Introduction to Computer and Programming

Manuel

Fall 2016

Table of contents

Chapter 1: Computers and Programming Languages
Chapter 2: Introduction to MATLAB
Chapter 3: Control statements
Chapter 4: Functions and recursion
Chapter 5: Plotting in MATLAB
Chapter 6: Data types and structures
Chapter 7: Introduction to C
Chapter 8: Data types in C
Chapter 9: Syntax and control statements
Chapter 10: Arrays and pointers
Chapter 11: Algorithm and efficiency
Chapter 12: Introduction to C++
Chapter 13: Object and class
Chapter 14: Inheritance and polymorphism
Chapter 15: Libraries and templates
Chapter 16: Beyond MATLAB, C, and C++

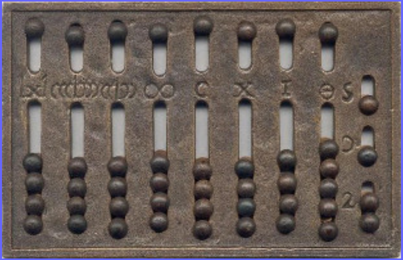
Chapter 1

Computers and Programming Languages

Outline

1. A brief history of computing
2. Understanding Computers
3. Understanding Programming

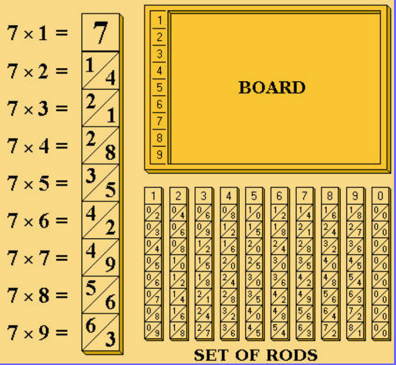
Ancient Era



Abacus (-2700)

Antikythera mechanism (-100)

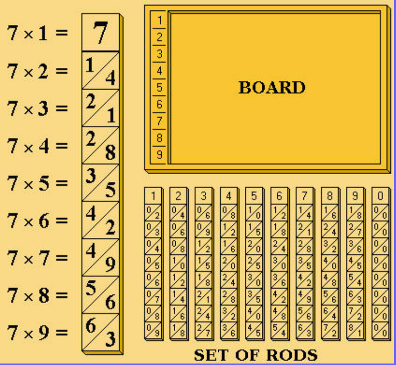
Calculating Tools



Napier's bones (1617)

Sliderule (1620)

Calculating Tools



Napier's bones (1617)

Sliderule (1620)

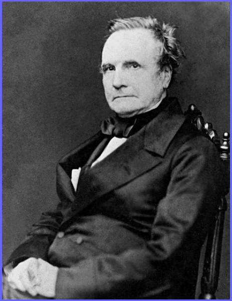
Note: first pocket calculator around 1970 in Japan.

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

- **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

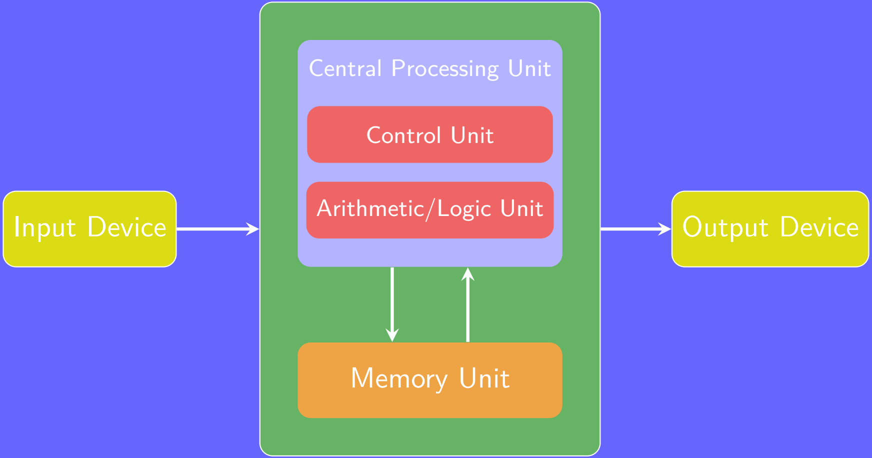
Apple I (1976)

- **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

Outline

1. A brief history of computing
2. Understanding Computers
3. Understanding Programming

Von Neumann architecture



What does a computer understand?

- Humans use *decimal* $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$
  e.g. $(253)_{10}$

What does a computer understand?

- 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. $(11111101)_2$

What does a computer understand?

- 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. $(\mathrm{FD})_{16}$

Number base conversion

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

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

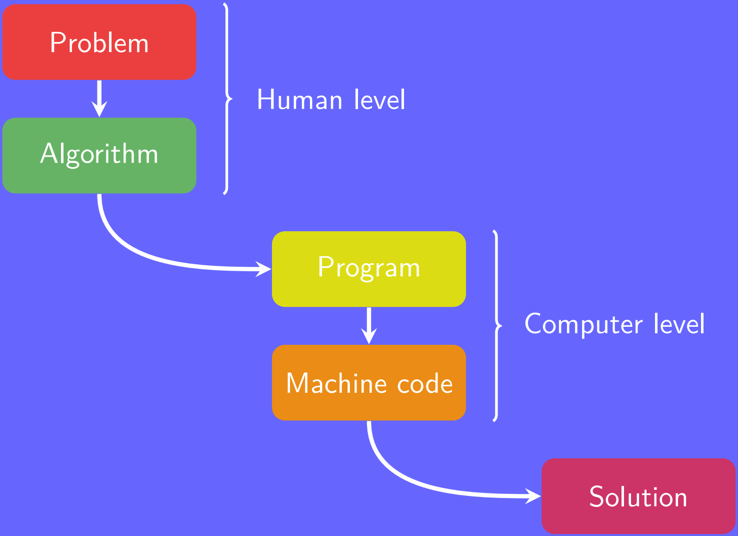
Quick examples

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

How to use a computer?



Outline

1. A brief history of computing
2. Understanding Computers
3. Understanding Programming

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**

$$\begin{array}{ccc} \text{Algorithm} & \longrightarrow & \text{Programming Language} \\ & & \downarrow \\ \text{Machine Code} & \longleftarrow & \text{Interpreter / Compiler} \end{array}$$

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

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

area.cpp

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

area.m

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

Running the 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`

Key points

- What is a programming language?
- What are the two main types of programming languages?
- What is an algorithm?
- How easy is it to write machine code?

Chapter 2

Introduction to MATLAB

Outline

1. Programming in sciences
2. Running MATLAB
3. Arrays and matrices

Mathematical softwares

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

MATLAB

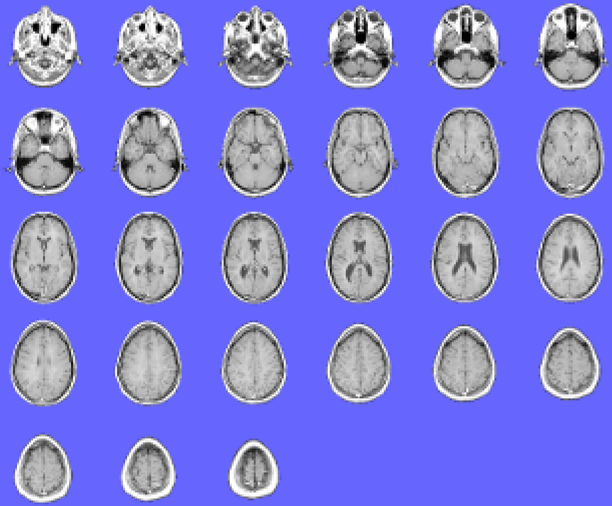
MATLAB=MATrix LABoratory

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

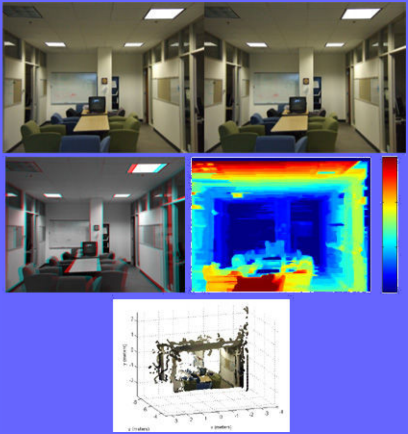
Why MATLAB?

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

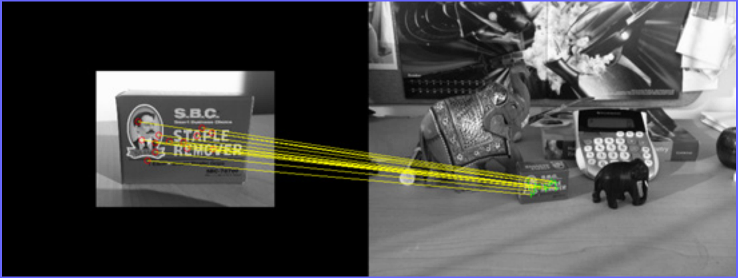
MRI slices



Stereo Vision



Object detection



Mathematics and Physics

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

What to do?

- 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

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

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 : $r = 1.496 \cdot 10^8,\ c = 4.379 \cdot 10^6,\ G = 6.674 \cdot 10^{-11},\ T = 365\mathrm{D}$
Output : Density of the Sun

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

Run the algorithm: 338110866080

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,
$G = 6.674 \cdot 10^{-11}$ m^3/kg/s^2, $T = 365 * 24 * 3600$ s

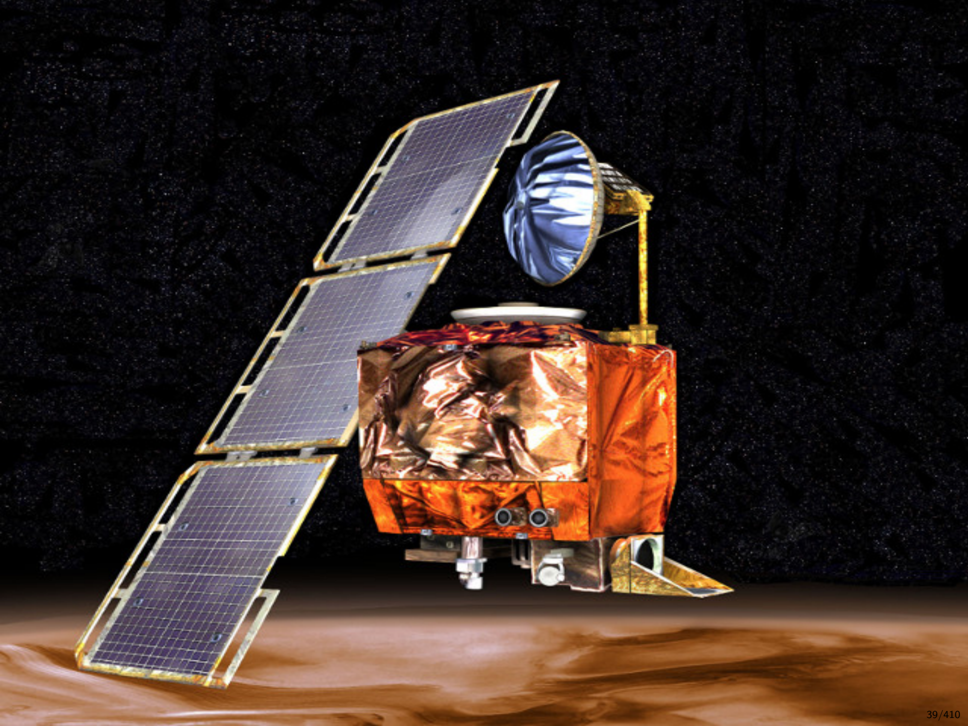
Output : Density of the Sun

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

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

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

Run the algorithm: 1404 kg/m^3



Outline

1. Programming in sciences
2. Running MATLAB
3. Arrays and matrices

Starting MATLAB

Two modes: desktop vs no desktop

In desktop mode:

- Command history
- Workspace
- Command window
- Files to run must be in *current directory* or in a *directory listed* in the path
- Help

Basic use

- 1+2 vs. 1+2;
- Variables: start with a letter, case sensitive.
  e.g. a=1+2; A=3+2; a123_=4+5;
- % to add comments
- , to separate commands
- ... to split a statement over 2 lines
- `namelengthmax`, `iskeyword`

Simple operations

- pi $= \pi$
- i $= \sqrt{-1}$
- j $= \sqrt{-1}$
- Inf $=$ Infinity
- NaN: Not a Number

Simple operations

- pi $= \pi$
- i $= \sqrt{-1}$
- j $= \sqrt{-1}$
- Inf $=$ Infinity
- NaN: Not a Number

- Addition: $+$
- Subtraction: $-$
- Multiplication: $*$
- Power: ^
- (Right) division: $/$
- Left division: $\backslash$
- Order of evaluation: $()$

More advanced operations

Large number of functions to solve mathematical problems:

- Elementary function: help elfun
- Special functions: help specfun
- Matrix functions: help elmat

Density of the Sun

MATLAB code to input in the workspace window:

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

M-File

- MATLAB code can be written is a file and then loaded
- All variables are added to the workspace
- To avoid variable conflicts make use of the functions: `clear, clear all, clc`
- Add *cell breaks* to debug the code

A first simple program

Exercise.
Prompt the user for two numbers, store their sum in a variable, and display the result.

A first simple program

Exercise.
Prompt the user for two numbers, store their sum in a variable, and display the result.

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

Outline

1 Programming in sciences

2 Running MATLAB

3 Arrays and matrices

Arrays and MATLAB

Array: arrangement of quantities in rows and columns

Arrays and MATLAB

Array: arrangement of quantities in rows and columns

$$\downarrow$$

Matrix: two-dimensional numeric array

Arrays and MATLAB

Array: arrangement of quantities in rows and columns

$\downarrow$

Matrix: two-dimensional numeric array

$\downarrow$

MATLAB: MATrix LABoratory

Arrays and MATLAB

Array: arrangement of quantities in rows and columns

$\downarrow$

Matrix: two-dimensional numeric array

$\downarrow$

MATLAB: MATrix LABoratory

$\Downarrow$

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

Generating arrays and matrices

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

Generating arrays and matrices

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

Generating arrays and matrices

- Generate a sequence of numbers: `a:b` or `a:b:c`
- Concatenate (join) elements: `[ ]`
- Generate a 1-dimensional array: `[a:b]` or `[a:b:c]`
- Generate a 2-dimensional array: `[a b c; d e f;]`
- Generate a list between a and b, with n elements: `linspace(a, b, n)`
- `zeros(a,b)`
- `ones(a,b)`

Dealing with matrices

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

Array vs Matrix

Arrays

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

Matrices

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

Given the array `A = [2 7 9 7 ; 3 1 5 6 ; 8 1 2 5]`, explain the results of the following commands:

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

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 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 left element has index 1 and the bottom right "number of elements"

Example.

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

Key points

- What does MATLAB mean?
- How to process a problem before implementing it?
- What can be said about units?
- How to write simple scripts in MATLAB?
- What is the difference between an array and a matrix?

Chapter 3

Control statements

Outline

1 Conditional expressions

2 Loops

3 Advanced usage

The `if` statement

If it rains, then I take an umbrella.

Structure in MATLAB:

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

Relational operators

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

Relational operators

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

Returns `True` or `False`

Boolean logic

Boolean logic was introduced by George Boole around mid 1800s

Truth table of 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 was introduced by George Boole around mid 1800s

Truth table of 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

Idea: run instructions depending on the thruth value of a given expression

Logical operators

- `&` and
- $|$ or
- $\sim$ not
- `xor`$(\cdot,\cdot)$ exclusive or

Logical operators

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

Example.

```
A=[0 1 1 0 1]; B=[1 1 0 1 0];
A & B, A | B, ~A, xor(A,B)
```

Short-circuit operators

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

Example.

```
exist('file') & load('file')
exist('file') && load('file')
```

Example

Press a key and return whether it is a digit or not

```
k=input('Press a key: ','s');
if k>='0' && k<='9'
   disp('Digit')
else
  disp('Not a digit')
end
```

The `switch` statement

Executes statements depending on the value of a variable.

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

The `switch` statement

Executes statements depending on the value of a variable.

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

The variable is expected to be a scalar or a string

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

Example

Prompt the user for a digit, then display 0 if it is 0, < 5 if it is between 1 and 4 or ≥ 5 if it is larger or equal to 5.

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

Outline

1. Conditional expressions
2. Loops
3. Advanced usage

What is a loop?

Group of statements repeatedly executed as long as the conditional expression is `True`

Three main types of loops in MATLAB:

- `while`
- `for`
- Vectorizing loops

The `while` loop

Instructions are executed until the expression becomes `False`

```
while expression
  statements
end
```

The `while` loop

Instructions are executed until the expression becomes `False`

```
while expression
  statements
end
```

Example.
Endless loop

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

Example

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

The for loop

Instructions are executed a predetermined number of times

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

The for loop

Instructions are executed a predetermined number of times

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

Example.
Display all the even numbers between 0 and 100

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

Vectorizing loop

MATLAB: array/matrix language

$\downarrow$

Convert `for`/`while` loops into vector/matrix operations

Vectorizing loop

MATLAB: array/matrix language

↓

Convert `for`/`while` loops into vector/matrix operations

Example.
`while` vs. `for` vs. vectorization

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

Outline

1 Conditional expressions

2 Loops

3 Advanced usage

The `continue` and `break` commands

- `continue`: skip the remaining statements in the loop to go to the next iteration
- `break`: exit the loop and execute the next statements outside the loop

The `continue` and `break` commands

- `continue`: skip the remaining statements in the loop to go to the next iteration
- `break`: exit the loop and execute the next statements outside the loop

Example.

Count the number of letters in a “word”

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

Efficiency

- Internal structure of matrices: linear memory
  e.g. $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
- Row or column first: 1 2 3 4 5 6 or 1 4 2 5 3 6
- MATLAB uses the “column-major order”
- Column should be in the outer loop

Example

Use a double loop to set all the element of the zero matrix to 1

```
N = 10000; a = zeros(N);
tic;
for j = 1:N
  for i=1:N
    a(j,i) = 1;
  end
end
toc;
```

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.

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

Key points

- Why are conditional statements useful?
- How the check some conditions?
- How to loop in MATLAB?
- How to exit a loop?
- How to choose which type of loop to use?

Chapter 4

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

- Function saved in a .m file
- The .m file must be in the “path”
- The function name must be the same as the filename
- `function [output1, output2,...]  = Functionname(input1,input2,...)`
- The function can be called from another .m file or from the workspace

Example

Script version:

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

Example

Script version:

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

Function version:

density.m

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

Example

Script version:

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

Function version:

density.m

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

The function can be applied to any orbital system; e.g. Jupiter – Europa: radius: 671034000 m, period: 306720 s, circumference: 439260000 m

Sub-functions

A .m file can contain:

- A “main function”
- Several sub-functions, only visible to other functions in the **same** file

Example.
Write a function returning the mean and the standard deviation, where the mean is calculated in a sub-function

Sub-functions

A .m file can contain:

- A "main function"
- Several sub-functions, only visible to other functions in the **same** file

Example.
Write a function returning the mean and the standard deviation, where the mean is calculated in a sub-function

stat.m

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

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

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

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

The `save` and `load` functions

Saving variables:

```
save('filename','var1','var2',...,'format')
```

- List of variables optional
- Common formats: `-mat` $\rightarrow$ binary, `-ascii` $\rightarrow$ text

The `save` and `load` functions

Saving variables:

```
save('filename','var1','var2',...,'format')
```

- List of variables optional
- Common formats: `-mat` $\rightarrow$ binary, `-ascii` $\rightarrow$ text

Loading variables:

```
load('filename')
```

Random or pseudorandom?

Problem:

- Computer cannot generate random numbers
- No way to generate real random numbers using a software
- Random human input is not random

Random or pseudorandom?

Problem:

- Computer cannot generate random numbers
- No way to generate real random numbers using a software
- Random human input is not random

Partial solution: pseudo random number generator

Generating pseudorandom numbers

- `rand(n,m)`: *nxm* matrix of random numbers, following the uniform distribution
- `randn(n,m)`: *nxm* matrix of random numbers, following the standard normal distribution
- `random('name',parameters)`: generate random numbers following the distribution *name*, parameters may vary depending on the distribution
- `rand('state',datenum(clock))`: use a specific seed
- `randperm(n)`: random permutation

The `sprintf` function

Purpose: write formated data into a string

Command: `sprintf(format,variable1, variable2...)`

- Format: string with special flag, replaced by value of the variables
- Special characters: carriage return/tab/backspace

Example.

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

File input/output

Basic idea: open a stream between a source and MATLAB
Different ways to access a file:

- Read only (`r`)
- Write to new file (`w`)
- Append to new/existing file (`a`)
- Read and write (`r+`)
- Read and overwrite (`w+`)
- Read and append (`a+`)

File input/output

Basic idea: open a stream between a source and MATLAB
Different ways to access a file:

- Read only (`r`)
- Write to new file (`w`)
- Append to new/existing file (`a`)
- Read and write (`r+`)
- Read and overwrite (`w+`)
- Read and append (`a+`)

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

The `fprintf` and `fscanf` functions

Writing in a file: `fprintf(fd, format, variables)`
(similar to `sprintf`)

The `fprintf` and `fscanf` functions

Writing in a file: `fprintf(fd, format, variables)` (similar to `sprintf`)

Reading from a file: `fscanf(fd, format, size)`

- Reads a file
- Converts into the specified format
- Only reads size elements if `size` is specified
- Returns a matrix containing the read elements
- Returns an optional parameter: number of elements successfully read

`fgetl(fd)`: returns one line

Example

Given a text file where each line is composed of three fields, namely firstname, 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, firstname and email

Example

sortnames.m

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

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

Example

In this simple example:

- 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?

Recursive acronyms

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

A short recursive story

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.

Recursion in computer science

Recursion: repeating items in a self-similar way

Given a process and some data, apply the same process using more simple data in order to describe this initial process and data

Strategy: a function calling itself

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.

Number in words

Input : A large integer n
Output : n, digit by digit, using words

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

Recursion vs. iteration

When using 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 using recursion over iteration:

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

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

Danger: memory usage

Key points

- Why should function 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 5

Plotting in MATLAB

Outline

1. 2D plotting
2. 3D plotting
3. Curve fitting

General plotting process

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. . .

Main 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

Changing the aspect of the figure:

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

Simple example

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

More plotting

- Polar graph: `polar(t,r)`
- Bar graph: `bar(x,y)`
- Horizontal bar graph: `barh(x,y)`
- Pie chart: `pie(x)`
- More than one plot: `subplot(mnp)`

Outline

1. 2D plotting
2. 3D plotting
3. Curve fitting

When?

Study data in more than one dimension
e.g. parametric equations:

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

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

Process

1. Define the function
2. Set up a mesh
3. "Study" the function:
    - Contour: `contour(x,y,z)`
    - Color map: `pcolor(x,y,z)`
    - 3D view: `surf(x,y,z)`

Example

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

More 3D plotting

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

Outline

1. 2D plotting
2. 3D plotting
3. Curve fitting

What, why, when?

Engineering, physics, or applied mathematics: many problems and experiments relate several variables

- How do they relate to each other?
- Is it possible to find a model or an equation?

What, why, when?

Engineering, physics, or applied mathematics: many problems and experiments relate several variables

- How do they relate to each other?
- Is it possible to find a model or an equation?

Problem: find the best values to match what is observed

- Parametric fitting
- Non-parametric fitting

Parametric fitting

0. Collect data
1. Import data into MATLAB
2. Open curve fitting tool
3. Determine the best fit
4. Extrapolate the data

Getting started

0. Collect data $\rightarrow$ done: US population from 1790 to 1990
1. Import data into MATLAB $\rightarrow$ `load census`
2. Open curve fitting tool $\rightarrow$ `cftool`

Applying a fit

1. Provide a name for the fit
2. Select *cdate* for "X data"
3. Select *pop* for "Y data"
4. Test various types with different fit names

Finding a better fit

1. View residuals: “View” $\rightarrow$ “Residuals Plot”
2. Change axis limits to predict the future: “Tools” $\rightarrow$ “Axes Limits”
3. Compare the “SSE” and the “Adj R-sq” for the different fits
4. Adjust confidence level: “View” $\rightarrow$ “Prediction Bounds”

Finding a better fit

1. View residuals: “View” $\rightarrow$ “Residuals Plot”
2. Change axis limits to predict the future: “Tools” $\rightarrow$ “Axes Limits”
3. Compare the “SSE” and the “Adj R-sq” for the different fits
4. Adjust confidence level: “View” $\rightarrow$ “Prediction Bounds”

On error/strange results, try to normalize the data

Nonparametric fitting

Goal: draw a smooth curve through some data or interpolate

Function: `interp1(X,Y,xi,m)`, X and Y are two vectors, find a corresponding y_i for x_i using method m

Example.

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

Nonparametric fitting

Goal: draw a smooth curve through some data or interpolate

Function: `interp1(X,Y,xi,m)`, X and Y are two vectors, find a corresponding y_i for x_i using method m

Example.

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

`interp2(X,Y,Z,xi,yi,m)`: find the value of a 2-dimensional function at an intermediate point

Key points

- How to perform 2D plotting?
- How to keep or erase the previous graph?
- What is the use of `plot3`, `contour`, `pcolor`, and `surf`?
- How to measure the quality of a fit?

Chapter 6

Data types and structures

Outline

1. Data types
2. Example of application
3. More data types

Main problematic

Previous chapters:

- 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 of that data

A simple example

Representing signed integers over 8 bits:

1. **Signed magnitude:** 7 bits for the numbers, 1 bit for the sign
   $\rightarrow$ 2 ways to represent 0

A simple example

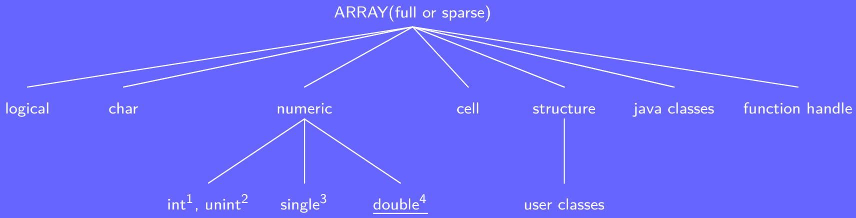
Representing signed integers over 8 bits:

1. **Signed magnitude:** 7 bits for the numbers, 1 bit for the sign
   $\rightarrow$ 2 ways to represent 0
2. **Two's complement:** Invert all the bits and add 1 to negate a number
   e.g. $00101010 \rightarrow 11010101 + 1 = 11010110$
   $00101010 = -0 \cdot 2^7 + 2^5 + 2^3 + 2 = 42$
   $11010110 = -1 \cdot 2^7 + 2^6 + 2^4 + 2^2 + 2 = 86 - 128 = -42$

Why data types?

- Different numbers (integer, real, complex...)
- Different range (short, long...)
- Different precision (single, double...)
- Different types $\Rightarrow$ different memory usage, performance

Data types in MATLAB



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

Numeric types

What is what:

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

Numeric types

What is what:

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

Two methods for numeric conversions:
e.g. `cast(a,'uint8')` or `unit8(a)`

Char type

String: array of Unicode characters, specified by placing data inside a pair of single quotes (e.g. `a='test'; whos a`)

Char type

String: array of Unicode characters, specified by placing data inside a pair of single quotes (e.g. `a='test'; whos a`)

Useful string functions:

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

Example

Exercise.

Input two numbers as strings and calculate their sum

Example

Exercise.
Input two numbers as strings and calculate their sum

Solution.

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

Outline

1 Data types

2 Example of application

3 More data types

The `fread` and `fwrite` functions

`fread(fd, count, precision)`: read count elements of type precision

`fwrite(fd, A, precision)`: write A as elements of type precision

The `fread` and `fwrite` functions

`fread(fd, count, precision)`: read count elements of type precision

`fwrite(fd, A, precision)`: write A as elements of type precision

Important to know what precision is (how many bytes the type is)

Lost in a file?

`ftell(fd)`: offset in bytes of the file position indicator

`fseek(fd,offset,origin)`: go to position offset starting at origin, where origin can be `'bof'`, `'cof'` or `'eof'` (i.e. beginning, current, end)

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

Lost in a file?

Alter the previous sample code by:

- Changing A
- Reading the numbers as `int64`
- Writing the numbers as `double` and reading them as `int8`
- Navigating in the file such as displaying consecutively: the first and fourth elements

Outline

1. Data types
2. Example of application
3. More data types

Structures

Structure: array with “named data containers” called fields.
Fields can be any kind of data

Structures

Structure: array with “named data containers” called fields.
Fields can be any kind of data

```
Student
 |__ Name _____ John Doe
 |__ Gender _____ Male
 |__ Marks _____ A, A+, B-
```

Structures in MATLAB

1. Initializing the structure

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

Structures in MATLAB

1. Initializing the structure

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

2. Using the structure

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

3. To go further: who got the best mark?

Structures in MATLAB

1. Initializing the structure

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

2. Using the structure

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

3. To go further: who got the best mark?

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

Key points

- What is a data type?
- Cite the most common data types
- What is a data structure?
- Why are data structure of a major importance?

Chapter 7

Introduction to C

Outline

1. Basics on C
2. From C to machine code
3. Functions and libraries

The birth of C

- Unix OS implemented in assembly
- New hardware $\rightarrow$ new possibilities
- New possibilities $\rightarrow$ new code
- AT&T Bell Labs 1969–1973
- Ken Thompson + Dennis Ritchie
- C as derived from B a strip-dwon version of BCPL

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
- Microsoft visual C++

Development environment

Common software to write C code:

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

Common C compilers:

- gcc (GNU C Compiler)
- icc (Intel C Compiler)

Outline

1. Basics on C
2. From C to machine code
3. Functions and libraries

A first example

gm-base.c

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

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

A first example

gm-base.c

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

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

Program structure:

- The `main` function is compulsory, and must be unique
- Generic function prototype:

```
OutputType FunctionName(InputType InputName,...){
  function's body
}
```

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

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

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

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

gm-ifndef.c

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

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

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

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. Basics on 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

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

A long program

ans-orig.c

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

Functions and operators used:

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

Organising a long program

ans-main.c

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

ans.c

```c
double answer(double d) {return d+1337;}
```

ans.h

```c
double answer(double d);
```

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

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

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 8

Data types in C

Outline

1. Basics on data types
2. More on data types
3. Beyond data types

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

Initialising variables

Common use:

- Variables for `#define` are UPPERCASE
- Other variables are lowercase
- Variables name not supposed to be longer than 31 characters
- Variable name start with _ or a character

Basic data types

C data types:

- Integer: `int`
- Number with a fractional part, single precision: `float`
- Number with a fractional part, double precision: `double`
- Character: `char`
- Valueless type: `void`

Basic data types

C data types:

- Integer: `int`
- Number with a fractional part, single precision: `float`
- Number with a fractional part, double precision: `double`
- Character: `char`
- Valueless type: `void`

Amount of storage and range of values for each type not defined (except `char`)

Optional specifiers

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`

Optional specifiers

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`

Outline

1. Basics on data types
2. More on data types
3. Beyond data types

Data types

Basic number types:

- `int` size limitation
- `float` 7 digits of precision, from 1.E-38 to 3.E+38
- `double` 13 digits of precision, from 2.E-308 to 1.E+308

Data types

Basic number types:

- `int` size limitation
- `float` 7 digits of precision, from 1.E-38 to 3.E+38
- `double` 13 digits of precision, from 2.E-308 to 1.E+308

The type must be defined before using a variable

Example.

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

Data types

Characters:

- No type for strings, only for single characters
- Strings as arrays of characters
- Characters are enclosed in single quotes: `char a='a';`
- Strings are enclosed in double quotes

Data types

Characters:

- No type for strings, only for single characters
- Strings as arrays of characters
- Characters are enclosed in single quotes: `char a='a';`
- Strings are enclosed in double quotes

American Standard Codes for Information Interchange (ASCII)

Wrong data type

types1.c

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

Wrong data type

types1.c

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

types2.c

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

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;`

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;`

Danger!

Think of the size...

Try double to char, int to float

Example

types3.c

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

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`?

Outline

1. Basics on data types
2. More on data types
3. Beyond data types

What, why data types?

More data types in C:

- Bits $\in \{0, 1\}$, 1 byte $=$ 8 bits
- Operating data at low level, e.g. shift `«`, `»`
- `char` does not necessarily contains a character
- Logical operation of a major importance
- Focus on efficiency, data representation
- Structures, enumerate, union

st.c

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

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

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

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?

Key points

- What are the main data types in C?
- How to perform type casting?
- Can a `char` contain something else than a character?
- How to define and use structures on C?

Chapter 9

Syntax and control statements

Outline

1. General syntax
2. Conditional statements
3. Loops

Program structure

Reminder:

- First lines
- Function name
- Brackets
- Input
- Output
- End of line

Blocks

blocks.c

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

Shorthand operators

Common shortcuts:

- Increment: ++, e.g. `a++` $\Leftrightarrow$ `a=a+1`
- Decrement: −−, e.g. `a--` $\Leftrightarrow$ `a=a-1`
- Add: `x+=y` $\Leftrightarrow$ `x=x+y`
- Subtract: `x-=y` $\Leftrightarrow$ `x=x-y`
- Multiply: `x*=y` $\Leftrightarrow$ `x=x*y`
- Divide: `x/=y` $\Leftrightarrow$ `x=x/y`

Jumping!

jump.c

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

Understanding the code:

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

Example

Write a short C program where the main function calls a function "apbp1" which takes two floats a and b as input and returns the nearest integer to $a + b + 1$.

Hint: how to round a real number to the nearest integer?

Solution

apbp1.c

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

Solution

apbp1.c

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

Questions: how are shorthand operators and casting used here?

Outline

1. General syntax
2. Conditional statements
3. Loops

Important operators

Basics on conditional statements:

- No boolean type
- $0 \Leftrightarrow$ `False`, $\neq 0 \Leftrightarrow$ `True`
- $<, <=, >, >=, ==, !=$: return 1 if `True`, 0 otherwise
- Not: !, and: &&, or: ||
- Operations on bits: and: &, or: |, xor: ^

Note: && and || are short-circuit operators; second operand only evaluated if result not fully determined by first one

A new operator

Conditional ternary operator: `?:`

```
condition ? expression1 : expression2
```

Useful especially in macros

E.g. Write a macro which returns the max of two numbers

A new operator

Conditional ternary operator: `?:`

```
condition ? expression1 : expression2
```

Useful especially in macros

E.g. Write a macro which returns the max of two numbers

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

The `if` and `switch` statements

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

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

Example

cards.c

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

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

Outline

1. General syntax
2. Conditional statements
3. Loops

The `while` and `do...while` statements

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

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

The while and do...while statements

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

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

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

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

The `for` statement

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

- `init`: executed at the beginning of the loop
- `test`: executed at the beginning of each iteration to determine whether to stop or continue
- `step`: executed at the end of each iteration

The `for` statement

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

- `init`: executed at the beginning of the loop
- `test`: executed at the beginning of each iteration to determine whether to stop or continue
- `step`: executed at the end of each iteration

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

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

The `break` and `continue` statements

Acting from within a loop:

- Exit a loop: `break`
- Go to the next iteration in the loop: `continue`

The `break` and `continue` statements

Acting from within a loop:

- Exit a loop: `break`

- Go to the next iteration in the loop: `continue`

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

Key points

- How are blocks defined?
- What are the shorthand operators?
- How to perform condition statements in C?
- How to write loops in C?

Chapter 10

Arrays and pointers

Outline

1. Arrays
2. Pointers
3. Pointers and arrays

What is an array?

Array: indexed collection of values of a same type (e.g. array of integers, floats...), first index being 0

Array declared through three parameters:

- type
- name
- size

e.g. `int a[3];` or `char b[4];`

What is an array?

Array: indexed collection of values of a same type (e.g. array of integers, floats. . .), first index being 0

Array declared through three parameters:

- type
- name
- size

e.g. `int a[3];` or `char b[4];`

What are the sizes in bits of the two previous arrays?

Example

```
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

Example

```
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

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

Arrays and functions

array-fct.c

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

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?

Problem

The following C program simulates multiple die rolls and print how many times each side appears. Adapt it to handle two dice.

die.c

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

Question: how is the array initialized?

Solution

dice.c

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

Solution

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

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

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 (10.210) using a function taking dice, sides, and rolls as input
- Explain how multi-dimensinoal 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

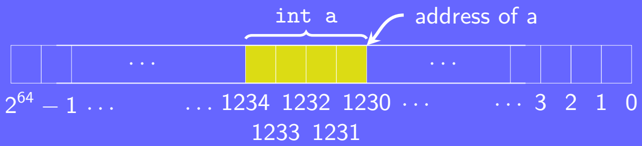
What is a pointer?

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



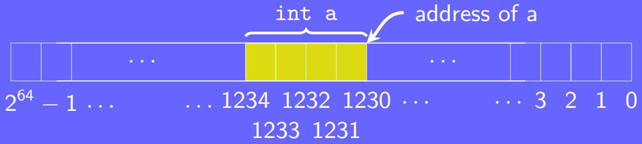
What is a pointer?

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

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

swap-p.c

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

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

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

Questions:

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

Dynamic memory

Possible to allocate memory as a block of a certain type (e.g. a block of *n* integers or floats)

- `malloc(n)`: allocates *n* bytes of memory, and returns a pointer on the first chunk (address of the first chunk)
- `calloc(n,s)`: allocates *n* chunks of memory, each one of size *s* bytes, and returns a pointer on the first chunk (address of the first chunk); memory is set to 0
- `realloc(ptr,s)`: changes the size of the memory block pointed to by *ptr* to *s* bytes
- `free(ptr)`: frees the memory space pointed to by *ptr*

Accessing memory

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

- Accessing first chunk
- Acessing the 5th chunk

Accessing memory

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

- Accessing first chunk

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

- Acessing the 5th chunk

Accessing memory

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

- Accessing first chunk

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

- Acessing the 5th chunk

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

Accessing memory

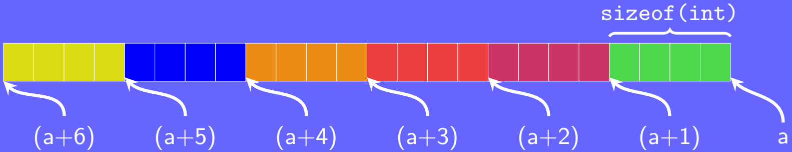
```c
int *a=malloc(6*sizeof(int));
```

- Accessing first chunk

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

- Acessing the 5th chunk

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



Question: what is (a+6)?

Pointers and structures

str-p.c

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct person {
  char* name; int age;
} person_t;
int main () {
  person_t al={"albert",32};
  person_t* group1=malloc(3*sizeof(person_t));
  group1->name="gilbert";
  group1->age=34;
  *(group1+1)=(person_t){"joseph",28};
  (*(group1+2)).name="emily";
  (group1+2)->age=42;
  printf("%s %d %d\n",al.name, al.age, sizeof(person_t));
  printf("%s %d\n",(group1+1)->name, (group1+2)->age);
  free(group1);
  return 0;
}
```

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:

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

Outline

1. Arrays
2. Pointers
3. Pointers and arrays

Pointer vs. array

arr-ptr.c

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

Pointer vs. array

arr-ptr.c

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

An array *contains* the elements, a pointer *points* to them.

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

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

Revisiting the dice

dice-mp.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void roll_dice(int dice, int sides, int rolls){
  int i, j, t;
  int *res=calloc((dice*sides-dice+1),sizeof(int));
  int *table=malloc(dice*rolls*sizeof(int));
  for(i=0;i<rolls;i++) {
    for (j=0; j < dice; j++) table[i*dice+j]=(rand()%sides)+1;
  }
  for (i=0;i<rolls;i++) {
    t=0; for(j=0;j<dice;j++) t+=table[i*dice+j]; res[t-dice]++;
  }
  for (i=0;i<dice*sides-dice+1;i++) printf("%d (%d) ",i+dice,res[i]);
  printf("\n"); free(table); free(res);
}
int main () {
  int dice=4, sides=6, rolls=1000000;
  srand(time(NULL)); roll_dice(dice,sides,rolls);
}
```

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?

Arrays, pointers and functions

Problem:

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

Arrays, pointers and functions

Problem:

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

Solution: pointers

Arrays, pointers and functions

Problem:

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

Solution: pointers

Back to the swap function (10.215)

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?
- When memory has been allocated and is not needed anymore, what must be done?
- How to have more than one output in a function?

Chapter 11

Algorithm and efficiency

Outline

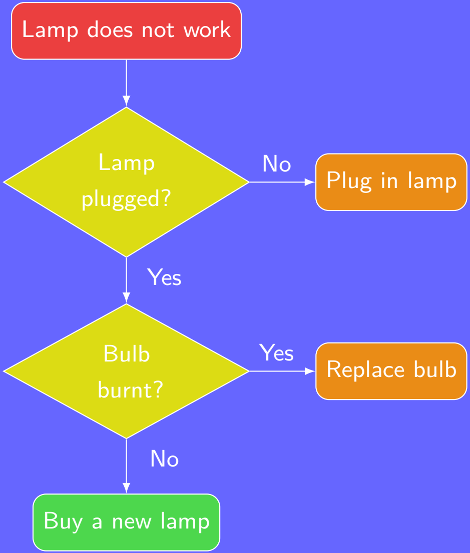
1. Algorithms
2. Standard library
3. A few final examples

What is already known

Reminders:

- Algorithm $\Leftrightarrow$ recipe
- 3 main components:
  - Input
  - Output
  - Instructions
- Clear algorithm often easy to implement
- Adjust algorithm to fit the language

Flowchart


Lamp does not work
Lamp plugged?
No
Plug in lamp
Yes
Bulb burnt?
Yes
Replace bulb
No
Buy a new lamp

Design paradigms

Most common types of algorithms:

- Brute force $\rightarrow$ often most obvious, rarely best
- Divide and conquer $\rightarrow$ often recursive
- Search and enumeration $\rightarrow$ model problem using a graph
- Randomized algorithms $\rightarrow$ feature random choices
  - Monte Carlo algorithms $\rightarrow$ correct answer with high probability
  - Las Vegas algorithms $\rightarrow$ alway correct answer but random running time
- Complexity reduction $\rightarrow$ rewrite a problem into an easier one

Efficiency

When writing a program:

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

Computational complexity

Complexity: measures how hard it is to solve a problem

Common complexity types:

- Best-case complexity
- Average-case complexity
- Worst case-complexity
- Time vs. space complexity

Outline

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

<stdio.h>

Traveling 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);`

`<stdio.h>`

Reading and writting:

- 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);`

<string.h>

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 is a string:
  `char *strchr(const char *s, int c);`

<string.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);`

`<ctype.h>`

Classifying elements (returns 0 if FALSE and nonzero if TRUE):

- `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 uppercase or lowercase

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

`<math.h>`

A few mathematical functions (input and output are doubles):

- 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)`

`<stdlib.h>`

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);`
  e.g. `atoi( "512.035" );` returns 512
- 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 *));
  ```

`<time.h>`

Useful functions for simple benchmarking:

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

Outline

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

Linear search

linear-search.c

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

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

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

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

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

Selection sort

- 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 12

Introduction to C++

Outline

1. Before starting with C++
2. C and C++
3. C++ syntax

A bit of history

- Bjarne Stroustrup
- BCPL too low level
- Simula too slow
- 1979: C with classes
- 1983: 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

Describing C++

C++ in a few words:

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

Reasons for using C++?

Highlights of C++:

- Performance
- Higher level than C
- Code often shorter/cleaner
- Safer (more errors caught at compile time)
- No runtime overhead

Outline

1. Before starting with C++
2. C and C++
3. C++ syntax

C vs. C++

What C++ brings:

- All aspects of C preserved
- Add new features
- Easier to write sophisticated programs

C++ is almost a superset of C

C or C++?

prg.cpp

```
#include <stdio.h>

int main () {

  int a=5;
  printf("%d\n",a);

}
```

Why easier?

A new appraoch:

- Easier to manage memory
- Object oriented programming
- New features for generic programming

Why easier?

A new appraoch:

- Easier to manage memory
- Object oriented programming
- New features for generic programming

Programmer focuses more on his problem and less on how to explain it to the machine

A difference

C

Implicit assignment from `*void`:

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

C++

No implicit assignment from `*void`:

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

Outline

1. Before starting with C++
2. C and C++
3. C++ syntax

Basics

Most of the syntax similar to C:

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

Novelties

- New type:

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

- New headers format:

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

Namespace

A wider perspective:

- C: function names conflicts among different libraries
- C++: introduction of *namespace*
- Each library/program has its own namespace
- Standard library: `std`

Input/Output

Handling I/O without `printf`, `scanf`

- Ouput:

```
cout << "Enter a number (-1 = quit): ";
```

- Input:

```
cin >> x;
```

Input

input-pb.cpp

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

Challenge: input a letter… and exit

input-ok1.cpp

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

Input

input-ok2.cpp

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

Formating output

Nicer display requires `#include <iomanip>`

- Field width: `setw(width)`
- Justification: `setiosflags(ios::left)`
- Precision: `setprecision(2)`
- Leading character: `setfill('z')`

Example

date.cpp

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
void showDate(int m, int d, int y) {
  cout.fill('0');
  cout << setw(2) << m << '/' << setw(2) << d << '/'
       << setw(4) << y << endl;
}
int main(){
  showDate(6,10,2014);
  cout << setprecision(3) << 1.2244 << endl;
}
```

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

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

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`

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

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

Strings

Improvements on strings:

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

```
#include <string>
string g="good "; string m="morning";
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)`

Example

Problem: copy the content of a text file into another text file and display each line on the console output

Example

Problem: copy the content of a text file into another text file and display each line on the console output

fio.cpp

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

Example

fio-c.cpp

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

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
- Sides 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

```
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 13

Object and class

Outline

1. Basic concepts
2. Writing and implementing a class
3. Dealing with objects

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

Outline

1. Basic concepts
2. Writing and implementing a class
3. Dealing with objects

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

Default behavior: private
Good practice: render public only if necessary

Example

circle-v0.h

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

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`

Example

main-v0.cpp

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

Implementation

Getting things ready:

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

Example

circle-v0.cpp

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

Outline

1. Basic concepts
2. Writing and implementing a class
3. Dealing with objects

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 $\sim$`classname`

Example

circle-v1.h

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

Example

circle-v1.cpp

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

Example

main-v1.cpp

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

Life span

Three kinds of objects:

- Static or global: same life span as the program
- Automatic or local: within a block
- Dynamic: created and deleted manually

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.

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

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

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

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

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

Key points

- How to describe an object?
- In what order should the attributes and methods be defined?
- What are private and public?
- How to use the constructor and destructor?

Chapter 14

Inheritance and polymorphism

Outline

1. Inheritance
2. Polymorphism
3. Multiple inheritance

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

cows-0.cpp

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

Managing a sick cow

What a sick cow does:

- Everything a cow does
- Take its medication

Managing a sick cow

What a sick cow does:

- Everything a cow does
- Take its medication

Two obvious possible strategies:

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

What is inheritance?

Definitions:

- Act of inheriting
- Transmitting characteristics from the parents to the children

What is inheritance?

Definitions:

- Act of inheriting
- Transmitting characteristics from the parents to the children

Example.
A sick cow inherits all the characteristics from a cow:

- Attributes and methods from a cow
- More attributes and methods can be added

cows-1.cpp

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

Private

Reminder on private members:

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

Private inheritance:

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

Public

Reminder on `public` members:

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

Public inheritance:

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

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

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

Inheritance or not inheritance?

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

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

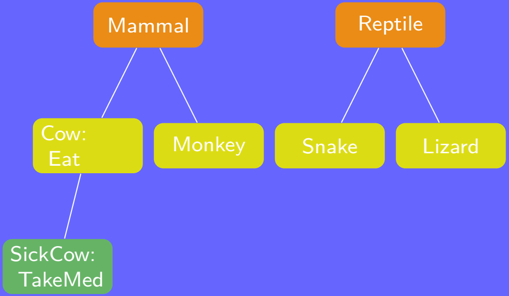
```
class Zoo {
  public:
    Mammal *m; Reptile *r;
    ...
};
```

Remark.
On a drawing:

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

Diagram

Representing the relationships using diagrams:



Zoo:
Reptile
Mammal
...

Outline

1. Inheritance
2. Polymorphism
3. Multiple inheritance

Polywhat????

Poly-morphism

poly: many

morphe: form/shape

Polywhat????

Poly-morphism

poly: many

morphe: form/shape

Simple idea:

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

Function overloading

cows-3.cpp

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

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

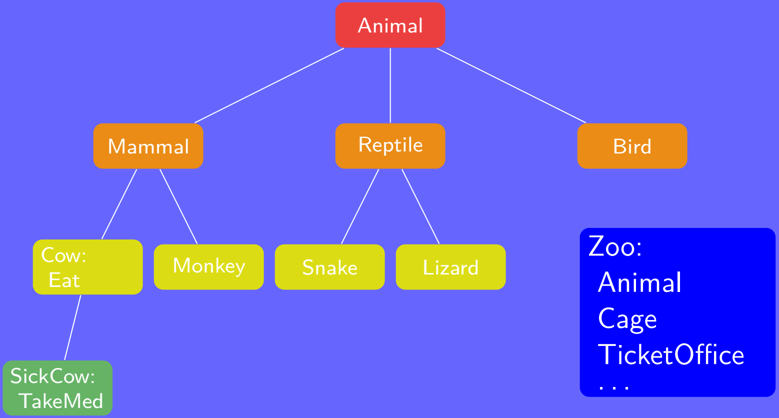
Fixing the cows

cows-4.cpp

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

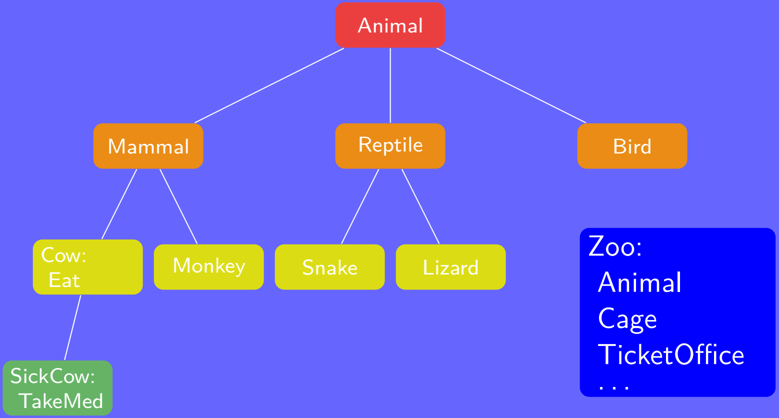
Extending the idea

Applying the same idea to generalize the diagram:



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.

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

Animals

animals.h

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

Animals

animals.cpp

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

Benefits of polymorphism

animals-main.cpp

```
#include <iostream>
#include "animals.h"
using namespace std;
void FeedAnimals(Animal **a, int ZooSize) {
  for(int i=0; i<ZooSize; i++) {
    cout << endl; a[i]->Speak(); a[i]->Eat();//a[i]->TakeMed();
    delete a[i];
  }
};
void ZooInit(Animal **a, int ZooSize) {
  for(int i=0; i<ZooSize; i++) {
    switch(i%4) {
      case 0: a[i]=new Cow; break; case 1: a[i]=new SickCow; break;
      case 2: a[i]=new Monkey; break; case 3: a[i]=new Monkey(1); break;
    }
  }
}
int main () {
  int ZooSize=10; Animal *a[ZooSize];
  ZooInit(a,ZooSize); FeedAnimals(a,ZooSize);
}
```

Outline

1. Inheritance
2. Polymorphism
3. Multiple inheritance

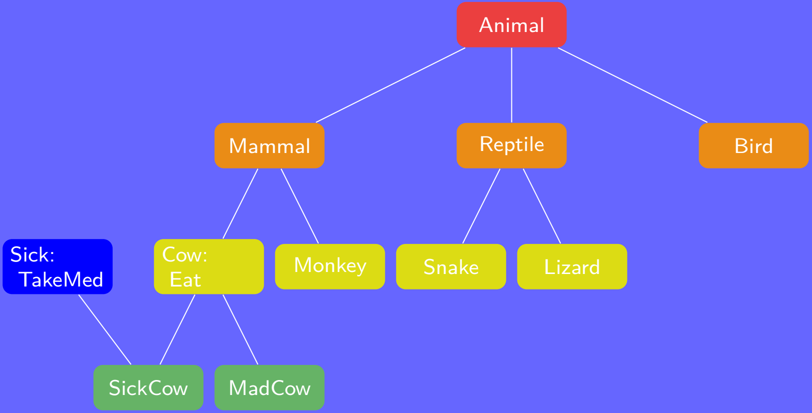
Multiple inheritance

Simple idea: a class can inherit from multiple classes

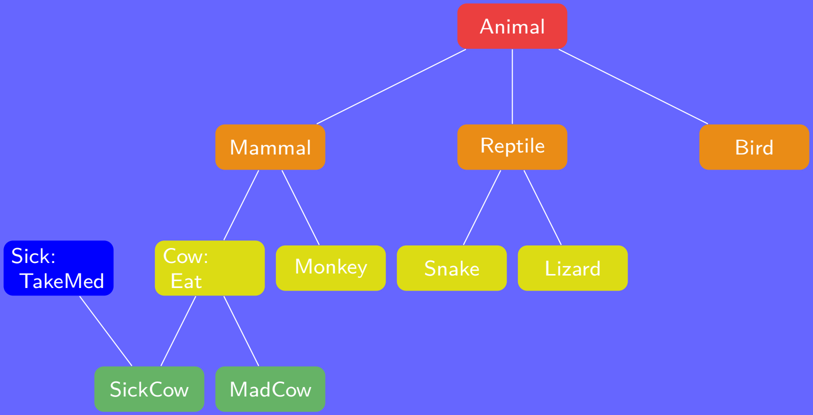
Example.
Any sick animal should be put under medication:

- 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




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

More cows

animals-m.h

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

More cows

animals-m.cpp

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

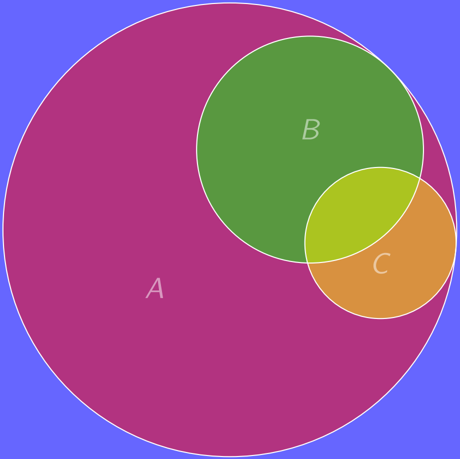
More cows

animals-main-m.cpp

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

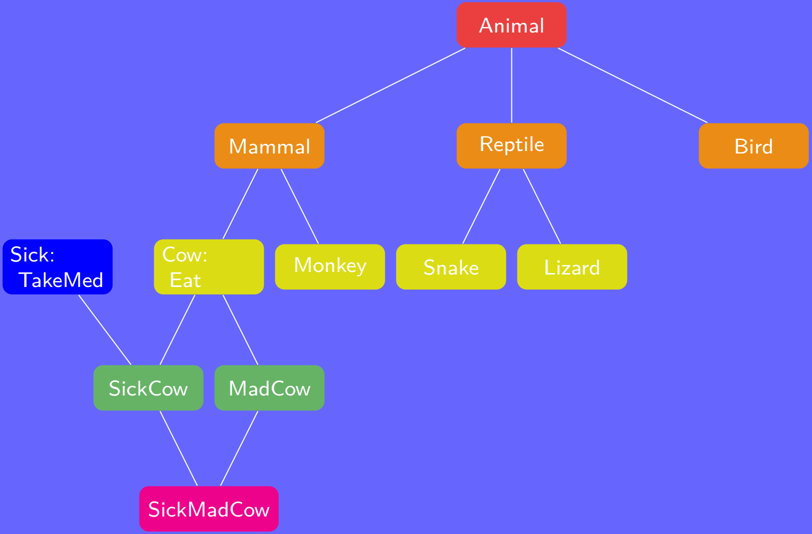
The diamond problem

Multiple inheritance can be tricky:



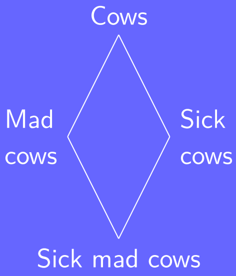
- A: Cows
- B: Sick cows
- C: Mad cows
- Sick mad cows are in $B \cap C$

The diamond problem

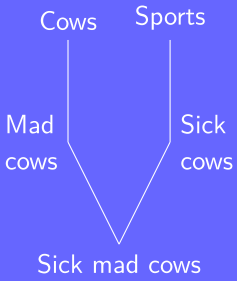


The diamond problem

Human perspective

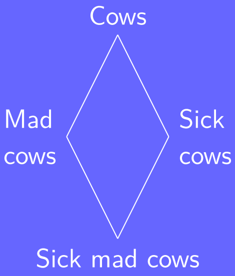


Computer perspective

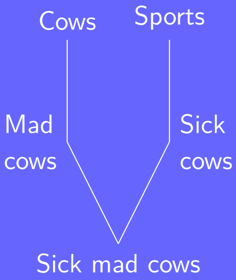


The diamond problem

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

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

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

The diamond problem

Solutions to overcome the problem:

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

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

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

Important: if the diamond problem appears in a diagram, redesign the whole hierarchy

Sick mad cows

animals-d.h

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

Sick mad cows

animals-d.cpp

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

Sick mad cows

animals-main-d.cpp

```cpp
#include <iostream>
#include "animals_d.h"
using namespace std;
int main () {
  SickCow c1(1,1);
  c1.Speak(); c1.Eat(); c1.TakeMed();
  c1.Eat(); c1.TakeMed();
  cout << endl;
  MadCow c2(1,1);
  c2.Speak(); c2.Eat(); c2.TakePills();
  c2.Eat(); c2.TakePills();
  cout << endl;
  SickMadCow c3(1,1,1);
  c3.Speak(); c3.Eat(); c3.TakePills(); c3.TakeMed();
  c3.Eat(); c3.TakePills(); c3.TakeMed();
}
```

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 are the three main concepts of object oriented programming?
- Why using inheritance?
- What is polymorphism?
- What are the pros and cons of the keyword `virtual`?
- What is the best way to solve the diamond problem?

Chapter 15

Libraries and templates

Outline

1. Using external libraries
2. Writing templates
3. Using the 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...

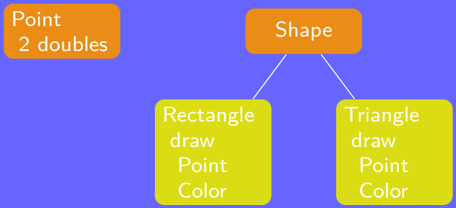
The OpenGL library

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...

Goal: wrap the C functions into classes to construct a home

Hierarchy diagram



Home:
- draw
  - 5 shapes
- zoom
  - in and out

Figures specification

home/figures.h

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

Figures implementation

home/figures.cpp

```cpp
#include <GL/glut.h>
#include "figures.h"
Shape::~Shape(){}
Rectangle::Rectangle(Point pt1, Point pt2,
    float red, float green, float blue) {
  p1=pt1; p2=pt2; r=red; g=green; b=blue;
}
void Rectangle::draw() {
  glColor3f(r, g, b); glBegin(GL_QUADS);
  glVertex2f(p1.x, p1.y); glVertex2f(p2.x, p1.y);
  glVertex2f(p2.x, p2.y); glVertex2f(p1.x, p2.y); glEnd();
}
Triangle::Triangle(Point pt1, Point pt2, Point pt3,
    float red, float green, float blue) {
  p1=pt1; p2=pt2; p3=pt3; r=red; g=green; b=blue;
}
void Triangle::draw() {
  glColor3f(r, g, b); glBegin(GL_TRIANGLE_STRIP);
  glVertex2f(p1.x, p1.y); glVertex2f(p2.x, p2.y); glVertex2f(p3.x, p3.y);
  glEnd();
}
```

Home specification

home/home.h

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

Home implementation (part 1)

home/home-part1.cpp

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

Home implementation (part 2)

home/home-part2.cpp

```
void Home::draw() {for(int i=0;i<5;i++) sh[i]->draw();}
void Home::zoom(double *width, double *height, double *owidth){
  int static i=0;
  if(h>=0.1 && i==0) zoomout(width, height, owidth);
  else if (h<=2) { i=1; zoomin(width, height, owidth); }
  else i=0;
}
void Home::zoomout(double *width, double *height, double *owidth){
  h/=1.01; *height=h; w/=1.01; *width=w; o/=1.01; *owidth=o;
}
void Home::zoomin(double *width, double *height, double *owidth){
  h*=1.01; *height=h; w*=1.01; *width=w; o*=1.01; *owidth=o;
}
void Home::setcolor(float *r, float *g, float *b) {
  *r=(float)rand()/RAND_MAX; *g=(float)rand()/RAND_MAX;
  *b=(float)rand()/RAND_MAX;
}
```

Home instantiation

home/main.cpp

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

Basics

Basic process with OpenGL:

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

Remarks

Understanding the code:

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

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

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

Outline

1. Using external libraries
2. Writing templates
3. Using the Standard Template Library

Classes

Limitations of inheritance and polymorphism:

- High level classes (boat, company, car...)
- Low level classes used to define high level ones
- Need to apply a function to more than one type: function overloading

Problem: duplicated code, more complex to debug...

Defining a template

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

Defining a template

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

complex.h

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

Using a template

Provide the class name with the data type:

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

Using a template

Provide the class name with the data type:

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

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

Solution

complex.cpp

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

A bit of history

A few dates:

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

Conclusion templates:

- Are very powerful, complex and new
- Are not always handled nicely
- Might display long and unclear error messages
- Do not always benefit from optimizations
- Require much work from the compiler

Basics on STL

C++ is shipped with a set of temples:

- 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

Outline

1. Using external libraries
2. Writing templates
3. Using the Standard Template Library

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

Vector: sequence representing arrays that can change size

- Size automatically adjusted
- Template: no specific initial type
- Example:

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

- A few useful functions: `push_back`, `pop_back`, `swap` vect.cpp

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

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

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

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

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

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

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

Example

iterator.cpp

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

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

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

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

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

Unique

Remove consecutive duplicate elements

unique1.cpp

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

Sort elements in ascending order

sort.cpp

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

Problem

Remove all duplicate elements from the color vector.

Solution

unique2.cpp

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

Reverse

Reverse the order of the elements

reverse.cpp

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

Any other possible strategy?

Remove

Remove elements and returns an iterator to the new end

remove.cpp

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

Random_shuffle

Randomly rearrange elements

random.cpp

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

Max and min

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

minmax.cpp

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

Key points

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

Chapter 16

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)

Makefile

Makefile

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

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

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

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

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

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

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

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

Constant pointers vs. pointer to constant

Constant pointer

```
int const *p;
```

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

Pointer to constant

```
const int *p;
```

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

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

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

ref.cpp

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

The `this` pointer

The `this` keyword:

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

boat.cpp

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

Pointer to function

Similar to pointer to variables:

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

fctptr.c

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

The enum and union keywords

enum-union.c

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

The argc and *argv[] parameters

arg.c

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

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

Compilation 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. . .) $\rightarrow$ 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. . .