B16: Software Engineering

Structured Programming

Lecture 1: Software engineering

Dr Andrea Vedaldi

4 lectures, Hilary Term

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

---

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

---

1. **Structured Programming**
   - Algorithms, data structures, complexity, correctness
   - Writing programs using structure programming languages

2. **Object Oriented Programming**
   - Object-oriented programming
   - Writing programs using object-oriented languages

3. **Computer Communications and Networking**
   - How devices and computers communicate, the Internet
   - Writing programs that can exchange information on a network

4. **Operating Systems**
   - Hardware abstraction and virtualisation
   - Writing programs that access hardware features transparently and concurrently

---

**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

```c
#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);
    memset(enc, 0,
           sizeof(TYPE) * 2 * 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,

```
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/

Software engineering

Aim
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**
  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

I_C = \beta I_B,
I_E = I_C + I_B,
V_{EB} = 60 \text{ mV}

operational amplifier

V_+ = V_-, \quad I_+ = 0 \text{ A}, \quad 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

**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

**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**
- Check that the program conforms to requirements

**Validation**
- Check that the program solves the problem (includes checking the requirements)

### Evolution

- Fix a bug
- Add a functionality
- Improve efficiency

### 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

---

**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

**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**
- Check that the program conforms to requirements

**Validation**
- Check that the program solves the problem (includes checking the requirements)

**Evolution**
- Fix a bug
- Add a functionality
- Improve efficiency

**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

Fundamental activities
1. Specification
2. Design & implementation
3. Validation
4. Evolution

Incremental models

- Develop the software by increments, exposing them to the user and eliciting changes, and go back to incorporate them.
- **Extreme programming**
  - Tight requirement-implementation-test loops
  - Applied at different temporal scales:
    - from seconds (implement a function)
    - to months (establish high-level goals for the next software release)
- Design as you go
- Good for small high-risk/speculative projects, bad for nuclear reactors

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)**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary 32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
<td>000001010101</td>
</tr>
<tr>
<td>1</td>
<td>48 89 e5</td>
<td>0100100010100101</td>
</tr>
<tr>
<td>4</td>
<td>c7 45 fc</td>
<td>0100001101010110 11111011</td>
</tr>
<tr>
<td>b</td>
<td>c7 45 f8</td>
<td>0100001101010110 11111010 00000000</td>
</tr>
<tr>
<td>12</td>
<td>8b 45 fc</td>
<td>0100001010101011 11111011</td>
</tr>
<tr>
<td>15</td>
<td>03 45 f8</td>
<td>0000001101010110 11111010 00000000</td>
</tr>
<tr>
<td>18</td>
<td>89 45 f4</td>
<td>0100100101010110 11111010 00000000</td>
</tr>
<tr>
<td>1b</td>
<td>8b 45 f4</td>
<td>0100001010101011 11111010 00000000</td>
</tr>
<tr>
<td>1e</td>
<td>5d</td>
<td>0000001101000110</td>
</tr>
<tr>
<td>1f</td>
<td>c3</td>
<td>0000001101100001</td>
</tr>
</tbody>
</table>

**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)

**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.

```
[\[a-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.

```
bool f(char const * str)
{
  bool match = true ;
  while (*str) {
    match &= ('a' <= *str && *str <= 'Z') ;
    str ++ ;
  }
  return match ;
}
```

---

This is a C implementation of the same program.
It specifies how to solve the problem in terms of elementary steps.
An overview of fundamental abstractions

Structure in imperative languages

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 ;
}
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 | PC 12 |
| 11 | wake up           |
| 12 | have breakfast   |

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)

- 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: \[
\begin{align*}
i & ← i + 1 \\
\text{print } i, \text{ "squared is ", } i \times i \\
\text{if } i ≥ 10 \text{ then goto end} \\
\text{goto more}
\end{align*}
\]

end: 

print “that’s all folks!”


Spaghetti code

Structured control flow

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

- Blocks (sequences of executable statements)
- Conditionals (execute a block if a condition is true)
- Loops (keep executing a block until a condition remains true)

Spaghetti monster

| 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!” |

Structured program

| 10 | i ← 0 |
| 19 | while (i < 10) |
| 20 | \{ |
| 21 | i ← i + 1 |
| 22 | print i, “squared is ", i * i |
| 23 | if i ≥ 10 then goto end |
| 24 | \} |
| 25 | goto more |
| 26 | end: print “that’s all folks!” |
Structured control flow

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

```plaintext
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!"
```

Structured program

```plaintext
{i
  i ← 0
  while (i < 10)
    { i ← i + 1
      print i, " squared is ", i * i
    }
  print "that's all folks!"
}
```

A way to create a software module or component is to wrap a sequence of statements in a `procedure`.

```plaintext
procedure print_n_squared_numbers(n)
{{
  i ← 0
  while (i < n)
    { i ← i + 1
      print i, " squared is ", i * i
    }
  print "that's all folks!"
}
```

A procedure implements a reusable functionality (behaviour) hiding the internal implementation details.

Examples

- `y = tan(x)` // compute the tangent of a number
- `printf("a string")` // display a string on the screen
- `window = createWindow()` // create a new window on the display
- `destroyWindow(window)` // destroy it

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 ",i,i*i);
 }
 printf("that's all folks!\n");
} /* the program entry point 

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 
   fprintf(’%d’,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, ...

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

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

% now reassign x and z
x = 'Oxford U';
z = x * y;

What is z?

What is the value of z? 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.

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;

% now reassign x and z
x = 'Oxford U';
z = x * y;

What is z?

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 $80 + 8 \times 1 \text{e6}$ bytes (~ 7.6 MB, efficiency ~ 100%)

Storing 1 million arrays of 1 number each uses $(80 + 8) \times 1 \text{e6}$ bytes (~ 83 MB, efficiency ~ 9%)

Overhead in dynamic data typing

<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 1x8 64-bit floats
- **vector-matrix mult.**: the scalar y is multiplied by this array
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

  **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;
```

```
error-pointer-by-integer.c:7: error: invalid operands to binary * (have ‘int *’ and ‘int’)
```

**MATLAB: program organisation**

A MATLAB procedure is called a **function**

- A MATLAB function is stored in a homonymous file with a `.m` extension

- A MATLAB function is demonstrated in a file `print_ten_squared_numbers.m`

```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
```

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

A script `my_script.m` can also contain a **script**.

A script does not define a function. It is more similar to cutting & pasting code into the MATLAB prompt.
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
    thats_all();
end

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

- `file: print_ten_squared_numbers.m`
- `% demonstrates the use of a function print_ten_squared_numbers()

An .m file defines a function that can be accessed by functions and scripts in other files.

- A .m file can contain also any number of local functions.
- Local functions are only visible from the file where they are defined.¹

- 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

```c
#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 get_an_awesome_number() {
    return 42 ;
}
```

- `file: usefulstuff.h`
- `file: usefulstuff.c`
- `file: myprogram.c`

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

```
usefulstuff module

declaration file:
usefulstuff.h
definition file:
usefulstuff.c
object file:
usefulstuff.o

myprogram module

declaration file: N/A
definition file:
myprogram.c
object file:
myprogram.o

linked

executable file:
myprogram
```

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

---

**More on declaring, defining, and calling functions**

**Declaration of the function prototype**

```c
void print_n_squared_numbers(int n);
```

**Definition of the function implementation**

```c
void print_n_squared_numbers(int n)
{
    /* do something */
}
```

**Invocation of the function**

```c
print_n_squared_numbers(10);
```

---

**Declaration of the function prototype**

```c
int get_awesome_number();
```

**Definition of the function implementation**

```c
int get_awesome_number()
{
    return 42;
}
```

**Invocation of the function**

```c
int x;
x = get_awesome_number();
/* x is now 42 */
```

---

**Declaration of the function prototype**

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

**Definition of the function implementation**

```c
function [a, b, c] = get_many_numbers()

    a = 42;
    b = 3.14;
    c = +inf;

    [x, y, z] = get_many_numbers();

end
```

**Invocation of the function**

```c
[x, y, z] = get_many_numbers();
% get eigenvectors and eigenvalues
[V, D] = eig(A);
```

---

**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)](http://en.wikipedia.org/wiki/Obfuscation_(software))):

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

---

**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
```matlab
global m;
m = 10;
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.
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

![Diagram of a function with input parameters, output values, hidden input, and hidden output]

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
- reading / writing a global variable
- 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.

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

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

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 represent 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 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 three instructions have the same effect */
a = 56 ;
*x = 56 ;
*(&a) = 56 ;
```

Pass by value vs reference

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

```c
void swap(int a, int b) {
    int temp = a ;
    a = b ;
    b = temp ;
}
/* example usage */
int x = 10 ;
iy = 20 ;
swap(x,y) ;
/* x = 10, y = 20 */
```

Using pointers (C or C++).

```c
void swap(int *a, int *b) {
    int temp = *a ;
    *a = *b ;
    *b = temp ;
}
/* example usage */
int x = 10 ;
iy = 20 ;
swap(&x,&y) ;
/* x = 20, y = 10 */
```

Using references (C++ only)

Points 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 */
int x = 10 ;
iy = 20 ;
swap(x,y) ;
/* x = 10, y = 20 */
```

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

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

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

Corresponding C function

\[
\text{int \ fact}(\text{int \ n})
\{
\text{int \ m} ;
\text{if} \ (n == 1) \ \text{return} \ 1 ;
m = n * \text{fact}(n - 1) ;
\text{return} \ m ;
\}
\]

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

```
int fact(int n)
{
    int m ;
    if (n == 1) return 1 ;
    m = n * fact(n - 1) ;
    return m ;
}
```
Example of recursive calls

int fact(int n)
{
    int m;
    if (n == 1) return 1;
    res:  m = n * fact(n - 1);
    return m;
}

/* 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:

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

**Euler method:** choose a step size \( h \) and an initial condition \( y_0 \) and then let:

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

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
```

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

The Euler solver needs to be modified:

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

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.

**Passing functions as parameters / C**

In C one uses a pointer to a function:

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

This declares a parameter \( \text{func} \).

The type of \( \text{func} \) 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[1] = solve(myF, y[n-1], h);
        printf("Y[%d] = %f\n", n, y[n]);
    }
}
```

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
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_ {
    double re ;
    double im ;
  } c ;
  c.re = 1.0 ;
  c.im = 0.0 ;
  ```

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

  ```c
  struct Complex_ {
    double re ;
    double im ;
  } ;
  typedef struct Complex_ Complex ;
  ```

### Example: VTOL state

- **VTOL state:**
  - height, velocity, mass (numbers)
  - landed (bool)
  - Use a single structure to store all numbers
  - data encapsulation
  - abstraction

- **Example**

  ```c
  typedef struct {
    double position ;
    double velocity ;
    double mass ;
    bool landed ;
  } VTOLState ;
  ```
Creating a structure

- As a local variable
  ```c
  VTOLState state;
  state.position = 10;
  state.velocity = 5;
  state.mass = 1000;
  state.landed = false;
  ```

- Dynamically
  ```c
  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
  ```c
  double getThrust(VTOLState state);
  double t = getThrust(state);
  ```

- By a pointer
  ```c
  double getThrust(VTOLState *statePtr);
  double t = getThrust(statePtr);
  double t = getThrust(&state);
  ```
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.

/* 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;
}

Array representation in C

In C an array is represented as a sequence of records at consecutive memory addresses.

/* 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.

/* A 2x5 array */
double A[2][5];

Static vs dynamic arrays in C

This C statement defines an array a of five integers


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:

int arrySize = 5 ;
int *A = malloc(sizeof(int) * arrySize) ;

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

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
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.


### Insertion

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

% Input: array A of size \( \geq n \) such that \( A[1] \leq \ldots \leq A[n-1] \)
% Output: permuted array such that \( A[1] \leq \ldots \leq A[n] \leq A[n] \)

function \( A = \text{insert}(A, n) \)

\[
\begin{align*}
i &= n \\
\text{while} & i > 1 \text{ and } A[i-1] > A[i] \\
& \quad \text{swap}(A[i-1], A[i]) \\
& \quad i = i - 1 \\
\text{end}
\end{align*}
\]

I.e., start with the first \( n-1 \) elements sorted, end with \( n \)

### Insertion sort

% Input: an array A with \( n \) elements

function \( A = \text{insertionSort}(A, n) \)

\[
\begin{align*}
i &= 1 \\
\text{while} & i < n \\
& \quad i = i + 1 \\
& \quad A = \text{insert}(A, i) \\
\text{end}
\end{align*}
\]

I.e., start with the first \( n-1 \) elements sorted, end with \( n \)

### Insertion: example

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

% Input: an array A with \( n \) elements

\[
\begin{align*}
i &= 9 \\
i &= 8 \\
i &= 7 \\
i &= 6 \\
i &= 5 \\
i &= 4 \\
i &= 3 \\
i &= 2 \\
i &= 1
\end{align*}
\]

% Input: an array A with \( n \) elements

\[
\begin{align*}
i &= 9 \\
i &= 8 \\
i &= 7 \\
i &= 6 \\
i &= 5 \\
i &= 4 \\
i &= 3 \\
i &= 2 \\
i &= 1
\end{align*}
\]

% Input: an array A with \( n \) elements

\[
\begin{align*}
i &= 9 \\
i &= 8 \\
i &= 7 \\
i &= 6 \\
i &= 5 \\
i &= 4 \\
i &= 3 \\
i &= 2 \\
i &= 1
\end{align*}
\]

% Input: an array A with \( n \) elements

\[
\begin{align*}
i &= 9 \\
i &= 8 \\
i &= 7 \\
i &= 6 \\
i &= 5 \\
i &= 4 \\
i &= 3 \\
i &= 2 \\
i &= 1
\end{align*}
\]
Insertion sort: example

Task: sort 5 4 1 2 3

<table>
<thead>
<tr>
<th>insert(2)</th>
<th>insert(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4 1 2 3</td>
<td>1 2 4 5 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 1 2 3</td>
</tr>
<tr>
<td>4 5 1 2 3</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 5 2 3</td>
</tr>
<tr>
<td>1 4 2 5 3</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4 1 2 3</td>
</tr>
</tbody>
</table>

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 \( \text{insert}() \) is \( O(m) \) as the while loop is executed at most \( m \) times. The time complexity of \( \text{insertionSort}() \) is \( O(n^2) \) because

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

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 and conquer
  - 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)
    A1 = A(1:k)
    A2 = A(k+1:end)
    A1 = mergeSort(A1)
    A2 = mergeSort(A2)
    A = merge(A1, A2)
end

function A = merge(A1, A2)
    i1 = 1, i2 = 1
    m1 = length(A1), m2 = length(A1)
    while i1 <= m1 and i2 <= m2
        if A1[i1] <= A2[i2]
            A[i1+i2-1] = A1[i1], i1 = i1 + 1
        else
            A[i1+i2-1] = A2[i2], i2 = i2 + 1
        end
    end
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
- \( \sim 1 \) 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.

```c
function bisect(f, a, b)
    m = (a + b) / 2
    if f(m) close to zero then 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 new one.

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

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

/* Create an element */
ListElement *element = malloc(sizeof(ListElement));
element->next = NULL;
element->value = 42.0;

/* Insert an element in a list */
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 \( O(1) \).

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

---

## Linked list

- Search, insertion, deletion
- Trees
  - Binary search trees
- Graphs
  - Minimum spanning tree
Removing an element from a linked list

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

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

Example usage

/* Remove the element after previous */
ListElement * removed ;
removed = remove(previous) ;

/* Do not forget to release the memory if needed */
if (removed != NULL) {
    free(removed) ;
}

---

**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) return
    visit(node.left)
    visit(node.right)
    print(node.value)
}
```

---

**Binary search tree**

- A **binary search tree** is a binary tree such that the value of each node is
  - at least as larger as the value of its left descendants
  - smaller all the values of its right descendants
- Its main purpose is to support the binary search algorithm.

---

Lecture 4 outline

- **Arrays**
  - In MATLAB and C
  - Pointer arithmetic
- **Sorting**
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- **Trees**
  - Binary search trees
- **Graphs**
  - Minimum spanning tree
**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.

```matlab
function node = binarySearch(node, x)
    if node == NULL return NULL end
    if node.value == x return node end
    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.

---

**Graphs**

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

```matlab
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 a \( n \times n \) matrix such that \( A(u,v) = 1 \) if, and only if, \((u,v) \in E\).

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{bmatrix}
\]

---

**Minimum spanning tree**

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

![Minimum spanning tree diagram](image)

- 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.