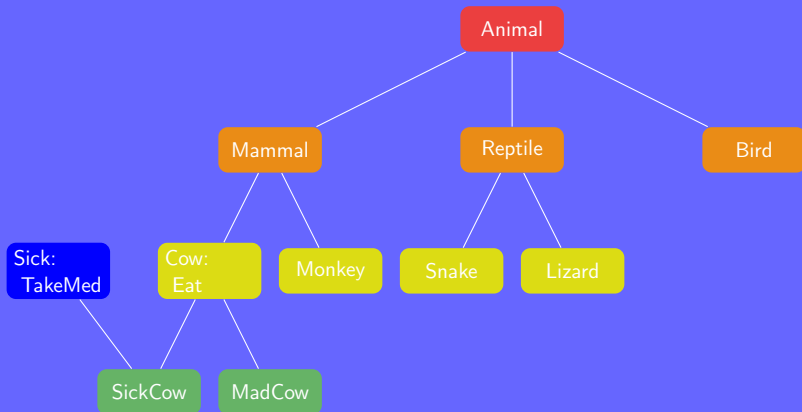
With multiple inheritance, a class can inherit from several classes

Example.

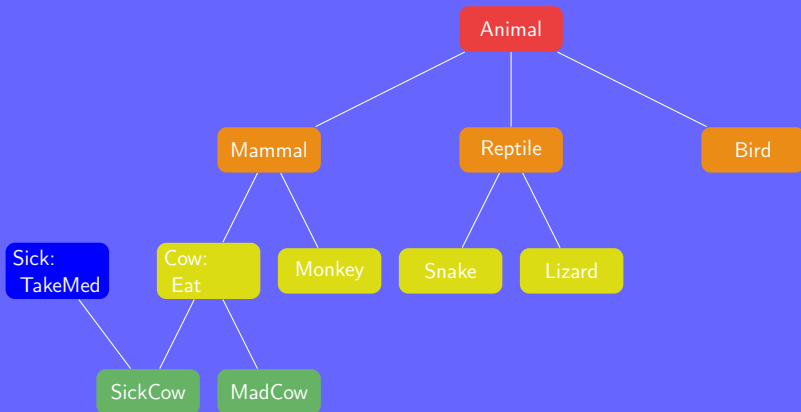
Any sick animal should be put under medication:

- 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



Multiple inheritance



```
1 class SickCow : public Cow, public Sick {  
2     ...  
3 }
```

More cows

animals-m.h

```
1  class Animal {
2      public:
3          virtual void Speak() = 0; virtual void Eat() = 0;
4  };
5  class Sick {
6      public: void TakeMed();
7      protected: int med;
8  };
9  class Cow : public Animal {
10     public: Cow(int f=0); virtual void Speak(); void Eat();
11     protected: int grass;
12 };
13 class SickCow : public Cow, public Sick {
14     public: SickCow(int f=0,int m=0); void Speak();
15 };
16 class MadCow : public Cow {
17     public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
18     protected: int pills;
19 };
```

More cows

animals-m.cpp

```
1  #include <iostream>
2  #include "animals_m.h"
3  using namespace std;
4  void Sick::TakeMed(){
5      if(med > 0) { med--; cout << "I feel better\n";}
6      else cout << "I'm dying\n";
7  }
8  Cow::Cow(int f) {grass=f;}
9  void Cow::Speak() { cout << "Moo.\n"; }
10 void Cow::Eat(){
11     if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
12     else cout << "I'm hungry\n";
13 }
14 SickCow::SickCow(int f,int m) {grass=f; med=m;}
15 void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
16 MadCow::MadCow(int f, int p) {grass=f; pills=p;}
17 void MadCow::Speak() { cout << "Woof\n";}
18 void MadCow::TakePills() {
19     if(pills > 0) {pills--; cout << "Moof, that's better\n";}
20     else cout << "Woof woof woof!\n";
21 }
```

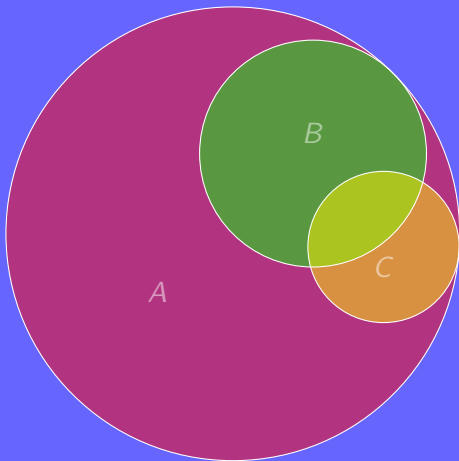
More cows

animals-main-m.cpp

```
1  #include <iostream>
2  #include "animals_m.h"
3  using namespace std;
4  int main () {
5      SickCow c1(1,1);
6      c1.Speak(); c1.Eat(); c1.TakeMed();
7      c1.Eat(); c1.TakeMed();
8      cout << endl;
9      MadCow c2(1,1);
10     c2.Speak(); c2.Eat(); c2.TakePills();
11     c2.Eat(); c2.TakePills();
12 }
```

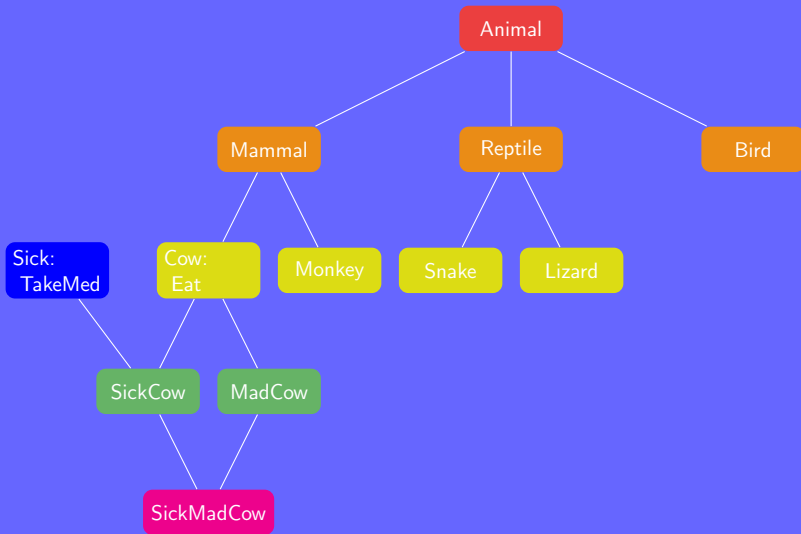
The diamond problem

Multiple inheritance can be tricky:



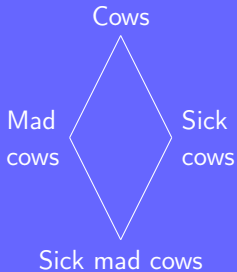
- A : Cows
- B : Sick cows
- C : Mad cows
- Sick mad cows are in $B \cap C$

The diamond problem

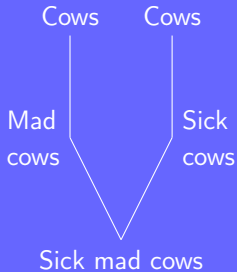


The diamond problem

Human perspective

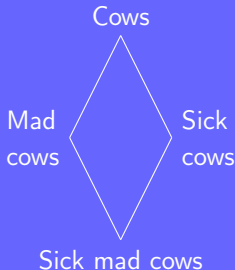


Computer perspective

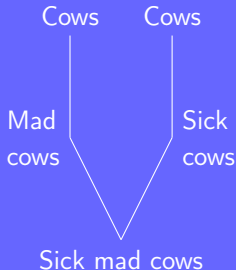


The diamond problem

Human perspective



Computer perspective



Questions:

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

The diamond problem

Solutions to overcome the problem:

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

```
1 class Cow {...};  
2 class SickCow : public virtual Cow {...};  
3 class MadCow : public virtual Cow {...};  
4 class SickMadCow : public SickCow, public MadCow {...};
```

Calling Eat or updating grass does not generate any problem

The diamond problem

Solutions to overcome the problem:

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

```
1 class Cow {...};  
2 class SickCow : public virtual Cow {...};  
3 class MadCow : public virtual Cow {...};  
4 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

```
1  class Animal {
2      public: virtual void Speak() = 0; virtual void Eat() = 0;
3  };
4  class Sick {
5      public: void TakeMed();
6      protected: int med;
7  };
8  class Cow : public Animal {
9      public: Cow(int f=0); virtual void Speak(); void Eat();
10     protected: int grass;
11 };
12 class SickCow : public virtual Cow, public Sick {
13     public: SickCow(int f=0,int m=0); void Speak();
14 };
15 class MadCow : public virtual Cow {
16     public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
17     protected: int pills;
18 };
19 class SickMadCow : public SickCow, public MadCow {
20     public: SickMadCow(int f=0, int m=0, int p=0); void Speak();
21 };
```

Sick mad cows

animals-d.cpp

```
1  #include <iostream>
2  #include "animals_d.h"
3  using namespace std;
4  void Sick::TakeMed() { if (med > 0) { med--; cout << "I feel better\n"; }
5      else cout << "I'm dying\n";
6  }
7  Cow::Cow(int f) {grass=f;}
8  void Cow::Speak() { cout << "Moo.\n"; }
9  void Cow::Eat(){ if (grass > 0) { grass-- ; cout << "Thanks I'm full\n"; }
10     else cout << "I'm hungry\n";
11 }
12 SickCow::SickCow(int f,int m) {grass=f; med=m;}
13 void SickCow::Speak() { cout << "Ahem... Moo\n"; }
14 MadCow::MadCow(int f, int p) {grass=f; pills=p;}
15 void MadCow::Speak() { cout << "Woof\n"; }
16 void MadCow::TakePills() {
17     if (pills > 0) {pills--; cout << "Moof, that's better\n"; }
18     else cout << "Woof woof woof!\n";
19 }
20 SickMadCow::SickMadCow(int f, int m, int p) {grass=f; med=m; pills=p;}
21 void SickMadCow::Speak() {cout << "Ahem... Woof\n"; }
```

Sick mad cows

animals-main-d.cpp

```
1  #include <iostream>
2  #include "animals_d.h"
3  using namespace std;
4  int main () {
5      SickCow c1(1,1);
6      c1.Speak(); c1.Eat(); c1.TakeMed();
7      c1.Eat(); c1.TakeMed();
8      cout << endl;
9      MadCow c2(1,1);
10     c2.Speak(); c2.Eat(); c2.TakePills();
11     c2.Eat(); c2.TakePills();
12     cout << endl;
13     SickMadCow c3(1,1,1);
14     c3.Speak(); c3.Eat(); c3.TakePills(); c3.TakeMed();
15     c3.Eat(); c3.TakePills(); c3.TakeMed();
16     SickMadCow c4(1,1,0); Cow *c5=&c4;
17     c4.Speak(); c4.Eat(); c4.TakePills(); c4.TakeMed();
18     c5->Speak(); c5->Eat(); //c5->TakePills(); c5->TakeMed();
19 }
```

Sick mad cows

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 (10.314) is uncommented? Why?

Project development

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?

Chapter 11

Libraries and templates

Outline

- ① Using external libraries
- ② Writing templates
- ③ 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

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

The OpenGL library

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...

The OpenGL library

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...

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

Hierarchy diagram

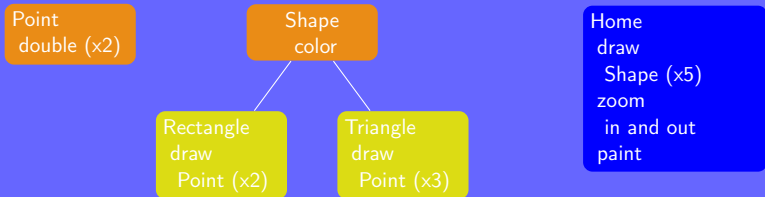
First steps:

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

Hierarchy diagram

First steps:

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



Figures specification

home/figures.h

```
1  #ifndef __FIGURES_H__
2  #define __FIGURES_H__
3  typedef struct _Point { double x,y; } Point;
4  class Shape {
5      public: virtual void draw() = 0; virtual ~Shape();
6      protected: float r, g, b;
7  };
8  class Rectangle : public Shape {
9      public: Rectangle(Point pt1={-.5,-.5}, Point pt2={.5,.5},
10                     float r=0, float g=0, float b=0);
11      void draw();
12      private: Point p1,p2;
13  };
14  class Triangle : public Shape {
15      public: Triangle(Point pt1={-.5,-.5}, Point pt2={.5,-.5},
16                     Point pt3={0,.5}, float r=0, float g=0, float b=0);
17      void draw();
18      private: Point p1,p2,p3;
19  };
20 #endif
```

Figures implementation

home/figures.cpp

```
1  #include <GL/glut.h>
2  #include "figures.h"
3  Shape::~Shape(){}
4  Rectangle::Rectangle(Point pt1, Point pt2,
5      float red, float green, float blue) {
6      p1=pt1; p2=pt2; r=red; g=green; b=blue;
7  }
8  void Rectangle::draw() {
9      glColor3f(r, g, b); glBegin(GL_QUADS);
10     glVertex2f(p1.x, p1.y); glVertex2f(p2.x, p1.y);
11     glVertex2f(p2.x, p2.y); glVertex2f(p1.x, p2.y); glEnd();
12 }
13 Triangle::Triangle(Point pt1, Point pt2, Point pt3,
14     float red, float green, float blue) {
15     p1=pt1; p2=pt2; p3=pt3; r=red; g=green; b=blue;
16 }
17 void Triangle::draw() {
18     glColor3f(r, g, b); glBegin(GL_TRIANGLE_STRIP);
19     glVertex2f(p1.x, p1.y); glVertex2f(p2.x, p2.y); glVertex2f(p3.x, p3.y);
20     glEnd();
21 }
```

Home specification

home/home.h

```
1  #ifndef __HOME_H__
2  #define __HOME_H__
3  #include "figures.h"
4  class Home {
5  public:
6      Home(Point pt1={0,-.25}, double width=1,
7           double height=1.3, double owidth=.175);
8      ~Home();
9      void draw();
10     void zoom(double *width,double *height,double *owidth);
11 private:
12     Point p; double w, h, o; Shape *sh[5];
13     void zoomout(double *width,double *height,double *owidth);
14     void zoomin(double *width,double *height,double *owidth);
15     void paint(float *r, float *g, float *b);
16 };
17 #endif
```

Home implementation (part 1)

home/home-part1.cpp

```
1  #include <ctime>
2  #include <cstdlib>
3  #include "home.h"
4  Home::Home(Point pt1, double width, double height, double owidth){
5      float r, g, b; Point p1, p2, p3;
6      p=pt1; w=width; h=height; o=owidth; srand(time(0));
7      p1={p.x-w/2,p.y-w/2}; p2={p.x+w/2,p.y+w/2};
8      paint(&r,&g,&b); sh[0]=new Rectangle(p1,p2,r,g,b);
9      p1={p.x-o,p.y-w/2}; p2={p.x+o,p.y};
10     paint(&r,&g,&b); sh[1]=new Rectangle(p1,p2,r,g,b);
11     p1={p.x-2*o,p.y+o}; p2={p.x-o,p.y+2*o};
12     paint(&r,&g,&b); sh[2]=new Rectangle(p1,p2,r,g,b);
13     p1={p.x+w/2-2*o,p.y+o}; p2={p.x+w/2-o,p.y+2*o};
14     paint(&r,&g,&b); sh[3]=new Rectangle(p1,p2,r,g,b);
15     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};
16     paint(&r,&g,&b); sh[4]=new Triangle(p1,p2,p3,r,g,b);
17 }
18 Home::~~Home(){ for(int i=0;i<5;i++) delete sh[i]; }
```

Home implementation (part 2)

home/home-part2.cpp

```
1 void Home::draw() {for(int i=0;i<5;i++) sh[i]->draw();}
2 void Home::zoom(double *width, double *height, double *owidth){
3     int static i=0;
4     if(h>=0.1 && i==0) zoomout(width, height, owidth);
5     else if (h<=2) { i=1; zoomin(width, height, owidth); }
6     else i=0;
7 }
8 void Home::zoomout(double *width, double *height, double *owidth){
9     h/=1.01; *height=h; w/=1.01; *width=w; o/=1.01; *owidth=o;
10 }
11 void Home::zoomin(double *width, double *height, double *owidth){
12     h*=1.01; *height=h; w*=1.01; *width=w; o*=1.01; *owidth=o;
13 }
14 void Home::paint(float *r, float *g, float *b) {
15     *r=(float)rand()/RAND_MAX; *g=(float)rand()/RAND_MAX;
16     *b=(float)rand()/RAND_MAX;
17 }
```

Home instantiation

home/main.cpp

```
1  #include <GL/glut.h>
2  #include "home.h"
3  void TimeStep(int n) {
4      glutTimerFunc(n, TimeStep, n); glutPostRedisplay();
5  }
6  void glDraw() {
7      double static width=1, height=1.5, owidth=.175;
8      Home zh({0, -.25}, width, height, owidth);
9      zh.zoom(&width, &height, &owidth);
10     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
11     zh.draw(); glutSwapBuffers(); glFlush();
12 }
13 int main (int argc, char *argv[]) {
14     glutInit(&argc, argv);
15     // glutInitWindowSize(500, 500);
16     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
17     glutCreateWindow("Home sweet home");
18     glClearColor(1.0, 1.0, 1.0, 0.0); glClear(GL_COLOR_BUFFER_BIT);
19     glutDisplayFunc(glDraw); glutTimerFunc(25, TimeStep, 25);
20     glutMainLoop();
21 }
```


Basics

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 ∈ [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 the `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?

Compilation

Compiling and running the home:

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

Compilation

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
```

Makefile

home/Makefile

```
1 CC = g++ # compiler
2 CFLAGS = -std=c++11 # compiler options
3 LIBS = -lglut -lGL # libraries to use
4 SRCS = main.cpp home.cpp figures.cpp
5 MAIN = home
6 OBJS = $(SRCS:.cpp=.o)
7 .PHONY: clean # target not corresponding to real files
8 all:      $(MAIN) # target all constructs the home
9     @echo Home successfully constructed
10 $(MAIN):
11     $(CC) $(CFLAGS) -o $(MAIN) $(SRCS) $(LIBS)
12 .cpp.o: # for each .cpp build a corresponding .o file
13     $(CC) $(CFLAGS) -c $< -o $@
14 clean:
15     $(RM) *.o *~ $(MAIN)
```

Outline

- ① Using external libraries
- ② Writing templates
- ③ Standard Template Library

Classes

Limitations of inheritance and polymorphism:

- High level classes (boat, company, car...)
- 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 that are more complex to debug

Defining a template

A *template* is a “special class” where the data type is a parameter
Example.

complex.h

```
1  #include <iostream>
2  using namespace std;
3  template<class TYPE>
4  class Complex {
5      public:
6          Complex(){ R = I = (TYPE)0; }
7          Complex(TYPE real, TYPE img) {R=real;I=img;}
8          void PrintComplex() {cout<<R<<'+'<<I<<"i\n";}
9      private:
10         TYPE R, I;
11 };
```


Using a template

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

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

Example.

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

Using a template

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

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

Example.

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

complex.cpp

```
1 #include "complex.h"  
2 typedef Complex<char> CComplex ;  
3 int main () {  
4     Complex<double> a(3.123,4.9876); a.PrintComplex();  
5     Complex<int> b; b = Complex<int>(3,4);  
6     b.PrintComplex();  
7     CComplex c('a','b'); c.PrintComplex();  
8 }
```

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

Outline

- ① Using external libraries
- ② Writing templates
- ③ Standard Template Library

Basics on STL

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

Sequence containers

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

Sequence containers

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

Other available containers: set, multiset, map, multimap, bitset, valarray, unordered_ { set, multiset, map, multimap }

Vectors

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.

```
1  #include <vector>
2  vector<int> vint;
3  vector<float> vfloat;
```


Vectors

vect.cpp

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

Container adaptors

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)

Example

queue.cpp

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4  int main () {
5      int i,j=0;
6      queue <int> line;
7      for(i=0;i<200;i++) line.push (i+1);
8      while(line.empty() == 0) {
9          cout << line.size () << " persons in the line\n"
10             << "first in the line: " << line.front() << endl
11             << "last in the line: " << line.back() << endl;
12         line.pop ();
13         if(j++%3==0) {
14             line.push (++i);
15             cout << "new in the line: " << line.back() <<endl;
16         }
17     }
18 }
```

Iterators

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

```
1  for(i=0;i<vct.size();i++) {  
2      ...  
3  }
```

```
1  for(it=vct.begin(); \  
2      it !=vct.end();++it) {  
3      ...  
4  }
```

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

Example

iterator.cpp

```
1  #include <iostream>
2  #include <set>
3  using namespace std;
4  int main() {
5      set<int> s;
6      s.insert(7);s.insert(2);s.insert(-6);
7      s.insert(8);s.insert(1);s.insert(-4);
8      set<int>::const_iterator it;
9      for(it = s.begin(); it != s.end(); ++it) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
```

Algorithms templates

Common algorithms implemented in templates:

- Manipulate data stored in the containers
- Mainly targeting range of elements
- Many “high low-level” functions
 - Sort
 - Shuffle
 - Find with conditions
 - Partition
 - ...

Count

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

count.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  int main () {
7      string colors[8] = {"red","blue","yellow","black",
8                          "green","red","green","red"};
9      vector<string> colorvect(colors, colors+8);
10     int nbcolors = count (colorvect.begin(),
11                           colorvect.end(), "red");
12     cout << "red appears " << nbcolors << " times.\n";
13 }
```

Find

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 (*use find with purple in the following code*)

find.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  int main () {
7      string colors[8] = {"red", "blue", "yellow", "black",
8                          "green", "red", "green", "red"};
9      vector<string> colorvect(colors, colors+8);
10     vector<string>::iterator it;
11     it=find(colorvect.begin(), colorvect.end(), "blue");
12     ++it;
13     cout << "following blue is " << *it << endl;
14 }
```


Unique

Remove consecutive duplicate elements

unique1.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  bool cmp(string s1, string s2) { return(s1.compare(s2)==0); }
7  int main () {
8      string colors[8] = {"red", "blue", "yellow", "black",
9                          "green", "green", "red", "red"};
10     vector<string> colorvect(colors, colors+8);
11     vector<string>::iterator it;
12     it=unique(colorvect.begin(), colorvect.end(), cmp);
13     colorvect.resize(distance(colorvect.begin(), it));
14     for(it=colorvect.begin(); it!=colorvect.end();++it)
15         cout << ' ' << *it;
16     cout << endl;
17 }
```

Sort

Sort elements in ascending order

sort.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  bool cmp(string s1, string s2) { return(s1.compare(s2)<0); }
7  int main () {
8      string colors[8] = {"red", "blue", "yellow", "black",
9                          "green", "green", "red", "red"};
10     vector<string> colorvect(colors, colors+8);
11     vector<string>::iterator it;
12     sort(colorvect.begin(), colorvect.end(), cmp);
13     for(it=colorvect.begin(); it!=colorvect.end();++it)
14         cout << ' ' << *it;
15     cout << endl;
16 }
```

Problem

Remove all duplicate elements from the color vector.

Solution

unique2.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  bool cmp1(string s1, string s2) {return(s1.compare(s2)<0);}
7  bool cmp2(string s1, string s2) {return(s1.compare(s2)==0);}
8  int main () {
9      string colors[8] = {"red","blue","yellow","black",
10         "green","green","red","red"};
11      vector<string> colorvect(colors, colors+8);
12      vector<string>::iterator it;
13      sort(colorvect.begin(), colorvect.end(),cmp1);
14      it=unique(colorvect.begin(), colorvect.end(),cmp2);
15      colorvect.resize(distance(colorvect.begin(),it));
16      for(it=colorvect.begin(); it!=colorvect.end();++it)
17          cout << ' ' << *it;
18      cout << endl;
19  }
```

Reverse

Reverse the order of the elements

reverse.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  int main () {
7      string colors[8] = {"red", "blue", "yellow", "black",
8                          "green", "green", "red", "red"};
9      vector<string> colorvect(colors, colors+8);
10     vector<string>::iterator it;
11     reverse(colorvect.begin(), colorvect.end());
12     for(it=colorvect.begin(); it!=colorvect.end(); ++it)
13         cout << ' ' << *it;
14     cout << endl;
15 }
```

Question. Any other possible strategy?

Remove

Remove elements and returns an iterator to the new end

remove.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  bool bstart(string s) { return(s[0]!='b'); }
7  int main () {
8      string colors[8] = {"red","blue","yellow","black",
9                          "green","green","red","red"};
10     vector<string> colorvect(colors, colors+8);
11     vector<string>::iterator it;
12     it=remove_if(colorvect.begin(),colorvect.end(),bstart);
13     colorvect.resize(distance(colorvect.begin(),it));
14     for(it=colorvect.begin(); it!=colorvect.end();++it)
15         cout << ' ' << *it;
16     cout << endl;
17 }
```

Random_shuffle

Randomly rearrange elements

random.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  int main () {
7      srand (unsigned(time(0)));
8      string colors[8] = {"red", "blue", "yellow", "black",
9                          "green", "green", "red", "red"};
10     vector<string> colorvect(colors, colors+8);
11     vector<string>::iterator it;
12     random_shuffle(colorvect.begin(), colorvect.end());
13     for(it=colorvect.begin(); it!=colorvect.end(); ++it)
14         cout << ' ' << *it;
15     cout << endl;
16 }
```

Max and min

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

minmax.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  using namespace std;
6  bool cmp(string s1, string s2) {return(s1.compare(s2)<0);}
7  int main () {
8      srand (unsigned(time(0)));
9      auto mm=minmax({"red","blue","yellow","black"},cmp);
10     cout << mm.first << ' ' << mm.second;
11     cout << endl;
12 }
```


Key points

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

Chapter 12

Beyond MATLAB, C, and C++

Outline

- ① Improving the coding style
- ② A few more things on C and C++
- ③ What's next?

Layer programming

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

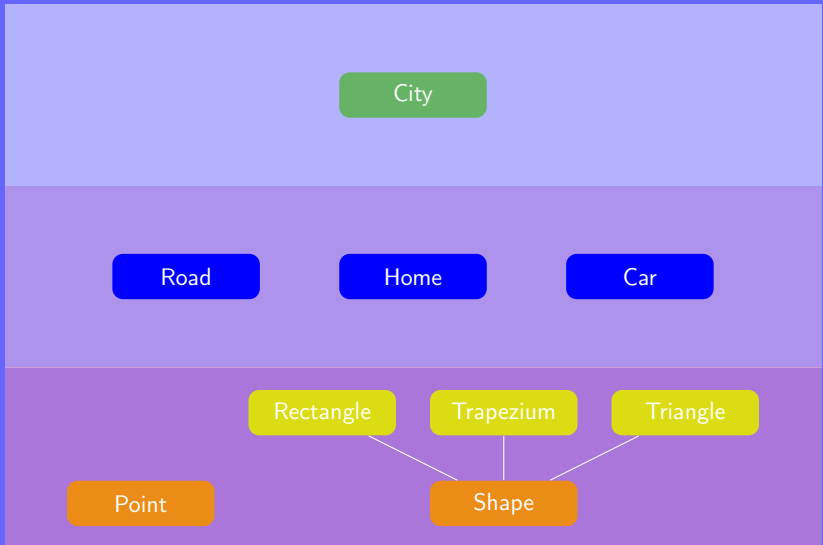
Layer programming

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

Makefile

```
1 CCC = g++
2 CCFLAGS = -std=c++11 -Wall -Wextra -Werror -pedantic
3 LIBS = -lglut -lGL
4 LLIBS = -L. -lhome -lfig
5 LFIG_SRC = figures.cpp
6 LFIG_OBJ = $(LFIG_SRC:.cpp=.o)
7 LFIG = libfig.a
8 LHOME_SRC = home.cpp
9 LHOME_OBJ = $(LHOME_SRC:.cpp=.o)
10 LHOME = libhome.a
11 MAIN_SRC = main.cpp
12 MAIN = home
13 .PHONY: clean hlibs
14
15 all: $(LFIG_OBJ) $(LHOME_OBJ) hlibs $(MAIN)
16     @echo Home successfully constructed
17
18 $(MAIN): $(MAIN_SRC)
19     $(CCC) $(CCFLAGS) -o $(MAIN) $(MAIN_SRC) $(LIBS) $(LLIBS)
20
21 .cpp.o:
22     $(CCC) $(CCFLAGS) -c $< -o $@
23
24 hlibs :
25     ar rcs $(LFIG) $(LFIG_OBJ);   ar rcs $(LHOME) $(LHOME_OBJ)
26
27 clean:
28     $(RM) *.o *.a *~ $(MAIN)
```

More compilation

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*

Outline

- ① Improving the coding style
- ② A few more things on C and C++
- ③ What's next?

The const keyword

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

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

Constant pointers vs. pointer to constant

Constant pointer

```
1 int const *p;
```

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

Pointer to constant

```
1 const int *p;
```

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

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

References

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 &a=3`

Remarks:

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

References

Example.

ref.cpp

```
1  #include <iostream>
2  using namespace std;
3  int square0(int x) {return x*x;}
4  void square1(int x, int &res) { res=x*x; }
5  //int& square2a(int x) { int b=x*x; return b; }
6  int& square2b(int x) { int b=x*x; int &res=b; return res; }
7  int& square2c(int x) { static int b=x*x; return b; }
8  int main () {
9      int a=2;
10     cout << square0(a) << ' ' << a << endl;
11     square1(a,a); cout << a << endl;
12     cout << square2b(a) << endl;
13     cout << square2c(a) << endl;
14 }
```

The this pointer

The this keyword:

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

boat.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Boat {
4  public:
5      Boat(string name, int tonnage, bool IsDocked) {
6          this->name=name; this->tonnage=tonnage; this->IsDocked=IsDocked;
7      }
8      void dock() { IsDocked=1; cout<<"Docked!\n"; }
9      void undock() { IsDocked=0; cout<<"Undocked!\n"; }
10     private: bool IsDocked; string name; int tonnage;
11 };
12 int main () {
13     Boat b("abc",1234,1); b.undock();
14 }
```

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

```
1  #include <stdio.h>
2  #include <string.h>
3  int gm(char *n) {
4      printf("good morning %s\n",n);
5      return strlen(n);
6  }
7  int main () {
8      int (*gm_ptr)(char *)=gm;
9      printf("%d\n",(*gm_ptr)("john"));
10 }
```

The enum and union keywords

enum-union.c

```
1  #include<stdio.h>
2  typedef struct _activity {
3      enum { BOOK, MOVIE, SPORT } type;
4      union {
5          int pages;
6          double length;
7          int freq;
8      } prop;
9  } activity;
10 int main() {
11     activity a[5];
12     a[0].type=BOOK; a[0].prop.pages=192;
13     a[1].type=SPORT; a[1].prop.freq=4;
14     a[2].type=MOVIE; a[2].prop.pages=123;
15     a[2].prop.length=92.5;
16     printf("%f",a[2].prop.length);
17 }
```


The argc and *argv[] parameters

arg.c

```
1  #include <stdio.h>
2  int main (int argc, char *argv[]) {
3      printf ("program: %s\n",argv[0]);
4
5      if (argc > 1) {
6          for (int i=1; i<argc; i++)
7              printf("argv[%d] = %s\n", i, argv[i]);
8      }
9      else printf("no argument provided\n");
10     return 0;
11 }
```

Compilation process

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

Outline

- ① Improving the coding style
- ② A few more things on C and C++
- ③ 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

Future

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, optimisations...) → improve efficiency

Key points

- Many things left to learn
- Before coding write an algorithm
- No better way to learn than coding
- Don't 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...