Overview

- Defining our own data-types (enumerated data-types),
- Control structures `case ... of ...`,
- Basic sorting algorithms,
- Compiler directives,
- Files (text files),
- Basic sorting algorithms.

How to pass an array as a parameter?

- In Classical Pascal we have to define our own data-type (show why the naive approach does not work).
- Turbo Pascal (and also Free Pascal) support open-array parameters.

Defining your own data type:

- Keyword `type` permits us to define a new data-type.
- trivial use: `type int=integer;`
- use: `type x=array[1..10] of integer;`

Example

```
program nnn;
type arr=array[1..10] of integer;
var p:arr;
procedure output(a:arr);
var i:integer;
begin
    for i:=1 to 10 do
        writeln(a[i]);
end;
begin
...output(p);
end.
```

Open-array parameters

- Available in Turbo Pascal and Free Pascal.
- We say that the argument is an array of a particular type, but we omit limits.
- Example: `procedure output(a:array of integer);`
- The argument is an array indexed from 0 to N.
- The value N can be determined using a function `high`.

Example using open-array parameters

```
procedure output(a:array of integer);
var i:integer;
begin
      for i:=0 to high(a) do
            writeln(a[i]);
end;
```

Further possibilities

how to use user-defined data-types

We want to calculate the days of a week. How we do that?

- We define constants: Monday=1, Tuesday=2,...
- But I'll change the numbering: Monday=0, Tuesday=1,...
- Then an American comes and enumerates: Sunday=1, Monday=2,...
- Thus we define a special data-type indexed with days of a week,
- the numbers get assigned by the compiler.

Enumerated data-type

- Gets defined in the `type`-section,
- individual values are in the brackets separated by commas.
- Example: `type daysofweek=(monday,tuesday, wednesday, thursday,friday, saturday, sunday);`
- Or we may directly define a variable of enumerated type: `var cal:(monday,tuesday,wednesday,thursday, friday, saturday,sunday);`

Example

- Let us implement a simple "calendar" for the year 2013, i.e., we output the date and day of week.
- For the sake of simplicity let's consider that each month consists of 30 days...
- Source code can be found on the web (kam.mff.cuni.cz/~perm/programovani/NPRG030/enum.pas).
- We see that the `write` function cannot output enumerated data-types.
- What should we do in order to write out the names of the days?
- Either we use large `if`-clause or `case variable of ...`

Structure `case ... of ...`

- It helps us to create many branches (in a program) depending on the value of a variable.
- Syntax:

```
case variable_name of
    value1: statement or blok
    value2: statement or blok
    else statement or blok
end;
```

- Only the branch labeled by current value of the variable gets executed. The `else`-branch gets executed otherwise (for other values).
- The `else`-clause is not compulsory!
- If the last clause is a block, we write the keyword `end` twice (the former closes the block, the latter finishes the `case` block.

Example – calendar

can be found at
`kam.mff.cuni.cz/~perm/programovani/NPRG030/case_of.pas`.

Compiler-directives

- Compiler tests many issues, e.g.:
- whether we are not violating array boundaries,
- whether the stack does not overflow,
- whether the input/output error occured...
- Usually it is a good idea to keep these tests switched on but sometimes we "know what we are doing".
- Then we can switch them off (but only if it is essential).
- We can do that using the *compiler-directives*.
- These directives look like a comment, i.e., they are in the braces,
- just the "comment" begins with the string-character ($). Then we place (usually 1-character long) name and a switch +/−.

Compiler-directives

- Example: $\{\$R-\}$ – switch the *range-checking* off.
- The most important:
  - $\$Q$ – overflow-checking,
  - $\$R$ – range-checking,
  - $\$I$ – input-output tests,
  - The full list can be found in the manual (some directives are compiler-dependent).

Files

and functions related to them

- This time we show handling of text files (binary files appear later).
- A text file is represented by a variable of type `Text`.
- This variable gets assigned to a given file by the `Assign`-function,
- then we open the file using `Reset`, `Rewrite` or `Append`,
- after that we read (using `Read` and `Readln` functions). This time we give the `Text`-type variable as the first argument,
- writing into the file is done in the same way by calling `Write` or `Writeln` functions.
- Finally we close the file using the `Close`-function.

Files

and functions related to them – syntax (1)

- `var f:Text;`
- `Assign(f,'file.txt');` – assing the variable `f` with `file.txt`.
- `Reset(f);` – open the file represented by `f` (for reading).
- `Rewrite(f);` – open `f` if it exists, destroy (erase) its contain.
- `Append(f);` – open `f` for appending (writing behind its current end).

Files

and functions related to them – syntax (2)

- `Writeln(f,'We are writing to the file');` – output the text into the file.
- `Read(f,a);` – Read from the file variable `a`.
- `Close(f);` – Close the file (we won't use it anymore).
- `eof(f);` – function returning `boolean` depending on whether we are (already) at the end of the file.
- `eof;` – function announcing the end of standard input (usually from keyboard).
- There are many further function `Rename`, `Erase`,...

Problems with files

- It can happen that a file we try to open with `Reset` does not exist.
- This causes an input/output error.
- To avoid this we can either destroy the file (calling `Rewrite` – this always creates a file): but this is usually very counter-productive! Alternatively, we use an appropriate compiler-directive (to switch the input/output error off) and if an error occurs, we find out about it by calling the `IOResult`-function.

Example

```
Assign(f,'file.txt');
{$I-} {Switch the tests on input/output errors off}
Reset(f);
{$I+} {Switch IO-error on again}
if IOResult<>0 then
begin writeln('A problem!'); halt;
end;
while not eof(f) do begin
      readln(f,s);
      writeln(s);
end;
```

Beware that `IOResult` is a function and thus after calling it, the error-value gets replaced by 0. Thus we have to store it into a variable (for further use).