B16 Part 1: Structured Programming

- **Software engineering principles**
  - Design, modularity, abstraction, encapsulation, etc.

- **Structured programming languages**
  - Interpreted (MATLAB) vs compiled (C) languages
  - **Control flow**
    - Sequencing, alternation, iteration
    - Functions and libraries
  - **Data**
    - Data types: primitive, aggregate, and compound
    - Local and global variables, parameters
    - The heap and the stack

- **Algorithms**
  - Proving algorithm correctness by mathematical induction
  - Time and space complexity
  - Recursion

**Texts**

- The C programming language, 2nd edition
  - Kernaghan & Ritchie
- Introduction to algorithms
  - Cormen, Leiserson, Rivest, Stein
The challenge of building software
- The size of code

Software engineering
- The aims and scope of software engineering
- Abstraction and modularity

Software processes & their models
- Specification, design & implementation, validation, evolution
- Waterfall and incremental models

Structured programming
- Structuring programs by using abstractions in a programming language
- Types of languages: imperative vs declarative
- Fundamental abstractions

The “size” of software
- SLOC: number of source lines of code

```
#include "fisher.h"
#include "gmm.h"
#include "mathop.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void
VL_XCAT_VL_fisher_encode_, SFX)
(TYPE * enc,
TYPE const * means, vl_size dimension, vl_size numClusters,
TYPE const * covariances,
TYPE const * priors,
TYPE const * data, vl_size numData,
int flags)
{
    vl_size dim;
    vl_index i_cl, i_d;
    TYPE * posteriors ;
    TYPE * sqrtInvSigma;

    posteriors = vl_malloc(sizeof(TYPE) * numClusters * numData);
    sqrtInvSigma = vl_malloc(sizeof(TYPE) * dimension * numClusters);
    
    for (i_cl = 0 ; i_cl < (signed)numClusters ; ++i_cl) {
        for (dim = 0; dim < dimension; dim++) {
            sqrtInvSigma[i_cl*dimension + dim] = sqrt(1.0 / covariances[i_cl*dimension + dim]);
        }
    }
    VL_XCAT(vl_get_gmm_data_posteriors_, SFX)(posteriors, numClusters, numData, 
                                             priors, 
About 50 SLOCs

worth of code

The impact of software
- Control
  - Digital manufacturing
  - Digital products, from kitchen appliances to nuclear plants

- Design
  - Numerical simulation
  - Computer-assisted design

- Data processing and analysis
  - Extremely large datasets (big data)
  - Sciences, humanities, healthcare

Communication
- Digital telephony
- Computer networks
- Internet, electronic commerce, digital economy

Entertainment
- Computer graphics, music
- Gaming
- Digital arts

large-scale computations
ubiquitous computations
interconnectivity
intelligence
What 130K lines of code look like

Apollo-11, 1969

Margaret Hamilton,
director of software engineering

Code available here: https://github.com/chrislgarry/Apollo-11/

Lecture 1 outline

- The challenge of building software
  - The size of code
- Software engineering
  - The aims and scope of software engineering
    - Abstraction and modularity
- Software processes & their models
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models
- Structured programming
  - Structuring programs by using abstractions in a programming language
  - Types of languages: imperative vs declarative
  - Fundamental abstractions

Software engineering

Aims

Software engineering seeks **principles** and **methodologies** to make programs that are:

- **Usable**
  - Meet their requirements, including being acceptable by the users
- **Dependable**
  - Reliable, secure, safe
- **Maintainable**
  - Can be updated with minimal effort
- **Efficient**
  - Run as fast as possible with limited resources

Scope

Software engineering is concerned with all aspects of software production. It includes:

- **Theory**
  - Computability, algorithms, correctness, complexity, formal languages
- **Tools and best practices**
  - Specific programming languages, programming environments, developer tools to build, debug, and analyse programs
- **Management**
  - Processes of software creation & maintenance

Abstraction and modularity

Software systems are far too complex to be tackled as a whole. Engineers adopt a reductionist approach based on:

- **Modularity**
  - Decompose the system into smaller components
- **Abstraction**
  - Each component behaviour has a simple description, independent of the other components and of the internal implementation

Benefits

- **Understandability**
  - Complexity of individual components is commensurate to human brainpower
- **Reuse**
  - The same component can be used in many applications (e.g. transistors)
- **Isolating changes**
  - The implementation of a component (e.g. transistor materials) can be changed as long as the behaviour (e.g. electrical properties) does not
**Examples**

**Abstraction and modularity**

- Transistor
  \[ I_C = \beta I_B, \]
  \[ I_E = I_C + I_B, \]
  \[ V_{EB} = 60 \text{ mV} \]

- Operational amplifier
  \[ V_+ = V_-, \]
  \[ I_+ = 0 \text{ A}, \]
  \[ I_- = 0 \text{ A} \]

---

**Lecture 1 outline**

- **The challenge of building software**
  - The size of code

- **Software engineering**
  - The aims and scope of software engineering
  - Abstraction and modularity

- **Software processes & their models**
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models

- **Structured programming**
  - Structuring programs by using abstractions in a programming language
  - Types of languages: imperative vs declarative
  - Fundamental abstractions

---

**Software processes**

A *software process* is a set of related activities that leads to the production of a software product [Sommerville].

Key activities of a software process:

- **Specification**
- **Design & implementation**
- **Verification & validation**
- **Evolution**
**Specification**

- **Elicit requirements** from users
- **Refine** requirements until they are sufficiently precise
- **Produce a requirement specification document**

**Example**
- **Elicitation**: “The program should calculate the trajectory of a lossy bouncing ball”
- **Refinement**: How is the output represented? How are the initial condition specified? The required numerical accuracy? Should air resistance be accounted for? How fast must the program run?
- **Requirement specification**:
  - Input: initial position and momentum, ball radius, restitution coefficient, ...
  - Output: simulated trajectory, 60 Hz sampling rate, 10⁻³ m tolerance, ...
  - Must complete simulation in 1ms

**Design & implementation**

- **Design**
  - Modules
    - Architecture which components
    - Data flow: how components inter-operate
    - Components: how components work
  - Interfaces
    - Programmatic interface
    - Text-based interface
    - Graphical interface
  - Databases
    - Data nature and relationships
    - Storage details
    - Security

- **Implementation**
  - A team of programmers writes each component
  - Low-level design details (e.g. specific names of local variables) are left to the programmers
  - Writing functionalities for debugging are part of the implementation (e.g. unit testing)

**Verification & validation**

- **Verification**
  - Black-box (from software specification) vs white-box (from code inspection)
  - Top-down (system testing) vs bottom-up (unit/component testing)

- **Coverage of the tests**
  - Exhaustive testing is impossible
  - Pick representative examples of “normal” inputs as well as “corner cases”
  - **Example**: Test a function to compute the tangent
    - normal input: \( \tan(1.1) \)
    - corner cases: \( \tan(-\pi/2) \), \( \tan(0) \), \( \tan(\pi/2) \)

**Evolution**

- **Cost of software evolution**
  - **Effort**: Engineers might not understand the system anymore, must study it
  - **Risk**: Changes may have unpredictable effects

- **Reengineering software**
  - **Refactoring**
    - Improve design, quality of code
  - **Rewrite** might be needed to:
    - Switch to a new, more modern basis (e.g. new programming language)
    - Scrap old design for a new one
    - Usually keep the interface or at least the functionality equivalent
Waterfall model

Ideally, activities are distinct stages to be signed off chronologically
In practice, activities partially overlap and interact

Lecture 1 outline

The challenge of building software
- The size of code

Software engineering
- The aims and scope of software engineering
- Abstraction and modularity

Software processes & their models
- Specification, design & implementation, validation, evolution
- Waterfall and incremental models

Structured programming
- Structuring programs by using abstractions in a programming language
- Types of languages: imperative vs declarative
- Fundamental abstractions

Imperative languages

The most common programming languages are imperative.
An imperative program is a list of instructions to be executed in the specified order to solve a problem.
Different imperative languages are characterised by different abstractions.

Machine Code (Intel x86)

```
0: 55
1: 48 89 e5
4: c7 45 fc 01 00 00 00
b: c7 45 f8 02 00 00 00
12: 8b 45 fc
15: 03 45 f8
18: 89 45 f4
1b: 8b 45 f4
1e: 5d
1f: c3
```

Machine Language (x86)

```
pushq %rbp
movq %rsp, %rbp
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
addl -8(%rbp), %eax
movl %eax, -12(%rbp)
popl %rbp
retq
```

Machine language's main abstraction are mnemonics (readable names for instructions, registers, etc.)
Imperative languages

- Abstractions can have a massive impact on the ease of use, understandability, maintainability, power, and efficiency of programming languages.

Machine Language (x86)

```
pushq %rbp
movq %rsp, %rbp
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
addl -8(%rbp), %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
popq %rbp
retq
```

C Language

```
int f()
{
  int x = 1;
  int y = 2;
  int z = x + y;
  return z;
}
```

Declarative (functional) languages

- A declarative program specifies the desired behaviour of the program, but now how this is achieved in term of elementary steps.

Regular expression

```
[a-z]*
```

This means that the program should match any string consisting only of letters from 'a' to 'Z'.

It says what the program should do.

Regular expression

```
[a-z]*
```

This is a C implementation of the same program.

It specifies how to solve the problem in terms of elementary steps.
Structure in imperative languages
An overview of fundamental abstractions

Data
Data types: elementary, aggregate and compound.
Variables.

Control flow
Blocks, conditionals, loops, switches.
Statements.

Procedural languages
Functions, data scoping, encapsulation, recursion.

Object-oriented programming (Part II of the course)
Attach behaviour to data.

Hello world!
Getting started with C programming

```
#include <stdio.h>
int main(int argc, char** argv)
{
  printf("Hello, world!\n") ;
  return 0 ;
}
```
Getting started with C programming

Hello world!

B16 Software Engineering
Structured Programming
Lecture 2: Control flow, variables, procedures, and modules

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see http://www.robots.ox.ac.uk/~vedaldi/teach.html

Lecture 2 outline

- Control flow
  - Imperative languages
  - Goto (considered harmful)
  - Blocks, conditionals, loops
- State
  - Variable
  - Data types
  - Static vs dynamic typing
- Compiled vs interpreted language
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files
- Practical notes
  - Clean vs obfuscated code
  - Avoid cut & paste
Control flow

An imperative program is a list of statements (instructions) to execute.

- Statements are executed sequentially.
- The program counter (PC) is a register pointing to the current instructions.
- It is incremented to move to the next instruction.

10 sleep eight hours
11 wake up
12 have breakfast

Branching

- Statements are executed sequentially.
- Branching statements allow for non-sequential execution, conditionally on the state of the program.
- Branching is performed by resetting the program counter.

13 if today is Saturday then goto 10
14 leave home

Goto and line numbers (don’t)

- To insert a line of code, one uses an intermediate line number (!)

10 i ← 0
20 i ← i + 1
30 print i, " squared is ", i * i
40 if i >= 10 then goto 60
50 goto 20
60 print "that’s all folks!"

Goto and labels (still don’t)

- Labels are an abstraction of line numbers and simplify the use of goto.
- A label is just a name given to a statement in the sequence.

i ← 0
more:
  i ← i + 1
  print i, " squared is ", i * i
if i >= 10 then goto end
goto more
end:
  print "that’s all folks!"

[Spaghetti code]

Structured control flow

Goto is (almost) never used. Any program can be expressed in terms of three simple control structures [Böhm-Jacopini 66]:

- Blocks (sequences of executable statements)
  - { do_this ; do_that ; do_something_else ; }

- Conditionals (execute a block if a condition is true)
  - if (condition) {}

- Loops (keep executing a block until a condition remains true)
  - while (condition) {}
Structured control flow

Code blocks

- The code is much easier to understand because each block has
  - only one entry point at the beginning
  - only one exit point at the end

Example

Spaghetti monster

```
more:
i ← 0
i ← i + 1
print i, " squared is ", i * i
if i >= 10 then goto end
goto more
end:
print "that's all folks!"
```

Structured program

```
{i
  i ← 0
  while (i < 10)
  {
    i ← i + 1
    print i, " squared is ", i*i;
  }

print "that's all folks!"
}
```

MATLAB vs C

### C version

```c
#include <stdio.h>

void print_n_squared_numbers(int n)
{
  int i = 0;
  while (i < n) {
    i = i + 1;
    printf("%d squared is %d\n",i,i*i);
  }
  printf("that's all folks!\n");
}

int main(int argc, char **argv)
{
  print_n_squared_numbers(10);
  return 0;
}
```

### MATLAB version

```matlab
function print_n_squared_numbers(n)
  i = 0;
  while i < n
    i = i + 1;
    fprintf(’%d squared is %d\n’,i,i*i);
  end
  fprintf(’that’s all folks!\n’);
end

% Example usage
print_n_squared_numbers(10);
```

Both MATLAB and C are imperative and procedural.

MATLAB is interpreted, C/C++ is compiled

MATLAB is dynamically typed, C/C++ is statically typed.

Lecture 2 outline

- Control flow
- Imperative languages
- Goto (considered harmful)
- Blocks, conditionals, loops

- State
  - Variable
  - Data types
  - Static vs dynamic typing

- Compiled vs interpreted language
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files

- Practical notes
  - Clean vs obfuscated code
  - Avoid cut & paste
Statements and the state

- Program state = program counter + content of memory
- Executing a statement changes the state
  - Updates the program counter
  - Almost always modifies the content of the memory as well
- Example
  \[50 \times x \times x + y;\]  // result is not remembered, no effect
  \[z = 50 \times x \times x + y;\]  // write the result to the variable z
- If a statement does not alter the content of the memory, it has essentially no effect
- Exceptions:
  - wasting time
  - in MATLAB, displaying a value on the screen
  - other side effects

Data types

- A (data) type specifies
  - a set of possible values
    - e.g. integers in the range \(-2,147,483,648\) to \(2,147,483,647\)
  - what one can do with them
    - e.g. create, assign, sum, multiply, divide, print, convert to float, ...
- A data type representation specifies how values are actually stored in memory
  - e.g. integer is usually represented as a string of 32 bits, or four consecutive bytes, in binary notation
- This is another example of abstraction
  - You never have to think how MATLAB represents numbers to use them!
- Most programming languages support several primitive data types:
  - MATLAB: numeric arrays (characters, integer, single and double precision), logical arrays, cell arrays, ...
  - C: various integer types, character types, floating point types, arrays, strings, ...

Memory and variables

- A computer's memory is just a large array of words (strings of bits)
- Addresses
  - 00000000
  - 00000001
  - ...
  - milesToGo: 001F034C
- The meaning depends on how it is interpreted:
  - 32 bit integer: 1718773092
  - 4 characters: "fred"
  - floating point: 1.68302e+022
- Write a 32-bit integer to memory in C
  \[*(\text{int } *) 0x001F034C) = 1718773092;\]
- Structured version: using a variable
  \[\text{milesToGo} = 1718773092;\]
- To a first approximation, a variable is just a name given to a memory address
  (and a data type)

Dynamic data typing

- Consider the following MATLAB fragment
  - \% x, y, and z are stored as 64-bit float
  - x = 5;
  - y = 10;
  - z = x * y;
- Each variable stores both:
  - the address of the data in memory and
  - the type of the data
- Use the MATLAB command who to get a list of variables and their types (classes):

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>y</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>z</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>
## Dynamic data typing

Consider the following MATLAB fragment

```matlab
% x, y, and z are stored as 64-bit float
x = 5;
y = 10;
z = x * y;
```

Now variable `x` refers to a new memory block and a different data type.

In MATLAB, the data type associated to a variable can be determined only at run-time, i.e. when the program is executed. This is called **dynamic typing**.

Overhead in dynamic data typing

In dynamic data typing each a variable is associated to both the actual data record as well as metadata describing its type.

While usually this is not a problem, in some cases the overhead may be significant.

Example: MATLAB uses about 80 bytes to store the data type descriptor.

- Storing **one array of 1 million numbers** uses $80 + 8 \times 10^6$ bytes ($\sim 7.6$ MB, efficiency $\sim 100\%$)
- Storing **1 million arrays of 1 number each** uses $(80 + 8) \times 10^6$ bytes ($\sim 83$ MB, efficiency $\sim 9\%$)

## Overhead in dynamic data typing

```
x = 'Oxford U';
y = 3.14;
z = x * y;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1x8</td>
<td>16</td>
<td>char</td>
</tr>
<tr>
<td>y</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>z</td>
<td>1x8</td>
<td>64</td>
<td>double</td>
</tr>
</tbody>
</table>

Two operations are involved in calculating $z$:

- **promotion**: the string $x$ is reinterpreted as an array of $1 \times 8$ 64-bit floats
- **vector-matrix mult.**: the scalar $y$ is multiplied by this array
**Static data typing and variable declaration**

- In C variables must be **declared** before they can be used.
  - A declaration assigns **statically a data type** to a variable.

**Examples**

```c
int anInteger; /* usually 32 bits length, but implementation dependent */
unsigned int anUnsignedInteger;
char aCharacter;
double aFloat;
int32_t a32BitInteger; /* C99 and C++ */
int16_t a16BitInteger;
```

- **Statically-typed variables**
  - have a well defined type before the program is run.
  - incorporate constraints on how a variable can be used.

Static typing allows for:
- smaller run-time overhead in handling variables
- better error checking before the program is run

**Compiled vs interpreted languages**

- MATLAB is an **interpreted language**
  - a MATLAB program is executed by an **interpreter**
  - the interpreted emulates a CPU capable of understanding MATLAB instructions
  - significant overhead at run-time

- C and C++ are **compiled languages**
  - a C/C++ program must be translated by a **compiler** into an executable format before it can be executed
  - no overhead at run-time
  - the compiler can spot programming error violating assumptions expressed in the code
  - for example, in a statically typed language the compiler can check that operations on variables involve data of the correct types

**Example**

Compiling the following fragment generates an error because the multiplication of an integer and a pointer (see later) is not defined:

```c
int * aPointerToInt = 0;
int anInt = 10;
int anotherInt = anInt * aPointerToInt;
```

**MATLAB: program organisation**

- MATLAB procedures are called **functions**
  - A MATLAB function is stored in a **homonymous file with a .m extension**
  - A MATLAB function is stored in a **homonymous file with a .m extension**

```matlab
function print_ten_squared_numbers(n)
i = 0;
while i < n
    i = i + 1;
    fprintf('%d squared is %d
',i,i*i);
end
fprintf('that’s all folks!
');
end
```

**file: print_ten_squared_numbers.m**

A .m file can also contain a **script**.

- A script does not define a function. It is more similar to cutting & pasting code in to the MATLAB prompt.

**Lecture 2 outline**

- Control flow
  - Imperative languages
  - Goto (considered harmful)
  - Blocks, conditionals, loops

- State
  - Variable
  - Data types
  - Static vs dynamic typing

- **Compiled vs interpreted language**
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files

- Practical notes
  - Clean vs obfuscated code
  - Avoid cut & paste
MATLAB: program organisation

- MATLAB procedures are called functions.
- A MATLAB function is stored in a homonymous file with a `.m` extension.

```matlab
function print_ten_squared_numbers(n)
    i = 0;
    while i < n
        i = i + 1;
        fprintf('%d squared is %d
', i, i*i);
    end
end

function thats_all()
    fprintf('that’s all folks!
');
end
```

MATLAB procedures are functions. A MATLAB function is stored in a homonymous file with a `.m` extension. The function `print_ten_squared_numbers` demonstrates the use of a function.

```matlab
% demonstrates the use of a function
print_ten_squared_numbers();
```

Local functions are only visible from the file where they are defined.\(^1\)

\(^1\)Advanced techniques allow to pass references to local functions, so that they can be called from other files.

MATLAB: grouping related functions

- Put related functions into a given directory.

```
directory: drawing/
drawAnArc.m
drawAnArrow.m
drawACircle.m
```

```
directory: math/
tan.m
atan.m
sqrt.m
```

```
directory: pde/
euler.m
```

MATLAB Toolboxes are just collections of functions organised in directories.

C/C++: program organisation

- C/C++ explicitly support the notion of modules.
- A module has two parts:
  - the declaration (.h), defining the interface of the functions i.e. the function names and the types of the input and output arguments
  - the definition (.c), containing the actual implementation of the functions

```
#include "usefulstuff.h"
#include <stdio.h>

void print_n_squared_numbers(int n) {
    int i = 0;
    while (i < n) {
        i = i + 1;
        printf("%d squared is %d\n", i, i*i);
    }
    printf("that's all folks!\n");
}
```

```
int main(int argc, char** argv) {
    print_n_squared_numbers(10);
    return 42;
}
```

```
#include <stdio.h>

void get_an_awesome_number() {
    return 42;
}
```

```
#include "usefulstuff.h"

int main(int argc, char** argv) {
    print_n_squared_numbers(10);
    return 42;
}
```

```
#include <stdio.h>

int get_an_awesome_number() {
    return 42;
}
```

```
myprogram module
```

```
object file: usefulstuff.o
executable file: myprogram
```

```
myprogram declaration file: N/A
```

```
myprogram definition file: myprogram.c
```

```
object file: myprogram.o
```
C/C++: compiling a program

Run the compiler `cc`
- Each `.c` file is compiled into an object file `.o`
  - This is the binary translation of a module

Run the linker, usually also implemented in `cc`
- The `.o` files are merged to produce an executable file

```
# Declaration file: morestuff.h
#include <stdio.h>

// Module definition file: morestuff.c
int morestuff_module(int n) {
    return n * n;
}

# Object file: morestuff.o

# Executive file: myprogram

```

More on declaring, defining, and calling functions

Declaration of the function prototype
- `void print_n_squared_numbers(int n);`

Definition of the function implementation
- `void print_n_squared_numbers(int n) {
    int x;
    x = n * n;
    printf("x = %d\n", x);
    }`

Invocation of the function
- `print_n_squared_numbers(10);`

- **Declaring** a function
  - defines its **prototype** = (name of the functions, type of input/output parameters)
  - specifies the **interface**
    - the compiler needs to know only the prototype to call the function

- **Defining** a function specifies its **implementation**. The parameters are said to be **formal** as their value is not determined until the function is called.

- **Calling** a function starts executing the function body. The parameters are said to be **actual** because they are assigned a value.

---

Return value(s)

- In C functions have a single output value, assigned by the `return` statement.

```
# Definition of the function
int get_awesome_number() {
    return 42;
}

# Invocation of the function
int x;
x = get_awesome_number();
```

- In MATLAB functions have an arbitrary number of output values.

```
# In MATLAB
function [a,b,c] = get_many_numbers()
    a = 42;
b = 3.14;
c = +inf;
    return;
end

# Definition of the function
[a,b,c] = get_many_numbers();

# Invocation of the function
[x,y,z] = get_many_numbers();
```

Lecture 2 outline

- Control flow
  - Imperative languages
  - Goto (considered harmful)
  - Blocks, conditionals, loops

- State
  - Variable
  - Data types
  - Static vs dynamic typing

- Compiled vs interpreted language
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files

- Practical notes
  - Clean vs obfuscated code
  - Avoid cut & paste
Some practical notes

- The look *is* important
- Use meaningful **variable names**
- Use **comments** to supplement the meaning
- **Indent** code for each block/loop
- Avoid to cut and paste code
- Use functions to encapsulate logic that can be reused
- Cutting and pasting code leads to guaranteed disasters
  - because when you need to change the code, you need to change all the copies!
- Top-down vs bottom-up
  - **Design** top-down
  - **Code** bottom-up or top-down, or a combination

Obfuscated code (don’t)

Here is a valid C program (http://en.wikipedia.org/wiki/Obfuscation_(software))

```
char M[3], A, Z, E = 40, J[40], T[40];
main(C) {
    for(*J = A = scanf("%d", &C);
        --E;
    }
    printf("._");
    for(; (A -= Z = !Z) || (printf("\n")
        , A = 39 , C = --))
        Z || printf(M[Z] = Z[A = E = A[J - Z]] & !C
        & A == T[A])
    |
    ""];
}
```

Can you figure out what this does?

Lecture 3 outline

- **Scope**
  - Local and global variables
  - Modularisation and side effects

- **Dynamic memory and pointers**
  - Memory organisation, dynamically allocating memory in the heap
  - Pointers, dereferencing, referencing, references
  - Passing by values or reference, side-effects

- **Recursion**
  - Procedures that call themselves
  - Recursion and local variables
  - The stack and stack frames

- **Passing functions as parameters**

- **Compound data types: structures**
The scope of a variable

The **scope of a variable** is the context in which the variable can be used.

- The scope of a **local variable** is the function where the variable is defined. Usually, local variables are created when the function is entered, and destroyed when it is left.
- **Global variables** can be accessed by all functions. They are created when the program starts, and destroyed when it ends.

**MATLAB example**

```matlab
function x = myFunction(n)
    m = 10;
    x = m * n;
end

% test script
myFunction(5) % 50
m = 20;
myFunction(5) % still 50!
```

- The two variables are distinct and accessible only from the respective context.

**MATLAB global variables**

- MATLAB strongly discourages the use of global variables.
- When they are really needed, they must be declared by the `global` operator.

```matlab
function x = myFunction(n)
    global m;
    x = m * n;
end

% test script
global m;
myFunction(5) % 50
m = 20;
myFunction(5) % still 50!
```

- You can always use MATLAB `whos` command to check your variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td>global</td>
</tr>
</tbody>
</table>

**C/C++ global variables**

- A variable is **implicitly global** if declared outside of any function.
- Question: which part of the program is responsible for initialising `m`?
- A global variable defined in a module is visible only to the functions of that module.
  - To make the variable visible from other modules it must be declared in the `.h` file, exactly like functions.
  - Furthermore, the `export` keywords must be used.

```c
#include "myfunction.h"
#include <stdio.h>

int m; /* global */

int myFunction(int n) {
    return m * n;
}
```

```c
/* global declaration */
export int m;

int myFunction(int n) {
    return m * n;
}
```

```c
/* global declaration */
#include "myfunction.h"

int m;

int myFunction(int n) {
    return m * n;
}
```
Procedure as functions

- **Procedures** are often intended as **functions**:
  - Then only effect of calling a procedure is to compute and return an output value.
  - The output value depends only on the value of the input parameters.

- **Side-effects** break the function-like semantics
  - e.g. a global variable is an implicit input/output parameter

---

Side-effects

- A procedure is useful only if its **behaviour is easy to predict and understand**.
- This is particularly important in **software libraries**:
  - e.g. C/C++ `math.h` (tan, cos, ...)
  - e.g. MATLAB toolboxes
- In practice, many procedures have **side-effects** beyond the simple function-like semantics:
  - reading a file, displaying a message, generating an error, ...
  - allocating and returning a new memory block
  - operating on data in the caller scope by means of references (see later)
  - ...
- A clean interface design (and documentation) is essential to control these side-effects.

---

Lecture 3 outline

- **Scope**
  - Local and global variables
  - Modularisation and side effects
- **Dynamic memory and pointers**
  - Memory organisation, dynamically allocating memory in the heap
  - Pointers, dereferencing, referencing, references
  - Passing by values or reference, side-effects
- **Recursion**
  - Procedures that call themselves
  - Recursion and local variables
  - The stack and stack frames
- **Passing functions as parameters**
- **Compound data types: structures**

---

Memory organisation

- A structured program organises the memory into four areas:
  1. The **code** area stores the program instructions.
    - The OS prevents the program from changing it.
  2. The **data** (or **heap**) area contains **dynamically** allocated records.
    - Implicit in MATLAB, using `malloc()` in C.
    - It grows towards the bottom as more memory is allocated.
  3. The **stack** area is used to handle recursive procedure calls and **local variables**.
  4. The **free** area is memory not yet assigned to a particular purpose.
Dynamic memory allocation / MATLAB

- In MATLAB dynamic memory allocation is **implicit**.

% allocate 80,000 bytes to store an array of 10,000 double
x = zeros(100,100);

Dynamic memory allocation / C

- In C/C++ dynamic memory allocation is **explicit**.

In C a new memory block is obtained by calling the `malloc()` function. Allocated memory must be disposed by calling `free`; otherwise the memory is **leaked**.

The output of `malloc` is the **address** of the allocated memory block.

- An **address** is stored into a variable of type **pointer to T**.

```c
/* declare a pointer x to a double */
double *x ;

/* allocate a double (eight bytes) and store the address in x */
x = malloc(8) ;

/* better: use sizeof to get the required size */
x = malloc(sizeof(double)) ;

/* write to the memory pointed by x */
*x = 3.14 ;

/* free the memory once done */
free(x) ;
```

Pointers and dereferencing

- A **pointer to T** is a variable containing the address to a record of type T. Its type is denoted T *

```c
/* Declare and assign a pointer to double */
double *x ;
x = malloc(sizeof(double)) ;

/* Dereference x to access the pointed memory */
*x = 3.14 ;

/* This changes the pointer, not the pointed data. */
x = 42 ;

/* This crashes the program because x does not contain the address of a valid memory block anymore */
free(x);
```

The operator * is called **dereferencing**. It allows accessing the value pointed by the pointer.

Null pointers

- By convention, the memory address 0 (0x0000000) is reserved.

- A **null pointer** is a pointer with value 0 (denoted by NULL).

Null pointers are commonly used to represents particular states. For example:

- `malloc()` returns NULL if the requested memory block cannot be allocated because the memory is exhausted (an error condition).
- In a linked list a NULL pointer may be used to denote the end of the list (see Lecture 4).

Note that writing to a null pointer (or as a matter of fact to any address not corresponding to a properly allocated memory block) crashes the program (or worse!). For example:

```c
int *myPointer = NULL ;
*myPointer = 42 ; /* crash */
```
Pointers

Pointers can be copied:

```c
unsigned int * x = malloc(sizeof(unsigned int)) ;
unsigned int * y = x ;
```

Pointers can also point to a local variable.

```c
unsigned int a = 42 ;
unsigned int * x = &a ;
```

The operator & is called referencing. It returns the address of a variable.

Now the same data record can be accessed by using the variable or the pointer:

```c
/* these tree instructions have the same effect */
a = 56 ;
*x = 56 ;
*(&a) = 56 ;
```

Pointers as input arguments

By using pointers, a procedure can access data that belongs to the caller.

**Example.** A procedure that swaps the value of two variables:

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

/* example usage */
```c
int x = 10 ;
int y = 20 ;
swap(x,y) ;
/* x = 10, y = 20 */
```

The function has no effect because calling it a and b are copies of x and y. x and y remain unaffected.

By passing pointers, the function can access the variables x and y in the caller and can swap them.

Pass by value vs reference

C++ (but not C) can pass parameters by reference instead of value

Think of references as implicit pointers

```c
void swap(int *a, int *b) {
    int temp = *a ;
    *a = *b ;
    *b = temp ;
}
```

/* example usage */
```c
int x = 10 ;
int y = 20 ;
swap(&x,&y) ;
/* x = 20, y = 10 */
```

Using pointers (C or C++).

Using references (C++ only)

Pointers and references: why?

**Uses.** Pointers/references are powerful:

- they allow a procedure to access data of the caller (e.g. `swap()`)
- they allow to pass data to a procedure avoiding copying (faster)
  
E.g. think of passing a 1000-dimensional vector
- they allow to construct interlinked data structures
  
E.g. lists, trees, containers in general

**Caveats.** Pointers/references allow side effects:

- they make a procedure behaviour harder to understand
- they make programming errors harder to find
  
An error inside a procedure may affect the caller in unpredictable ways

In MATLAB

- there are (almost) no references nor pointers
- it is only possible to assign or copy the value of variables
  
Under the hood, however, all data are passed by reference. The pass-by-value semantic is ensured by sharing copies as much as possible.
Lecture 3 outline

- Scope
  - Local and global variables
  - Modularisation and side effects
- Dynamic memory and pointers
  - Memory organisation, dynamically allocating memory in the heap
  - Pointers, dereferencing, referencing, references
  - Passing by values or reference, side-effects
- Recursion
  - Procedures that call themselves
  - Recursion and local variables
  - The stack and stack frames
- Passing functions as parameters
- Compound data types: structures

Recursion

Recursion is one of the most powerful ideas in computer programming

- Algorithmic techniques such as divide & conquer map directly to recursion
- Many data structures are recursive (e.g. trees)
- Procedures can also be called recursively

Example: computing the factorial of \( n \)

Mathematical definition

\[
\text{fact}(n) = \begin{cases} 
1, & n = 1, \\
n \text{fact}(n - 1), & n > 1.
\end{cases}
\]

Corresponding C function

```c
int fact(int n) {
  int m;
  if (n == 1) return 1;
  m = n * fact(n - 1);
  return m;
}
```

Recursion and local variables

The local variables constitute the “private” state of the function. Each execution has its own state.

In this manner, recursive calls do not interfere with each other. And memory for local variable is allocated only when needed.

Recursion and the stack

The **stack** is a memory area used to handle (recursive) calls to procedures.

- A **stack frame** is pushed on top of the stack when a procedure is entered, and popped when it is left. It contains:
  - a return location (PC) to enable resuming the caller upon completion of the procedure
  - the input and output parameters
  - the local variables
Example of recursive calls

```c
int fact(int n)
{
    int m;
    if (n == 1) return 1;
    return m = n * fact(n - 1);
}
/* example usage */
x = fact(5);
lab: printf("fact(5) is %d", x);
```

Recursion: a more advanced example

- **Multiple recursion.** A procedure can call itself multiple times.

This example paints the image region of colour `old_colour` containing the pixel `(x, y)` with `new_colour`.

```c
const int SIZE = 256;
int im[SIZE][SIZE];

void fill(int x, int y, int old_colour, int new_colour)
{
    if (x >= 0 && x < SIZE && y >= 0 && y < SIZE) {
        if (im[y][x] == old_colour) {
            im[y][x] = new_colour;
            fill(x-1,y,old_colour,new_colour);
            fill(x+1,y,old_colour,new_colour);
            fill(x,y-1,old_colour,new_colour);
            fill(x,y+1,old_colour,new_colour);
        }
    }
    return;
}
```

Lecture 3 outline

- **Scope**
  - Local and global variables
  - Modularisation and side effects
- **Dynamic memory and pointers**
  - Memory organisation, dynamically allocating memory in the heap
  - Pointers, dereferencing, referencing, references
  - Passing by values or reference, side-effects
- **Recursion**
  - Procedures that call themselves
  - Recursion and local variables
  - The stack and stack frames
- **Passing functions as parameters**
- **Compound data types: structures**
A function can be passed as a parameter of another function.
In this manner, a behavior can be communicated.

**Example.** Consider implementing an algorithm for the numerical solution of a first order ODE:

**Euler method:**

\[ \dot{y}(t) = -y(t), \quad t \geq 0. \]

Choose a step size \( h \) and an initial condition \( y_0 \) and then let:

\[ y(0) = y_0, \]
\[ y(h n) = -h y(h(n-1)) + y(h(n-1)), \quad n = 1, 2, \ldots, N - 1 \]

**MATLAB implementation:**

```matlab
function y = solve(y0, h, N)
    y = zeros(1, N);
    y(1) = y0;
    for n = 2:N
        ydot = -y(n-1);
        y(n) = y(n-1) + h * ydot;
    end
end
```

More in general, there is one ODE problem for each function \( F \):

\[ \dot{y}(t) = F(y(t)), \quad t \geq 0. \]

The Euler solver needs to be modified:

\[ y(0) = y_0, \]
\[ y(h n) = hF(y(h(n-1))) + y(h(n-1)), \quad n = 1, 2, \ldots, N - 1 \]

To avoid writing a new program for each \( F \) pass the latter as a parameter:

```matlab
function y = solve(F, y0, h, N)
    y = zeros(1, N);
    y(1) = y0;
    for n = 2:N
        ydot = F(y(n-1));
        y(n) = y(n-1) + h * ydot;
    end
end
```

The @ operator returns a handle to a function. A handle is similar to a pointer.

More in general, there is one ODE problem for each function \( F \):

\[ \dot{y}(t) = F(y(t)), \quad t \geq 0. \]

The Euler solver needs to be modified:

\[ y(0) = y_0, \]
\[ y(h n) = hF(y(h(n-1))) + y(h(n-1)), \quad n = 1, 2, \ldots, N - 1 \]

In C one uses a pointer to a function:

```c
double myF(double y) {
    return -y;
}
```

This declares a parameter `func`.

The type of `func` is "pointer to a function that takes a double as input and returns a double as output".

```c
int main() {
    int n;
    double y[200], h = 0.05;
    y[0] = 1.0;
    for (n = 1; n < 200; n++) {
        y[n] = solve(myF, y[n-1], h);
        printf("Y[%d] = %f\n", n, y[n]);
    }
}
```

In C one uses a pointer to a function:

```c
double myF(double y) {
    return -y;
}
```

This declares a parameter `func`.

The type of `func` is "pointer to a function that takes a double as input and returns a double as output".

```c
int main() {
    int n;
    double y[200], h = 0.05;
    y[0] = 1.0;
    for (n = 1; n < 200; n++) {
        y[n] = solve(myF, y[n-1], h);
        printf("Y[%d] = %f\n", n, y[n]);
    }
}
```
Custom and structured data types

- All languages support natively a number of **primitive types**
  - C/C++: `char`, `int`, `float`, `double`, ...
  - MATLAB: arrays of `char`, `int16`, `int32`, `single`, `double`, ...

- Most languages support defining novel data types. Often these are **compound types** combining primitive types.
  - C: array and structures (**struct**)
  - C++: array, structures (**struct**), and classes (**class**)
  - MATLAB: `cell arrays`, structures, and classes (**class**)

- **Structures** can be used to group related information together into a single data record.

- **Classes** add a behavior to structures in term of a custom set of operations that can be applied to the data (see the next lecture series).

C/C++ structures: **struct**

- In C, each new type of structure must be declared before a corresponding variable can be defined and assigned.

  **Declaration**
  ```c
  struct Complex_ {
    double re ;
    double im ;
  } ;
  ```

  **Definition and assignment**
  ```c
  struct Complex_ c ;
  c.re = 1.0 ;
  c.im = 0.0 ;
  ```

- **typedef** can be used as a shorthand.

  ```c
  typedef struct Complex_ Complex ;
  ```

MATLAB structures

- A **structure** is a compound data type which comprises related data into a single entity.

- In MATLAB, a structure is defined by assigning a variable using the `.` operator.

- **Example:** create and assigns a new variable `person`:

  ```matlab
  person.name = 'Isaac' ;
  person.surname = 'Asimov' ;
  person.age = 66 ;
  person.occupation = 'writer' ;
  ```

- The variable `person` is a structure with the following fields: name, surname, age and occupation.

- Structures can contain other structures, recursively:

  ```matlab
  person.address.city = 'New York' ;
  person.address.zipCode = '12345' ;
  ```

Example: VTOL state

- VTOL state:
  - height, velocity, mass (numbers)
  - landed (bool)

- Use a single structure to store all numbers
  - data encapsulation
  - abstraction

- **Example**

  ```matlab
  typedef struct {
    double position ;
    double velocity ;
    double mass ;
    bool landed ;
  } VTOLState ;
  ```
Using structures in C/C++

- **Creating a structure**
  - % As a local variable
    ```
    VTOLState state;
    state.position = 10;
    state.velocity = 5;
    state.mass = 1000;
    state.landed = false;
    ```
  - % Dynamically
    ```
    VTOLState *statePtr;
    statePtr = malloc(sizeof(VTOLState));
    statePtr->position = 10;
    statePtr->velocity = 5;
    statePtr->mass = 1000;
    statePtr->landed = false;
    ```

  **Note**: `x->` combines dereferencing and structure access. It is the same as `(*x)`.

- **Passing a structure to a function**
  - % By value
    ```
    double getThrust(VTOLState state);
    double t = getThrust(state);
    ```
  - % By a pointer
    ```
    double getThrust(VTOLState *statePtr);
    double t = getThrust(statePtr);
    ```
  - % By a pointer and by value
    ```
    double t = getThrust(&state);
    ```

### B16 Software Engineering
Structured Programming
Lecture 4: Programs = Algorithms + Data structures

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

### Lecture 4 outline

- **Arrays**
  - In MATLAB and C
  - Pointer arithmetic
- **Sorting**
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- **Divide & conquer**
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
- **Linked list**
  - Search, insertion, deletion
- **Trees**
  - Binary search trees
- **Graphs**
  - Minimum spanning tree

- **Divide & conquer**
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
Arrays

- An array is a data structure containing a numbered (indexed) collection of items of a single data type.
- In MATLAB arrays are primitive types.
- In C, arrays are compound types. Furthermore, C arrays are much more limited than MATLAB’s.

```c
/* Define, initialise, and access an array of three integers in C */
int a[3] = {10, 20, 30};
int sum = a[0] + a[1] + a[2];

/* Arrays of custom data types are supported too */
VTOLState states[100];
for (t = 1; t < 100; t++) {
    states[t].position = states[t-1].position + states[t-1].velocity + 0.5*g;
    states[t].velocity = states[t-1].velocity + g - getThrust(states[t-1], burnRate) / states[t-1].mass;
    states[t].mass = states[t-1].mass - burnRate * escapeVelocity;
}
```

Static vs dynamic arrays in C

- This C statement defines an array `a` of five integers
  ```c
  int A[5];
  ```
  The size is static because it is specified before the program is compiled. What if the size needs to be adjusted at run-time?
  - The solution is to allocate dynamically the required memory:
    ```c
    int arraySize = 5;
    int *A = malloc(sizeof(int) * arraySize);
    ```
  - Note that `a` is declared as a pointer to an int, not as an array. However, the array access operator [] can still be used. E.g. `a[1] = 2`
  - Pointer math: `a[n]` is the same as `(*a + n)`
    - E.g. `a[0]` is the same as dereferencing the pointer (`*a`)
    - Under the hood, the address stored by `a` is incremented by `n * sizeof(int)` to account for the size of the pointed elements

Array representation in C

- In C an array is represented as a sequence of records at consecutive memory addresses.
  ```c
  /* array of five doubles */
  double A[5];
  /* get a pointer to the third element */
  double * pt = &A[2];
  ```

- Two (and more) dimensional arrays are simply arrays of arrays.
  ```c
  /* A 2x5 array */
  double A[2][5];
  ```

Lecture 4 outline

- Arrays
  - In MATLAB and C
  - Pointer arithmetic
- Sorting
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- Divide & conquer
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
- Linked list
  - Search, insertion, deletion
- Trees
  - Binary search trees
- Graphs
  - Minimum spanning tree
Sorting

Problem: sort an array of numbers in non-decreasing order.

There are many algorithms to do this: bubble sort, merge sort, quick sort, ...

We will consider three aspects:

- Describing the algorithm formally.
- Proving its correctness.
- Evaluating its efficiency.

We start from the *insertion sort algorithm*

Input: an array of numbers.
Output: the numbers sorted in non-decreasing order.
Algorithm: initially the sorted output array is empty. At each step, remove an element from the input array and insert it into the output array at the right place.

See http://www.sorting-algorithms.com/ for illustrations

Insertion

The insertion procedure extends a sorted array by inserting a new element into it:

% Input: array $A$ of size $\geq n$ such that $A[1] \leq \cdots \leq A[n-1]$  
% Output: permuted array such that $A[1] \leq \cdots \leq A[n-1] \leq A[n]$ 

function $A = \text{insert}\left(A, n\right)$
  
  $i = n$
  % the invariant is true here  
  while $i > 1$ and $A[i-1] > A[i]$
    swap($A[i-1], A[i]$)  
    $i = i - 1$
  % the invariant is true here  
end  
end

A *loop invariant* is a property that is valid before each loop execution starts. It is usually proved by induction. For `insert()` the loop invariant is:

- $A[1] \leq A[2] \leq \cdots \leq A[i-1] \leq A[i+1] \leq \cdots \leq A[n]$ and
- $A[i] \leq A[i+1]$

### Insertion: example

$n = 9$; \(\leq\) means \(\leq\)

% Input: an array $A$ with $n$ elements  

\[
\begin{align*}
&\text{function } A = \text{insertionSort}\left(A, n\right) \\
&\quad i = 1 \\
&\quad % the invariant is true here (A)  
&\quad \text{while } i < n \\
&\quad \quad i = i + 1 \\
&\quad \quad A = \text{insert}\left(A, i\right) \\
&\quad \quad % the invariant is true here (B)  
&\quad \text{end}  
&\quad \text{end} \\
\end{align*}
\]

**Loop invariant:** the first $i$ elements are sorted: $A[1] \leq A[2] \leq \cdots \leq A[i]$

**Proof** by induction

- **base case** ($A[1]$) is sorted
- **inductive step** ($A[i] \geq 1$): at iteration $i$ the `insert()` procedure sorts $A[1], \ldots, A[i]$ provided that $A[1], \ldots, A[i-1]$ are sorted. The latter is given by the invariant at iteration $i - 1$. 

Insertion sort
### Algorithmic complexity

- **The time complexity** of an algorithm is the maximum number of elementary operations $f(n)$ required to process an input of size $n$. Its **space complexity** is the maximum amount of memory required.

- It often suffices to determine the **order of the complexity** $g(n)$: linear $n$, squared $n^2$, polynomial $n^k$, logarithmic $\log(n)$, exponential $\exp(n)$, ... We say that the order of $f(n)$ is $g(n)$, and we write $f(n) = O(g(n))$, if:

$$\exists a, n_0 : \forall n \geq n_0 : f(n) \leq ag(n)$$

- **Example: insertion sort**
  - The size of the input is the number $n$ of elements to sort.
  - The space complexity is $O(n)$ as the algorithm stores only the elements and a constant number of local variables.
  - The time complexity of `insert()` is $O(m)$ as the while loop is executed at most $m$ times. The time complexity of `insertionSort()` is $O(n^2)$ because

$$\sum_{m=1}^{n} m = \frac{(n + 1)n}{2} = O(n^2)$$

### Divide and conquer

- **Divide and conquer** is a recursive strategy applicable to the solution of a wide variety of problems.

- The idea is to split each problem instance into two or more smaller parts, solve those, and recombine the results.

```matlab
% Divide and conquer pseudocode
solution = solve(problem)
If problem is easy, compute solution
Else
    Subdivide problem into subproblem1, subproblem2, ...
sol1 = solve(subproblem1), sol2 = solve(subproblem2), ...
    Get solution by combining sol1, sol2, ...
```

- Note the **recursive call**. Divide and conquer is naturally implemented as a recursive procedure.

- Some of the best known and most famous (and useful) algorithms are of this form, notably quicksort and the Fast Fourier Transform (FFT).
**Complexity of divide and conquer**

- **Assume** that the cost of splitting and merging a subproblem of size $m$ is $O(m)$ (linear) and that the cost of solving a subproblem of size $m = 1$ is $O(1)$.

Given a problem of size $n$, at each level $O(n)$ work is done in order to split & merge or solve subproblems. Since there are $\log_2(n)$ levels the total cost is $O(n \log_2 n)$.

**Merge sort**

The **merge sort** algorithm sorts an array $A$ by divide and conquer:

- **Split**: divide $A$ into two halves $A_1$ and $A_2$.
- **Merge**: iteratively remove from the beginning of the sorted $A_1$ and $A_2$ the smallest element and append it to $A$.
- **Base case**: if $A$ has one element only it is sorted.

**Function**

```plaintext
function A = mergeSort(A)
    n = length(A)
    if n == 1 then
        return A
    end
    k = floor(n / 2)
    A_1 = A(1:k)
    A_2 = A(k+1:end)
    A_1 = mergeSort(A_1)
    A_2 = mergeSort(A_2)
    return merge(A_1, A_2)
end

function A = merge(A_1, A_2)
    i_1 = 1, i_2 = 1
    m_1 = length(A_1), m_2 = length(A_2)
    while i_1 <= m_1 and i_2 <= m_2
        if A_1[i_1] <= A_2[i_2]
            A[i_1+i_2-1] = A_1[i_1], i_1 = i_1 + 1
        else
            A[i_1+i_2-1] = A_2[i_2], i_2 = i_2 + 1
        end
    end
    A[i_1+m_1-1] = A_1[m_1], i_1 = i_1 + 1
    A[i_2+m_2-1] = A_2[m_2], i_2 = i_2 + 1
end
```

**Insertion vs Merge Sort**

- The two sorting algorithms have **different complexities**:
  - insertion: $O(n^2)$
  - merge: $O(n \log(n))$

- Plotting time vs size in loglog coordinates should give a line of slope:
  - 2 for insertion sort
  - ~ 1 for merge sort

- This is verified experimentally in the figure.
Root finding

- **Problem**: find a root of a non-linear scalar function \( f(x) \), i.e. a value of \( x \) such that \( f(x) = 0 \).

- **Assumption**: \( f(x) \) is a continuous function defined in the interval \([a, b]\); furthermore, \( f(a)f(b) < 0 \).

- The **bisection algorithm** is a divide and conquer strategy to solve this problem.

```plaintext
function bisect(f, a, b)
    m = (a + b) / 2
    if f(m) close to zero
        return mu
    if f(m) * f(a) > 0
        return bisect(f, m, b)
    else
        return bisect(f, a, m)
end
```

Lecture 4 outline

- Arrays
  - In MATLAB and C
  - Pointer arithmetic

- Sorting
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity

- Divide & conquer
  - Solving problems recursively
  - Merge sort
  - Bisection root finding

Linked lists

- A limitation of arrays is that inserting an element into an arbitrary position is \( O(n) \).

- This is because existing elements must be shifted (moved in memory) in order to make space for the a one.

- **Linked lists** solve this problem by using pointers:

```plaintext
/* List element datatype */
typedef struct
    ListElement {
        struct ListElement_*next;
        double value;
    } ListElement;

/* List datatype */
typedef ListElement List;
```

- **Example usage**

```plaintext
/* Create an empty list */
List list;
list->next = NULL;

/* Insert at the beginning of the list */
/* Insert an element into alist, use pointers to create a “bypass” at cost \( O(1) \). */
void insert(ListElement*prev, ListElement*element) {
    element->next = prev->next;
    prev->next = element;
}
```

Inserting an element into a linked list

- To **insert** an element into a list, use pointers to create a “bypass” at cost \( O(1) \).

```plaintext
/* Insert an element in a list */
void insert(ListElement*prev, ListElement*element) {
    element->next = prev->next;
    prev->next = element;
}
```

- **Example usage**

```plaintext
/* Create an empty list */
List list;
list->next = NULL;

/* Insert at the beginning of the list */
insert(&list, element);

/* Insert after element */
insert(element, element2);
```

A limitation of arrays is that inserting an element into an arbitrary position is \( O(n) \). This is because existing elements must be shifted (moved in memory) in order to make space for the a one.

**Linked lists** solve this problem by using pointers:

```plaintext
/* List element datatype */
typedef struct
    ListElement {
        struct ListElement_*next;
        double value;
    } ListElement;

/* List datatype */
typedef ListElement List;
```

- **Example usage**

```plaintext
/* Create an empty list */
List list;
list->next = NULL;

/* Insert at the beginning of the list */
/* Insert an element into a list, use pointers to create a “bypass” at cost \( O(1) \). */
void insert(ListElement*prev, ListElement*element) {
    element->next = prev->next;
    prev->next = element;
}
```

Inserting an element into a linked list

- To **insert** an element into a list, use pointers to create a “bypass” at cost \( O(1) \).

```plaintext
/* Insert an element in a list */
void insert(ListElement*prev, ListElement*element) {
    element->next = prev->next;
    prev->next = element;
}
```

- **Example usage**

```plaintext
/* Create an empty list */
List list;
list->next = NULL;

/* Insert at the beginning of the list */
insert(&list, element);

/* Insert after element */
insert(element, element2);
```
Removing an element from a linked list

To insert an element into a list, use pointers to create a "bypass" at cost O(1).

Example usage

```c
/* Remove an element */
ListElement *remove(ListElement *prev) {
    ListElement removed = prev->next;
    if (removed != NULL) {
        prev->next = removed->next;
    }
    return removed;
}
```

Lecture 4 outline

- Arrays
  - In MATLAB and C
  - Pointer arithmetic
- Trees
  - Binary search trees
- Sorting
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- Divide & conquer
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
- Linked list
  - Search, insertion, deletion
- Graphs
  - Minimum spanning tree

Binary tree

- Each node in a binary tree has one left child and one right child.
- There are no backward links (no cycles).

Example C data type

Similar to a linked list:

```c
typedef struct Node_ {
    struct Node_ *left;
    struct Node_ *right;
    double value;
} Node;
```

Depth first traversal

This algorithm visits recursively all the nodes in a tree.

```c
function visit(node) {
    if node == NULL then return
    visit(node.left)
    visit(node.right)
    print(node.value)
    end
```
Binary search algorithm

- **Problem**: find a node with value \( x \) in a binary search tree.
- The **binary search algorithm** searches for \( x \) recursively, using the **binary search tree property** to descend only into one branch every time.

```java
function node = binarySearch(node, x)
if node == NULL return NULL
if node.value == x return node
if x > node.value return binarySearch(node.right, x)
else return binarySearch(node.left, x)
end
end
```

- The **cost** is \( O(h) \) where \( h \) is the **depth** of the binary tree.
- Typically \( h = O(\log n) \), where \( n \) is the number of nodes in the tree. Hence the **search cost** is \( O(\log n) \), sub-linear.
- Compare this with the \( O(n) \) cost of searching in an array or a linked list.

Balanced binary trees

- A binary tree of height \( h \) has at most \( n = 2^h \) leaves.
- Proof:
  - Let \( n(h) \) the maximum number of leaves of a binary tree of height \( h \)
  - Split it at the root in two subtrees of height \( h' \leq h - 1 \)
  - Then at best \( n(h) = 2 n(h - 1) \)
- An "optimal" binary tree is balanced

Decision tree

- Many algorithms can be described as decision trees: at every step one changes the state based on a binary test applied to the input.
- For example, the following algorithm decides whether a person is a father, a mother, or neither.

```
Genred = male?

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has children?</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Father</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has children?</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Mother</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sorting algorithms as decision trees

- An algorithm that sorts an array \( A \) based on pairwise comparisons can be thought of as a decision tree.

```

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

```
Sorting algorithms as decision trees

For a sorting algorithm:
- Leaves = possible permutations of the array
- Path from root to leaf = steps in a run of the algorithm
- Tree height = maximum number of steps required
- For an array of length $n$, there are $n!$ possible permutations
- The tree height is at least $\log_2(n!)$ (achieved by a balanced tree)
- Hence the best possible algorithm requires at least $t = \log_2(n!)$ steps in the worst case

$t \geq \log_2 n! = \log_2 n \cdot (n-1) \cdot \ldots \cdot \frac{n}{2} \cdot \ldots \cdot 2 \cdot 1 \geq \frac{n}{2} \log \frac{n}{2}$

We can never do better than $O(n \log n)$

Hence merge sort is optimal*!

*in the asymptotic worst case sense

Lecture 4 outline

Arrays
- In MATLAB and C
- Pointer arithmetic
Sorting
- The sorting problem
- Insertion sort
- Algorithmic complexity
Divide & conquer
- Solving problems recursively
- Merge sort
- Bisection root finding
Linked list
- Search, insertion, deletion
Trees
- Binary search trees
Graphs
- Minimum spanning tree

An (directed) graph is a set of vertices $V$ and edges $E \subseteq V \times V$ connecting the edges. An undirected graph is a graph such that for each edge $(u,v)$ there is an opposite edge $(v,u)$.

% MATLAB representation
edges = [1 2 2 3 4 5 5 6 2 3 6 4 5 6 8 7 2 3 6 4 5 6 8 7 1 2 2 3 4 5 5 6] ;

An alternative representation of a graph is the adjacency matrix $A$. $A$ is an $n \times n$ matrix such that $A(u,v) = 1$ if, and only if, $(u,v) \in E$.

$A = [0 1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1] ;$
Minimum spanning tree

Consider a weighed undirected graph with non-negative weights on the edges:

A spanning tree is a subset of the edges forming a tree including all the nodes.

A minimum spanning tree (MST) is a spanning tree such that the sum of the edge weights is minimal.

A famous algorithm to compute the MST is explored in the tutorial sheet.

Concept summary

- **Software engineering processes**
  - Specification, design & implementation, validation, evolution
  - Waterfall and extreme programming

- **Software engineering tools**
  - Abstraction and modularity
  - Procedures
  - Variables, data type, scoping
  - Dynamic memory allocation
  - Pointers, references
  - Recursion, stack, stack frames
  - Pointers to functions
  - Compound data types

- **Data structures and algorithms**
  - Complexity and correctness
  - Arrays, lists, trees, graphs
  - Sorting, searching, numerical problems

- **Exam questions?** See tutorial sheet to follow.