Algorithms and processes

Recall: An algorithm is a description of how to carry out a process.

- A process interacts with its environment (by accepting input and producing output).
  Example: the process of computing weekly wages requires information about wage rates and hours worked; it produces information about wages payable and taxes deducted.

- A process can terminate (e.g. the sweater is completed) or not (e.g. staying happy).
Nontermination

- Even processes carried out by computers may never terminate (e.g., controlling traffic signals).
- Under certain circumstances the process described may not terminate, whereas it is always supposed to do so.

Nontermination is a major source of error in the design of an algorithm!
Recall: An algorithm to be executed by a computer is written in a programming language and is then called a program.

Natural languages like English are not suitable for expressing algorithms:

- Enormous vocabulary and grammatical rules
- Interpretation is complex and not well-defined (e.g. metaphors)
  Example: “funny situation” (peculiar or humorous?)

⇒ We must express an algorithm in a simpler form, using a programming language.
Why are there so many programming languages?

- Computer programming is a comparatively recent activity; new programming paradigms are developed, e.g. object orientation or aspect orientation
- Special-purpose languages have been developed for expressing algorithms in specific application areas.
- Tendency to reinvent the wheel ("Our own wheels are better than anyone else’s")

Example

**COBOL**  MULTIPLY price BY quantity GIVING cost

**Pascal**  cost := price * quantity
A processor must be able to interpret an algorithm to carry out the process the algorithm describes. That is, it must be able to

1. understand the form in which the algorithm is expressed (e.g. a knitting pattern or a musical score).
2. carry out the corresponding actions.

To understand the expression of an algorithm the processor

- must know the vocabulary and grammar of the used language, e.g., “sleeve” is an English noun, “=” is a relation between two numbers
- must be able to recognize and make sense of the symbols in which the algorithm is expressed, e.g. “sleeve the seems” is a violation of the rules of English, $a+ = b$ is an incorrect mathematical expression
Definition: Syntax

The set of grammatical rules which govern how the symbols in a language may be legitimately used is called the *syntax* of the language. For example, the syntax of English governs the use of the word *sleeve*.

- A program which adheres to the syntax of the language in which it is written is called *syntactically correct*.
- A deviation from the syntax of the language is called a *syntax error*.
- Syntactic correctness is a necessary prerequisite for interpreting a computer program.
- Programming languages in contrast to natural languages have a relatively simple syntax.
Meaning of expressions

Second stage in understanding the expression of an algorithm: attach meaning to each step, in terms of the operations the processor is intended to carry out.

Example: the meaning of the Pascal expression

\[
\text{cost} := \text{price} \times \text{quantity}
\]

is that two numbers called \textit{price} and \textit{quantity} are to be multiplied together to give a third number called \textit{cost}.

\textbf{Remark} When we express algorithms to be executed by a computer, we will use a programming-language-like notation.
Definition: Semantics

The meaning of particular forms of expression in a language is called the **semantics** of the language.

- Programming languages in contrast to natural languages have relatively simple semantics

Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>the elephant ate the peanut</strong></td>
<td>meaningful sentence</td>
</tr>
<tr>
<td><strong>the peanut ate the elephant</strong></td>
<td>syntactically correct, semantically meaningless</td>
</tr>
<tr>
<td><strong>write down the name of the 1st month of the year</strong></td>
<td>meaningful step in an algorithm</td>
</tr>
<tr>
<td><strong>write down the name of the 13th month of the year</strong></td>
<td>syntactically correct, semantically meaningless</td>
</tr>
</tbody>
</table>
Detection of **semantic inconsistencies** relies on knowledge of the objects being referred to, in particular their attributes and the relations among them, e.g. relative attributes of peanuts and elephants.

Consider e.g. a processor faced with the command *Write down the name of the 13th month of the year*.

If the processor knows that there are only 12 months in the year it can detect the semantic inconsistency in this command before trying to execute it. Otherwise it will attempt to execute the command, possibly by looking up the name in a calendar. Then, inconsistencies become apparent only during execution (*run-time error*).
More subtle semantic inconsistencies

- When an inconsistency is a result of executing an earlier part of an algorithm there is in general no chance of detecting it beforehand.

- Example:

  think of a number from 1 to 13; call this number N; write down the name of the Nth month of the year

contains a potential inconsistency
Steps in interpreting an algorithm

1. Make sense of the symbols in which the step is expressed
2. Attach meaning to the step in terms of operations to be performed
3. Perform the appropriate operations

Syntax errors can be detected in stage 1, and certain semantic errors can be detected in stage 2. Other semantic errors will not be detected until stage 3.

When the processor is a computer, stages 1 and 2 are performed by a *translator*. 
Logical errors

Third class of error in addition to syntactic and semantic errors. A program may be syntactically correct and contain no semantic inconsistencies, but may not properly describe the desired process.

Example:

*compute the circumference of a circle by multiplying the radius by $\pi$*

is syntactically and semantically correct, but produces the wrong result.

Logical error: omission of the factor of 2.

A processor cannot detect logical errors, since it has no idea what process the algorithm is intended to describe.
Design of nontrivial algorithms is very difficult.

Designers tend to overlook certain circumstances.

Often there are unforeseen circumstances, such that the algorithm *usually* results in the intended process being carried out but not always.
Difficulty of designing algorithms

- Design of nontrivial algorithms is very difficult.
- Designers tend to overlook certain circumstances.
- Often there are unforeseen circumstances, such that the algorithm *usually* results in the intended process being carried out but not always.
- Example: calculation of flight time from an airline timetable:
  
  *look up departure time and arrival time; subtract departure time from arrival time*
Difficulty of designing algorithms

- Design of nontrivial algorithms is very difficult.
- Designers tend to overlook certain circumstances.
- Often there are unforeseen circumstances, such that the algorithm *usually* results in the intended process being carried out but not always.
- Example: calculation of flight time from an airline timetable:
  
  *look up departure time and arrival time; subtract departure time from arrival time*

- This algorithm will fail if the departure point and the destination are in different time zones, or daylight saving time is used only at one end of the route but not at the other.
The idea of **stepwise refinement** is to (repeatedly) break a complex process into a number of steps, each of which can be described by an algorithm which is smaller and simpler than that for the entire process.

Also known as **top-down design** or divide and conquer.

Designer must know what sort of step the processor can interpret to know when a particular step in an algorithm is sufficiently primitive to need no further refinement.
Example: Stepwise refinement (1)

Initial version of an algorithm describing how to make a cup of instant coffee:

1. *boil water*
2. *put coffee in cup*
3. *add water to cup*

Detailed enough for a human servant but not for a robot. Refinement of the step *boil water* into a sequence of simpler steps:

1. *boil water*
   1.1 *fill kettle*
   1.2 *switch on kettle*
   1.3 *wait until boiling*
   1.4 *switch off kettle*
Example: Stepwise refinement (2)

Refinement of the step *put coffee in cup* into a sequence of simpler steps:

2. *put coffee in cup*
   - 2.1 *open coffee jar*
   - 2.2 *extract one spoonful of coffee*
   - 2.3 *tip spoonful into cup*
   - 2.4 *close coffee jar*

Refinement of the step *add water to cup*:

3. *add water to cup*
   - 3.1 *pour water from kettle until cup is full*
Example: Stepwise refinement (3)

Refinement of the step *fill kettle* into a sequence of simpler steps:

1.1 *fill kettle*
   - 1.1.1 *put kettle under tap*
   - 1.1.2 *turn on tap*
   - 1.1.3 *wait until kettle is full*
   - 1.1.4 *turn off tap*

and so on . . .
Final version of the coffee-making algorithm (1)

{Algorithm for making a cup of coffee}
{first boil water}

1.1.1 put kettle under tap
1.1.2 turn on tap
1.1.3 wait until kettle is full
1.1.4 turn off tap
1.2 switch on kettle
1.3.1 wait until kettle whistles
1.4 switch off kettle
Final version of the coffee-making algorithm (2)

{put coffee in cup}

2.1.1 take coffee jar from shelf
2.1.2 remove lid from jar
2.2 extract one spoonful of coffee
2.3 tip spoonful into cup
2.4.1 put lid on coffee jar
2.4.2 replace coffee jar on shelf

{add water to cup}

3.1 pour water from kettle into cup until cup is full
Terminating the refinement

The designer of an algorithm must know when a particular step in the algorithm is sufficiently primitive to need no further refinement.

⇒ The designer must know what sort of step the processor can interpret.

Example: *switch on kettle* is primitive, while *fill kettle* is not.

The designer of a computer algorithm refines the algorithm in such a way that its steps can be expressed in an appropriate programming language, and terminates the refinement when every step is expressed in the language concerned.
The simplest form of an algorithm is a **sequence** of steps, meaning that:

- The steps are executed one at a time.
- Each step is executed exactly once; none is repeated and none is omitted.
- The order in which the steps are executed is the same as that in which they are written.
- Successive steps are separated with “;”.
- Termination of the last step implies termination of the algorithm.
- An algorithm which is solely a sequence of steps is extremely inflexible, because its course of execution is fixed and cannot be modified by circumstances.
Example:

- What happens in the coffee-making example if the coffee jar is empty? The robot will probably either halt in confusion or serve a hot cup of water.
- How could the robot handle a request for several cups of coffee? Would it boil the water separately for each cup?
- How could the sequence of steps be extended to cater for optional milk and sugar?
Sequence (3)

Another example:

*catch Underground train to Heathrow Airport, London;*  
*fly by plane to Kennedy Airport, New York;*  
*take a cab from Kennedy Airport to downtown New York*
Another example:

*catch Underground train to Heathrow Airport, London; fly by plane to Kennedy Airport, New York; take a cab from Kennedy Airport to downtown New York*

No provision is taken for the not unlikely circumstance if there being either an Underground strike in London or a cab drivers’ strike in New York.

⇒ The sequence is a very primitive structure for an algorithm. We need less primitive structures.
Selection

- The ability to execute a certain step of an algorithm dependent on different cases is called **selection**.

- General form: `if condition then step endif`
  
  `condition` specifies the circumstance under which `step` is to be executed.

- The processor must be able to interpret the conditions in an algorithm just as it must be able to interpret the steps. 
  \[\Rightarrow\] Conditions must be refined, too!

- **if, then, and endif** are so-called **keywords**.

- Example:
  
  ```
  take coffee jar off shelf
  if jar is empty then get new jar from cupboard endif
  remove lid from jar
  ```

- The condition `jar is empty` will be evaluated to decide if the step `get new jar from cupboard` is executed.
Selection with two alternative steps

- An extension of the selection is a form which determines which of two alternative steps is to be executed.

- General form: \texttt{if condition then step\_1 else step\_2 endif}

- \texttt{if Underground on strike}
  \hspace{1cm} \texttt{then catch airport bus}
  \hspace{1cm} \texttt{else catch Underground train}
  \hspace{1cm} \texttt{endif}

- The condition \texttt{Underground on strike} will be evaluated to decide if the step \texttt{catch airport bus} or the step \texttt{catch Underground train} is executed.

- \texttt{if condition then step endif}
  is equivalent to
- \texttt{if condition then step else do nothing endif}
Nesting

- We can design algorithms which contain nested occurrences of selection.
- Consider a set of traffic signals:
  ```c
  if no signal
    then proceed with great caution
  else if signal is red or signal is amber
    then stop
  else proceed
  endif
  endif
  ``
- The second selection is nested inside the first and is executed only if the signal is working.

**Summary:** The power of selection is that it allows a processor to follow different paths through an algorithm according to circumstance. Without selection it would be impossible to write algorithms of any significant practical use.
Indentation

```plaintext
if no signal
    then proceed with great caution
else if signal is red or signal is amber
    then stop
    else proceed
endif
endif
```

is more readable than

```plaintext
if no signal
    then proceed with great caution
else if signal is red or signal is amber
    then stop
else proceed
endif
endif
```

⇒ Indentation is important for human readers!
Develop an algorithm for determining the largest of three numbers \( x, y, z \), given that the processor can compare only two of them at a time.

First version

\[
\text{if } x > y \\
\quad \text{then } \text{choose between } x \text{ and } z \\
\quad \text{else } \text{choose between } y \text{ and } z \\
\text{endif}
\]

Refining the choice between \( x \) and \( z \):

\[
\text{if } x > z \\
\quad \text{then } \text{choose } x \\
\quad \text{else } \text{choose } z \\
\text{endif}
\]

The choice between \( y \) and \( z \) can be refined similarly.
Indentation/Example (2)

Final algorithm:

```plaintext
if \( x > y \)
    then if \( x > z \)
        then choose \( x \)
        else choose \( z \)
    endif
else if \( y > z \)
    then choose \( y \)
    else choose \( z \)
endif
endif
```

Note how indentation makes clear which steps of the algorithm are part of each of the three selections.
Consider the process of looking up a person's address, given the name, from a list of names and addresses:

*consider the first name in the list*

**if** this name is the given name  
  **then** extract the corresponding address  
  **else** consider the next name in the list  
  **if** this name is the given name  
  **then** extract the corresponding address  
  **else** consider the next name in the list  
  **if** this name is the given name  
  **then** . . .

Problem: When should we stop writing?

⇒ Sequence and selection are not sufficient to express algorithms whose length varies according to circumstance.
The ability of repeating certain steps of an algorithm an arbitrary number of times is called iteration.

General form:
```
repeat body of loop until terminating condition endrepeat
```

Lookup algorithm:
```
consider the first name in the list
repeat
  if this name is the given name
    then extract the corresponding address
    else consider the next name in the list
  endif
until given name is found or list is exhausted endrepeat
```

Use indentation to make the body of the loop stand out from the surrounding text.
Power of iteration: allows a process of indeterminate duration to be described by an algorithm of finite length.

However: possibility of non-termination arises.

Responsibility: ensure that the iteration does indeed terminate when (and if) intended.

Example: the termination condition *given name is found* or *list is exhausted* eventually becomes true.

However: leaving out the second part *or list is exhausted* is potentially disastrous. (If the name being sought is not present the processor will run off the end of the list.)

Failure to specify a terminating condition correctly is one of the most common errors in algorithm design!
Infinite loops and nested loops

- Some processes are not intended to terminate, and an algorithm describing such a process must contain a loop which is executed forever.

- General form:
  \textit{repeat} body of loop \textit{forever}

- We can define so called \textbf{nested loops}: In this case an algorithm contains two loops, the \textit{inner loop} nested inside the \textit{outer loop}. 
Example: Prime number (1)

Task: calculate the first prime number which is greater than a given “starting” number (which is assumed to be a positive integer).

obtain starting number;
repeat
   add 1;
   test number for primeness
until number is prime endrepeat;
write down number

Why does this algorithm terminate?
The step *test number for primeness* must be refined, because it is unlikely that the processor can execute it directly.

Test if a number *potential prime* has any nontrivial factors.

\[
\text{divide potential prime by every number between 1 and itself; if no division is exact then potential prime is prime else potential prime is not prime endif}
\]

The first step clearly involves iteration, in which the potential prime is divided by successive possible factors starting with 2.
Example: Prime number (3)

set possible factor to 2;
repeat
  divide potential prime by possible factor;
  add 1 to possible factor
until division is exact or possible factor > $\sqrt{\text{potential prime}}$
endrepeat;
if no division is exact
  then potential prime is prime
  else potential prime is not prime
endif

We can stop looking for possible factors when division is exact or possible factor > $\sqrt{\text{potential prime}}$

This saves processing time!
Example: Prime number, final algor. (nested loops)

obtain starting number;
set potential prime to starting number;
repeat
  add 1 to potential prime;
  set possible factor to 2;
  repeat
    divide potential prime by possible factor;
    add 1 to possible factor
  until division is exact or possible factor > \sqrt{potential prime} endrepeat;
if no division is exact
  then potential prime is prime
else potential prime is not prime
endif
until potential prime is prime endrepeat;
write down potential prime
A **repeat loop** has its terminating condition at the end, after the **until** which follows the loop body.

This implies that the body is always executed at least once, even if the terminating condition is initially true, since the terminating condition is arrived at only after the body has been executed.

Example: determine largest of a list of numbers

```plaintext
set largest so far to first number in list;
repeat
    consider next number in list;
    if this number > largest so far
        then set largest so far to this number
    endif
until list is exhausted endrepeat;
write down largest so far
```
Pre-tested loop

The above algorithm seems correct, but it contains a serious error which becomes apparent if the list has only one number in it. We must put the terminating condition at the start of the loop rather than at the end:

- A **while loop** has its terminating condition at the start, before the **do** which precedes the loop body.
- General form:
  ```
  while terminating condition do body of loop endwhile
  ```
- Using a while loop means that the body of the loop is to be executed repeatedly as long as the terminating condition is true.
Example: Pre-tested loop

Correct algorithm:

_set largest so far to first number in list;_

while list is not exhausted do
   consider next number in list;
   if this number > largest so far
      then set largest so far to this number
   endif
endwhile;
write down largest so far

Remember: A post-tested loop is always executed at least once, whereas the body of a pre-tested loop may not be executed at all.

Note: The input list must be non-empty!
Example: Greatest common divisor (GCD)

Mathematical laws:

\[
\begin{align*}
GCD(x, y) &= GCD(y, \text{remainder of } x/y) & \text{if } y > 0 \\
GCD(x, y) &= x & \text{if } y = 0
\end{align*}
\]

where \( x \) and \( y \) are any nonnegative integers.

Example:
\[
GCD(24, 9) = GCD(9, 6) = GCD(6, 3) = GCD(3, 0) = 3
\]

Algorithm:

\[
\begin{align*}
\textbf{while } y \neq 0 \textbf{ do} \\
& \quad \text{calculate remainder of } x/y; \\
& \quad \text{replace } x \text{ by } y; \\
& \quad \text{replace } y \text{ by remainder} \\
\textbf{endwhile;} \\
\text{write down the answer } x
\end{align*}
\]

A post-tested loop could result in division by 0!
Definite iteration

- Particularly simple form
- Number of repetitions is known before the loop is executed
- General form:
  
  \[
  \text{repeat } N \text{ times} \\
  \text{body of loop} \\
  \text{endrepeat}
  \]

  where \( N \) is an arbitrary positive integer

- Example: calculate \( x^N \)

  obtain values of \( x \) and \( N \);
  set product to 1;
  repeat \( N \) times
  multiply product by \( x \)
  endrepeat;
  write down product
Definite iteration is safer than indefinite iteration, because it is guaranteed to terminate, but it is by no means as powerful.
Alternative notation

- **repeat** for each instance of an object
  body of loop

- Examples
  - **repeat** for each wheel
  check tyre pressure
  endrepeat
Alternative notation

- **repeat** for each instance of an object
  body of loop

- Examples
  - **repeat** for each wheel
    check tyre pressure
  - endrepeat

- calculate Factorial of $N$

  obtain value of $N$;
  set product to 1;
  **repeat** for each integer from 1 to $N$
    multiply product by integer
  endrepeat;
  write down product
Remark:

- Algorithms use **variables**.
- In the factorial algorithm, the used variables are $N$, \textit{product}, and \textit{integer}.
- \textit{integer} is not a good variable name, because it usually denotes the \textit{type} of integers (and is a keyword).
- Variables may have **values** of some specific \textit{type}.
- The initial value of the variable should always be defined by the algorithm (for example, \textit{set product to 1}).
- The variable values change during the execution of the algorithm (\textbf{assignment}).
  Notation $x := t$, where $x$ is a variable and $t$ is an expression of the same type as $x$; example: $x := x + 1$.
- New variables may be introduced during algorithm design.
Modularity: motivation

- Stepwise refinement divides algorithms into smaller components.
- These are often independent of the context in which they are to be used.
- “Plug-in” components, called modules (in programming languages: procedure, routine, function, method)
- A module is a self-contained algorithm.
- An algorithm that uses a module is said to call the module.
Example: Picture-drawing robot (1)

- A robot is equipped with wheels which allow it to move around over a sheet of paper.
- It has a pen which it can lower onto the paper when it wants to draw a line.
- It can interpret and execute commands of the form

  move(x)  move forward x cm
  left(x)  rotate x degrees to the left
  right(x) rotate x degrees to the right
  raise pen  lift pen off paper
  lower pen  put pen down on paper
Example: Picture-drawing robot (2)

move to point A;
draw a square of side 10 cm;
move to point B;
draw a square of side 20 cm

- Drawing squares is a self-contained process quite independent of the rest of the algorithm.
- Hence, we can design a square-drawing algorithm without reference to the larger algorithm in which it is to be used.
- Once that algorithm is designed we can simply “plug it in” at the appropriate points of the above algorithm.
For a square drawing module to be generally useful, it must be capable of drawing a square of *any* size (formal parameter).

When the module is called, the *specific* size required must be indicated (actual parameter).

Example: `drawsquare(10)` and `drawsquare(20)`

Formal parameters represent the information the module needs when it is called.

Actual parameters are the pieces of information supplied for a particular call.
Module for drawing squares

module drawsquare(size)
{Draws a square of size cm. The square is drawn counterclockwise, starting from the current position of the robot. The first edge is drawn according to the robot's current orientation. The robot returns to its initial position and orientation, with the pen raised.
Assumption: Initially the pen is raised.}

lower pen;
repeat 4 times
  move(size);
  left(90)
endrepeat;
raise pen
endmodule

Note: Comments (and specifications) are important!
Syntax for defining and calling modules

- **Definition**
  - Procedure: is called for the effects it causes
    ```
    module modulename(formal parameters)
    ... body of module ...
    endmodule
    ```
  - Function: is called to obtain the function value
    ```
    module modulename(formal parameters): Type
    ... body of module ... return ...
    endmodule
    ```

- **Call**
  - Procedure: `modulename(actual parameters)`
  - Function: `x := modulename(actual parameters)`
    Body of module is executed, with the formal parameters replaced by the actual parameters
  - Number and types of formal and actual parameters must be the same!
Refinement of example algorithm

\{ Assumption: robot is initially at point \( X \) \}

\begin{align*}
left(45); \\
\text{move}(\sqrt{50}); & \quad \{ \text{move to point } A \} \\
left(135); \\
\text{drawsquare}(10); & \quad \{ \text{draw inner square} \} \\
right(135); \\
\text{move}(\sqrt{50}); & \quad \{ \text{move to point } B \} \\
left(135); \\
\text{drawsquare}(20) & \quad \{ \text{draw outer square} \}
\end{align*}
Abstraction of the square-drawing algorithm

Modules can be nested, too.

module drawpolygon(size, N)
{Draws an \(N\)-sided polygon with sides \(size\) cm long. The polygon is drawn counterclockwise, starting from the current position of the robot. The first edge is drawn according to the robot’s current orientation. The robot returns to its initial position and orientation, with the pen raised. Assumption: Initially the pen is raised.}
lower pen;
repeat \(N\) times;
    move(size);
    left(360/N)
endrepeat;
raise pen
endmodule
Using *drawpolygon*

```plaintext
module drawsquare(size : Real)
{Draws a square of *size* cm. . . .}
drawpolygon(size, 4)
endmodule

module drawtriangle(size : Real)
{Draws a triangle of *size* cm. . . .}
drawpolygon(size, 3)
endmodule
```

Note:
- The new definition of *drawsquare* does not require any alterations of the algorithms in which it is called!
- The input parameters must be positive numbers!
Advantages of using modules

- Modules fit naturally into stepwise refinement, giving a top-down design.
- A module is a self-contained component of any larger algorithm which calls it. The design of the module and of the calling algorithm can be considered separately.
- To incorporate a module in an algorithm it is necessary to know only *what* the algorithm does, but not *how* it does it.
- Modules not only simplify the design, but also the understanding of algorithms.
- Once a module has been designed it can be incorporated in any algorithm which needs it. It is therefore possible to build up a "library" of modules.
Summary: basic concepts

- The concept of an *algorithm* is central to computer science.
- Algorithms are timeless and do not depend on the technology of the day.
- An algorithm describes a *process* to be carried out by a *processor*.
- To execute an algorithm on a computer, it must be transformed into a *program*, expressed in some programming language.
- The description of an algorithm must be free of *syntactic errors*, *semantic inconsistencies*, and *logical errors*.
Summary: expressing algorithms

- **Control structures** for algorithms are *sequence*, *selection*, and *iteration*. In combination with basic steps (assignments, evaluation of expressions), they suffice to express all conceivable algorithms.

- To re-use algorithms, we use **modules**.

- Modules have **formal parameters**, and they are called with **actual parameters**.

- For each algorithm, it must be shown that it **terminates**.
Algorithms can be developed by **stepwise refinement**.

In stepwise refinement, we start out with a coarse-grained algorithm and gradually replace non-basic steps by further algorithms.

This approach corresponds to a **top-down** problem decomposition.

Stepwise refinement terminates when all steps of the algorithm are **basic**, i.e., can be carried out by the envisaged processor.

Therefore, it is necessary to know what steps are basic for a given processor.
For each algorithm, we must write down the intended process that algorithm is expected to perform; otherwise we cannot know if the algorithm is correct.

This description is called the **specification** of the algorithm.

Such a specification consists of a **precondition** and a **postcondition**.

They form a **contract** between a module/algorithm and its callers.
Design by Contract

Source:
Bertrand Meyer
Object-Oriented Software Construction

online see:
Contracts in daily life

- Contractual partners are clients and sellers or service providers.
- Both expect advantages from the contract and are willing to make a commitment.
## Example: traveling

I want to travel from Berlin to Duisburg.

<table>
<thead>
<tr>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Passenger</strong></td>
<td><strong>Getting to Duisburg</strong></td>
</tr>
<tr>
<td>Pay ticket</td>
<td>Has advantages from the postcondition</td>
</tr>
<tr>
<td>Be there at departure time</td>
<td></td>
</tr>
<tr>
<td>Must keep precondition</td>
<td></td>
</tr>
<tr>
<td><strong>Traffic provider</strong></td>
<td>Receives the price for the ticket; does not have to take passengers who have not paid or did not arrive in time</td>
</tr>
<tr>
<td>Must take the passenger to Duisburg</td>
<td>Can assume precondition</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Must guarantee postcondition</td>
<td></td>
</tr>
</tbody>
</table>
Example: tailor services

I want to get my trousers shortened.

<table>
<thead>
<tr>
<th></th>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>Bring trousers</td>
<td>Get shortened trousers</td>
</tr>
<tr>
<td></td>
<td>Pay</td>
<td>Has advantages from the postcondition</td>
</tr>
<tr>
<td></td>
<td>Must keep precondition</td>
<td></td>
</tr>
<tr>
<td><strong>Tailor</strong></td>
<td>Must shorten my trousers</td>
<td>Receives payment; does not have to shorten trousers that are not taken to the workshop or have not been paid for</td>
</tr>
<tr>
<td></td>
<td>Must guarantee postcondition</td>
<td>Can assume precondition</td>
</tr>
</tbody>
</table>
Advantages of explicit contracts

Meyer:

*A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.*
A contract is a formal agreement between a module/algorithm and its callers. It specifies the rights and duties for both sides:

**caller** must ensure precondition of module/algorithm can assume that postcondition holds after the module/algorithm has terminated

**module/algorithm** can assume precondition must ensure postcondition

If the caller calls the module/algorithm with the precondition not satisfied, the module/algorithm has no obligations whatsoever!

It is not the module/algorithm that has to test if the precondition holds, but the caller of the module!
In order to develop correct algorithms (i.e., algorithms that indeed describe the intended process), the algorithm development task must be stated unambiguously by describing the precondition and the postcondition of the algorithm.

- In the following, we will not use verbal descriptions any more, neither for specifications nor for algorithms.
- For specifications, we will use formulas.
- We will furthermore classify the variables used in the formulas expressing the pre- and postcondition of an algorithm (instead of read- or write-statements).
Stating algorithm development problems precisely

We specify algorithms as follows:

- **pre** precondition, expressed as a formula
- **post** postcondition, expressed as a formula
- **reads** variables that are allowed to be read by the algorithm, but not changed
- **changes** variables that are allowed to be changed (and read) by the algorithm
- **mem** variables that are not allowed to occur in the algorithm; they serve to memorize the state in which the algorithm is called.

where

- **reads**, **changes** and **mem** contain all (free) variables used in **pre** and **post**
- **reads**, **changes** and **mem** are disjoint
Correctness of algorithms

An algorithm is *correct* with respect to an algorithm design problem iff

- the algorithm, started in a state where \( \text{pre} \) holds, terminates in a state where \( \text{post} \) holds (*total correctness*, as opposed to *partial correctness*, where the termination is not shown) and
- the algorithm does not contain any variables from \( \text{mem} \) and does not change any variables from \( \text{reads} \).

We can introduce new variables during algorithm design, that are not contained in \( \text{reads}, \text{changes} \) and \( \text{mem} \). There is no restriction on the use of the newly introduced variables. Recall that

- the precondition characterizes the conditions under which the algorithm may be called
- the postcondition describes what the algorithm does, i.e., its effect
**Summary: algorithm specifications**

**Precondition** describes input (**reads**, possibly **changes**-clause) and pre-state (**pre**-clause)

**Postcondition** describes relation between input/pre-state and output/post-state (**changes**, possibly **mem** and **post**-clauses)
If precondition is not satisfied

If we call an algorithm with the precondition not satisfied
  - we do not know if there is any output and – if so – how it looks like
  - we do not know if the module will terminate and – if so – how the post-state will look like
Important container type: arrays

Arrays
- are collections of elements of the same type
- can be accessed via an *index*
- have a fixed size
- can have more than one dimension (e.g., matrices)
- we will e.g. consider arrays $a[1..n]$ of integer values

Example:

<table>
<thead>
<tr>
<th>John</th>
<th>Kate</th>
<th>Fred</th>
<th>Bill</th>
<th>Sam</th>
<th>Jill</th>
<th>Mary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
We will use the following statements to express algorithms:

**assignment** \( x := t \) or \( x := module(\ldots) \) or \( a[i] := t \)
assigns the value of the expression \( t \) or the return value of \( module() \) to the variable \( x \) or the value at the \( i \)th position of the array \( a \)

**composition/sequence** if \( S_1 \) and \( S_2 \) are statements, then
\( S_1; S_2 \) is a statement

**conditional/selection** if \( b \) is a boolean expression and \( S_1, S_2 \) are statements, then
\[ \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ endif} \] and
\[ \text{if } b \text{ then } S_1 \text{ endif} \] are statements
**Algorithm notation II**

**while loop**

if $b$ is a boolean expression and $S$ is a statement, then

**while $b$ do $S$ endwhile** is a statement

**repeat loop**

if $b$ is a boolean expression and $S$ is a statement, then

**repeat $S$ until $b$ endrepeat** is a statement

**procedure call**

$modulename(\ldots)$ is a statement
Example: procedure for GCD

Specification:

pre \quad x \geq 0 \quad \textbf{and} \quad y \geq 0
post \quad r = \text{GCD}(x, y)
reads \quad -
changes \quad r, x, y
mem \quad -

Algorithm (using a procedure):

module gcd(x, y : integer)
while y \neq 0 do
\quad r := mod(x, y);
\quad x := y;
\quad y := r
endwhile;
\quad r := x
endmodule
Rule for memory variables

Problem: the algorithm

\[ x := 1; y := 1; r := 1 \]

would also be correct with respect to the above specification, because the postcondition refers to the values of the variables after termination of the algorithm.

To prevent “cheating”, we have the following

**Specification rule for algorithms where the input may be changed**

If the input of an algorithm is allowed to be changed, introduce memory variables and express the postcondition using the memory variables.
Corrected specification for GCD problem

pre \quad x \geq 0 \quad \textbf{and} \quad y \geq 0 \quad \textbf{and} \quad x = x' \quad \textbf{and} \quad y = y'

post \quad r = GCD(x', y')

reads \quad -

changes \quad r, x, y

mem \quad x', y'
Example: function for GCD

Specification:

\[
\text{pre} \quad x \geq 0 \quad \text{and} \quad y \geq 0 \quad \text{and} \quad x = x' \quad \text{and} \quad y = y' \\
\text{post} \quad \text{gcd}_fct(x, y) = \text{GCD}(x', y') \\
\text{reads} \quad - \\
\text{changes} \quad x, y \\
\text{mem} \quad x', y'
\]

Algorithm (using a function):

\[
\text{module } \text{gcd}_fct(x, y : \text{integer}) : \text{integer} \\
\text{while } y \neq 0 \text{ do} \\
\quad \text{tmp} := \text{mod}(x, y); \\
\quad x := y; \\
\quad y := \text{tmp} \\
\text{endwhile}; \\
\text{return } x \\
\text{endmodule}
\]
# Commitments and advantages

<table>
<thead>
<tr>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller Call $gcd(x, y)$ only with integer numbers $x \geq 0$ and $y \geq 0$</td>
<td>Get result $r$ with $r = GCD(x, y)$. Has advantages from post-condition</td>
</tr>
<tr>
<td>Must keep precondition</td>
<td></td>
</tr>
<tr>
<td>Module Make sure that $r$ really is the GCD of $x$ and $y$</td>
<td>Unnecessary to treat the case where $x$ or $y$ is not a non-negative integer number Can assume precondition</td>
</tr>
<tr>
<td>Must guarantee post-condition</td>
<td></td>
</tr>
</tbody>
</table>
Example: draw polygon

pre \hspace{1cm} size > 0 \textbf{and} N \geq 3 \textbf{and} \textit{raised}(pen)

post \hspace{1cm} \textit{drawn}(\textit{paper, polygon}(size, n)) \textbf{and} \textit{raised}(pen)

reads \hspace{1cm} size, N

changes \hspace{1cm} \textit{paper}

mem \hspace{1cm} -
## Commitments and advantages

<table>
<thead>
<tr>
<th></th>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
</table>
| **Caller** | Call the module only when the pen of the robot is raised and with inputs $size > 0$ and $N \geq 3$  
Must keep precondition | Get a polygon drawn  
Has advantages from post-condition |
| **Module** | Make sure that the polygon is drawn correctly  
Must guarantee post-condition | Unnecessary to treat the case where the the pen is not raised or $size \leq 0$ or $N \leq 3$  
Can assume precondition |
Sequence, selection and iteration are sufficient for constructing any algorithm!!

We want to develop a sorting algorithm in the context of sorting names into an alphabetical order to review these three basic forms.

A well-known and simple algorithm to fulfill this requirement is called **bubble sort**.

The basic idea of bubble sort is to pass through the list of names, stored in an array, comparing each name with its successor and interchanging them if they are out of order; if at the end of the pass all the names are in the correct order nothing further needs to be done; if not, another pass is made through the list and the passes must be repeated as long as any names are out of order.
Sorting: specification

Let $a$ be an array with lower index 1 and upper index $n$.

\begin{align*}
\text{pre} & \quad n \geq 1 \text{ and } a = a' \\
\text{post} & \quad \text{ordered}(a) \text{ and permutation}(a, a') \\
\text{reads} & \quad n \\
\text{changes} & \quad a \\
\text{mem} & \quad a'
\end{align*}

In general, we can sort all data for which an ordering relation such as $<$ or $\leq$ is defined.
## Sorting: commitments and advantages

<table>
<thead>
<tr>
<th>Caller</th>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call sorting algorithm with an array ( a ) and an index ( n \geq 1 )</td>
<td>Get a sorted array</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Must keep precondition</td>
<td>Has advantages from post-condition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>Commitments</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make sure that the input array is sorted correctly</td>
<td>Unnecessary to treat the case where the input is not an array and a positive index</td>
<td></td>
</tr>
<tr>
<td>Must guarantee post-condition</td>
<td>Can assume precondition</td>
<td></td>
</tr>
</tbody>
</table>
## Actions of bubble sort

<table>
<thead>
<tr>
<th>original list</th>
<th>after pass 1</th>
<th>after pass 2</th>
<th>after pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>John</td>
<td>Fred</td>
<td>Bill</td>
</tr>
<tr>
<td>Kate</td>
<td>Fred</td>
<td>Bill</td>
<td>Fred</td>
</tr>
<tr>
<td>Fred</td>
<td>Bill</td>
<td>John</td>
<td>Jill</td>
</tr>
<tr>
<td>Bill</td>
<td>Kate</td>
<td>Jill</td>
<td>John</td>
</tr>
<tr>
<td>Sam</td>
<td>Jill</td>
<td>Kate</td>
<td>Kate</td>
</tr>
<tr>
<td>Jill</td>
<td>Mary</td>
<td>Mary</td>
<td>Mary</td>
</tr>
<tr>
<td>Mary</td>
<td>Sam</td>
<td>Sam</td>
<td>Sam</td>
</tr>
</tbody>
</table>

Names bubble upward or sink downward to their correct position in the list.
Example: Bubble sort – Initial version

We use a pre-tested loop since the processor may be fortunate enough to find that the list is already in order and there is nothing to do:

```module bubblesort(a : array, n : integer)
while a is not sorted do
   make a pass through a, exchanging adjacent names as necessary
endwhile
endmodule```

```
Example: Bubble sort – First refinement step

The body of the while-loop can be refined into:

```plaintext
if n > 1 then
    i := 1;
    repeat n - 1 times
        if a[i] > a[i + 1] then
            auxvar := a[i];
            a[i] := a[i + 1];
            a[i + 1] := auxvar
        endif;
        i := i + 1
    endrepeat
endif
```
Since the processor is incapable of telling whether or not the list is sorted, the terminating condition in line 1 needs further refinement.

One way of telling whether the array is sorted is to remember whether any names needed exchanging during the last pass through it.

At least one pass through the list is required to see whether it is sorted to start with. Thus, we should replace the pre-tested loop by a post-tested loop.
Example: Bubble sort – Second refinement step (2)

```plaintext
repeat
    exchanged := false;
    if n > 1 then
        i := 1;
        repeat n - 1 times
            if a[i] > a[i + 1] then
                auxvar := a[i];
                a[i] := a[i + 1];
                a[i + 1] := auxvar;
                exchanged := true
            endif;
            i := i + 1
        endrepeat
    endif
until exchanged = false endrepeat
```
■ Bubble sort is a particularly simple sorting algorithm, but it is not always the fastest

■ Best case: array is (almost) sorted

■ Worst case: last name in array is alphabetically first; then \( n - 1 \) passes are needed

■ Average: number of passes roughly proportional to \( n \)

■ Since each pass involves \( n - 1 \) executions of the inner loop, the average number of steps executed is roughly proportional to \( n^2 \)

■ The number of steps involved in the execution of the algorithm, and hence the time needed to execute it, are attributes of the algorithm which fall under the general term **complexity**