Introdução à Computação

2º semestre 2004-2005

cursos: BQ, M, EFT, FQ, Q

---

aulas teóricas

1	15 Fev	Apresentação		
2	16 Fev	Computadores		
3	22 Fev	Memória		
4	23 Fev	PASCAL		
5	1 Mar	Variáveis, Write e WriteLn		
6	2 Mar	Atribuição, Constantes, Write formatado		
7	8 Mar	Read e ReadLn, Cálculo		
8	9 Mar	if ... then ... else, comparações		
9	15 Mar	Boolean álgebra, case ... of		
10	16 Mar	Ciclos I: For		
11	29 Mar	Ciclos II: While-Do Repeat-Until		
12	30 Mar	Programação Modular I Procedures		
13	5 Abr	Programação Modular II Funçoes		

mini testes

1		
2		
3		
4		
5		
6		
7		
8		
9		
10 11		
12 13		
14		
15		
16		
17 18		
19		
20		

aulas práticas

1	1,2 Mar	Windows 1 Windows 2 Internet	
2	8,9 Mar	Compilador Primeiros programas	soluções
3	15,16 Mar	Variáveis, Constantes WriteLn	soluções
4	29,30 Mar	Read, ReadLn, Atribuição, if ... then ... else	soluções
5	5,6 Abr	Case ... Of Ciclos I: For	soluções
6	12,13 Abr	Ciclos e Procedimentos	soluções
7	19,20 Abr	Matemática	soluções
8	26,27 Abr	Vários assuntos	soluções
9	3,4 Maio	Arrays / records	soluções
10	17,18 Maio	Vários assuntos alternativo	soluções soluções
11	24,25 Maio	Ficheiros	soluções
12	31-Maio, 1-Jun	Vários assuntos	soluções
13	7-8 Jun	Vários assuntos	

14	6 Abr	Funções matemáticas		
15	12 Abr	Âmbito das variáveis *Passing by value* / *Passing by reference*		
16	13 Abr	Programação recursiva		
17	19 Abr	Arrays		
18	20 Abr	Records		
19	19 Abr	Definir novos tipos		
20	27 Abr	Apontadores		
21	3 Maio	Ficheiros		
22	4 Maio	Algoritmos		
23	17 Maio	aula de dúvidas I		
24	18 Maio	Resumo		
	24 Maio	explicação de um exame		
24	25 Maio	Outras linguagens		
	31 Maio			
	1 Jun			
	7 Jun			
	8 Jun			

Avaliação:

regras de avaliação

exemplos de exames dos anos passados:
Frequência: 3 de Junho 2002: doc, pdf.
Exame época normal: 7 de Junho 2002: doc, pdf.
Frequência: 11 de Junho 2003 enunciado, solução

Exame época normal: 30 de Junho 2003 enunciado
Exame época normal: 25 de Junho 2004 enunciado, solução

exemplo de um outro exame: DOC, PDF

Horário:

aulas teóricas			
turno	*cursos*	*quando*	*sala*
IC-T1A	BQ, EFT, FQ, M, Q	Terça 12:00-13:00	CP Anf.C
IC-T1B	BQ, EFT, FQ, M, Q	Quarta 12:00-13:00	CP Anf.C

aulas práticas				
curso	*turno*	*quando*	*sala*	*docente*
BQ 2º ano	P3	Quarta 9:30-12:30	C1 0.23	Peter Stallinga
EFT 1º ano	P2	Terça 14:30-17:30	C1 0.23	João Lima
FQ 1º ano	P2	Terça 14:30-17:30	C1 0.23	João Lima
M 1º ano	P2	Terça 14:30-17:30	C1 0.23	João Lima
Q 1º ano	P2	Terça 14:30-17:30	C1 0.23	João Lima

Inscrições nos turnos das aulas práticas no secretaria 2 (esquerda) do edifício 2

Trabalho prático 2005:

English: click here
Português: clique aqui

Mirror sites:

pagina oficial na universidade: http://www.adeec.fct.ualg.pt/IC/

---

compilador pascal

o compilador DevPASCAL foi tirado deste site porque contem addware (Dialer.Gen)

Peter Stallinga, Fev. 2005

Introdução à Computação

Aula 1: apresentação 

Índice

Objectivos da cadeira
Professores
Bibliografia
Regras de avaliação
Data e local dos exames
Copianço
Programa
Recomendação para os alunos

Descrição e objectivos da cadeira

Esta é uma cadeira de introdução à computação e programação. Nas primeiras duas semanas de aulas iremos dar uma prespectiva global sobre as várias facetas do mundo da computação. Daremos as noções de hardware, software, sistema operativo, linguagens de programação, compiladores, programas de aplicação, a Internet e a sua utilização.

O resto das aulas vai incidir sobre os fundamentos de programação de computadores. Como linguagem de programação iremos usar a linguagem Pascal mas os conceitos que vão aprender aplicam-se a quase todas as linguagens de programação. No final da cadeira os alunos devem saber os fundamentos de programação e devem ser capazes de escrever programas simples. A cadeira não requer conhecimentos prévios na área de informática.

Nos apontes de aula e nas páginas na rede:

- Todos os blocos de texto em itálica são informação geral e não precissam de ser estudiados. Palavras isoladas em itálica estão escritas em Inglês. (por exemplo: *file* em vez de ficheiro)
- Qualquer texto com fonte com esta: `"fixed width"`, representa código em Pascal.

Professores

NOME	E-MAIL	HORÁRIO DE DÚVIDAS
Peter Stallinga	pjotr@ualg.pt	2ª e 6ª 9:30-11:30 Ed. 1 - sala 2.68 ou 2.78
João Lima	jlima@ualg.pt	Quarta-feira 16:30-18:30 Quinta-feira 10-12 Ed. 1 - gabinete 2.63

Os alunos devem tirar as suas dúvidas preferencialmente nas aulas. Só se a dúvida persistir é que devem então contactar os docentes no horário acima referido.

Bibliografia

- Apontamentos dados nas aulas estarão disponíveis na página web da cadeira, http://diana.uceh.ualg.pt/IC e *mirror site* http://w3.ualg.pt/~pjotr/ic
- "Programação em Pascal", Byron S. Gottfried, Schaum, McGraw Hill, ISBN 9729241589
- ("Schaum's Outline of Programming with Pascal", Byron S. Gottfried, ISBN 007023924X)
- "Pascal Programming Made Simple", P.K. McBride, ISBN: 0750632429
- "Learn Pascal" e "Learn Pascal in Three Days", Sam A Abolrous, ISBN: 155622706X e ISBN: 1556228058 resp.
- http://www.devq.net/pascal/

Regras de avaliação (2004-2005)

- Não há testes (frequências).
- As aulas práticas são obrigatórias.
- No fim do semestre os alunos devem entregar um trabalho. O enunciado do trabalho será dado em Maio.
- Os alunos podem ser chamados a qualquer altura para defender o trabalho (a afixação na porta do meu gabinete é só para comunicar as notas provisórias). Quem não consegue defender o trabalho obterá a classificação de 0 valores.
- Serão admitidos ao exame os alunos com uma nota do trabalho igual ou superior a 10 e uma assiduidade nas aulas práticas não inferior a 2/3 das aulas (9 em caso de 13 aulas, 8 em caso de 12 aulas).
- O trabalho prático conta 20%, o exame conta 80%.
- Nota mínima do trabalho prático: 10.0
- Nota mínima do exame: 9.0
- Nota final mínima: 9.5
- Os alunos repetentes já admitidos ao exame num ano passado estão dispensados de fazer o trabalho ou assistir às aulas práticas. Neste caso, o exame conta 100%. A nota do trabalho do ano passado não conta para a nota final.
- Os alunos repetentes podem optar por entregar o trabalho de este ano. A entrega significa aceitação da avaliação 20%-80%.
- Qualquer forma de fraude causa directamente a anulação de qualquer qualificação obtida e possivelmente um processo disciplinar no Conselho Pedagógico.

Data e local dos exames

a anunciar mais tarde.
(salas a anunciar oportunidamente)

Fraude

Quem copiar, deixar copiar, ou fizer qualquer outro tipo de fraude durante os momentos de avaliação, terá zero valores e leva um processo disciplinar para o Conselho Pedagógico.

Programa

Apresentação, descrição e objectivos da cadeira.
Noções introdutórias sobre computadores: tipos de computadores, componentes de um computador, características de um computador, sistema operativo, linguagens de programação, compiladores, programas de aplicação.
Noções e utilização da Internet.
Noções básicas de programação: constantes, variáveis, expressões, operadores, instrução de atribuição, instruções de input/output, funções pré-definidas.
Noções de programação estruturada: sequência, selecção, iteração.
Instruções de selecção: if, if-else, case.
Instruções de iteração: for loops, while loops, do-while loops, repeat-until loops.
Funções e procedimentos.
Vectores e matrizes: arrays de uma e duas dimensões.
(Caracteres e cadeias de caracteres.)
Records e definição de noos tipos de dados.
Noção de algoritmo. Algoritmos de ordenação simples. Algoritmo de pesquisa sequencial e de pesquisa binária.
(Apontadores. Passagem de parâmetros.)

Ficheiros de texto.

Recomendação para os alunos

A programação de computadores não é difícil. Pelo contrário, é uma tarefa relativamente fácil e divertida que envolve apenas meia dúzia de conceitos. No entanto, requer um tipo de raciocínio a que as pessoas normalmente não estão muito habituadas. Como tal, trata-se de uma tarefa que exige bastante prática e por isso recomendo que treinem bastante fora do horário das aulas. Se fizerem isso ao longo do semestre, nem sequer precisam de estudar para os testes e passarão à cadeira com boa nota quase de certeza.

Peter Stallinga. 17 fevereiro 2004

Lecture 2: Computers

What is a computer? According to the Columbia Encyclopedia it is "a device capable of performing a series of arithmetic or logical operations. A computer is distinguished from a calculating machine, such as an electronic calculator, by being able to store a computer program (so that it can repeat its operations and make logical decisions), by the number and complexity of the operations it can perform, and by its ability to process, store, and retrieve data without human intervention."

Existem vários tipos de computadores. Today, when we say 'computer' we mean a digital computer. Historically, there also existed mechanical and analog (electrical) computers.

The first mechanical computer was designed by Charles Babbage in the beginning of the 19th century (around 1815)! For this reason, Babbage is often called "The father of computing". Famous is also his Difference Engine. If you want to know more about Charles Babbage, click here.
The first electronic computer, processing data in digital format was called ENIAC just before the second word war (1939). The first commercially available computer was the UNIVAC (in 1951). At that time it was thought that a handful of computers, distributed around the world, would be sufficient to take care of all computer calculations.
Nowadays, nearly everybody in the western world has a computer at home, much more powerful than those first computers.

Computers are categorized by both size and the number of people who can use them at the same time:

Supercomputers	sophisticated machines designed to perform complex calculations at maximum speed; they are used to model very large dynamic systems, such as weather patterns. An example is the Cray SV2 (see picture), which is the size of an average living room. 
Mainframes	the largest and most powerful general-purpose systems, are designed to meet the computing needs of a large organization by serving hundreds of computer terminals at the same time. Imagine insurance companies with all their documents internally shared over a network. All employees can retrieve and edit the same data.
Minicomputers	though somewhat smaller, also are multiuser computers, intended to meet the needs of a small company by serving up to a hundred terminals.
Microcomputers	computers powered by a microprocessor, are subdivided into personal computers and workstations. **Personal computers are what most people have at home**.

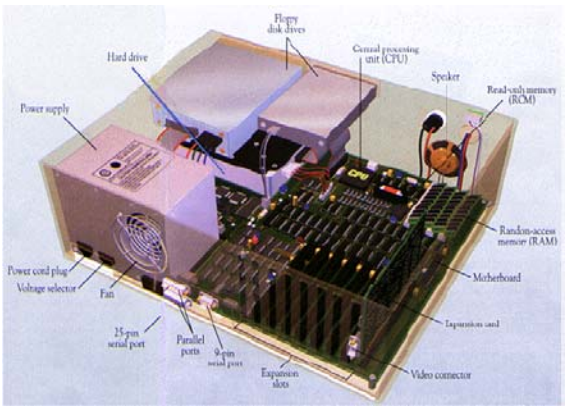
	Examples:  IBM PC and compatibles with a microprocessor like the Intel Pentium IV, 1.5 Ghz. Portable computers are a portable variant of microcomputers   Apple Macintosh encorporating the Motorola processor family.
Processors	Many home appliances, like washing machines and ovens, contain a small processor that is controlling the machine. These are very small computers that have been programmed in the factory in hardware and cannot be programmed by the user. As such, they can probably not be considered computers, but are still important to mention. Some more-advanced home appliances, like satellite receivers or home-cinema equipment, are running quite sophisticated programs which follow the rules which will be presented in this lecture.

Note: with the speed the technology is advancing we can say that "the supercomputers of today are the (personal) microcomputers of tomorrow"

Hardware vs. software

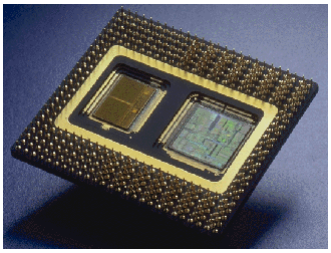
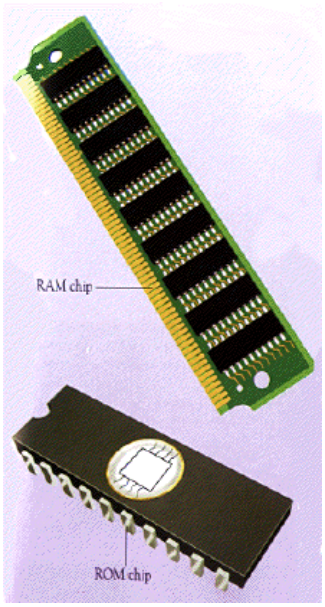
Two important things to distinguish are hardware and software. Hardware is everything that you can feel and touch. Sofware is the programs that are running on the hardware. Example are given below.

Every working computer consists at least of the following:
1) A processor
2) Memory to store the program
3) An output device
4) A program running
Most computers also have
5) An input device, to either change the program running, enter new data to be processed, or control the processes running.



Let's take a look of things we can find in a computer system:

Table: hardware elements

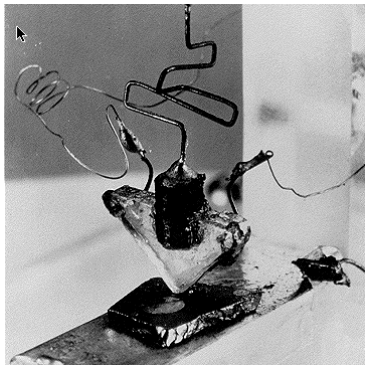
Mouse		input device	to control the processes of the computer
CPU		processor	Central Processing Unit is what is doing all the work. Calculating, controlling data flow, etc.
Joystick		input device	input data to a game
Keyboard		input device	to give instructions to the computer or enter data to be processed
Memory		storage	store program and data to be processed
Monitor		output device	show results of processes
Printer		output device	show results of processes
Modem		input/output	MOdulator-DEModulator communicate with other computers over a telephone line

Network card		input/output	communicate with other computers over a high bandwidth network
Harddisk/floppy disk		storage	store data or programs in non-volatile format (data will stay when the computer is switched off)
CD-ROM		input	load programs or data into the computer memory
Sound card		output	play music or other sounds
Scanner		input	scan an image

Physical and logical layers of a computer

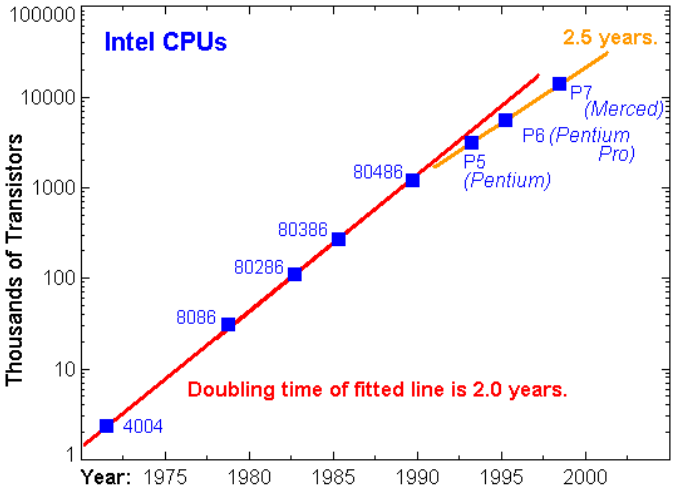
At the lowest level we can determine the level of **Physics**. Electrons (and holes) are responsible for the electrical conduction of the material. The materials used in computers is called semiconductors, meaning that they have a resistance in-between metals (like copper and gold) and isolators (like glass and plastic).

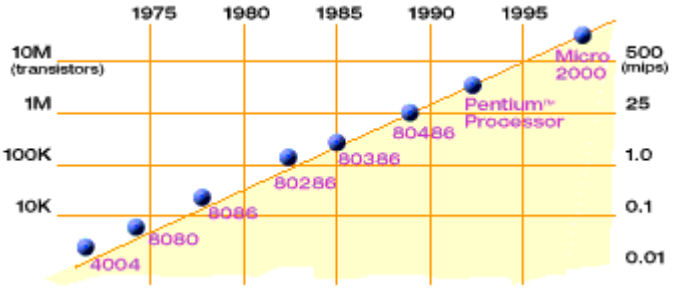
At the next level we find **Electronics**. Electronics connect materials with different properties and the resulting electronic components show remarkable and usefull behavior. Note the diode which conducts current only in one direction, or the transistor, whose conductivity is controlled by an external voltage.

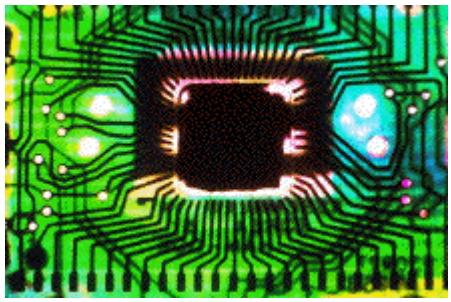


The picture shows the first transistor, as invented at Bell Labs in 1947.

Next we find the level of **Digital Electronics**. This involves so-called gates: OR, NOR, AND, NAND, XOR, to name a few. Such gates are made up of basic electronic components like transistors. Gates are the cornerstone of digital computers. We have to remember that all calculations of computers at the end are done on this level. When we add 2+2, somewhere in the computer gates are switching and are performing calculation like "1 OR 1 = 1".
Another important component of digital electronics is memory. These are also made up of transistors (and capacitors in case of dynamic RAM) and can temporarily store information.







The next level is that of **Integrated Circuits**. In these circuits, millions and millions of gates are connected and this allows for complex programs to run.
In the years since the first integrated circuits, the number of transistors on a single IC has been doubling every two years, approximately. This is calles Moore's law and is still avlid, although the physical limits seem to be in sight. See the picture on the left.

To make a popular comparison. If the same industrial advance had been made in the car industry, a modern car would be able to run at 5 million km/h, consume 1 drop of petrol per 100.000 km and could seat about 10 thousand people

At the next level the real programming starts. We start with binary programs, or (in readable format) **Micro-assembler**. This is directly programming the processor: put address xxx in address register, enable addressing lines, wait xx ns, add register X to register Y, etc. ("registers" are small memory units *inside* the processor)

The next level consists of **Machine language**. This is directly programming the processor with binary code like
1010001000000101000
which might mean: put the contents of address 0000101000 into the A register. Such code is nearly impossible for humans to read. Therefore, very rapidly macro-assembler were invented. When a program is executed, this code is (without further translation) directly put in the memory and executed. Files with extensions like ".exe" and ".com" in MS-DOS and Windows are of this type.

For the next level, which already starts looking like real programming, we have **Macro-assembler**. In this level we instruct the processor to execute the small programs written in microassembler. These are equal to machine language, but in a more readable format and with the first hints of higher level programming (for instance "labels" and "variables"). In macro-assembler we write commands like
ADDA $2050
which might mean "add the contents of memory at address $2050 to register A" and which is translated

by the assembler into machine code.

At the next level we finally have our modern **Programming languages**. These languages are often called "fourth generation languages", because they evolved from earlier languages such as assembler, etc. Many of these programming languages were invented during the 1960s and and 1970s. For every application there exists a programming language. In 1995 there existed about 2500 different programming languages. (for people interested, see http://cui_www.unige.ch/langlist).
For professional programmers there is C and C++, for simple applications, there is BASIC. For educational purposes PASCAL was invented, especially to teach the ideas of programming.
Examples:

BASIC	`IF A=20 THEN PRINT"Hello World"`
PASCAL	`if (a=20) then writeln('Hello World');`
C	`if (a==20) printf("Hello world\n");`
FORTRAN	`IF (A .EQ. 20) PRINT ,'Hello World'`

Lately new generations of programming languages are evolving. All of them involve the concept of **Object-oriented programming**, a concept that we will not use in our lectures, but has become indispensible in modern program environments. We might call this "fifth generation languages"

Modern programming languages are very flexible. We can write a variety of applications in the languages. We might, for instance, write a program that simulates the workings of a diode, or calculate a transistor at the physical level. Then we are full-circle; we will use the computer to make the basic components - and thus the computer - better and faster.
Also, bear in mind, that if we write in Pascal

```
writeln('Hello world');
```

and run the program, we are, in fact, controlling the flow of electrons on the lowest levels. This might give you a good feeling of control .....

Compilers

As said before, modern programming langauges have to be translated from aform that the user understands to a form that the computer understands. When we write

```
writeln('Benfica - Sporting   3 - 0');
```

This has to be translated into

```
MOVAI $0102      ; load 'B' into A register
MOVAO $1245      ; move contents of register to Video Card
```

or, one level deeper

```
00110111000111111100011110100011111
```

For this purpose exist compilers. They translate text that is readible to humans into something that is executable for the computer. When we start with a file containing our program `myprogram.pas`, we translate this with a compiler which will generate a file called

```
myprogram.exe
```

which we can call with

```
myprogram
```

and the following output will appear on the screen when everything goes correct

```
Benfica - Sporting   3 - 0
```

In most modern version of a programming language we will work within a so-called IDE (integarted development environment), which means that we can write the program and with a single keypress we can compile the program, see the errors of our writing, and, in case there were no errors during compilation, execute the program and see the results. Such environments greatly speed up the

development of software, but we should not forget that in fact a compiler is translating the program for us.

We will discuss compilers later and the use of compilers will also be explained in the practical lessons.

Operating Systems

Operating systems are programs that are constantly running on our computer and are interpreting the commands we give it. For instance, when we want to 'run' our program, we can write its name or click on its icon, or something like that. The operating system will then

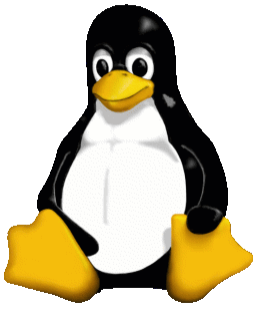
1) load our program from harddisk into memory
2) start executing it

When our program is finished it is normally removed from the memory again, but the operating system will continue running, waiting for our next instruction. In fact, a computer left alone is doing nothing but checking if we already typed something or clicked on something. What a waste of energy .....

The most famous operating system was probably MS-DOS by Microsoft. This was a command-line operating system, meaning that you had to type-in your instructions to the computer using a keyboard. For example

```
DIR C:\
```

Later, a graphical interface was added to MS-DOS and it was called Windows. Underlying it, was still the same MS-DOS command line operating system, but our mouse-clicks on icons and objects was translated into commands. We might click on a 'folder' and see its contents. Clicking on a folder would then be equal to typing DIR, with the output presented in graphical format.
Over the years, Windows has become more advanced and nowadays it is a multitasking operating system (meaning that more than one program can run at the same time) and most people in the world are using it. Because of the monopoly that Microsoft has, the 27% of the shares that co-founder Bill Gates has in this company had a value of $20 billion (20.000.000.000 dolars) in 1995. By 2000 it had risen to $65 billion. Note, this is about $10 for every person on earth, be it an American or a Chinese in whatever remote village in China.
Alternatives to Windows are Unix/Linux, which has the advantage that it is for free and MacOS, which



runs on Macintosh computer, as described above.

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this **NOT** the form the final test takes!

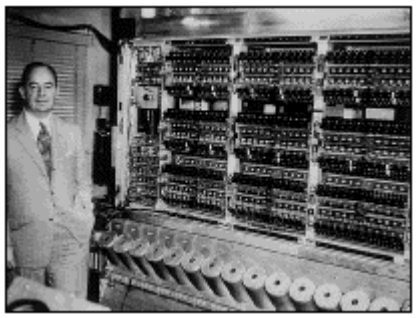
Peter Stallinga, Universidade do Algarve, 6 fevereiro 2002

Lecture 3: Units of Information / Memory

As described in the previous lesson, the memory is an essential part of the computer. It stores

- the program
- the data the program is working with

Note: The idea to seperate the program form the data it is working on and to seperate the hardware from the software (or the "the machine" and the "program") comes from Von Neumann. He designed the first electronic computer capable of running a flexible program (1940-1952). All modern computers are Von Neumann computers.
Click here if you want to know more about Von Neumann.



Before writing programs, it is useful to take a closer look at the memory.

Memory is filled with **information**. This information can either be program code or data. Let's take a look at some types of information.

BIT

The smallest unit of information is a **bit**. One bit can contain inforamtion of the type "TRUE or FALSE". For example, it can contain the information
"Did the student pay his tuition fees (propinas), yes or no?",
"Can the student go to the frequencia, yes or no?",
"Is *x* larger than *y*?".
We have to remember that, at the electronics level, the computer is calculating with these bits of information. In the previous lecture we have seen how digital electronic components ("AND gates", etc.) handle these bits of information. For these electronic components, there are two levels: 0 V and +5 V (or any pair of discrete voltage levels). In our language, we can call it 'TRUE' and 'FALSE', or '1' and '0', or 'green' and 'red', or whatever pair of symbolic names we want to give it.
Because a bit of information can be only one of two possible values, we call it a **binary** unit. Since all units of inforamtion are derived from these bits, we call a computer a binary calculator. Although computers can easily be constructed using other basic untis of information (for instance ternary or quaternary), all modern computers are of the type binary calculators.

electronic levels of AND gates	0 V	+5 V
binary	0	1
logical	FALSE	TRUE
bicolor	green	red

These ideas can be mixed at our wish. For example, PASCAL uses the ideas 'TRUE' and 'FALSE' for

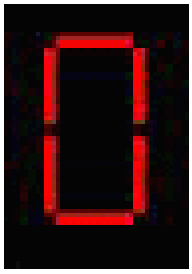
logical calculations, while C uses '1' and '0'.

NIBBLE

The next unit of information is a **nibble**. This is a set of 4 bits. In these 4 bits we can for instance store information of the type 0..9. Nibbles are therefore used in many digital displays, such as alarm clocks etc, where each digit is stored in a nibble. We call this binary-coded-decimal (**BCD**):

binary	*BCD*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

binary	*BCD*
1000	8
1001	9
1010	not used
1011	not used
1100	not used
1101	not used
1110	not used
1111	not used



example of an LED display

Looking at the table we see that when the binary code is

abcd

the decimal code is

$a*8 + b*4 + c*2 + d$

or, more general:

$a*2^3 + b*2^2 + c*2 + d$

This, we will see, is always the relation between binary and decimal numbers.

Note also that some of the possible combinations of bits are not used in BCD. A way to represent these four bits of information with all possbile combinations used is the **hexadecimal** system. The bit combinations from '1010' to '1111' are then represented by the letters A to F. In a hexadecimal representation we get the following translation table:

binary	*hexa-decimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

binary	*hexa-decimal*
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

and we see that all 16 binary combinations have a counterpart in the hexadecimal system The hexadecimal system is widely used in computer technology.

As an example: 21F in the hexadecimal system is equal to $2*16^2 + 1*16^1 + 15*16^0 = 2*256 + 1*16 +$

15*1 = 543.

BYTE

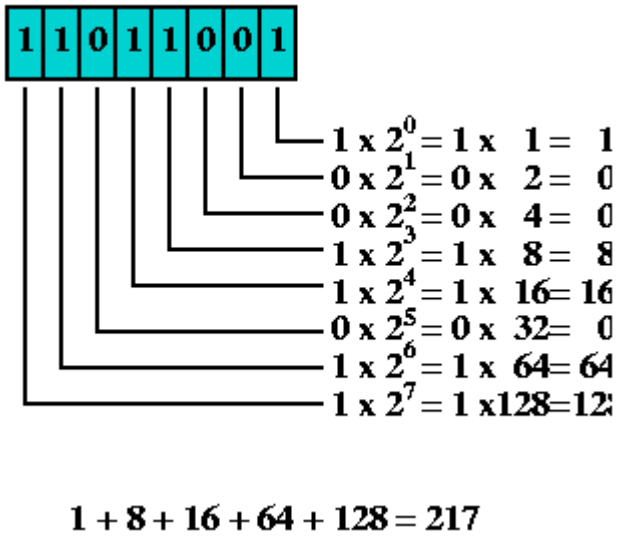
The next unit of information is a byte. A byte is a combination of two nibbles and thus of 8 bits. In this we can store numbers from 0..255 because

$00000000 = 0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = 0$

$11111111 = 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 =$
$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =$
$= 255$

another example:

$11011001 = 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 =$
$= 128 + 64 + 0 + 16 + 8 + 0 + 0 + 1 =$
$= 217$



Alternatively, a byte can represent all the letters of the alphabet, in uppercase ('A' .. 'Z') and lowercase ('a' .. 'z'), plus all the digits '0' .. '9', some special letters like '{', '}', '(', ')', space, etc, and other things like control codes. The most often used way to do this is **ASCII** (American Standard Code of Information Interchange) in which, for example 'A' is 65 (decimal), or 01000001 (binary), or 41 (hexadecimal). Other examples are:

binary	*decimal*	*hexadecimal*	*ASCII*
01000001	65	41	'A'
01000010	66	42	'B'
01100001	97	61	'a'
00010000	32	20	' '

Memory

In many computers, the byte is the smallest unit that can be 'addressed'. To understand this idea, we have to look at how the memory is organized. Imagine the memory as a (very long) street with houses. Each house has an address. If we want to put something in a house or take something out of it, we have to specify the address of the house.
In a computer, the memory takes the place of the street and the byte takes the place of the house. In each 'house' live 8 bits, or a byte.

As we will se later, some 'people' (units) are very big and occupy two or even more adjacent houses. These 'people' are called 'integers', 'words', 'reals', etc. Still, addresses are in most computers 1 byte and thus 8 bits apart.
(*Note [not needed to study]: especially in supercomputers the distance between two addresses can be much longer than a byte, for example 63 bits instead of 8. We call this distance the 'word length' of a computer.*)

Note that all three units described until now are also units of food in English. Bit, Nibble, Byte. Further units disband this idea.

Memory Size of Computers

Very often you read in the advertisement things like:
Computer, with 1.7 GHz processor, 256 MB RAM, 40 GB harddisk, 1.4 MB floppy

The 1.7 GHz specifies the speed of the CPU (the central processing unit). 1.7 GHz thus means that it can do 1.7 million simple instructions per second.
(Since most commands given to the processor take more than one simple instruction, the actual number of commands per second is lower. For example an addition of two floating point number might consist of 1) loading the first number from memory, 2) loading the second number from memory, 3) adding the number (possibly in several steps), 4) saving the result to memory. Nevertheless, the overall speed of a computer is largely determined by the speed of the processor and the speed with which it can load the data from memory.)

The other numbers determine the size of the memory of the computer (RAM = Random-access memory), the harddisk and the floppy disk repectively, in number of bytes (B). To give an idea what these numbers mean, let's analyze them more carefully:

BYTE: The basic unit to describe memory is the byte (B). As said before, a byte is large enough to contain a single letter, or a number from 0 .. 255.

KILOBYTE: 1024 bytes is called a kilobyte (kB). In science, 'kilo' means exactly 1000, a nice round number in the decimal system. For computers 'kilo' means a little more, 1024. This is based on the fact that 1024 is exactly 2^{10}, or in binary 10000000000, a nice round number.
To give an idea of how much a kilobyte is: A page of A4 with text is approximately 4 kB.

MEGABYTE: 1024 kilobytes is a megabyte (MB). This is then equal to 1024 x 1024 bytes, or 1048476 bytes.
To give an idea how much a megabyte is: 250 pages of text, or, let's say a book.
Most floppies are 1.4 MB and this is thus enough to store a book of about 350 pages.
(without images etc.)

GIGABYTE: 1024 megabytes is a gigabyte (GB). This is enough to store a nice library of a thousand books.
Most CD-ROMs have 650 MB (0.65 GB), enough to store a small library

Modern harddisks have about 40 GB space. On this we can therefore store a large library, some 40.000 books.

TERABYTE: 1024 gigabytes is a terabyte. Although disks of this size do not exist yet, some companies have computer systems with many disks totalling disk space in the order of terabytes. This is enough to store all the books in the world.

To give an idea about the total amount of diskspace on the world: there exist about 500 million computer owners in the world. On the average, each has a hardisk of about 10 GB. That makes a total of approximately 5.000.000.000.000.000.000 bytes. It would take a person reading 10 books per day more than a billion years to complete reading all the information!

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this **NOT** the form the final test takes!

Peter Stallinga, Universidade do Algarve, 13 fevereiro 2002

Lecture 4: Introduction to PASCAL

PASCAL

PASCAL is a french acronym for "Program Appliqué à la Selection et la Compilation Automatique de la Literature" a high-level ("fourth generation") computer-programming language. Designed by Niklaus Wirth in the 1960s as an aid to teaching programming. It is still widely used as such in universities, and as a good general-purpose programming language. Most professional programmers, however, now use C or C++. Pascal was named after the 17th century French mathematician Blaise Pascal.

Blaise Pascal (1623-1662)

French philosopher and mathematician. He contributed to the development of hydraulics,
the calculus, and the mathematical theory of probability.
His most famous invention was probably the "Pascal Triangle":
Each number is the sum of the two numbers immediately above it, left and right, like this:

```
      1
    1   1
  1   2   1
1   3   3   1
1   4   6   4   1
```

Start

A program is a sequence of **instructions**, or statements which inform the computer of a specific task we want it to do.
Most modern program languages are in a very readible format, close to English, making it easy for humans to read and write programs. This in contrast to earlier programming languages, which were closer to things the computer understand. See for example the assembler language (aula 2). PASCAL is the language which most resembles a natural human language and as such is best suited for explaining the art of programming.

Every PASCAL program has the same essential format, (called the template):

```
PROGRAM Title;
begin
  program_statement_1;
  program_statement_2;
          |
  program_statement_n;
end.
```

Let's take a look at this program.

- Every program starts with the word **PROGRAM** followed by the title of the program.
- **begin** defines the starting point of the program.
- **end.** defines the end of the file and thus the end of the program. Note the full stop "." at the end. This is only used for the "end" statement at the end of the program. All other statements end with a semicolon ";"
- The combination of "begin" and "end" is a way to group instructions so they form a block. All the statements between "begin" and "end" are treated by the compiler as one statement.
- In between "begin" and "end" we can put our instructions. These instructions can be either contain instructions that PASCAL already knows, or instructions that we are going to define (so called procedures or functions, which we will dsicuss later).
- PASCAL is not case sensisitive: "Program", "PROGRAM", "program", etc, are all the same.

Identifiers

Identifiers, as the name already says, are used for identifying things. This can be name of **programs** (as above), name of **procedures** and **functions** and names of **variables** and **constants**. This we will see in later aulas. Like in most languages, names of identifiers have some restrictions:

- They should start with a letter; "PROGRAM 20hours" is not allowed.
- Followed by any combination of letters, digits or the underscore character "_".
- Spaces are not allowed, nor are characters like "(", "{", "[", "%", "#", "?", etc, except "_". The reason why this is so is that these characters are used for other things in PASCAL. They are called **reserved characters**.

```
{ } [ ] ( ) - = + / ? < > . , ; : ' " ! @ # $ % ^ & * ~ ` \ |
```

- Identifiers cannot be equal to **reserved keywords** of PASCAL, such as word like "program", "integer", "begin". Note that identifiers like "program1", or "program_" are allowed, although it is advised to avoid such confusing names. Note: In many programming environments (like the Turbo Pascal 7 that we will use in the practical lectures), we will notice when we are using a reserved keyword because they will change color when we type them in.
- Choose your identifiers well. When a program is calculating interest rates, call it, for example "PROGRAM InterestRates" and not "PROGRAM program1". Although it is not an error to give a program the name "program1", it is much more intelligent to give it a more meaningful name. This helps other people to understand your program (or yourself when you come back to the program after a long time).
- The minimum length of identifiers is 1, the maximum length 255. Make use of this possibility of long names, but also remember that also long names can make the program unreadable. Choose a "golden middle". Which of the following do you think is best:

```
  r := r + a;
  money := money + interest;
  themoneyintheaccountofpersonwithnameJohnson :=
themoneyintheaccountofpersonwithnameJohnson +
thecurrentinterestrateatthetimeofthiswriting;
```

- They are not case sensitive, "i" is equal to "I", etc. To make programs more readable, follow a convention all through your program(s). The most often used convention is lowercase for variables and UPPERCASE for CONSTANTS.

Structured programming

The most important thing in programming is to write clear, logical and structured programs.

- Use meaningful names for variables, procedures and functions.
- Use indentation. Compare the following two programs:

```
PROGRAM Myprogram;begin writeln("Hello world!");end.
```

and

```
PROGRAM MyProgram;

begin
  writeln("Hello world!");
end.
```

Both programs do exactly the same, but the second one is much more readible. The difference is

  - Only put one statement per line.
  - Use indent. Put (2) extra spaces in the beginning of the line every time we are one level "deeper" in the structures.
  - Seperate blocks of text (functions and procedures) with blank lines.
- Avoid the use of "goto label" statements. With these statements, the program rapidly starts looking like spaghetti. Whereas in BASIC (Beginner's All-purpose Symbolic Instruction Code) the use of the GOTO statement is nearly unavoidable, in any itself-respecting language, the goto statement should be avoided.
- Comment. Since PASCAL is nearly like English, the program itself should be self-explanatory. Still, in places where the idea of the program might not be clear to the programmer, use comments. In PASCAL comments are placed within a set of accolades: "{" and "}". The compiler stops interpreting the text after the first "{" and resumes compiling after it encounters the matching "}". Therfore, any text can be placed within this context.

  Alternatively, in Turbo Pascal, comments can be placed within a combination of "(*" and "*)". Note that in that case even the accolades "{" and "}" are skipped by the compiler.
- Use procedures and functions wherever it makes the text more organized. If at many different places the program has to do basically the same thing (for instance reading a line of text form a file), consider putting it in a procedure or function (for example PROCEDURE FileReadLn;). This will make the program more readable, more efficient and shorter.

Reserved keywords in Turbo Pascal

The following words cannot be used for identifiers. Most of these keyword are explained in the lectures in the chapters described by the subject in the second column.

keyword	*subject*
and	boolean algebra
asm	
array	arrays
begin	introduction
case	if .. then
const	variables and constants

keyword	*subject*
goto	
if	if .. then
implementation	
in	
inherited	
inline	
interface	

keyword	*subject*
program	introduction
record	records
repeat	loops
set	
shl	
shr	
string	variables

constructor	
destructor	
div	algebra
do	loops
downto	loops
else	if .. then
end	introduction
exports	
file	
for	loops
function	procedures and functions

label	
library	
mod	algebra
nil	pointers
not	boolean algebra
object	
of	if .. then
or	boolean algebra
packed	
procedure	procedures and fucntions

then	if .. then
to	loops
type	variables
unit	
until	loops
uses	
var	variables
while	loops
with	records
xor	boolean algebra

The following words are related to variables and constants. Use of these words for identifiers is disadvised:

boolean
byte
char
double
integer
real
string
text
word

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this **NOT** the form the final test takes!

Peter Stallinga. Universidade do Algarve, 17 fevereiro 2002

Lecture 5: Variables

`Write` and `Writeln`

The instructions `write` and `writeln` are used for showing information on the screen: text, numbers, values of variables, etc.. They are very often used; nearly every program has somewhere a write or writeln instruction. As an example, consider the following program

```
PROGRAM WritelnExample;

begin
  writeln('Today is a very nice day');
end.
```

when we compile and execute the program the screen will show

```
Today is a very nice day
_
```

(note, the underscore character _ represents the position of the cursor).
The `write` and `writeln` instructions are not only used for text, as in the example above, but also to show numbers, or other types of information. Later we will learn how to control the format of the output (the way the information is shown). For the moment, let's look at the brother of `writeln`, namely `write`: The difference between `writeln` and `write` is that `writeln` puts the cursor on a newline, so that the next output (the time time we use write or writeln) will appear below the first text. With `write`, the cursor stays directly after the text:

```
PROGRAM WriteExample;

begin
  write('Today is a very nice day');
end.
```

when we compile and execute the program the screen will show

```
Today is a very nice day_
```

When we want to print more things at the same time, we can do this with the same write instruction

```
PROGRAM WriteMultiple;

begin
  write('Today is ', x, ' a very nice day');
end.
```

which might show (when `x` is equal to 34)

```
Today is 34 a very nice day_
```

Compare the following programs:

program

```
PROGRAM WriteLnsExample;

begin
  writeln('Today is a ');
  writeln('very nice day');
end.
```

output

```
Today is a
very nice day
_
```

program

```
PROGRAM WritesExample;

begin
  write('Today is a ');
  write('very nice day');
end.
```

output

```
Today is a very nice day_
```

program

```
PROGRAM WritesExample;

begin
  writeln('Today is a ');
  writeln;
  write('very nice day');
end.
```

output

```
Today is a

very nice day_
```

program

```
PROGRAM WritesMulti;

begin
  writeln('Today ','is',' a ');
  write('very nice day');
end.
```

output

```
Today is a
very nice day_
```

Types of Variables

Variables store values and information. They allow programs to perform calculations and store the results for later use. Imagine a variable as a box that can contain information. When we want to know this information, we open the box and read the value. At the end, we put the information back in the box and leave it there, until we need the information again.

For ease of identification, to avoid confusion, every box must have a name, an identifier (see aula 4).

The type of the box defines the size of the box and the type of information that can be found in there. Variables come in many sizes and flavours.

Variables can store numbers, names, texts, etc. In modern versions of PASCAL there are many types of basic variables. The most important are

group	*variable type*	*range*	*size it occupies in memory*
I	boolean	TRUE / FALSE	1 bit
II	byte	0.. 255	8 bit = 1 byte
II	integer	–32768 .. 32767	16 bit = 2 bytes
II	word	0 .. 65535	16 bit = 2 bytes
II	longint	–2147483648 .. 2147483647	32 bit = 4 bytes
III	real	–2.9E–39 .. 1.7E38	48 bit = 6 bytes
III	double	–5.0E–324 .. 1.7E308	64 bit = 8 bytes
III	extended	–3.4E–4932 .. 1.1E4932	80 bit = 10 bytes
IV	char	#0 .. #255	8 bit = 1 byte
IV	string	text	256 bytes

Note the convention of writing exponents in the scientific notation: 2.9E–39 means 2.9×10^{-39}, 1.7E308 means 1.7×10^{308}, etc.

I Boolean is used to store and manipulate information of the type true-or-false. As we have seen in lecture 3, this is one bit of information.

II Byte, **integer**, **word**, and **longint** all store values of complete numbers. This is used for things that can be counted; number of people in the room, number of doors of a car, number of phonecalls somebody made, ano lectivo, day of the month, etc. **Byte** and **word** only store positive numbers, while integer and longint can store both positive and negative numbers, at the cost of limited maximum value. The **byte** occupies the least memory, only 8 bits, but the range of values it can take is therefore very limited, only 0 .. 255. If we need to store larger positive complete numbers, we have to use **word**. If we want to store large numbers and want positive and negative numbers we use **longint**.

Note: since a **byte** has 8 bits, it can store $2^8 = 256$ different numbers: 0 .. 255. The same calculation we can make for the 16-bit units **word** and **integer**: $2^{16} = 65536$, numbers from 0 .. 65535 for **word**, and -32768 .. 32767 for **integer**. Remember the calculations of lecture 3.

III Real, **double** and **extended** are examples of variables that can store floating point numbers (for

example 3.1415926535). These are used for things that are not countable, like the length of the car, the time elapsed between events, the height of a building, the square-root of 3, etc. The smallest is **real**, it occupies only 6 bytes, at the cost of smaller precision in our calculations. The best is **extended**, with 80 bits (10 bytes), the calculations will have very high precision, but the calculations will be slower and it occupies more space in memory. A good middle way is the **double**.

IV The last two types are used for storing text. **Char** is used for a single character, while **string** can store up to 255 characters.

Later we will learn how we can define our own type of variables, now let's take a look at how we use them in PASCAL

Variables: Declaration with VAR

In most modern compiled languages, all variables that we will use have to be defined first. This is called **declaration**. To be more precise, declaration means reserving space in memory and associating a name to it, so that later we can use the name instead of the memory address when we want to retrieve the information.
In PASCAL we declare variables with the instruction **VAR**, followed by the variable(s) with type that we want to use. The place to do that is before the begin of the program, but after the declaration of the name of the program. For example:

```
PROGRAM VarExample;

VAR i: integer;
    a: real;
    b: word;

begin
  writeln('Today is day ',i);
end.
```

If we want to define more variables of the same type, we can do that with several instructions or do it on a single line. Note that in any case, we don't have to write VAR again, although it is not prohibited:

```
PROGRAM VarsExample;

VAR i, j, k: integer;
    a1, a2: real;
VAR b1: word;
    b2: word;
    b3: word;

begin
  writeln('Today is day ',i);
end.
```

Variables: Initialization

The VAR instruction does not assign a value to the variable, it only reserves space for it in memory! In the last example program above, the output might have been

```
Today is day 23741
```

When a computer is switched on, the memory is normally filled with 0's, but after a while, after many programs have been using the memory and left there their garbage, the contents of a memory address is unpredictable. To ensure that we are working with well defined values we always must assign a value to each variable. In the next lecture we will learn how we can do that with assignment instructions in the program. Here it suffices to say that: "don't assume that your variables are set to 0 in the beginning".
Note: In many programming languages it is possible to assign a value to a variable at the moment of declaration. Also in PASCAL it is possible (via variable constants), but this will add to the confusion and we better avoid doing that.

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this **NOT** the form the final test takes!

Peter Stallinga. Universidade do Algarve, 23 fevereiro 2002

Lecture 6: Assignment and Constants

Formatted `Write` and `Writeln`

Using write and writeln without specifying the format will show the values in standard form, which is the scientific notation for reals and 'as many places as is needed' for integers (inteiros):

```
PROGRAM WritelnUnformatted;

Var r: real;
    i: integer;

begin
  writeln('Real r = ', r);
  writeln('Integer i = ', i, '!!');
end.
```

output:

```
Real r = 3.00000E-2
Integer i = 1!!
```

To change the way the values are presented, we can specify the width of the text, and, for variables of floating point type (real, double, extended), the number of cases after the floating point. Note the following example:

```
PROGRAM WritelnFormatted;

Var r: real;
    i: integer;

begin
  writeln('Real r = ', r:8:6);
  writeln('Integer i = ', i:5, '!!');
end.
```

output:

```
Real r = 0.030000
Integer i =     1!!
```

The real `r` is shown with a total of 8 places reserved, and 6 digits after the floating point. The integer `i` is written with 5 places of text. Since in this case only one digit was enough, the rest, 4 places, is filled with empty spaces.

Assignment

In the previous lecture (aula 5) we have seen how we can define variables. Now we will take a look at how the value of a variable can be assigned a value

Giving a new value to a variable is called **assignment**. In PASCAL this is done with the operator

On the left side op this operator we put the variable, on the right side we put the new value or expression that will produce a value **of the same type** as the variable. Examples:

```
a := 3.0;
b := 3.0*a;
c := Cos(t);
```

wrong (assuming `a` is of type real):

```
1.0 := a;
a := TRUE;
a := 1;
```

In reality, for PASCAL the last example was correct. PASCAL knows what we want and helps us by converting the integer 1 to the real 1.0. In other languages (C, FORTRAN) this is not so. In any case, it is bad practice to mix reals with integers and we'd better write: `a := 1.0;`

The symbol `:=` is pronounced as 'will be', or 'becomes'. This to distinguish it from the normal mathematical symbol =. At this stage, it is interesting to make a comparison between the mathematical symbol = and the assigment symbol in programming languages (:= in PASCAL). As an example take the following mathematical equation

$a = a + 1$

This, as we all know, does not have a solution for a. (In comparison, another example: $a = a^2 - 2$, has a solution, namely $a = 2$).
In programming languages, however, we should read the equation differently:

```
a := a + 1;
```

means

(the new value of a) (will be) (the old value of a) (plus 1)

Or, in other words: first the value of `a` is loaded from memory, then 1 is added to it, and finally the result is put back in memory. This is fundamentally different than the mathematical equation. Exactly for this reason, in PASCAL the symbol `:=` was invented for assignment. To avoid confusion. In most other modern languages the symbol `=` is used, which is confusing, especially for the beginning programmer. That is why the student is advised to pronounce the assigment symbol as 'becomes' or 'will be' instead of 'is' or 'equals'.

To summarize, the **:=** symbol is

NOT part of a comparison ("a is equal to b?")
NOT part of an equation ("$a^2 = 2a - 1$")
it IS an assignment ("a becomes b")

Later, we will learn how to make comparisons (in the lecture on "if .. then" and "boolean algebra"), for which we will use the = symbol.

Constants

Constants differ from variables in that they cannot change their value during the running of the program. Once assigned a value, it cannot be changed. The advantage of this is that we can define a value at one place in the program and from that moment on use it wherever we need it. Consider the example of π. It is much easier to once define this value than to every time have to type 3.1415926535 when we need it's value. The program becomes more readable in this way.
The definition of constants we do at the same place as the variables, but with the word CONST, so for example:

```
PROGRAM ConstExample;

VAR angle: real;
    tan: real;
CONST PI = 3.1415926;

begin
  angle := 45.0;
  angle := PI*angle/180.0;
  tan := Sin(angle)/Cos(angle);
end.
```

Note the way the constant is written, namely in UPPERCASE. Remember that PASCAL is not case sensitive and we can mix uppercase and lowercase whenever we want. Still, the use of uppercase for constants is a useful convention which the student is advised to follow. Any programmer reading our program can rapidly see that he is dealing with a consatnt instead of a variable.

Interesting technical detail: To be more precise, for constants no space in memory is reserved like it is for variables. Instead, the compiler, everytime it encounters the name of our constant in our text, will substitute it's value. We do not have to worry about this detail, though. For all good purposes, we can assume that a constant is a variable that cannot change value.

Consider the following example:

```
PROGRAM InterestRateExample;

VAR money: real;
CONST TAXA = 4.3;

begin
  money := 99.0;
  money := money*(1.0+TAXA/100.0);
end.
```

In case the interest rates change, we only have to change our program in one place (probably somehwere in the beginning). This avoids errors and inconsistencies in the changed program. If we forget to change it in one place the program will produce erroneous output, and now imagine a program with thousands of lines of PASCAL code. It is easy to miss one instance of our interest rate.

Example

The following shows an example of a program using assignment statements and the use of variables and constants. The right side shows the value of the variables after each line

	value of `a` after executing the line	value of `b` after executing the line

`PROGRAM Assign;`
 	
`VAR a: real;`
`b: real;`
`CONST C = 4.3;`
 	
`begin`
`a := 1.0;`
`b := C;`
`a := a + b + C;`
`end.`
	`undefined`
	`1.0`
	`1.0`
	`9.6`
		`undefined`
		`undefined`
		`4.3`
		`4.3`

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this is **NOT** the form the final test takes!

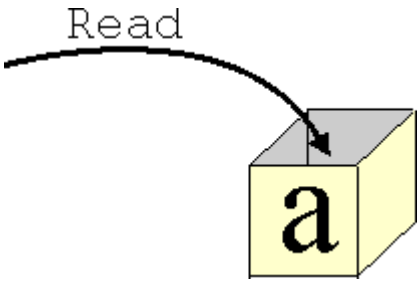
Peter Stallinga. Universidade do Algarve, 23 fevereiro 2002

Lecture 7: Math

Read and ReadLn

In the previous lectures (see aula 5 and aula 6) we have learned how to show the results of our calculations on the screen. With `write` and `writeln` we can make our program have *output*. In many cases, we would also like our program to have *input*. The user enters his name, or enters numbers that our program has to process. Or even more simple, we want that the users can control the program. For example, we want that the user can stop the program with a simple stroke of the escape key.

PASCAL: With `Read` and `ReadLn` we can get information from the keyboard. They can read values and characters from the keyboard and directly store them into specified variables. The general form the instructions take is




```
Read(variable_list);

ReadLn(variable_list);
```

Here `variable_list` is a list of variables to be read, seperated by commas, for example

```
Read(a, b, i);
```

Note that these variables do not have to be of the same type. In the above example they could be of type real, boolean and integer respectively. As an example

```
PROGRAM ReadExample;

VAR n1, n2: integer;

begin
  WriteLn('Please enter two numbers separated by a space');
  Read(n1);
  Read(n2);
  WriteLn('n1 = ', n1 ,'    n2 = ', n2);
end.
```

When run, the program will display the message

```
Please, enter two numbers separated by a space
```

The user is expected to type the two numbers

```
128 31<return>
```

(note the conventions we will use in this text: things the user will enter will appear in *green italics*, and `<return>` signifies pressing the return key).
after which the program shows

```
n1 = 128  n2 = 31
```

The difference between `Read` and `ReadLn` is that the `ReadLn` statement discards all other values on the same line, while `Read` doesn't. In the above example, if we had replaced the `Read` instructions with `ReadLn` instructions, the program would have assigned 128 to `n1`, then discard the rest of the text the user typed and consequently wait for more input to assign to `n2`.

Math

After learning how to assign values to variables in lecture 6, we can now take a look how to perform calculations with our programs. The most basic operations are adding, subtracting dividing and multiplying:

operator	operation
+	addition
–	subtraction
*	multiplication
/	division

These four *operators*, when used for calulations, need two *operands*. In PASCAL one is placed on the left side and one on the right side. As examples:

correct	*wrong*
`3 * a`	`* a`
`a + b`	`3 a +`

These expressions will result in values that can be assigned to variables as we have seen in lecture 6. As an example:

```
c := 3 * a;
```

Note again, on the left side of the assignment symbol := we have a variable and on the left side we put our expression resulting in a new value for the variable.

Integer Math

The operators shown in the previous section are used for floating point calculations. For integer calculations (used for types like byte, word, integer, longint, etc), the division operator is not used. Instead, we have two new operators that also have an equivalent in modern mathematics. Imagine the calculation of, for example, 7/3. As we have learned in primary school, this is equal to **2** with a remainder of **1** to be divided by 3:

$$\frac{7}{3} = 2 + \frac{1}{3}$$

In PASCAL exist two operators `Div` and `Mod` that reproduce these results. Example:

expression	result
`7 Div 3`	2
`7 Mod 3`	1

These replace the floating point operator /. The other three operators (*, +, -) are the same for integer numbers.

Priority

In case there is more than one operator in an expression, the normal rules of mathematics apply as to which one is evaluated first. The multiplication and division operators have higher precedence. So, when we write

```
a := 1 + 3 * 2;
```

The result will be 7.
If we want to change the order at which the operators in the expression are evaluated, we can always place parenthesis (and). So, for example

```
a := (1 + 3) * 2;
```

will result in 8. Putting parenthesis never hurts!

```
a := (1 + 3) - (4 + 5);
```

Examples

```
PROGRAM SimpleCalculations;

VAR x, y, sum, diff, divis: real;

begin
  WriteLn('Give the value of the first variable x:');
  ReadLn(x);
  Writeln('Give the value of the second variable y:');
  ReadLn(y);
  sum := x + y;
  Writeln('The sum of ', x:0:4, ' and ', y:0:4, ' is ', sum:0:4);
  diff := x - y;
  Writeln('The difference between ', x:0:4, ' and ', y:0:4, ' is ', diff:0:4);
  divis := x / y;
  Writeln(x:0:4, ' divided by ', y:0:4, ' is ', divis:0:4);
end.
```

will produce, when running:

```
Give the value of the first variable x:
3.4
Give the value of the second variable y:
1.8
```

```
The sum of 3.4000 and 1.8000 is 5.2000
The difference between 3.4000 and 1.8000 is 1.6000
3.4000 divided by 1.8000 is 1.8889
```

Note the format specifiers in the writeln statement (:0:4), as described in lecture 6.

```
PROGRAM IntegerCalculations;

VAR x, y, modd, divv: integer;

begin
  WriteLn('Give the value of the first variable x:');
  ReadLn(x);
  Writeln('Give the value of the second variable y:');
  ReadLn(y);
  sum := x + y;
  Writeln('The sum of ', x, ' and ', y, ' is ', sum);
  diff := x - y;
  Writeln('The difference between ', x, ' and ', y, ' is ', diff);
  divv := x Div y;
  modd := x Mod y;
  Writeln(x, ' divided by ', y, ' is ', divis, ' plus ', modd, '/', y);
end.
```

will produce, when running:

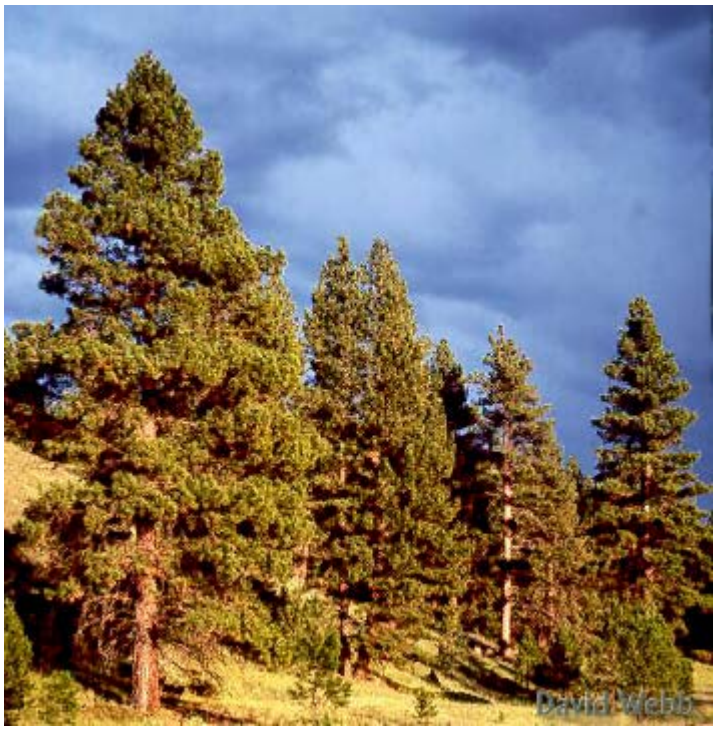
```
Give the value of the first variable x:
13
Give the value of the second variable y:
5
The sum of 13 and 5 is 18
The difference between 13 and 5 is 8
13 divided by 5 is 2 plus 3/5
```

Quick test:

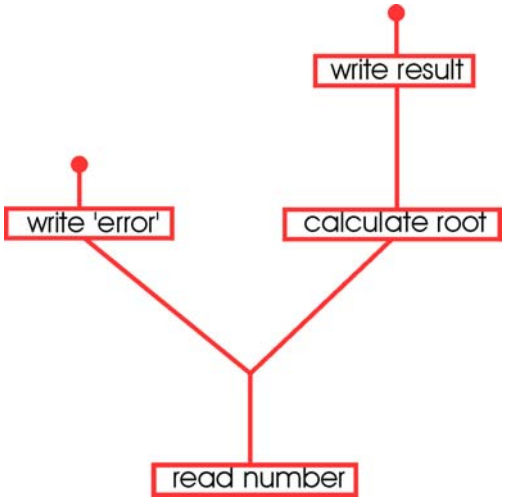
To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this is **NOT** the form the final test takes!

Peter Stallinga. Universidade do Algarve, 1 março 2002

Lecture 8: Branching I (if ... then ... else)



Until now, all the instructions that were put within the program were executed. Moreover, they were executed exactly in the order that they were placed. The first line of the program was executed first, then the second, then the third, and so on. This is not always the case. With *branching* (a *branch* is a part of a tree) we can control the flow of the program.



Imagine a program that specifies a number and the computer will calculate the square-root of the number. Taking the square-root of a negative number doesn't make sense (unless you are working with complex numbers, off course), so you would like your program to generate an error and stop when the user enters a negative number. You want text like

```
Negative numbers are not allowed!
```

to appear on the screen. Obviously, you do not always want this text to appear on the screen; in case the user enters a positive number you just want the square-root to be calculated and appear on the screen:

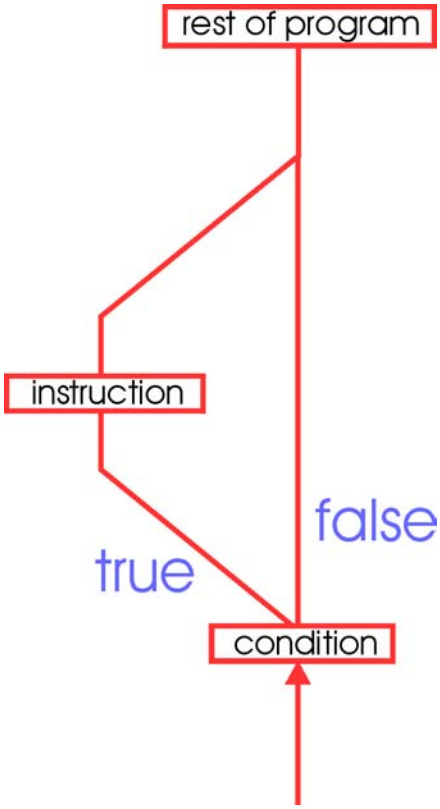
```
The square-root of 5 is 2.23607
```

You would like to have some way to check the number and depending on this result, execute parts of the program.

if ... then ...

The simplest way to have a control over the instructions that will be executed is with the statement if .. then. The full syntax of the statement is

```
if condition then instruction;
```



For `condition` we will substitute our condition and for `instruction` we will put our instruction(s) that will be executed if and only if the condition was true.
The `condition` is an expression that results in a value of type boolean (see lecture 4). This can be a variable, for instance: were `b` declared as `boolean`, the following is correct:

```
if b then instruction;
```

On the other hand, more common are conditions with expressions that compare variables, such as

```
if (x = y) then instruction;
if (x < y) then instruction;
```

comparison	meaning
(a = b)	a equal to b
(a <> b)	a not equal to b
(a < b)	a smaller than b
(a > b)	a larger than b
(a <= b)	a smaller or equal to b
(a >= b)	a larger or equal to b

Remember that if we want more than one instruction to be executed, we can group them with a begin ... end combination, so that for the if ... then statement they appear as one.

```
if (a = b) then
  begin
    instruction1;
    instruction2;
  end;
```

In this case, both `instruction1` and `instruction2` will be executed when a is equal to b.

The normal execution of the program will resume after the block of instructions. In the following example, `instruction3` and `instruction4` will be executed, regardless of the condition (a = b).

```
if (a = b) then
  begin
    instruction1;
    instruction2;
  end;
instruction3;
instruction4;
```

to be executed:

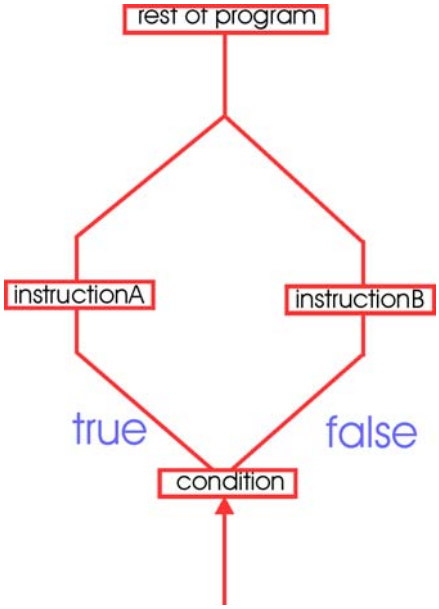
(a=b)	(a<>b)
instruction1 instruction2 instruction3 instruction4	instruction3 instruction4

Note that here the analogy with branching in trees stops. In a tree, the branches never meet again; once we are on a branch, it is never again possible to join the main trunk.

if ... then ... else ...

If we also want to program to do things in case the condition is not true we can do this with if ... then ... else statement. The general form of this instruction is

```
if condition then
  instructionA
else
  instructionB;
```



Note the peculiarity of PASCAL: **the instruction immediately before else is not terminated with ;**

example:

```
if (a=b) then
  begin
    instruction1;
    instruction2;
  end
else
  begin
    instruction3;
    instruction4;
  end;
instruction5;
instruction6;
```

to be executed:

(a=b)	(a<>b)
instruction1	instruction3
instruction2	instruction4
instruction5	instruction5
instruction6	instruction6

Here a complete program that shows the use of branching to calculate the square-root of a number:

```
PROGRAM SquareRoot;

Var x: real;
       root: real;

begin
  writeln('Give a number');
  readln(x);
  if (x<0) then
    writeln('Negative numbers are not allowed!')
  else
    begin
      root := Sqrt(x);
      writeln('The square-root of ', x:0:4, ' is ', root:0:4);
    end;
  writeln('Have a nice day');
end.
```

Running the program; two examples:

```
Give a number
3.68
The square-root of 3.6800 is 1.9183
Have a nice day
```

```
Give a number
-3.68
Negative numbers are not allowed!
Have a nice day
```

Lecture 8: ... of roots and branches 

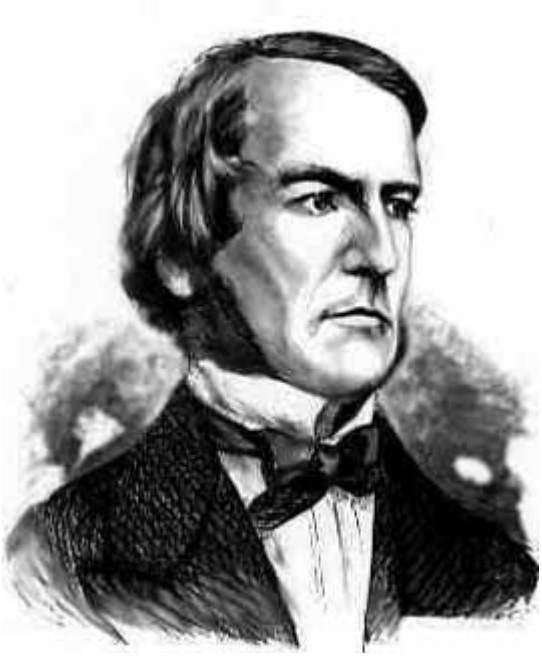
Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this is **NOT** the form the final test takes!

Peter Stallinga. Universidade do Algarve, 4 março 2002

Lecture 9: Branching II / Boolean Algebra

Boolean Algebra



George Boole (1815-1864)

English mathematician. His work "The mathematical analysis of logic" (1847) established the basis of modern mathematical logic, and his boolean algebra can be used in designing computers.
Boole's system is essentially two-valued. This can be symbolized by

0 or 1	*"binary representation"*
TRUE or FALSE	*"truth representation"*
0 V or 5 V	*"TTL electronics (transistor-transistor logic)"*
0 pC or 1 pC (pC = pico-Coulomb)	*"the charge in a condensator, the elementary memory unit in (dynamic) RAM"*

In the previous lecture (aula 8) we have seen how we can control the flow of the program by the branching instructions `if ... then` and `if ... then ... else`. We used conditions like (x<1.0). Now imagine we apply this to calculate the square-root of (x^2 – 4). Clearly, this doesn't have an answer for x between –2 and +2. It would be nice to check if x is in this range or not. We could solve this with

```
if (x<2) then
  if (x>-2) then
    WriteLn('Error');
```

Much nicer would be if we could do this in a single condition. We will now see that exactly that is possible

```
if ((x<2) AND (x>-2)) then WriteLn('Error');
```

which obviously mean that both conditions (x<2 and x>–2), should be true for the complete condition to be true.
This is an example of a Boolean calculation

```
condition3 := condition1 AND condition2;
if condition3 then instruction;
```

Other **Boolean algebra** operators that can be used in PASCAL are `OR` and `XOR`. (Others that are not implemented include NAND and NOR). The `AND` and `OR` operations are obvious. The `XOR` operation is a little complicated. `a XOR b` means "a or b true, but not both!"
Finally, there is the boolean negator `NOT`. It means taking the opposite; if `a` is `TRUE`, `NOT a` is `FALSE`, and vice cerse. Wheras `AND`, `OR` and `XOR` need two operands (for example `a XOR b`), `NOT` needs only one (`NOT a`).
With these four operators we can calculate all possible conditions we will need.

For completeness sake, here is the complete calculation tables ("truth tables") for the four Boolean operators used in modern programming languages.

Boolean operator
AND
OR
XOR
NOT

AND

a	b	a AND b
TRUE	**TRUE**	**TRUE**
TRUE	FALSE	FALSE
FALSE	**TRUE**	FALSE
FALSE	FALSE	FALSE

OR

a	b	a OR b
TRUE	**TRUE**	**TRUE**
TRUE	FALSE	**TRUE**
FALSE	**TRUE**	**TRUE**
FALSE	FALSE	FALSE

XOR

a	b	a XOR b
TRUE	**TRUE**	FALSE
TRUE	FALSE	**TRUE**
FALSE	**TRUE**	**TRUE**
FALSE	FALSE	FALSE

NOT

a	NOT a
TRUE	FALSE
FALSE	**TRUE**

Examples:

```
a := 1;
b := -2;
c := 3;
d := 3;

if (a>0) then
  WriteLn('TRUE')
else
  WriteLn('FALSE');
```

```
(a>0)                    TRUE
(b>0)                    FALSE
(a>0) AND (b>0)          FALSE
(a>0) OR (b>0)           TRUE
(a>0) XOR (b>0)          TRUE
NOT (a>0)                FALSE
(NOT (a>0)) OR (b>0)     FALSE
(2=b) OR (NOT (2=b))     TRUE
(NOT (c>0)) XOR (d>0)    TRUE
```

Boolean algebra for integers

We can also apply boolean algebra to complete numbers (byte, word, integer, longint, see aula 5). Although this was not in the original idea of Boole, we can easily perform the same type of calculations with numbers, as long as we do this "one bit at a time" and use the representations "1 = TRUE" and "0 = FALSE". With this convention, the truth tables for bit-wise boolean algebra become.

AND

a	b	a AND b
1	**1**	**1**
1	0	0
0	**1**	0
0	0	0

OR

a	b	a OR b
1	**1**	**1**
1	0	**1**
0	**1**	**1**
0	0	0

XOR

a	b	a XOR b
1	**1**	0
1	0	**1**
0	**1**	**1**
0	0	0

NOT

a	NOT a
1	0
0	**1**

49 = 1 1 0 0 0 1
24 = 0 1 1 0 0 0
49 OR 24 = 1 1 1 0 0 1 = 57

As an example, imagine we want to calculate 49 OR 24
First we have to convert these numbers to the binary system (see aula 3):
49 = 1*32 + 1*16 + 0*8 + 0*4 + 0*2 + 1*1 = 110001
24 = 0*32 + 1*16 + 1*8 + 0*4 + 0*2 + 0*1 = 011000
Then we do a bitwise calculation with the conventions as in the table above (1 OR 1 = 1, 1 OR 0 = 1, 0 OR 1 = 1, 0 OR 0 = 0), which will give 111001
This we then convert back to the decimal system and we get
111001 = 1*32 + 1*16 + 1*8 + 0*4 + 0*2 + 1*1 = 57

Multiple branching: Case ... Of

In the previous lecture (see aula 8) we have seen how to use the `if ... then` instruction to branch between two possible parts of the program. In some cases, we want that there are more than two possible ways the program continues. For this we have the `Case ... Of` instruction.
Imagine the following program that asks from what year the student is:

```
PROGRAM Years;

VAR ano: integer;

begin
  WriteLn('From what year are you?');
  ReadLn(ano);
  if (ano=1) then
    writeln('primeiro ano: MAT-1, CALC-1')
  else
    if (ano=2) then
      writeln('segundo ano: INF, LIN-ALG')
    else
      if (ano=3) then
        writeln('terceiro ano: ELEC, FYS')
      else
        if (ano=4) then
          writeln('quarto ano: QUI, MAT-2')
        else
          if (ano=5) then
            writeln('quinto ano: PROJECTO')
          else
            writeln('>5: AINDA NAO ACABOU?');
end.
```

(Note the structure of the program, with indentations. Also note the missing ; before every `else`).
This program will run without problem

```
From what year are you?
 1
primeiro ano: MAT-1, CALC-1

From what year are you?
 4
quarto ano: QUI, MAT-2
```

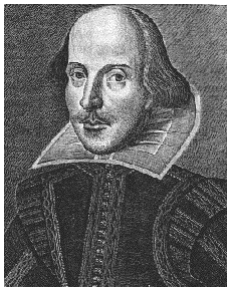
The structure is not very readable, though. To make it better, we can use `case ... of`. Whereas in "`if condition then instruction1`" the condition is necessarily of the boolean type (TRUE or FALSE), in `case ... of` we can use any type of variable that has discreet values (in contrast to floating point variables which do not have discreet, whole number values).

```
Case expression of
  value1: instruction1;
  value2: instruction2;
        |
  valueN: instructionN;
  else instructionE;
end;
```

The `expression` needs to result in a value of any countable type (for example: integer, byte, boolean, but also char. Not, for example: real or string). This can be a simple variable or a calculation resulting in a value. The values `value1` to `valueN` also have to be of the same type;
As an example the above program rewritten to make use of `case ... of`:

```
PROGRAM Years;

VAR ano: integer;

begin
  WriteLn('From what year are you?');
  ReadLn(ano);
  Case ano of
    1: writeln('primeiro ano: MAT-1, CALC-1');
    2: writeln('segundo ano: INF, LIN-ALG');
    3: writeln('terceiro ano: ELEC, FYS');
    4: writeln('quarto ano: QUI, MAT-2');
    5: writeln('quinto ano: PROJECTO')
    else writeln('>5: AINDA NAO ACABOU?');
  end;
end.
```

This progam will have the same output as the program before, but now it is more readable.



Aula 9: (2*B) OR (NOT (2*B))

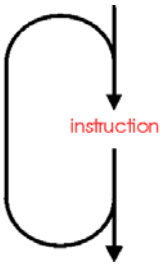
Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this is **NOT** the form the final test takes!

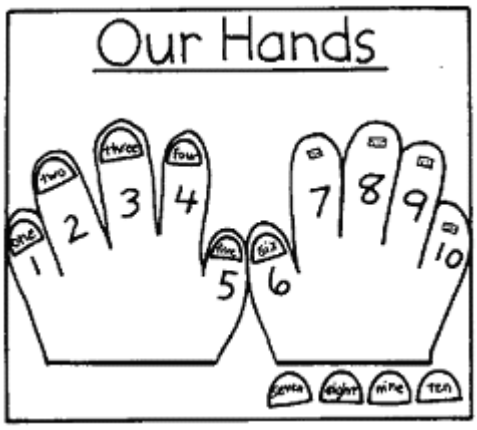
Peter Stallinga. Universidade do Algarve, 1 março 2002

Lecture 10: Loops I: For

For loop



The most common loop in PASCAL is the For loop. This loop is used to execute things a predetermined number of times in a **countable** way. This in contrast to loops that will run while a certain condition is true, as we will learn in the next lecture. The general structure of the For loop is




```
For variable := start_value To end_value Do
     instruction;
```

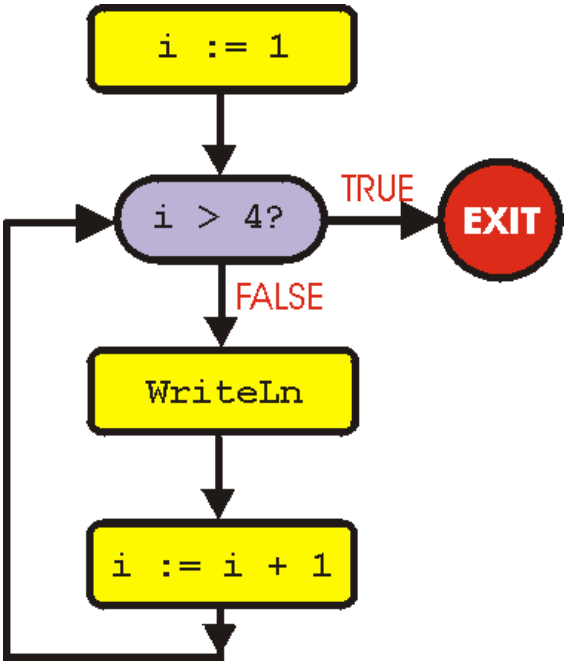
The `instruction` is repeated a number of times, determined by the control parameters `start_value` and `end_value`.
Since the loop is for doing things in a **countable** way, the control parameters `variable`, `start_value` and `end_value` all have to be of any integer type. Do not forget to declare the variable (see lecture 5)
Example:

program code

```
PROGRAM ForLoopExample;

Var i: integer;

begin
  For i := 1 To 4 Do
    WriteLn('Ola');
end.
```

output

```
Ola
Ola
Ola
Ola
```



This program is doing the following
1) it is attributing 1 to `i`
2) it is checking if `i` is larger than 4
3) if so: EXIT LOOP else
4) execute `WriteLn('Ola');`
5) add 1 to `i`
6) go to step 2)

Multiple instructions

Just like with the `if ... then ... else` structure, we can also group instructions together with `begin`

and `end` in loops:

program code	*output*

`For i := 1 To 4 Do`
`  begin`
`    WriteLn('Ola');`
`    WriteLn('Eu chamo-me Peter');`
`  end;`
	`Ola`
	`Eu chamo-me Peter`
	`Ola`
	`Eu chamo-me Peter`
	`Ola`
	`Eu chamo-me Peter`
	`Ola`
	`Eu chamo-me Peter`

Use of the loop variable

Inside the loop the variable can be used, but **don't mess** with it

Good code:

program code	*output*

`For i := 1 To 4 Do`
`  WriteLn(i, ' Ola');`
	`1 Ola`
	`2 Ola`
	`3 Ola`
	`4 Ola`

Bad code:

program code	*output*

`For i := 1 To 4 Do`
`  begin`
`    WriteLn(i, ' Ola');`
`    i := i + 1;`
`  end;`
	`1 Ola`
	`3 Ola`

The program on the right is an example of bad code. Such style of programming, although at some times it will save space and maybe executing time, makes your program unstructured and very difficult to understand for others! If you want to achieve things like in the program on the right, use other loops, like `while` or `repeat-until`, or , better, use something like in the program below.

program code	*output*
`For i := 1 To 2 Do` `  WriteLn(2*i-1, ' Ola');`	`1 Ola` `3 Ola`

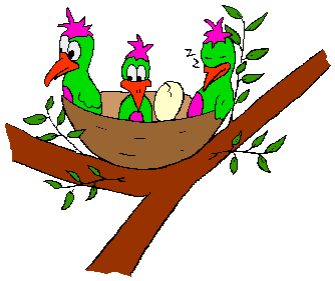
Expressions

The `start_value` and `end_value` can also be variables or expressions that return a value of integer type or variables of the same type, for example:

program code	*output*

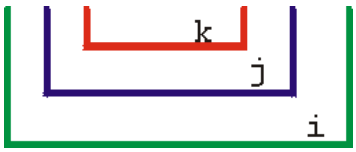
`PROGRAM ForLoopExample;`
 	
`Var i: integer;`
`  j: integer;`
 	
`begin`
	`5 Ola`
	`6 Ola`
	`7 Ola`
	`8 Ola`
	`9 Ola`

```
  j := 5;
  For i := j To 2*4+1 Do
    WriteLn(i, 'Ola');
end.
```



Nested loops

The For loops (and any other loop as well) can also be 'nested', which means that they can be put within eachother. We can create double loops, or triple loops (like in the figure below on the left) or any other level. Such structures look like nests of birds and hence the name '**nesting**' of loops. Here are some examples



program code

```
PROGRAM NestedLoop3;

Var i, j, k: integer;

begin
  For i := 1 To 2 Do
    For j := 1 To 2 Do
      For k := 1 To 2 Do
        WriteLn('i=',i,' j=', j,
          ' k=',k);
end.
```

output

```
i=1 j=1 k=1
i=1 j=1 k=2
i=1 j=2 k=1
i=1 j=2 k=2
i=2 j=1 k=1
i=2 j=1 k=2
i=2 j=2 k=1
i=2 j=2 k=2
```

program code

```
PROGRAM NestedLoop3;

Var i, j, k: integer;

begin
  For i := 1 To 2 Do
    For j := 1 To 2 Do
      begin
        For k := 1 To 2 Do
          WriteLn('i=',i,' j=', j,
            ' k=',k);
        For k := 1 To 2 Do
          WriteLn('i=',i,' j=', j,
            ' k=',k);
      end;
end.
```

output

```
i=1 j=1 k=1
i=1 j=1 k=2
i=1 j=1 k=1
i=1 j=1 k=2
i=1 j=2 k=1
i=1 j=2 k=2
i=1 j=2 k=1
i=1 j=2 k=2
i=2 j=1 k=1
i=2 j=1 k=2
i=2 j=1 k=1
i=2 j=1 k=2
i=2 j=2 k=1
i=2 j=2 k=2
i=2 j=2 k=1
i=2 j=2 k=2
```

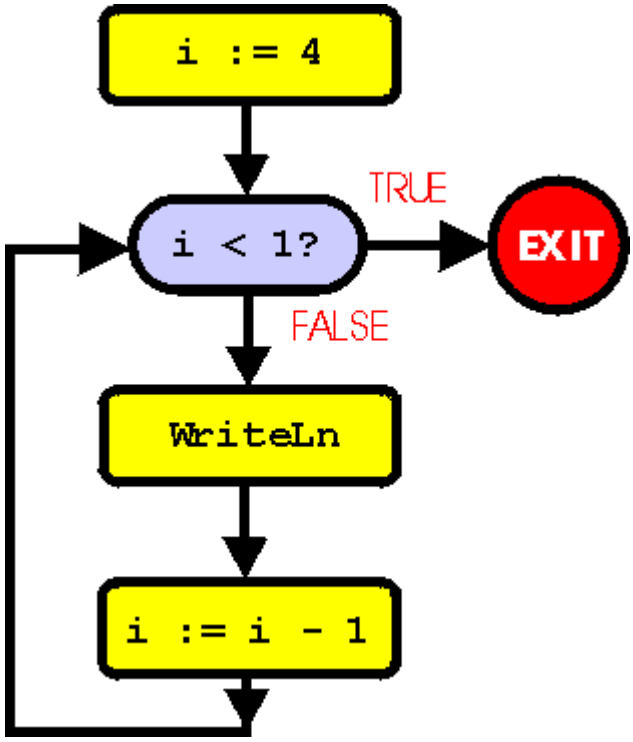
Down-counting loops

Instead of a loop with a variable that is counting up, we can also make a loop with a variable that is counting down. This we do by substituing the word 'To' in the structure with the word 'DownTo'. For example:

program code	output

`For i := 4 DownTo 1 Do`
`  WriteLn(i, ' Ola');`
	`4 Ola`
	`3 Ola`
	`2 Ola`
	`1 Ola`

This program is doing the following
1) assign 4 to `i`
2) check if i is smaller than 1
3) if so: EXIT loop,
   if not
4) execute `WriteLn('Ola');`
5) subtract 1 from `i`
6) go to step 2)



Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 9 março 2002

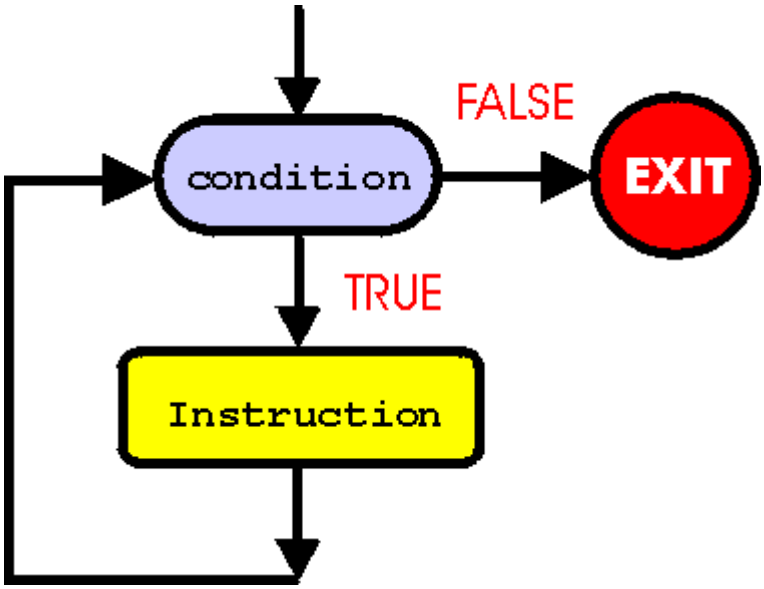
Lecture 11: Loops II: `While ... Do` and `Repeat ... Until`

`While ... Do`

The loops in this lecture, while-do and repeat-until are used for repeating things that are not exactly countable. Normally we use this when it is not exactly clear when the loop will finish, for instance because the control variable changes within the loop (as is strictly disadvised in For loops). Also, when we want to loop over something with a floating point type variable we use the while-do and repeat-until loops.

The general format of the while-do loop is

```
While condition do
  Instruction;
```



This structure is repeating the instruction as long as the condition is true. The `condition` is any condition that results in a boolean value (TRUE or FALSE), as we have discussed in lecture 8. This can be a comparison, or anything else (still information in file?, user pressed a key?, etc).
Note that the structure while-do does not attribute a starting value to any variable like the for-loop was. So, we have to do this ourselves.
Example:

program code

```
PROGRAM WhileExample;

Var x: real;

begin
  x := 0.0;
  While (x<=1.0) do
    begin
      WriteLn(x:0:1);
      x := x + 0.1;
```

output

```
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
```

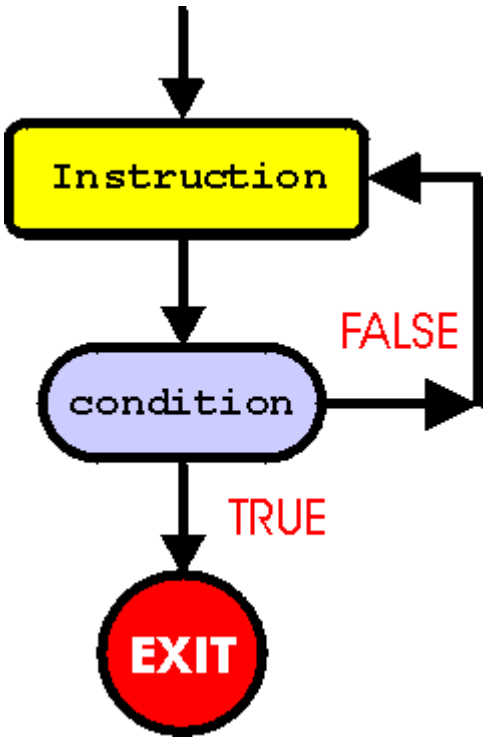
```
    end;
end.
```

```
1.0
```

repeat ... until

The repeat-until structure is very similar to the while-do structure. The repeat-until structure repeats an instruction (or instructions) until a condition is met. The general format is

```
Repeat
 Instruction;
until condition;
```



Another very important difference between while-do and repeat-until is that with while-do the condition is checked in the **beginning** of the loop, whereas in repeat-until it is checked at the **end**. Therefore, **the instructions in repeat-until are at least executed once.**
Look at the following programs (also an example with a for-loop is included). Only the code with the repeat-until structure has output.

program code

```
x := 100;
repeat
  writeln('Ajax');
  x := x + 1;
until (x>10);
```

output

```
Ajax
```

program code

```
x := 100;
while (x<=10) do
  begin
    writeln('Ajax');
    x := x + 1;
  end;
```

output

program code

```
for i := 100 to 10 do
  begin
    writeln('Ajax');
  end;
```

output

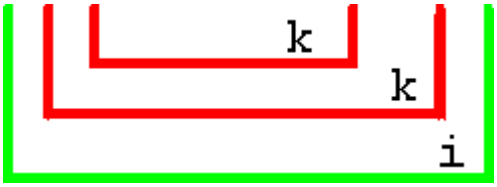
Note also the conditions. To make the program do the same thing, we have to make the condition for repeat-until (x>0) the opposite of the condition for while-do (x<=10).

Nesting II

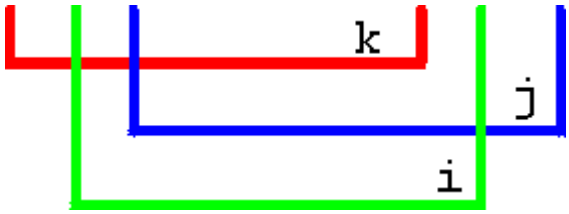
Now that we know all the types of loops, let's take a look at the rules-of-good-behavior related to nesting:

- Each For loop must use its own seperate control variable.
- The inner loop must begin and end entirely within the outer loop

Examples of bad code:




```
for i := 1 to 10 do
  for  k := 1 to 10 do
    begin
      for k := 1 to 10 do
         writeln('Hello');
    end;
```




```
for  k := 1 to 10 do
  begin
    i := k;
    repeat
      j := i-1;
      while (j<10) do
        begin
  end; {end of for}
  i := i + 1;
    until i>20;  {end of repeat-until}
          j := j + 1;
        end; {end of while-do}
```

Good indentation of your program will always avoid such errors.

Summarizing

Three rules for loops:

- each loop with seperate **control variable**
- loops start and finish completely **inside** other loop
- give a **possibility to exit** the loop. Remember the anecdote of the programmer found dead in the shower with a bottle of shampoo "wash, rinse and repeat!"

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 15 março 2002

Lecture 12: Modular Programming I: Procedures

Until now we have only used the things that were supplied by PASCAL. We have learned how to write loops (for, while-do, repeat-until), how to have input and output (readln, writeln), how to control the flow of the program (if-then, if-then-else, case-of), how to declare variables (var), constants (const), how to assign values to constants (:=) and how to calculate, even up to complicated Boolean algebra, but we have never invented anything NEW. With Procedures and Functions we can do exactly that.
We already met a couple of procedures which are standard in PASCAL, namely `Write`, `WriteLn`, `Read` e `ReadLn`. Now we will define our own procedures.

Procedures and Functions are small subprograms or **modules** of the main program. Each of theses modules performs a certain task. This helps to organize the program and make it more logic and can also increase the programming efficiency by avoiding repetitions of parts of the program.

There are two type of modules

1. Modules which don't return anything. In PASCAL these are called **Procedures**. Of these there are two subtypes
   - Procedures that accept parameters and
   - Procedures that don't accept parameters as input
2. Modules that are returning values. In PASCAL these are called **Functions**. Again, there are two subtypes
   - Functions that accept parameters and
   - Functions that don't accept parameters as input

Procedures

Modules that are not returning any value are called **Procedures**. They just do something and don't return anything. They can either accept parameters (and work with them) or do things without any input from outside. In any case, Procedures are like programs-within-programs. They have

- A name. The same rules for identifiers applies to the names of procedures. See aula 5
- Variables and constants. These have to be declared in the Procedure
- a `begin` and `end;` combination indicating the start and end of the procedure. Note that this end is finished with a ;
- The program code

A prototype of the declaration of a procedure:

```
Procedure ProcedureName;

Var <variable_list>;
Const <const_list>;
```

```
begin
  instructions;
end;
```

The place to declare a procedure is **inside** the program (so **after** the program name declaration), but **before** the beginning of the main program, so before the first `begin` statement.

Calling

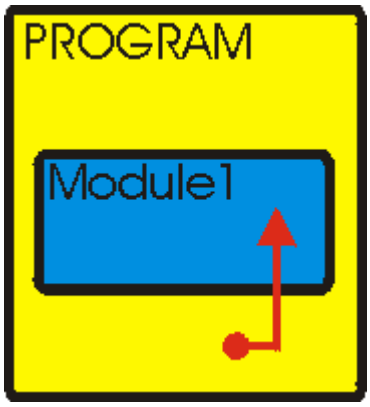
After declaring a new procedure, we can use it in the main program. This is called **calling** the procedure. We do this by writing the name of the procedure. As a complete example

program code

```
PROGRAM WithProcedure;

Var x: real;

PROCEDURE Module1;
var y: real;
begin
  WriteLn('Now I am entering Procedure Module1');
  WriteLn('Give a value for y:');
  ReadLn(y);
  Writeln(y/3:0:3);
end;

begin
  WriteLn('Starting the program');
  Module1;
end.
```

output

```
Starting the program
Now I am entering Procedure Module1
Give a value for y:
 12
4.000
```



A program calling one of its procedures

Procedures calling procedures

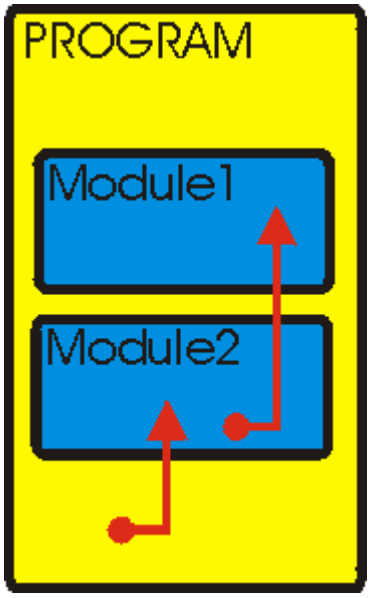
Procedures can also be called by other procedures or functions. Under normal circumstances, in PASCAL these procedures have to be declared after the procedure to be called, though. Take a good look at the following program and the output it generates when run:

program code

```
PROGRAM TwoProcedures;

Var x: real;

PROCEDURE Module1;
var y: real;
begin
  WriteLn('Now I am entering Procedure
Module1');
  WriteLn('Give a value for y:');
  ReadLn(y);
  Writeln(y/3:0:3);
  WriteLn('Leaving Module1');
end;

PROCEDURE Module2;
var z: real;
begin
  WriteLn('Now I am entering Procedure Module2');
  Module1;
  WriteLn('Give a value for z:');
  ReadLn(z);
  Writeln(z*2:0:3);
  WriteLn('Leaving Module2');
end;

begin
  WriteLn('Now I am starting the program');
  Module2;
end.
```

output

```
Now I am starting the program
Now I am entering Procedure Module2
Now I am entering Procedure Module1
Give a value for y:
 12
4.000
Leaving Module1
Give a value for z:
 10
20.000
Leaving Module2
```



A program calling one of its procedures which, in turn, is calling one other procedure

Use of variables

Yes: Procedures can make use of the variables of the main program, but not the other way around:

No: The main program cannot make use of the variables of the procedures. Think of the **procedure as a a box with one-way-mirror windows**. From inside you can look outside, but from outside you do not see what is going on inside.

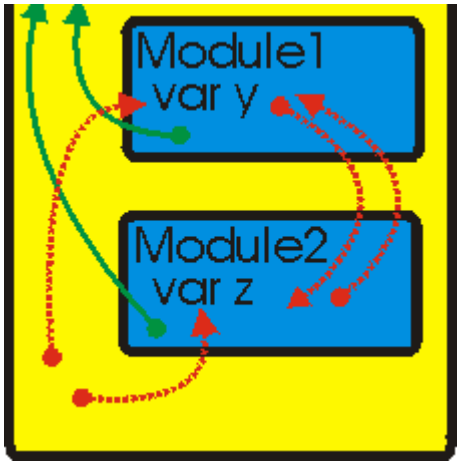
No: Procedures cannot use variables of the other procedures, for the same reason as why the main program cannot see the variables of the procedures. They can see the other procedures, but cannot look inside.

See the example program below. All the instructions in ***red bold italics*** are forbidden.

program code

```
PROGRAM ProcedureVariables;
```

```
Var x: real;

PROCEDURE Module1;
var y: real;
begin
  WriteLn(x);
  WriteLn(y);
  WriteLn(z); (* this is not allowed *)
end;

PROCEDURE Module2;
var z: real;
begin
  WriteLn(x);
  WriteLn(y); (* this is not allowed *)
  WriteLn(z);
end;

begin
  WriteLn(x);
  WriteLn(y); (* this is not allowed *)
  WriteLn(z);   (* this is not allowed *)
end.
```



forbidden (red dashed arrows) and allowed (green solid arrows) variable use.

Later we will take a look at these things in more detail.

Nesting

In most programming languages Procedures can be nested. We can write procedures within procedures within procedure within ....
Take a look at the program example below. The procedure within the procedure is shown in **bold**. Just like the variables of the procedure Module1, the sub-procedure Module2 is NOT visible outside the Module1.

program code

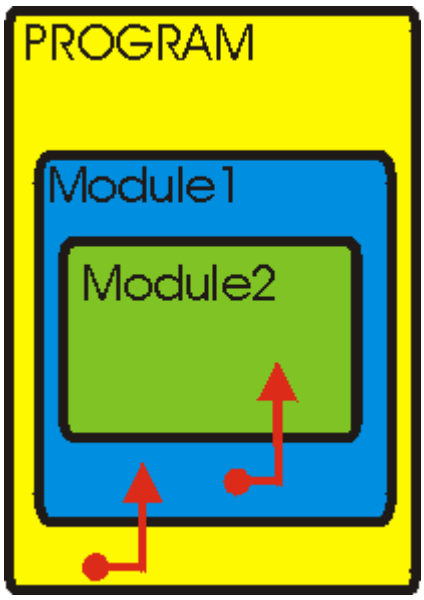
```
PROGRAM NestedProcedures;

Var x: real;

PROCEDURE Module1;
var y: real;
  PROCEDURE Module2;
  var z: real;
  begin
    WriteLn('Now I am entering Procedure
```

output

```
Now I am starting the program
Now I am entering Procedure
Module1
Give a value for y:
 12
4.000
Now I am entering Procedure Module2
Give a value for z:
 10
20.000
```

```
Module2');
    WriteLn('Give a value for z:');
    ReadLn(z);
    Writeln(z*2:0:3);
    WriteLn('Leaving Module2');
  end;
begin
  WriteLn('Now I am entering Procedure Module1');
  WriteLn('Give a value for y:');
  ReadLn(y);
  Writeln(y/3:0:3);
  Module2;
  WriteLn('Leaving Module1');
end;

begin
  WriteLn('Now I am starting the program');
  Module1;
    (* cannot use variables or procedures *)
    (*  of Module1 here:                  *)
    (* cannot call Module2 here           *)
    (* cannot use variable y or z here    *)
end.
```

```
Leaving Module2
Leaving Module1
```



Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 15 março 2002

Lecture 13: Modular Programming II:

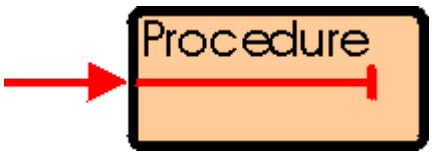
Procedures with input and output

In the previous lecture (aula 12) we have seen Procedures that are not accepting parameters and are not generrating return values. These are simple procedures. Now we are going to look at Procedures that are accepting parameters and Procedures that are producing return values (Functions).

Parameters to pass to Procedures

We can pass parameters to procedures. The procedure can then work with these parameters. In PASCAL the parameters the procedure expects are put after the name of the procedure inside parenthesis:

```
 Procedure
ProcedureName(parameter_list);

 Var <variable_list>;
 Const <const_list>;

 begin
   instructions;
 end;
```



The variables on the parameter list are declared in the same way as the normal variables of a program or procedure, namely we have to specify the type of the variable. Inside the procedure we can use the parameter as iff it were a normal variable. We can calculate with it, use it in conditions, and even change its value.

As an example, the following program will calculate and show the square of a variable `x`: Note the way the parameter `r` is declared and used.

program code | *output*

```
PROGRAM WithParameters;

Var x: real;

PROCEDURE WriteSquare(r: real);
var y: real;
begin
  r := r*r;
  y := r;
  Writeln('The suare of ',r:0:1,' is ',y:0:1);
end;
```

```
The square of 4.0 is 16.0
The square of 3.0 is 9.0
```

```
begin
  x := 4;
  WriteSquare(x);
  WriteSquare(3.0);
end.
```

As seen in the program above, the procedure with parameters can now be called with a variable, as in `WriteSquare(x)` or with a constant.as in `WriteSquare(3.0)`.

Another example, that uses two parameters:

program code

```
PROGRAM WithParameters;

Var x: integer;
    y: integer;

PROCEDURE WriteSum(i1, i2: integer);
  ( will write the sum of i1 and i2 *)
var j: integer;
begin
  j := i1+i2;
  Writeln('The sum of ',i1,' and ',
    i2, ' is ', j);
end;

begin
  x := 4;
  y := 5;
  WriteSum(x, y);
  WriteSum(3, 4);
end.
```

output

```
The sum of 4 and 5 is 9
The sum of 3 and 4 is
```

Finally, an example with a parameter list of mixed types. Just like in normal variable decalration, variables to be declared in the parameter list are separated by ;

program code

```
PROGRAM WithParameters;

PROCEDURE WriteNTimes(r: real; n: integer);
  (* Will write n times the real r *)
var i: integer;
begin
  for i := 1 to n do
    WriteLn(r:0:3);
end;


PROCEDURE WriteFormatted(r: real; n: integer);
  (* Will write the real r with n decimal cases *)
begin
  WriteLn(r:0:n);
end;
```

output

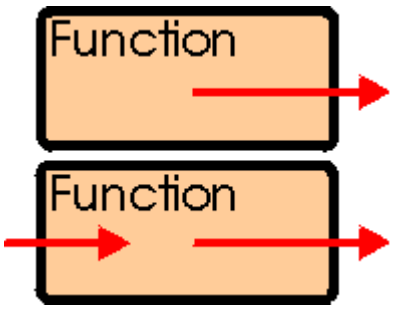
```
3.000
3.000
3.000
3.000
5.00000
5.0
```

```
begin
  WriteNTimes(3.0, 4);
  WriteFormatted(5.0, 5);
  WriteFormatted(5.0, 1);
end.
```

Functions

Functions are procedures that are returning an output value. The **type** of the returning value has to be specified at the moment of declaring the function, after the parameterlist (if any), preceded by a colon **:** Example:

```
 Function
FunctionName(parameter_list):
   type;

 Var <variable_list>;
 Const <const_list>;

 begin
   instructions;
 end;
```



Functions without and with input parameters

Somewhere in the instructions we have to specify a **returning value** by.

```
FunctionName := Value;
```

For example

```
FUNCTION Square(r: real): real;
  (* will return the square of of the parameter r *)
begin
  r := r*r;
  Square := r;
end;
```

At the place where the function will be called, we can assign this value to a variable (of the same type as the returning value of the function!), for example

```
y := Square(3.0);
```

use it as part of an expression, for example

```
y := 4.0 * Square(3.0) + 1.0;
```

or use it in another function or procedure, for example

```
Writeln(Square(3.0):0:1);
```

A full example:

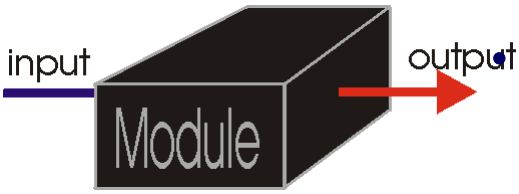
program code | *output*

```
PROGRAM WithParameters;

Var x, y: real;

FUNCTION Square(r: real): real;
  (* will return the square of of the parameter r *)
begin
  r := r*r;
  Square := r;
end;

begin
  x := 4.0;
  y := Square(x);
  WriteLn('The square of ', x:0:1, ' is ', y:0:1);
  WriteLn('The square of ', 3.0:0:1, ' is ',
    Square(3.0):0:1);
end.
```

```
The square of 4.0 is 16.0
The square of 3.0 is 9.0
```

Why?

Now the big question is "why?". Why write procedures and functions if we can do the same thing with normal lines of instructions? Indeed, the first languages (for example BASIC) didn't have the possibility to write functions and still we could write programs to solve any problem with it. There are however two important reasons why to use modules.



With modules, because they are like **black boxes**, we can distribute our programming tasks over several people or groups of people without having to have much communication between the people. We can tell somebody that we need a function that diagonalizes a matrix and we do not have to say how we want it done. We will only specify the type of parameters to pass to the procedure.

For the same reason, we can easily use parts of other programs or libraries for our purpose. In the ideal case we will just "link" those procedures that we will need to our program, without knowing exactly how they work. (Of course with knowing what they will do and how to call them)

- With procedures and functions the program becomes shorter, more efficient and more readible by avoiding repetitions of code and by organizing it more logical.

Quick test:

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 17 março 2002

Lecture 14: Mathematical Functions

Most languages already have many procedures and functions implemented. As an example, we will discuss the most usefull functions of PASCAL. It is not very important to remember the exact format and use of these functions. Much better is to remember that such things exist and when you need them, read the manual of the language you are using and chose the function you need.

Function	*description*	*argument*	*result*	*examples*
Abs	absolute value of parameter parameter can be real or integer Abs returns the same type	real integer	real integer	Abs(-23.2) = 23.1 Abs(12.3) = 12.3 Abs(-10) = 10
Cos	Cosine of argument. Argument in radians (2π rad = 360^o)	real	real	Cos(1.0) = 0.5403
Sin	Sine of argument. Argument in radians (2π rad = 360^o)	real	real	Sin(1.0) = 0.8415
ArcTan	Inverse tangent of argument	real	real	ArcTan(1.0) = $\pi/4$
Exp	Exponent of argument	real	real	Exp(1.0) = 2.718
Ln	Natural logarithm of argument	real (>0)	real	Ln(10.0) = 2.303
Odd	Test whether argument is odd or even	integer	boolean	Odd(3) = TRUE
Round	Rounding of argument to closest complete number	real	integer	Round(3.4) = 3 Round(3.5) = 4
Int	Rounding down to complete number	real	real	Int(3.99) = 3.00
Frac	returns fractional part of argument	real	real	Frac(3.99) = 0.99
Trunc	Rounding down to complete number	real	integer	Trunc(3.99) = 3
Sqrt	Square root of argument	real (>0)	real	Sqrt(3.0) = 1.732
Sqr	Square of argument	real	real	Sqr(2.0) = 4.0
Random	Generates random number	integer	real integer	Random = 0.0234 Random(10) = 3
Randomize	Randomizes the random number generator.			

Trigonometric functions

As trigonometric functions we have `Sin` and `Cos` which return the sine and cosine of the argument, respectively. Note that the Tangent function doesn't exist, but as we all know (hopefully) Tan(x) is equal to Sin(x)/Cos(x). Note that the arguments (the parameters to pass to the function) have units radian (2π rad = 360^o). As an example:

program code

```
PROGRAM Trigonometry;

Var xdeg, xrad, ysin, ycos, ytan: real;

begin
  xdeg := 45.0;
  xrad := Pi*xdeg/180.0;
   (* convert to radians! Note that the
    value Pi is defined in PASCAL *)
  ysin := Sin(xrad);
  ycos := Cos(xrad);
  ytan := ysin/ycos;
  Writeln('x = ',xdeg:0:3);
  WriteLn('Sin = ', ysin:0:4);
  WriteLn('Cos = ', ycos:0:4);
  WriteLn('Tan = ', ytan:0:4);
end.
```

output

```
x = 45.000
Sin = 0.7071
Cos = 0.7071
Tan = 1.0000
```

There exists only one inverse trigonometric function, namely $\tan^{-1}$ which is called `ArcTan` in PASCAL (and in some languages `ATan`). The others (ArcCos and ArcSin) can be derived from ArcTan. In the next lecture I will prove this.

program code

```
PROGRAM InverseTrigonometry;

Var x, y, angle: real;

begin
  x := 0.5;
  y := ArcTan(x);
  angle := 180.0*y/Pi;
  Writeln('x = ', x:0:3);
  WriteLn('ArcTan = ', angle:0:4);
end.
```

output

```
x = 0.500
ArcTan = 26.5651
```

Exponential and power functions

The function `Exp` returns the exponent of the argument:

$\mathrm{Exp}(x) = e^x$

`Ln` returns the (natural) logarithm of the argument. The argument should be larger than zero, of course. The ${}^{10}\mathrm{Log}(x)$ function or in general the ${}^{n}\mathrm{Log}(x)$ function and the general power fucntion x^n do not exist in PASCAL, but we can easily derive them from the functions above:

$x^n = \mathrm{Exp}(\mathrm{Ln}(x^n)) = \mathrm{Exp}(n*\mathrm{Ln}(x))$

${}^{n}\mathrm{Log}(x) = {}^{n}\mathrm{Log}(e^{\mathrm{Ln}(x)}) = \mathrm{Ln}(x)* {}^{n}\mathrm{Log}(e) = \mathrm{Ln}(x) / {}^{e}\mathrm{Log}(n) = \mathrm{Ln}(x) / \mathrm{Ln}(n)$

In many languages the power function (for example in C: pow(x,n)) is implemented in this way any way. In most modern computers with mathematical coprocessors (all Pentium processors have one built in) these complicated calculations of Ln, Exp, but also Cos, Sin and ArcTan are executed very fast because they make use of tables with precalculated values inside the processor.

In some languages, confusingly, with Log, the natural logarithm Ln is meant. Therefore, always READ THE INSTRUCTIONS!

Example:

program code

```
PROGRAM ExponentAndLogarithm;

Var x, y: real;

begin
  x := 100.0;
  y := Ln(x)/Ln(10.0);
  WriteLn('Log(',x:0:1,') = ',y:0:1);
  x := 3.0;
  y := Exp(x*Ln(10));
  WriteLn('10^',x:0:1,' = ',y:0:1);
end.
```

output

```
Log(100.0) = 2.0
10^3.0 = 1000.0
```

Although any power function x^n can be expressed with the functions Exp and Ln as described above, two of these powers are so often used that they are implemented with their own functions: `Sqr` gives x^2 and `Sqrt` gives $x^{1/2}$. Example:

program code

```
PROGRAM SquareRootAndSquare;

Var x, y: real;

begin
  x := 3.0;
  y := Sqrt(x);
  WriteLn('The square-root of ', x:0:3,
    ' is ',y:0:1);
  y := Sqr(x);
  WriteLn('The square of ',x:0:3,
    ' is ',y:0:1);
end.
```

output

```
The square-root of 3.0 is 1.732
The square of 3.0 is 9.000
```

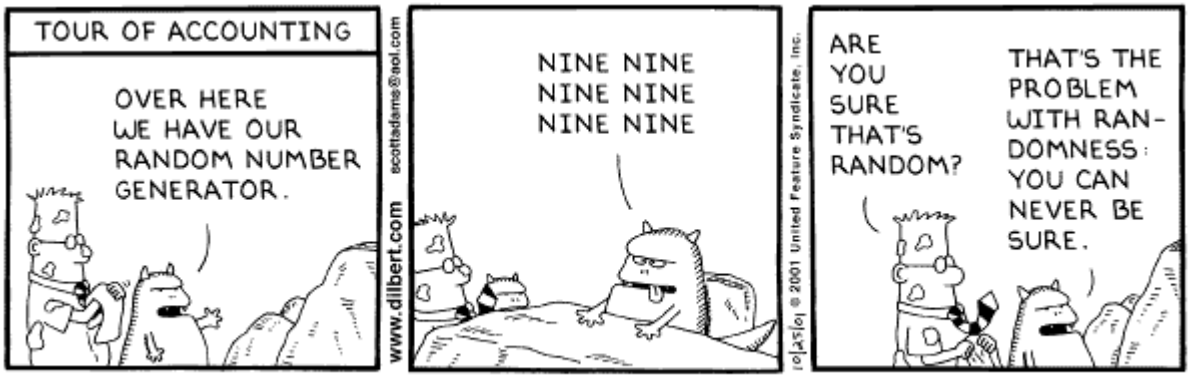
Rounding Functions

Four rounding fucntions are implemented in PASCAL, Round, Int, Frac, and Trunc.
`Round` returns the closest complete number. The returned value is of integer type. Examples: `Round(1.2)` = 1, `Round(2.5)` = 3, `Round(4.99)` = 5.
`Int` returns the part before the floating point, so, the closest complete number below. The returned value is of type real. This may cause some confusion; `Int` doesn't convert the number to integer, like in some other languages (C). Examples `Int(1.2)` = 1.0, `Int(2.5)` = 2.0, `Int(4.99)` = 4, `Int(5.0)` = 5.0.
`Frac` returns the part of the number after the floating point, so a number between 0 and 1. It is therefore also of real type. Examples: `Frac(1.2)` = 0.2, `Frac(2.5)` = 0.5, `Frac(4.99)` = 0.99, `Frac(5.0)` = 0.0.
`Trunc` does the same as Int, but converts to integer. Examples: `Trunc(1.2)` = 1, `Trunc(2.5)` = 2, `Trunc(4.99)` = 4, `Trunc(5.0)` = 5.

The odd function

An odd function is the function `Odd`. It returns TRUE if the argument is odd, FALSE otherwise. This function `Odd(x)` is therefore equal to `(x MOD 2)>0` which also returns TRUE and FALSE, depending on the oddness of the argument x.

Random numbers



One of the nicest things in programming is the use of random numbers. With randowm numbers even the programmer doesn't know the outcome of his own program. It is like flipping a coin or throwing dice. Even though you know exactly what a coin or a die is, you do not know the outcome of an exepriment with them. Generating random numbers with a computer is not easy. Computers are good at doing things in a predictable way, unlike humans, but not good at doing things randomly. There exist fucntions in PASCAL which generate random numbers that we can use. For all our purposes we can consider these numbers to be really random. Only upon closer scrutiny do they prove to be not random at all.
To generate a random number between 0 and 1 we can use the function Random. For example, calling this function five times might generate the series 0.3354, 0.2134, 0.2200, 0.9876, 0.0230.
If we want to have integer numbers between 0 and n-1 we can convert this number with the functions shown in the previous section: `Trunc(n*Random)`. This result can also be obtained directly with the `Random` fucntion by supplying a parameter to the function: `Random(n)` returns integer numbers between 0 and n-1.
Note that everytime the program is ran, the random numbers follow the same sequence. If we want to start another sequence, we can call the procedure `Randomize`.

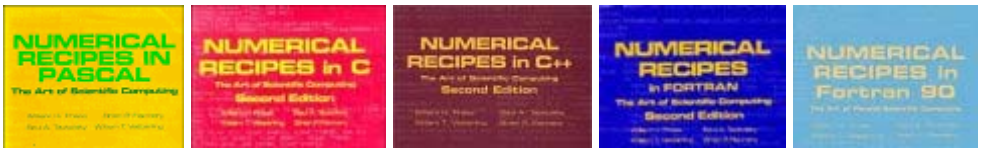
program code

```
PROGRAM RandomNumbers;

Var x, y: real;
    i: integer;

begin
  Randomize;
  for i := 1 to 5 do
    Writeln(Random:0:4);
  for i := 1 to 5 do
    WriteLn(Random(10));
end.
```

output

```
0.7132
0.5111
0.0638
0.7837
0.3810
8
5
0
8
1
```

With these functions we can make cardgames, simulate nuclear decay, simulate traffic or any other kind of random process. The function Random is a homogeneous distribution; all numbers between 0 1nd 1 are equally likely to occur. If we want other distributions we can use the Random function as a starting function and convert it to these distributions. In other programming courses these things will be discussed further. For who wants to know more, I can recommend the book "Numerical Recipes in Pascal" (or in C or in Fortran).



Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 17 março 2002

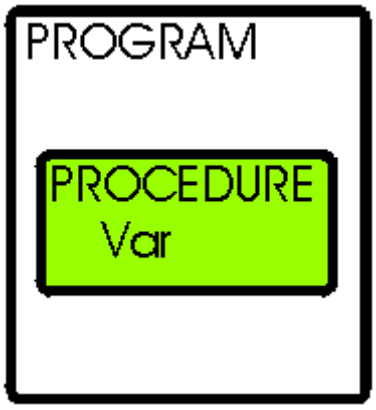
Lecture 15: of variables and procedures

This lecture:

- Global and local variables
- Passing by value, passing by reference.

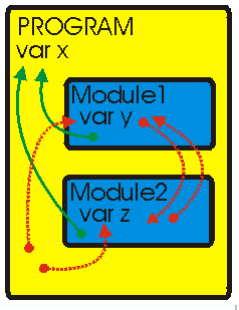
Scope of variables: Global or local

Global variables are variables that can be used everywhere in the program

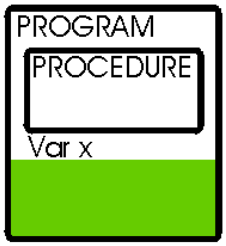


Local variables are only defined inside a procedure or function.

With this new information, it is much more clear which variable we can use when. Inside procedures and functions we can use the global and the local variables. Outside the procedures and functions we can only use the global variables. Remember again what we learned in lecture12 (see the image on the right). The main program cannot use the variables of the modules, but the other way around is allowed.



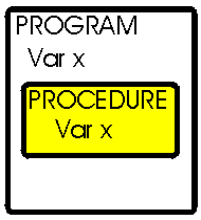
Warning: try to avoid the use of global variables in procedures as much as possible. The reason why is simple. If we want to copy the procedure for another program this will be more difficult, becuase in the new program probably will not have the same global variables. Using only local variables in procedures is therefore much better. If you want to use the global variables, pass them as parameters to the procedure. Ideally, a procedure is a stand-alone unit.



One more rule, typically for PASCAL and languages alike (*single-pass compilers*): variables can only be used in places AFTER their declaration in the program, so, if we put the declaration of a variable after a procedure, this procedure cannot use the variable.
Let's take a look at some examples. First a program of lecture 13:

```
PROGRAM WithParameters;
 (* global variables x and z *)
Var x, y: real;

FUNCTION Square(r: real): real;
 (* local variable localr *)
var localr: real;
begin
  localr := r*r;
  x := localr;
  Square := localr;
end;

begin
  x := 4.0;
  y := Square(x);
end.
```

Priority



Variable x is a local and a global variable. Inside the procedure, the local variable will be used.

When local and global variables exist with the same name, the local variable has higher priority and will therefore be used inside the procedure. Anyway, this is confusing, so **always try to avoid using the same identifier again**!

Some languages do not have the difference between local and global variables (for example BASIC). This will mean that we cannot use the same name for a variable twice.

Passing by value or by reference

When passing parameters to procedures or functions, we can do this in two different ways, either passing by value, or passing by reference.

Passing by value:

Until now we have only seen the first type. In this way, only a value is passed to the functions. Whatever we do with that value in the procedure will have no effect on the original value of the variable used in calling the procedure. As an example, to make this more clear. Assume we have a procedure that writes the square of the parameter p. To calculate the square we assign a new value (p*p) to p. The value of p will therefore change inside the procedure:

```
  PROCEDURE WriteSquare(p: real);
  begin
```

```
    p := p*p;
    WriteLn(p:0:1);
  end;
```

When we now call the procedure with a variable x, the value of this variable will not change by calling the procedure. In the main program:

```
  begin
    x := 2.0;
    WriteSquare(x);
    WriteLn(x:0:1);
  end.
```

After returning from the procedure, the value of x has not changed. The total output of the program above will therefore be

```
  4.0
  2.0
```

Passsing by reference:

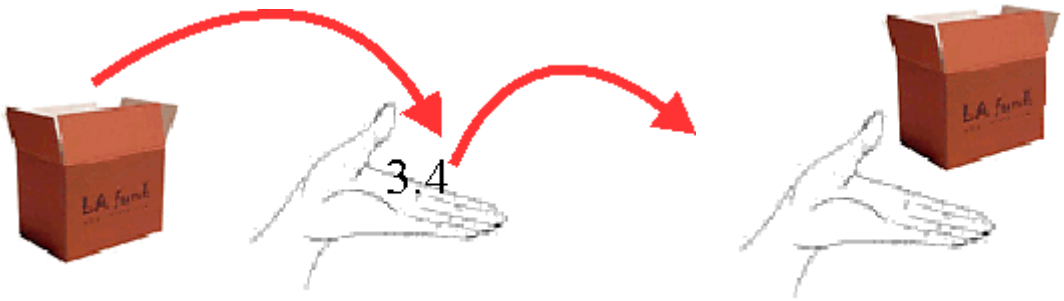
On the other hand, if we do want to change the value of the variable used in calling the procedure, we can specify this at the definition of the procedure by placing the word Var in front of every parameter that we want to change permanently:

```
  PROCEDURE WriteSquare(Var p: real);
  begin
    p := p*p;
    WriteLn(p:0:1);
  end;
```

If we now run the program, the output will be

```
  4.0
  4.0
```

because the value of x has changed simultaneously with the value of the parameter p.



Passing by value *Passing by reference*

In the analogon of boxes visualizing variables: passing by reference is handing over the box (variable) to the procedure which can then use and change the value in the box and at the end handing it back, while passing by value is equivalent to opening the box, copying the value and handing only that value over to the procedure. Obviously, then the original value stays in the box.

Or in another example: I can tell you how much is on my bank account, which you can then use to calculate how much it is in dollars, or I can give you the right to change the amount on my bank account, in which case, the amount will probably change.

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 5 Abril 2002

Lecture 16: Recursive programming

Recursive

A function or procedure is recursive if it is defined in terms of itself (i.e., it calls itself)

Example 1: Factorial

The classic example is the calculation of the factorial function *n*! We might do this with a loop, as described in lectures 11 and 12:

```
FUNCTION Factorial(n: integer): integer;
 (* returns n! *)
var result: integer;
    i: integer;
begin
  result := 1;         (* initialize the variable *)
  for i := 1 to n do
    result := i*result;
  Factorial := result;  (* return result *)
end;
```

This function, indeed, will return the factorial of the argument, for instance `Factorial(5)` = 120. Check this.

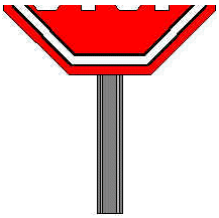
A much more interesting solution is if we define the function Factorial in terms of itself, just as we have learned in school,

$n! = n*(n-1)!$

Let's do exactly that:

```
FUNCTION Factorial(n: integer): integer;
 (* returns n! *)
begin
    (* the value to be returned is expressed in terms of itself: *)
  Factorial := n*Factorial(n-1);
end;
```

This function is already nearly correct. The only problem is that it will never stop calling itself. For instance, we can call it with Factorial(4), which will then try to calculate 4*Factorial(3) and hence call Factorial(3). Factorial(3) will try to calculate



3*Factorial(2), which will call Factorial(2), ... which will call Factorial(1) ... which will call Factorial(0) ... which will call Factorial(−1) ... which will call Factorial(−2) ... and this will never end. The program will never return anything and the computer will crash, probably generating a so-called "stack overflow" error. Clearly we have to build in a way to stop calling itself. Now remember that in the mathematical way of defining a function in terms of itself we also always had to build in a stop, for the Factorial function, this was

$1! = 1$

Let's also built this into our function:

```
FUNCTION Factorial(n: integer): integer;
 (* returns n! *)
begin
  if n=1 then
    Factorial := 1
  else
    Factorial := n*Factorial(n-1);
end;
```

The idea we should learn from this is that we should give it a possibility to come up with an answer and exit the recursive calculation. Just like the way we had to give a loop the possibility to end we should also give this possibilty to recursive functions. If not, the program will continue forever (or crash).

Example 2: Fibonacci

Remember from the practical lessons, the definition of a Fibonacci number is in terms of itsef:

$f_n = f_{n-2} + f_{n-1}$

with the stopping conditions

$f_1 = 1$

$f_2 = 1$

For instance,

$f_3 = 1 + 1 = 2$

$f_4 = 1 + 2 = 3$

$f_5 = 2 + 3 = 5$

$f_6 = 3 + 5 = 8$

$f_7 = 5 + 8 = 13$

We can implement this in a function, note the stopping condition:

```
FUNCTION Fibonacci(n: integer): integer;
begin
  if (n=1) OR (n=2) then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-2) + Fibonacci(n-1);
end;
```

Variables

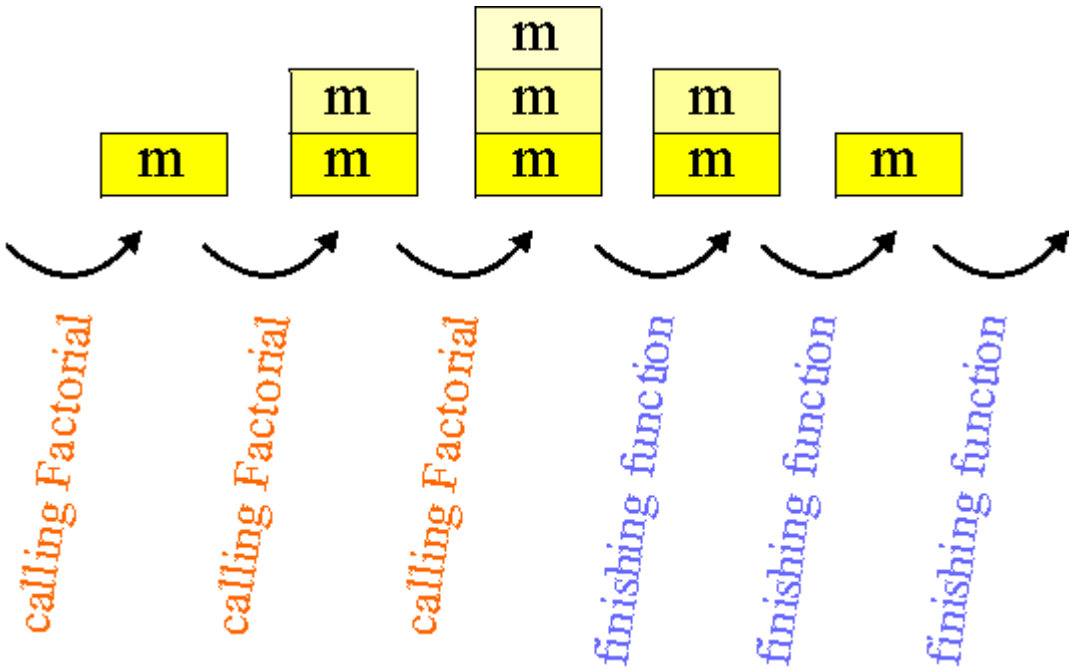
Variables that are declared inside recursive procedures and functions are all local. Moreover, everytime the function is called, a new instance of the variable is created that exists until the procedure finishes. (*in*

C we can prevent this with the word "static" in front of the variable declaration). Each of these variables, although they have the same name, will point to a different space in memory. As an example, let's create a local variable in our Factorial function:

```
FUNCTION Factorial(n: integer): integer;
Var m: integer;
begin
  m := 2*n;
  if n=1 then
    Factorial := 1
  else
    Factorial := n*Factorial(n-1);
  WriteLn(m);
end;
```

When we call this function with argument 3 the following will happen:

```
  Factorial(3) is called
    a variable with name m is created
    2*3 is assigned to it
    Factorial(2) is called
      a variable with name m is created (different than m above!)
      2*2 is assigned to it
      Factorial(1) is called
        a variable with name m is created (different than the two above)
        2*1 is assined to it
        ... we reach the stop condition and Fortorial := 1;
        The value of m is writen: 2
      we exit Factorial(1)
      The value of m is written: 4
    we exit Factorial(2)
    The value of m is written: 6
  We exit Factorial(3)
```



What we learn from this is that the variable `m` is not a static object in memory, pointing to a certain address, but instead, the variable is created at the time we run the program. Each time we enter the function, a new one is created that lives until we exit the function. Moreover, as can be seen in the example above, the last variable created is the one to be used locally.

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 7 Abril 2002

Lecture 17: Arrays

Array

Imagine that we want to write a program to calculate the average of 10 numbers. With the knowledge we gained until now, we could do this by defining ten different variables, for instance

```
Var a1, a2, a3, a4, a5, a6, a7, a8, a9, a10: real;
    average: real;
```

and in the program code:

```
average := (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10;
```

I hope you will agree that this is very cumbersome. And, it could be even worse: imagine we want the user to select how many numbers to use in the calculation of the average:

```
Var n: integer;
   |
ReadLn(n);
Case n of
   1: average := a1;
   2: average := (a1 + a2) / 2;
   3: average := (a1 + a2 + a3) / 3;
        |
  10: average := (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10;
end;
```

For these purposes exist the arrays. An array lets us define a set of variables of the same type with an easy way of access, namely with an index. Just like in mathematics, a_i is the i-th element of vector or series a, `a[i]` gives the i-th element of array `a`.

An array is a set of indexed variables of the same type.

Declaration of an array

To declare an array we use the following syntax:

Var name: array[startindex .. endindex] of type;

`name` is the identifier of the array, just like a name for other variables.
`startindex` and **`endindex`** define the limits of the indices of the array. Note that the start index is not necessarily 0, instead we can let the array start at any index we want. That is nice, because humans are more used to working with indices that start with 1 rather than 0.
`type` is any variable type, for instance real or integer, but it can even be another array as we will see

later.

Examples:

```
Var account: array[1..100] of real;
```
This might be used to store the information of 100 bankaccounts.

```
Var prime: array[1..10] of longint;
```
This might be used to store the first 10 prime numbers..

```
Var propinas: array[1000..2000] of boolean;
```
This might be used to store some boolean information of the status of the students with numbers between 1000 and 2000, for instance if they paid their tuition fees or not.

Use of an array

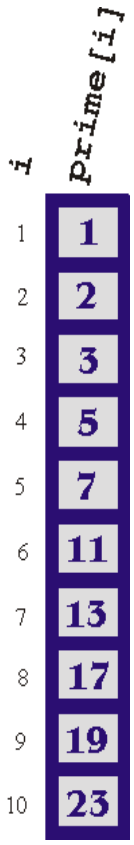
Inside the program we can use the elements of an array.

name[index]

`name[index]` will return the value element number `index` of the array `name`. This is then a value of the type as described in the declaration of the array. Examples of the arrays declared in the previous section:

```
account[20]
```
is the value - of type real - of element 20 of the array with name `account`.

```
prime[8]
```
is the value - of type longint - of element 8 of the array `prime`. The numbers on the right might represent this array, so element number 8 would be equal to 17. Of course, our program has to fill this array in some way before the array really contains the prime numbers.




```
propinas[1055]
```
is TRUE or FALSE (type boolean). Element 1055 of the array `propinas`. Did student 1055 pay his tuition fees? Probably our university administration has somewhere in their computers an array with this information.

We can also use a variable for the index in addressing a single element of an array. Naturally, this variable needs to be of any integer type, because the index is something countable; index 3.4981 does not make sense. Index 3 does, it will address the third element of the array. The following code will show the entire array of 20 accounts:

```
for i := 1 to 20 do
  WriteLn(account[i]);
```

Multiple arrays

Just like in mathematics, where we have vectors (tensors of 1 dimension) and matrices (tensors of 2 dimensions) we can have arrays of 1 dimension or 2 dimensions or even more. We can specify this in the following way, for instance a 'double array' (an array of two dimensions):

```
Var name: array[startindex1 .. endindex1, startindex2 .. endindex2] of type;
```

Technically speaking we can also do it in the way

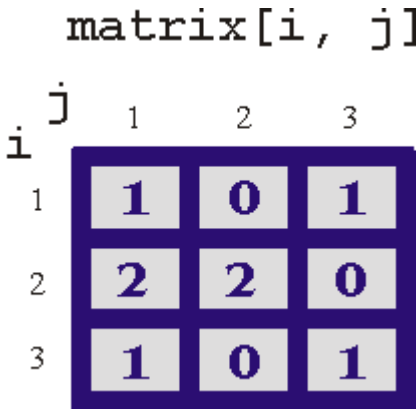
```
Var name: array[startindex1..endindex1] of array[startindex2..endindex2] of type;
```

which nicely shows what we are dealing with, namely an array of arrays. This is how the computer organizes our array. The first form of declaring a double array is preferred though, because it is more logical from a human point of view.

The use of a double array is similar to that of a single array. We separate the indices with a comma, or by putting them in separate square parenthesis:

```
name[index1, index2]
name[index1][index2]
```

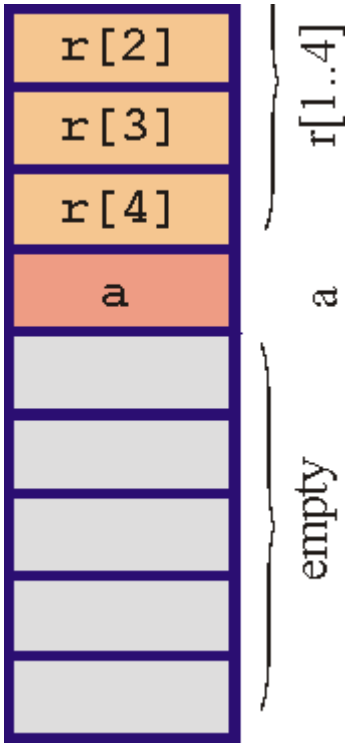
As an example: to write the matrix on the left we might do the following in a complete program. Note that the array consists of 9 (3x3) elements of type integer.




```
PROGRAM ShowMatrix;

Var matrix: array[1..3, 1..3] of integer;

begin
  matrix[1, 1] := 1;
  matrix[1, 2] := 0;
  matrix[1, 3] := 1;
  matrix[2, 1] := 2;
  matrix[2, 2] := 2;
  matrix[2, 3] := 0;
  matrix[3, 1] := 1;
  matrix[3, 2] := 0;
  matrix[3, 3] := 1;
  for i := 1 to 3 do
    begin
       for j := 1 to 3 do
         Write(matrix[i, j],' ');
      WriteLn;
    end;
end.
```

Caution

When we are using an array we also have to be careful not to use an index that is 'out of bounds'. This means that we always have to use an index that



is inbetween the lower index and the higher index limit. If we use a too large or too low index, the results of our program canl be very odd. This is best illustrated in an example. The following program defines an array of 4 integers `r[1..4]` and a normal integer `a`. The figure on the left shows how they might be placed in memory. What will happen when our program assigns a value to `r[5]`? If `r[5]` had existed, it would have occupied the place that is now taken by `a`, and an assignment to `r[5]` would be putting a value in the box of what is now occupied by `a`. Most computer languages don't care and put the value for `r[5]` there anyway, thereby **overwriting the value of `a`**.

```
PROGRAM Test;

Var r: array[1..4] of integer;
Var a: integer;

begin
  a := 0;
  WriteLn('a=',a);
  r[5] := 1;
  WriteLn('a=',a);
  ReadLn;
end.
```

The output of the program will probably be

```
a=0
a=1
```

Some programming languages will check for this at run rime. With our Turbo Pascal compiler we can select the option to verify for such an index-out-of-bounds event at run time. This is called range-checking and when the program tries to use a wrong index, a 'range-check error' will be generated. The disadvantage of doing this is that the program becomes slower and the compiled program will occupy more space in memory and on disk.

"Hurray! I know everything about arrays."

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Lecture 18: Records

Record

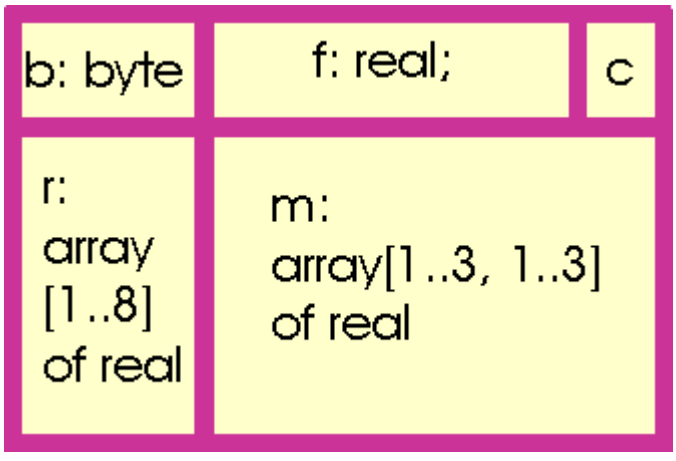
In the last lecture we learned how an array can store variables of the same type in a nicely ordrered, indexed way, like in a file cabinet with in each drawer the same type of information. If we want to group variables together that are not of the same type, we can do this in a **record**.

*The three file cabinets store things of the **same type**, just like **arrays**. The left one could be "bytes", the middle one "integers" and the right one "reals".*

*The file cabinet in the center is used for storing things of mixed type. In the same way, a **record** is used for storing variables of **different type**, integers, real, or whatever, all together in the same box.*

A record is a set of variables of mixed type.

Declaring a record



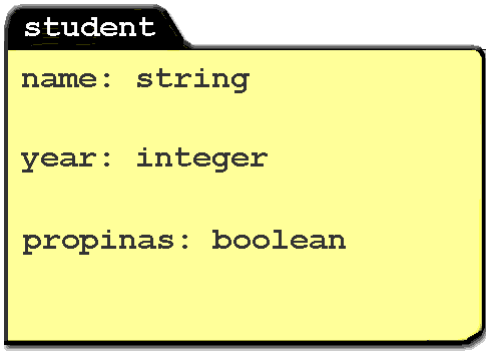
A visualization of a record, namely a motley set of variables. Each variable inside a record is called a field. Here we have 5 fields: a byte (b), a real (f), a boolean (c), a single array of reals (r) and a double array of reals (m).

To declare a record we do the following

```
Var name:
  record
      item1: type1;
      item2: type2;
          |
      itemN: typeN;
  end;
```

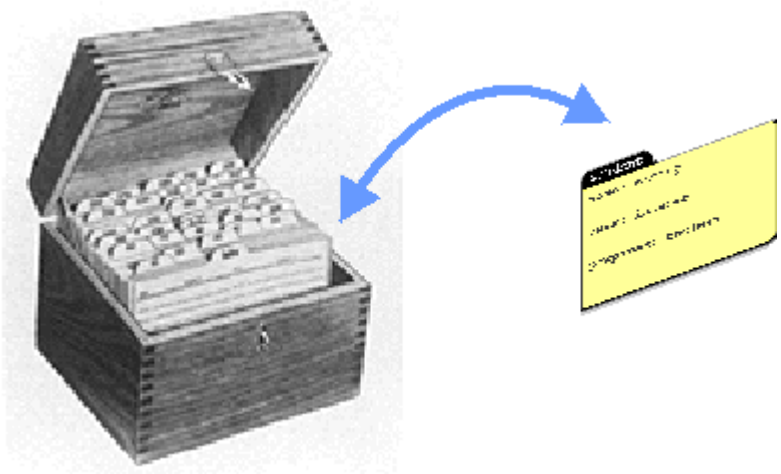
with
name: the name for the variable holding the record.
item1..itemN: the name for the fields in the record. These have the same rules as the other identifiers for variables, constantes, procedures, etc. Note that we can put as many fields in the record as we want, with any combination of types.
type1..typeN: the type of the fields of the records.

As an example, the definition of a record containing the information of a student might have fields for name, year, and tuition fees paid:




```
Var student:
  record
    name: string;
    year: integer;
    propinas: boolean;
  end;
```

This record can only contain the information of a single student. With the information of the previous lecture (lecture 17), however, we know how to build an array that can store the information of may equal objects of any type, even records. let's build an array of 1000 students:



```
Var students: array[1..1000] of
  record
    name: string;
    year: integer;
    propinas: boolean;
  end;
```

Later we will see that we can much more easily do this with a new type definition (see lecture 19).

Using a record

To access a record we use the format

name.field

For example, to assign values to the record `student` we can do the following

```
student.name := 'Peter Stallinga';
student.year := 2002;
student.propinas := TRUE;
```

or in the example of the array of records `students`:

```
i := 1055;
students[i].name := 'Peter Stallinga';
students[i].year := 2002;
students[i].propinas := TRUE;
```

Note the structure of arrays of records. `students` is an array of records, therefore, `students[i]` is one of these records and if we want to assign something to a field we use the period and the fieldname, so `students[i].name` is a string containing the name of student number i.
Wrong syntax would be `students.name[i]` (we could use this if we had a single record `students` containing a field `name` that is an array)
Also wrong: `students.[i]name`, which doesn't make sense at all.

Another example:

```
Var coordinate:
  record
    x: real;
    y: real;
  end;

coordinate.x := 1.0;
coordinate.y := 0.0;
```

This is not exactly a set of variables of mixed type, so in principle we could also do this with an array:

```
Var coordinate: array[1..2] of real;

coordinate[1] := 1.0;
coordinate[2] := 0.0;
```

but, the first version, with the record is more logical.
Another example:

```
Var address:
  record
    street: string;
    housenumber: integer;
    andar: integer;
    porta: char;
  end;
```

```
address.street := 'Rua Santo Antonio';
address.housenumber := 34;
address.andar := 3;
address.porta := 'E';

writeln(address.street, '  ', address.housenumber);
writeln(address.andar, address.porta);
```

In the above example we had to write many time the word "address". To save time, in PASCAL exists the combination with recordname do, which means that in the instruction after it (everything between begin and end) the variables (that the compiler doesn't know otherwise!) get the recordname in front of it. Therefore, the above two lines of code can be rewritten in a more readable form as

```
with address do
   begin
     writeln(street, '  ', housenumber);
     writeln(andar, porta);
   end;
```

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 15 Abril 2002

Lecture 19: Defining new types

Type

Sometimes it is nice to be able to define a new type of variable to be used later in the program. Just to have a more readable code, or to avoid having to retype code many times. Defining new varibale types can be done with the word type.

Type typename = description;

with `typename` the name we want to give to the type and `decription` any type of variable we have learned until now, including arrays, records and all the simple variable types. It can also be of the type pointer which we will learn in the next lesson.

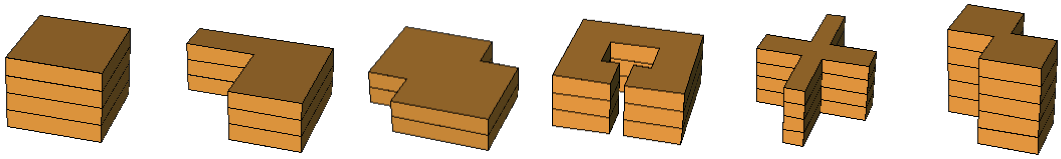
Examples:

```
Type float = real;
```

This is useful for people that are used to programming in C. After writing the line above, we can use variables of type float, just like in C.

```
Type realarray = array[1..10] of real;

Type myrecord =
  record
    name: string;
    length: real;
    width: real;
    height: real;
  end;
```

Note that this definition of a new type does not create a variable! It does not save space in memory and it does not assign a name to a variable. It is just a description of a type that we can use later in declaring a variable.



... defining new boxes for variables.

Using a new type

After the definition of a type, we can declare variables of this type:

Var varname: typename;

with `varname` the name of a new variable and the type of this variable is `typename`, as decribed before. After the declaration we can use the variable as if it were declared in the normal way.

Examples:

```
Var f: float;
```

This looks already much more like C (*in C it would be "`float f`"*)

```
Var ra: realarray;
```

And in the code we can use this array:

```
ra[1] := 2.68;
```

This is completely equivalent with

```
Var ra: array[1..10] of real;
ra[1] := 2.68;
```

The last example

```
Type myrecord =
  record
    name: string;
    length: real;
    width: real;
    height: real;
  end;
Var mydata: array[1..100] of myrecord;
mydata[23].length := 3.1;
```

More examples

```
PROGRAM WithTypeDefinition;

Type ra = array[1..6] of integer;
Var x: ra;
    y: array[1..7] of integer;

FUNCTION AreEqual(r: ra): boolean;
  (* Note that the definition can also be used for parameters *)
begin
  if r[1]=r[2] then
    AreEqual := TRUE;
  else
    AreEqual := FALSE;
end;

begin
  x[1] := 1;
  x[2] := 0;
  WriteLn(AreEqual(x));
  y[1] := 1;
  y[2] := 0;
   (* The following line of code is not allowed because the type of y and the type the
```

```
function 'AreEqual' expects are different. Look at the size of the arrays. *)
  WriteLn(AreEqual(y));
end.
```

```
PROGRAM WithTypeDefinition;

Type time =
  record
    hour, minute, second: integer;
  end;

PROCEDURE ShowTime(t: time);
  (* Will show the time in format h:m:s *)
begin
  WriteLn(t.hour, ':', t.minute, '.', t.second);
end;

Var atime: time;

begin
  atime.hour := 23;
  atime.minute := 16;
  atime.second := 9;
  ShowTime(atime);
end.
```

---

record of records:

```
Type date =
  record
    day, month, year: integer;
  end;
Type time =
  record
    hour, minute, second: integer;
  end;
Type dateandtime =
  record
    dattime: time;
    datdate: date;
  end;
Var x: datendtime;

x.dattime.hour := 1;
```

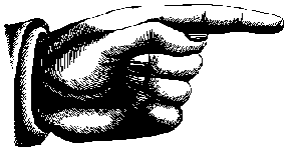
`x` is a record containing two fields. One field is `dattime` which is a record of three fields. One of these fields is `hour`.

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Lecture 20: Pointers

Pointer



A pointer is a special type of variable. In itself it contains no useful information. It is only an address to what (might) contain useful information.

A pointer contains the address of a place in memory

This is like keeping the address of an appartment in my addressbook. It is only an address and nothing more.
To declare a pointer we can use the following syntax:

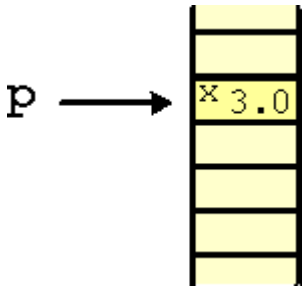
```
Var p: pointer;
```

Pointer instructions

There are two pointer operations in PASCAL

@x or **Addr(x)** returns a pointer to variable **x**

p^ is what pointer **p** points to



In the example on the left, `x` is a variable of type real that has a value of 3.0. When we want the pointer `p` to point to this variable, we can use

```
p := @x;
```

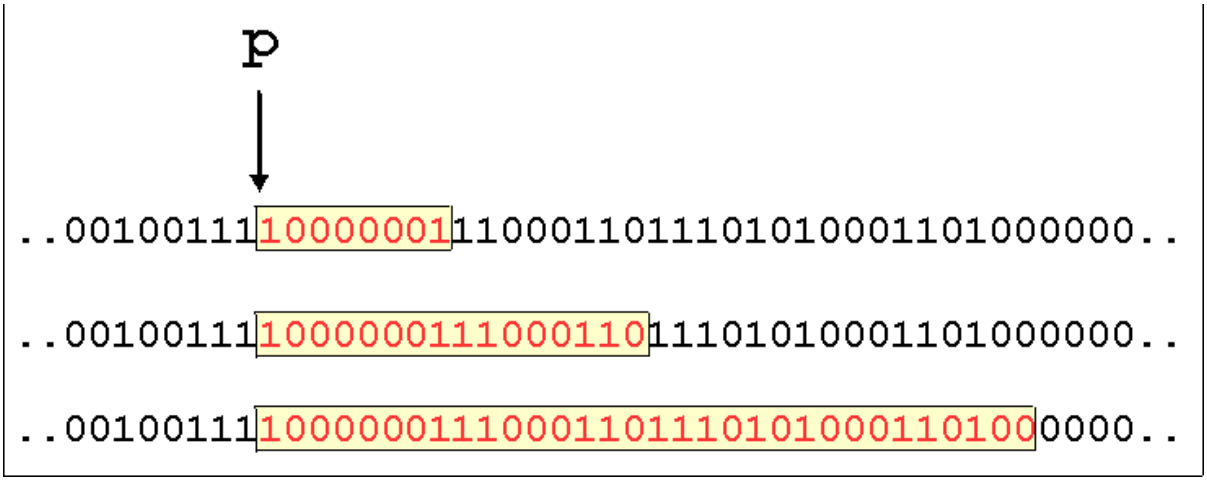
or

```
p := Addr(x);
```

The value of `p` is now a memory address, namely the address that contains the variable `x`. To show the contents of the memory of what p points to, we might think that we can do this with

```
WriteLn(p^);
```

There is only one thing wrong with this. The compiler only knows that `p` is of type pointer. When we run the program we let it point to a certain place in memory, but the program doesn't know what (type of information) the pointer is pointing to. Picture the following situation. A pointer `p` points to a place in memory. If it is pointing to a byte, the value is different than if it is pointing to an integere or a word. In the example below, `p^` pointing to a byte is 129 (binary: 10000001), while the same `p^` pointing to the same place in memory, but pointing to a word is equal to 25473 (binary: 0110001110000001), or `p^` pointing to a longint (4 bytes, 32 bits) will give 743924609 (binary: 00101100010101110110001110000001). (Note that in Intel-processor-based computers numbers are stored with their lowest value bit [LSB=least significant bit] first).



Therefore, we have to specify the type the pointer is pointing at. For that we have to go back to the declaration of the pointer. Instead of just using 'pointer' for the type, we will specify what kind of variable the pointer will point to:

Declaring a pointer

To declare a pointer we can use the following forms:

```
Var p: pointer;
Var p: ^type;
```

The first form declares a general pointer, without specifying what type of data it will point to.
The second form also specifies the type of data it points to. For type we can use any type we have learned. From the simple variables (integer, real, etc) to the complicated types (record, array), to combinations of these (arrays of records, records of arrays).
Examples:

```
Var byteptr:  ^byte;
    wordPtr:  ^word;
    real6arrayptr: ^array[1..6] of real;
```

Now let's see an example to show check if we have everything under control. We are going to create a pointer of type 'pointing to a word', and let it point to a word:

```
Var wordptr: ^word;
    w: word;

begin
   (* assign a value to the word *)
  w := 25473;
   (* let a 'pointer to word' point to our word *)
  wordptr := Addr(w);
   (* show the contents of the memory wordptr points to *)
  WriteLn(wordptr^);
end.
```

output:

```
25473
```

Now let's see a more complicated example. We are going to create a pointer of type 'pointing to a **byte**',

and let it point to our **word**:

```
Var byteptr: ^byte;
    w: word;

begin
   (* assign a value to the word *)
  w := 25473;
   (* let a 'pointer to word' point to our word *)
  byteptr := Addr(w);
   (* show the contents of the memory wordptr points to *)
  WriteLn(byteptr^);
end.
```

output:

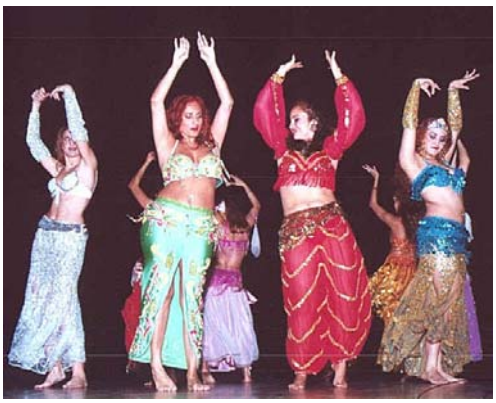
```
129
```

This shows that we have to be careful what our pointer points to. The value depends on the type!

Why?

Why use pointers? There are several reasons to use pointers instead of normal variables. The most important ones are

- Rapidez



- Flexibilidade

Speed: Imagine you write a procedure that has an array as parameter. Every time the program calls the procedure, the entire array has to be copied in memory and the procedure executed. If, instead of telling the procedure the value of every element of the array, it would be much faster to only tell the address of the array. This means copying only a single variable (a pointer is just 4 bytes in Intel computers).

```
PROGRAM TestSpeed;

Type ra = array[1..1000] of real;
Var r: ra;
    i: longint;

PROCEDURE SlowProc(a: ra);
begin
  a[1] := 1.0;
end;
```

```
PROGRAM TestSpeed;

(* define an array and a pointer to
  that arrray: *)
Type ra = array[1..1000] of real;
     rap = ^ra;
Var r: ra;
    i: longint;

PROCEDURE FastProc(a: rap);
begin
```

```
begin
  for i := 1 to 4000000 do
    SlowProc(r)
end.
```

execution time on a Pentium II 450 MHz: **115** s.

```
  (* a is a pointer to an array *)
  (* a^ is what it points to, the *)
  (* the array. a^[1] is the first *)
  (* element of that array *)
  a^[1] := 1.0;
end;

begin
  for i := 1 to 4000000 do
    FastProc(@r)
   (* pass only a pointer to *)
   (* the procedure *)
end.
```

execution time on a Pentium II 450 MHz: **1** s.

(In fact, the program on the right does the same as a program that passes the array by reference (see lecture 15). `PROCEDURE SlowProc(Var a: ra);` is also fast. Passing by reference means that a pointer is passed to the procedure.)

Flexibility: If, at the beginning of the program we do not know yet how many variables we need we would have to reserve space for all possible eventualities. If we want to write a program that calculates the first N prime numbers, with N given by the user, we would have to declare an array of maximum size to be sure that we can fit the users request in it. Something like this:

```
Var prime: array[1..10000000] of longint;
```

With this we would completely occupy the memory of the computer. Nothing else can run anymore. Much nicer would be if we could declare the array (the variables) dynamically so that we only use memory if we really need it. With pointers this is easily possible.

The pointers and the idea of dynamic creation of variables also lies at the basis of object-oriented programming, which is the type of programming of every modern computer language. Object oriented programming is outside the scope of this lecture, though.

NIL

A pointer that is not pointing to anything has the value NIL. NIL is a predefined constant in PASCAL.

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 23 Abril 2002

Lecture 21: Files

Output to File

Today we are going to learn how to read from files and write to file. As an example we will only learn how to read and write text files, files where the infomation is stored in ASCII format. Such files differ from binary format because they are also readable by humans. We already are using with text files, because all our programs written in the practical lessons are of this type.
These files can be placed on floppy disks, on the harddisk, or even on CD-ROMs (in which case they can - of course - only be read and not written).

Instructions

The following instruction are related to file access:

```
Var text
Assign
Rewrite
Reset
Close
Read, ReadLn
Write, WriteLn
Eol, Eof
```

Declaring a variable for file access:

Before we can open a file and have access to it (either reading or writing) we have to declare the existence of a file. In PASCAL we can declare a text file in the following way:

```
Var filehandle: text;
```

With `filehandle` the logical name of the file. This is not equal to the actual name of the file, as we will see in a moment. The place to declare this is together with the files.
Example:

```
Var f: text;
```

Assigning a name to the file

Inside the program we should assign an actual name to the file before we can open it:

```
Assign(filehandle,filename);
```

The `filehandle` is the one declared above and the `filename` is a string (constant or variable) containing the name of our file. For example:

```
Assign(f, 'MYFILE.TXT');

ReadLn(s);
Assign(f, s);
```

Input or output?

To open the file, we can use two forms, depending if we want to open the file for input (reading the file) or output (writing to the file):

```
Reset(filehandle);
Rewrite(filehandle);
```

For example:

```
Reset(f);
Rewrite(f);
```

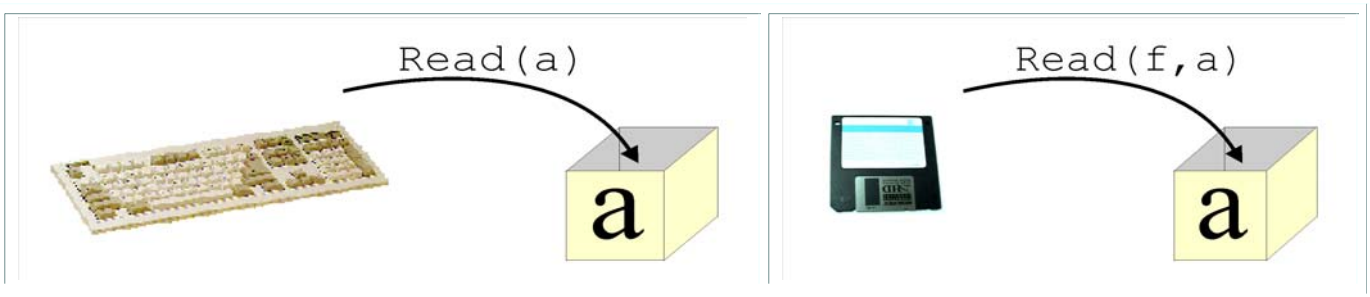
Reading and Writing

The reading from file and writing to file are now the same as if we were reading from the keyboard or writing to the screen. We use the same instructions (Read, ReadLn, Write, and WriteLn). The only difference is that the first parameter has to be the file we have just opened:

```
Read(filehandle,...);
ReadLn(filehandle,...);
Write(filehandle,...);
WriteLn(filehandle,...);
```

Note that we can only use the `Read` and `ReadLn` instructions for files that have previously been opened for input and `Write` and `WriteLn` are only to be used for files opened for output. Examples:

```
WriteLn(f, r:0:2);
ReadLn(f, opcao);
```



Read a variable with the keyboard	Read a variable from file

Closing the file

When we are ready with the file, we must close it. This is especially the case for output files. If we forget to close the file before ending the program, probably not all the information will be written to the file. To close the file we use

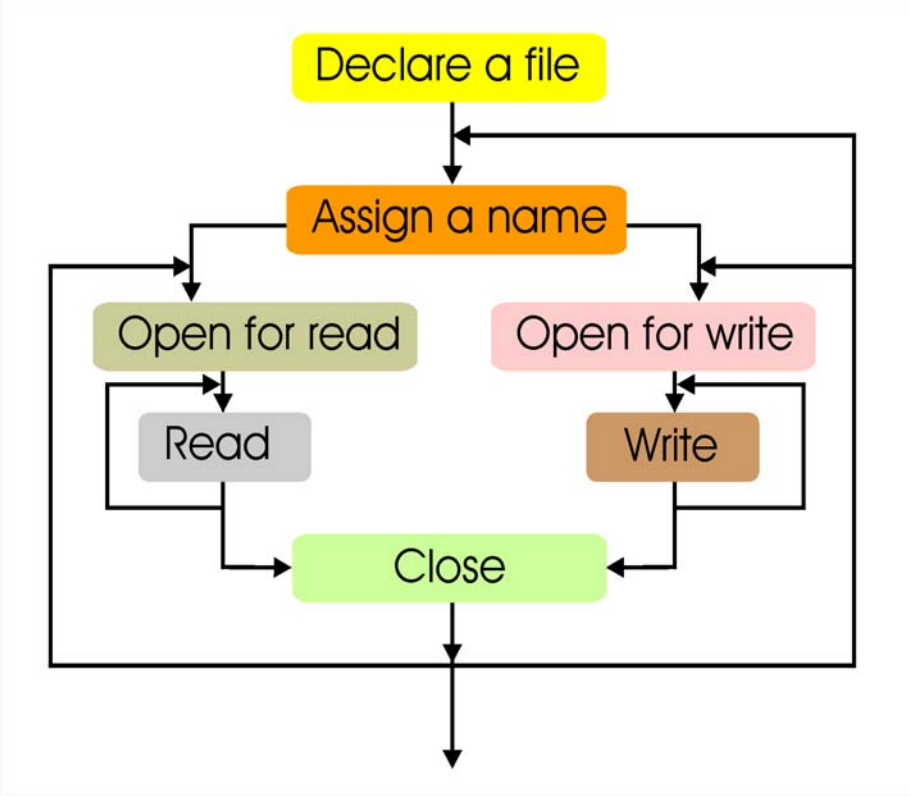
```
Close(filehandle);
```

for example:

```
Close(f);
```

Summary

File input and output consist of the following steps:



Eol, Eof

Two instructions can be useful

Eol(filehandle): returns true if we are reading at the end of a line in the file

Eof(filehandle): returns true if we are reading at the end of the file.

Example:

```
While NOT Eof(f) do
```

```
ReadLn(s);
```

which will read from the file until the end of the file is encountered.

Examples

code	*screen*	*file TEST.TXT **after** running the program*

`PROGRAM WithFileOutPut;`
 	
`Var f: text;`
`  s: string;`
`  i: integer;`
 	
`begin`
`  WriteLn('Name of File:');`
`  ReadLn(s);`
`  Assign(f, s);`
`  Rewrite(f);`
`  for i := 1 to 10 do`
`  WriteLn(f, i, ' Hello');`
`  Close(f);`
`end.`
	`Name of File:`
	`TEST.TXT`
		`1 Hello`
		`2 Hello`
		`3 Hello`
		`4 Hello`
		`5 Hello`
		`6 Hello`
		`7 Hello`
		`8 Hello`
		`9 Hello`
		`10 Hello`

code	*screen*	*file TEST.TXT **before** running the program*

`PROGRAM WithFileInPut;`
 	
`Var f: text;`
`  c: char;`
`  s: string;`
`  i: integer;`
 	
`begin`
`  WriteLn('Name of File:');`
`  ReadLn(s);`
`  Assign(f, s);`
`  Reset(f);`
`  While NOT Eof(f) do`
`  begin`
`  (* read a character`
`  from file: *)`
`  Read(f, c);`
`  (* show it on the screen: *)`
`  Write(c);`
`  end;`
`  Close(f);`
`end.`
	`Name of File:`
	`TEST.TXT`
	`1 Hello`
	`2 Hello`
	`3 Hello`
	`4 Hello`
	`5 Hello`
	`6 Hello`
	`7 Hello`
	`8 Hello`
	`9 Hello`
	`10 Hello`
		`1 Hello`
		`2 Hello`
		`3 Hello`
		`4 Hello`
		`5 Hello`
		`6 Hello`
		`7 Hello`
		`8 Hello`
		`9 Hello`
		`10 Hello`

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 30 Abril 2002

Lecture 22: Algorithms

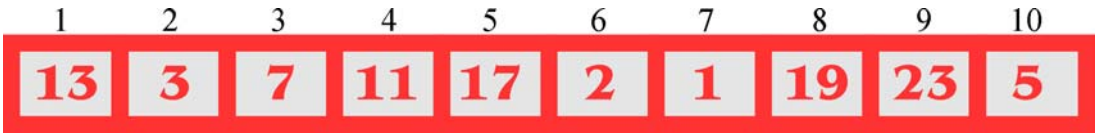
Algorithm

The term algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa Al-Khowarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850 AD.

(http://searchvb.techtarget.com)

Sorting

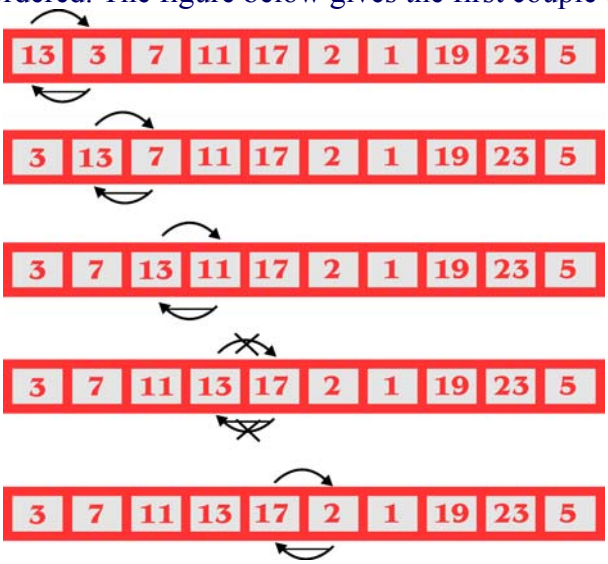
For our lectures it means that before we even touch the computer we are going to design a method for solving the problem. As an example we use the problem of sorting of an array. Imagine we have an array of integers of determined length N and we want to sort them. How we are going to do that?



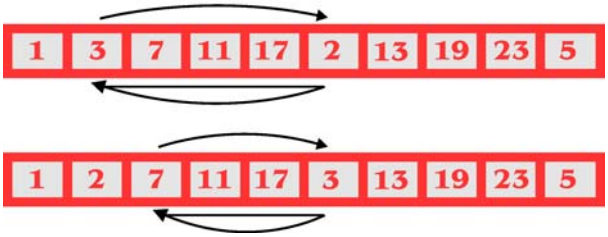
The figure shows an array of 10 elements that we are going to sort.

Without looking at the details, we for example can have the ideas:

1. Go through the list. If two consecutive elements are not ordered we will exchange them. Repeat doing this until the list is ordered. The figure below gives the first couple of steps.

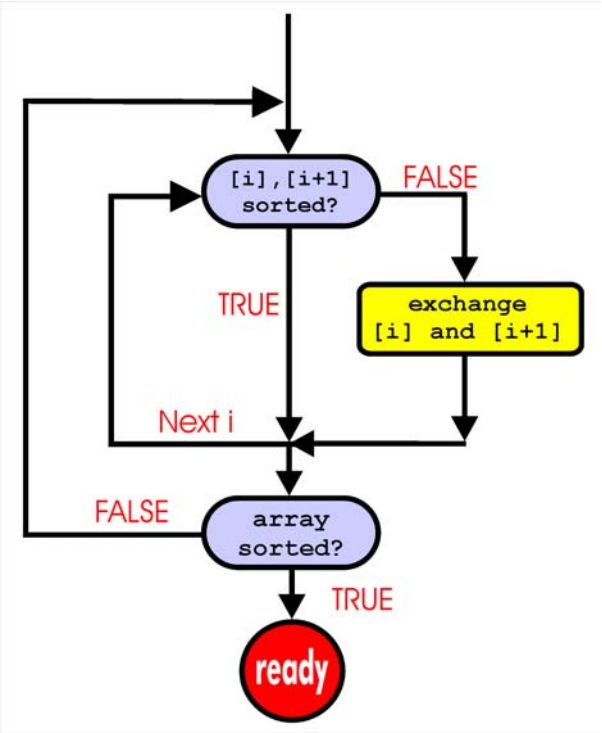


2. Go through the list, find the smallest element. Put this in the first place. Then find the smallest of the rest. Put this in the second place, etc. Repeat this N times. The figure below shows the fist couple of steps for our array.



Bubble sort

These are two examples of algorithms. The first one is normally refered to as "Bubble sort", because it resembles the slowly moving up of air bubbles in water. Let us implement this algorithm in PASCAL. Let's first put the idea into a so-called flow diagram.



Flow diagram of the Bubble sort algorithm. [i] signifies array element i.

In this flow diagram we can already distinguish a little bit of programming. We can see at least two loops: one with i to check if two consecutive elements are ordered and one loop which repeats until the entire array is sorted. Apart from tis there is an instruction for exchanging two elements.
Now we are going to look at the details. We will program the three distinct parts of the figure above. Assume we have an array `n[1..N]` as presented in the figure above.

1) To check if two consecutive numbers are ordered:

```
if n[i] < n[i+1] then ...
```

2) To exchange two numbers is also not difficult. We can do this in the following way:

```
n[i] := n[i] + n[i+1];
n[i+1] := n[i] - n[i+1];
n[i] := n[i] - n[i+1];
```

Although this method really exchanges two numbers (check this), it is a little bit clumsy. Normally we exchange two numbers by using a new variable for temporarily storing information:

```
temp := n[i];
n[i] := n[i+1];
n[i+1] := temp;
```

3) To check if the entire array is ordered we need to use a variable that saves the information if any exchange took place in the previous cycle of checking all consecutive pairs. For this we declare a variable `change` that will be `TRUE` when there was an exchange in the last pass of the array.

With this, the complete procedure becomes:

```
repeat
  change := FALSE;  (* no changes so far in this pass *)
  for i := 1 to N-1 do (* check all consecutive pairs: *)
    if n[i]>n[i+1]  (* not ordered? *)
      begin
          (* exhange the two numbers: *)
        temp := n[i];
        n[i] := n[i+1];
        n[i+1] := temp;
          (* and signal there was a change: *)
        change := TRUE;
      end;
until NOT change;
```

Speed

In the analysis of our algorithm we can also say something about the speed, the efficiency of our method. What are the minimum, maximum and average execution times. If we consider that the execution is proportional to the number of times an element is read, we can say the following:

The Bubble sort algorithm has to

- **minimally** has to go once through the array, hence read 2*(N–1) numbers
- **maximally** has to go N–1 times through the array, hence read $2*(N-1)^2$ numbers.
- on the **average**: N^2–N (the average of the two numbers above)

with N the array size.
In comparison, the other algorithm given in this chapter would have

- **minimally** has to go through the array N times, but each time read a little less (the first time it has to read all N elements, the second time only starting from element 2, then from element 3, etc.), hence read N + (N–1) + (N–2) + (N–3) ... 1 = $N^2/2$ numbers
- The program always has to read the same amount of numbers, independent of the distribution of the numbers. Therefore, **maximally** it also has to read $N^2/2$ numbers.
- on the **average**: $N^2/2$ (the average of the two numbers above)

Which of the two algorithms is more efficient? For large numbers, for example 100, the second one becomes more and more efficient (5000 vs. 9900). Also note that in the Bubble sort algorithm we have more exchanges of numbers. This is easy to see why. If a number is all the way at the other end of the

array, it has to move its way slowly up to the other side. This will take N−1 exchanges. In the other algorithm, the value is copied only once. In fact, Bubble sort is one of the most inefficient algorithms. Probably the best is Quick-sort (which falls outside the scope of this lecture). The reason to use Bubble sort is that it is a very elegant algorithm and very useful for explaining sorting or algorithms in general.

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.

Peter Stallinga. Universidade do Algarve, 6 Maio 2002

Keywords

word	description	PASCAL	example
module	program within the program designed to do a specific task.	`Procedure` `Function`	`Procedure Square(x:real);` `Function Square(x: real): real;`
declaration	reserving space for a variable and associating a name to it	`Var`	`Var x: real;`
constant	like variable but cannot change value	`Const`	`Const x = 3.0;`
assignment	assigning a value to a variable	`:=`	`x := 3.0;`
variable	a place in memory with a name used to store information of a specified type		`x`
condition	any expression that returns a value of type TRUE/FALSE. normally used for branching		`(x<0)`
expression	a calculation producing a value		`3*a + 2*b + c`
instruction	instruction to tell the computer what to do		`a := 2*Sqrt(3.0);`
operator	any symbol that operates on values and produces another value		`+`
operand	what the operator works on		`3.0`
operation	operator plus operands		`3.0 + a`
program	complete set of instructions		
local variables	variables that can only be used inside a procedure or function		
global variables	variables that can be used throughout the program		
scope of a variable	the places where a variable can be used (local or global)		
compiler	the program that translates our (PASCAL) code to machine language		
machine language	translated program consisting of instructions for the CPU (processor)		
output	results shown by computer (normally to monitor or file)		
input	data fed into the program (normally from keyboard or file)		

argument	parameter passed to function or procedure, equal to parameter.		
parameter	see argument		
type	type of variable, etc.		boolean, integer, word.
branching	deciding which part of a program to execute based on a condition		
memory	place to store the program and the variables		

Programming skills

Here follows a summary of good programming skills:

- Use procedures and functions whenever
  - the same thing has to be done in several places of the program.
  - it makes the program more readable.
- Make these procedures and functions independent of the rest of the program. Therefore, also
- Don't use global variables in procedures or functions.
- Don't use names for local variables equal to names for global variables.
- Use meaningful names for variables, constantes, functions and procedures.
- Write (* comment *)
- Use indentation. This makes the program more readable and easier to debug.
- Don't change the value of the control variable in for-loops. If you want to do this, use another form of loop (do-while, repeat-until)

Peter Stallinga. Universidade do Algarve, 16 Maio 2002

Quick Test 2: Computers

1. The first computer was designed by

- Bill Gates for Microsoft
- Blaise Pascal
- Charles Babbage
- IBM

2. The computer most people have at home is of the type

- Supercomputer
- Mainframe
- Minicomputer
- Microcomputer
- Micro processor

3. Indicate for each piece of hardware what function it has

	input	output	storage	processing
Mouse	○	○	○	○
Keyboard	○	○	○	○
Memory	○	○	○	○
Monitor	○	○	○	○
Printer	○	○	○	○
CPU	○	○	○	○

4. To translate a PASCAL program into something the computer understands we use

- A compiler
- A dictionary
- An Operating system
- A Hard disk

Quick Test 3: Units of Information / Memory

1. The smallest unit of information is called

- A bit
- A byte
- A nibble
- An integer

2. The smallest unit of information that can be addressed independently is

- A bit
- A byte
- A nibble
- An integer

3. 1101 in the binary system is in the decimal system equal to

- 1101
- 15
- 13
- D

4. 2A in the hexadecimal system is in the decimal system equal to

- 2A
- 42
- 20
- 0

5. The most common way to code text is

- binary
- hexadecimal
- decimal
- ASCII

6. How much information can be stored approximately on a standard floppy disk

- 1 byte: one ASCII letter
- 1 kilobyte (1 kB): a quarter of a page in ASCII format
- 1 megabyte (1 MB): a book in ASCII format
- 1 gigabyte (1 GB): a small library in ASCII format

Quick Test 4: Introduction to PASCAL

---

1. Indicate for each of these identifiers if they are valid

	valid	not valid	explanantion
birthday8	○	○	
1shot	○	○	
hot?	○	○	
OLD_TIME	○	○	
down.to.earth	○	○	

---

2. In PASCAL we write comment

- ○ after "REM"
- ○ after "//"
- ○ in between "{" and "}"
- ○ after "comment"

Quick Test 5: Variables

1. The difference between `write` and `writeln` is

- `write` writes to screen, `writeln` to the printer.
- `write` is used in C, `writeln` in PASCAL.
- `writeln` puts the cursor on the next line.
- `writeln` is used for formatted output.

2. To store complete values we use variables of the type

- boolean
- byte, integer, or word
- real or double
- string

3. The range of an integer is

- 0 .. 255
- 0 .. 65535
- -32768 .. 32767
- -2147483648 .. 2147483647

4. Declaring a variable means

- Reserving space in memory and associating a name to it.
- Assigning a name and a value
- Initializing a variable
- Showing its value on the screen

5. Variables

- are all set to 0 at the beginning of the program
- are declared by their first use
- have to be assigned a value at the time of declaration
- have unpredictable values at the start of the program

6. For highest-precision floating-point calculations, we use variables of the type

- boolean
- real
- longint
- extended

Quick Test 6: Assignment and Constants

1. When we want to assign a value of 8.3 to a variable `r` we do this with

- `r := 8.3;`
- `r = 8.3;`
- `r == 8.3;.`
- `8.3 -> r;`

2. After the assignment of question 1, which of the following lines of PASCAL will produce

```
8.3000
```

- `writeln(8.3000);`
- `writeln(6*r,4);`
- `writeln('%6.4f', r);`
- `writeln(r:6:4);`

3. What is wrong with the following program?

```
PROGRAM Error1;

VAR x: real;
CONST c = 1.0;

begin
  x*x := 2*c;
end.
```

- A constant cannot change a value
- The left side of the `:=` can only contain a (single) variable
- Variable `x` is not well defined
- The right side of `:=` cannot contain constants

4. What is wrong with the following program?

```
PROGRAM Error2;

VAR x: real;
CONST c = 2.0;

begin
  c := 2*x*c + 1;
end.
```

- A constant cannot change a value
- The right side of the `:=` can only contain a (single) variable
- The equation does not have a solution
- Constants must be written with CAPITALS

5. What is the output of the next program

```
PROGRAM Variable;

VAR x: real;
CONST C = 1.0;

begin
  x := C + 1;
  x := 2;
  x := x + 3;
  writeln(x:4:1);
end.
```

- 7.0
- 5.0
- 3.0
- 1.0

Quick Test 7: Input and Math

1. The difference between `Read` and `ReadLn` is

- `ReadLn` is used for reading from file.
- `ReadLn` discards the rest of the information on the line
- `ReadLn` reads formatted values
- `ReadLn` is used for constants, `Read` is used for variables

2. What is the result of the expression "`33 Mod 2`"?

- 1.5
- 16
- 16.5
- 1

3. What is the result of the expression "`33 Div 2`"?

- 1.5
- 16
- 16.5
- 1

4. What is the result of the expression "`1.0 + 2.0 * 3.0 - 6.0 / 2.0`"?

- 4.0
- 1.5
- 14
- 9.0

Quick Test 8: if ... then ... else

1. Which is <u>not</u> a valid comparison in PASCAL:

- `(a == b)`
- `(a <> b)`
- `(a = b)`
- `(a > b)`

2. The proper syntax for simple branching is

- `if condition`
  `instruction;`
- `case condition`
  `instruction;`
- `if condition then`
  `instruction;`
- `case condition of`
  `instruction;`

3. What is displayed when the following program is executed?

```
PROGRAM IfThenElse;

Var a, b, c, d: integer;

begin
  a := 5; b:= 3; c := 99; d := 5;
  if a>6 then Write('A');
  if a>b then Write('B');
  if b=c then
    begin
      Write('C');
      Write('D');
    end;
  if b<>c then Write('E') else Write('F');
  if a>=c then Write('G') else Write('H');
  if a<=d then
    begin
      Write('I');
      Write('J');
    end;
end.
```

Check I give up

4. To declare a constant called PI with a value of 3.1415927

- Constant PI = 3.1415927;
- Const PI = 3.1415927;
- Const PI 3.1415927;
- Constant PI 3.1415927

Quick Test 9: Boolean Algebra / Case ... Of

1. What will be the output of the following program

```
PROGRAM Test;

Var a, b: real;
Const c = 10.0;

begin
  a := 9.0;  b := 2.0*c;
  if (a>0) XOR (b>0) then
    Write('Fixe!')
  else
    Write(' Uma pena');
end.
```

- `Fixe!`
- `Fixe! Uma pena`
- `Uma pena`
- the program doesn't have output!

2. What is wrong with the following program

```
PROGRAM Test;

Var a: real;
Const C = 2;

begin
  a := 3.0;
  Case a+1.0 Of
    1: Write('Fixe!');
    C: begin
         WriteLn('Cool!');
         WriteLn('Ingles');
       end;
    3: Write('Super!');
    else Write('Language?');
  end;
end.
```

- `Case .. Of` cannot contain expressions (`a+1.0`)
- `Case ... Of` cannot have expressions of type real (`a+1.0`)
- In the structure `Case ... Of` we cannot use constants (`C`)
- In the structure `Case ... Of` we cannot use `else`

3. What will be the result of the Boolean calculation `(43 AND 33)`?

Check Help I give up

4.

```
(3*4 + 12/6*i - j*2)
```

is an exemplo of

- an expression
- a condition
- an assignment
- an operation

Quick Test 10/11: Loops

1. In which type of loop is the instruction executed at least once?

- For
- While-Do
- Repeat-Until
- Such a loop doesn't exist

2. We want to write a program that asks the user to supply a number. The program then should show all the prime numbers up to that number. In this case, the best loop to use is

- For
- While-Do
- Repeat-Until
- Other structure

3. What are the two basic rules for nesting of loops?

1:
2:

Help Correct answer

4. What is the diference between loops of type While-Do and Repeat-Until?

- While-Do is for integer numbers, Repeat-Until is for variables of floating point type.
- Repeat-Until is for integer numbers, While-Do is for variables of floating point type.
- In loops of type Repeat-Until the condition is checked in the beginning, whereas in loops of type While-Do the condition is checked at the end.
- In loops of type While-Do the condition is checked in the beginning, whereas in loops of type Repeat-Until the condition is checked at the end.

5. What is wrong with the following code?

```
x := 0.0;
while (x<10.0) do
  begin
    y := x*x;
    z := x*y;
    writeln('The square of ',x:0:2, ' is
', y:0:2);
    writeln('The cube of ',x:0:2, ' is ',
z:0:2);
  end;
```

- This loop will never finish
- We should use a loop of Repeat-Until instead.
- We should use a For-loop instead.

6. We want to write a program that asks the user to choose a type of calculation or to exit the program (1=adding, 2=subtracting, 0=finish). It has to continue doing this forever (except, of course when the user selects 0). In this case, the best loop to use is

- For
- While-Do
- Repeat-Until
- Other structure

- The condition cannot contain variables of type real.

Quick Test 12,13: Modular Programming

1. What two types of modules do you know in PASCAL?

1:
2:

Correct Answer

2. What are the advantages of writing modules

1:
2:

Correct Answer

3. What will be the result of the next program?

```
PROGRAM Procs;
Var x: real;

PROCEDURE WriteFormatted(r: real; n:
integer);
begin
  WriteLn(r:0:n);
end;

begin
  x := 10.0;
end.
```

- 10.0
- r:0:n
- 0
- This program doesn't have output. We forgot to CALL the procedure!

4. What is the diference between a Procedure and a Function?

- A Function has input parameters, a Procedure doesn't
- A Function returns an output value, a Procedure doesn't
- A Procedure has input parameters, a Function doesn't
- A Procedure returns an output value, a Function doesn't

Quick Test 14: Mathematical Functions

1. What will be the result of the next function call?

```
Round(3.53)
```

- 3
- 3.0
- 4
- 4.0

2. How to implement the function ArcCos(x)?

for x>0:

for x=0:

for x<0:

Correct Answer

3. What will be the result of the next function call?

```
Random(100)
```

- 100
- 0, because we forgot to call Randomize;
- Unpredictable. Any number between 0 and 99 (inclusive)
- Unpredictable. Any number between 1 and 100 (inclusive)

4. What will be is the value of x after the next instruction?

```
x := Sqr(Sqr(Sqr(2.0)));
```

- This construction is not allowed!
- 4.0
- 16.0
- 256.0

Quick Test 15: Scope of variables, passing by value vs. passing by reference

1. What is the scope of each object in the following program

```
PROGRAM VariableTypes;
Var a: real;

PROCEDURE Proc1(b: real);
Var c: real;
Const d = 10.0;
begin
  c := b+d;
  Writeln(c);
end;


Function Proc2(Var e: real): real;
Const f = 20.0;
begin
  Proc2 := e+f;
end;

Var g: real;

begin
```

2. Consider the program below

```
PROGRAM QuickTest15
Var x: integer;
PROCEDURE Show(Var a: integer);
begin
  write(a,' ');
  a := a + 1;
end;

begin
  x := 0;
  Write(x,' ');
  Show(x);
  Write(x);
end.
```

The procedure is called using the technique of

- ○ Passing by value
- ○ Passing by reference

And, hence the result will be

[ ]

Correct Answer

```
  a := 10.0;
  Proc1(a);
end.
```

	local	global	parameter	neither
a	○	○	○	○
b	○	○	○	○
c	○	○	○	○
d	○	○	○	○
e	○	○	○	○
f	○	○	○	○
g	○	○	○	○

3. What will be the output of the next program?

```
PROGRAM DoubleNames;
Var x: integer;

PROCEDURE Show;
Var x: integer;
begin
  x := 1;
  x := x*x;
  Write(x,' ');
end;

begin
  x := 0;
  Show;
  Write(x);
end.
```

- ○ Cannot have two identical names for variables!
- ○ 0 0
- ○ 1 0
- ○ 0 1
- ○ 1 1

Quick Test 16: Recursive Programming

Consider the following program:

```
PROGRAM CalculateN;
Var a: real;

Function XfuncN(x: real; n: integer): real;
Var c: real;
begin
  c := 0.0;
  if n=0 then
    XfuncN := 1.0
  else
    XfuncN := x*XfuncN(x, n-1);
end;

begin
  WriteLn(XfuncN(3.0, 3):0:1);
end.
```

1. What is the output of the program?

Check Correct Answer

2. How many copies of the local variable c exists at maximum?

Check Correct Answer

Quick Test 16 extra: Recursive Programming

Consider the following program:

```
PROGRAM CalculateN;
Var a: real;

Function XfuncN(x: real; n: integer): real;
Var c: real;
begin
  c := 0.0;
  if n=0 then
    XfuncN := 1.0
  else
    XfuncN := Exp(n*Ln(x)) + XfuncN(x, n-1);
      (* remember a^b = Exp(b*Ln(a)) *)
end;

begin
  WriteLn(XfuncN(2.0, 2):0:1);
end.
```

1. What is the output of the program?

Check Correct Answer

2. How many copies of the local variable c exist at maximum?

Check Correct Answer

Quick Test 17, 18: Arrays and Records

1. What is the difference between an array and a record?

- An array can store only countable things, whereas a record can store anything.
- A record can only store countable things, whereas an array can store anything.
- Arrays are for combining variables of different type, records store variables of the same type.
- Records are for combining variables of different type, arrays store variables of the same type.

```
FUNCTION Maximum(a, b: real): real;
var max: real;
begin
  if a>b then max := a else max := b;
 ......
end;
```

2. How to let this function return the value of max to the calling instruction?

- Nothing, this is done automatically.
- `Maximum := max;`
- `return max;`
- This function has no output and will not return anything!

```
Var a: array[1..10] of
  record
    x: record
         z: array[1..10] of real;
         i: array[1..3] of integer;
       end;
    y: record;
         r: real;
         p: double;
       end;
  end;
```

3. How to assign a value of 0 to (the first) `i` of this program?

Check | Correct Answer

4. We want to make a database with the information of 1000 students with their name and year. We can do this best with a variable

-
```
Var a: record
     number: integer;
     name: string;
     year: integer;
end;
```
-
```
Var a: array[1..1000] of
     record
     name: string;
     year: integer;
end;
```
-
```
Var a: record
     number: array[1..1000] of
integer;
     name: array[1..1000] of
string;
     year: array[1..1000] of
integer;
end;
```
-
```
Var a: array[1..1000] of
     record
       name: array[1..1000] of
string;
       year: array[1..1000] of
integer;
     end;
```

Quick Test 19: Type

1. What is `Type` used for?

- For typing text on the screen.
- For defining a new type of variable.
- For making combinations of arrays and records.
- For declaring variables of mixed types.

```
Type b = real;
 ......
b := 3.1;
```

2. Why doesn't the above code work?

- We should use '`Type b: real`' instead.
- `real` is already defined.
- The syntax is wrong, we should use '`typedef`' instead.
- `Type` only specifies a type for variables to be declared later and doesn't create a variable itself.

```
Type a = array[1..10] of
  record
    ri: record
         x: array[1..10] of real;
         y: array[0..3] of integer;
       end;
    rd: record;
         v: real;
         w: double;
       end;
  end;
Var b: a;
```

3. How to assign a value of 0 to a (the first) `y` of this program?

Check | Correct Answer

4. What will be the output of the next code?

```
Type floats = array[1..10] of real;

PROCEDURE WriteIt(r: floats);
begin
  WriteLn(r[1]);
end;

Var x: array[1..10] of integer;

begin
  x[1] := 3;
  WriteIt(x);
end.
```

- Undetermined. We forgot to initialize the array `r`!
- `3.0`
- Nothing; type mistmatch in calling the procedure.
- `3`

Quick Test 20: Pointers

1. How to declare a pointer to a word?

- `Var a: ^word;`
- `Var a: word;`
- `Var a: @word;`
- `Var a: word^;`

2. How to attribute the address of variable `x` to pointer `p`?

- Depends on the type of `x`.
- `p := ^x;`
- `p := @x;`
- `p := x^;`

```
Var b: array[1..20] of ^integer;
```

3. How to assign a value of 0 to the first integer of `b`?

Check Correct Answer

4. What means `p := NIL`?

- 0.
- The contents address `p` will be 0.
- The pointer `p` will point to nothing.
- The pointer `p` will point to address 0.

Aula Prática 1-a

Sumário

Noções sobre o Windows
Comandos básicos do Windows
O ambiente de trabalho

Noções sobre o Windows

O Windows é um sistema operativo. Um sistema operativo é um conjunto de programas que gerem os recursos do computador, permitindo-nos trabalhar com ele.

O Windows é um sistema operativo multi-utilizador e multi-tarefa (significa que várias pessoas podem usar o mesmo computador ao mesmo tempo e correndo
programas diferentes).

O Windows tem um mecanismo de segurança que impede os utilizadores normais de danificarem ficheiros que são essenciais para o bom funcionamento do sistema.
Portanto, não tenham medo de experimentar. O computador "não morde", e o pior que pode acontecer é perderem os vossos ficheiros pessoais.

Cada utilizador tem um nome (login name) e uma password, que o identifica no sistema. Tem também uma área de trabalho só dele, aqui encontram-se todos os
ficheiros que lhe pertencem.

O Windows não é sensível às maiúsculas e minúsculas.

Para começar uma sessão em Windows temos de fazer login:

login - identifica os utilizadores que entram no sistema.

> Login: nome do utilizador
> Password: **********

Nota: É importante não esquecer a password.

Comandos Básicos do Windows

A partir de abrir um Command-Shell ("cmd" em "Start:Run", ou "MS-DOS Prompt") podemos entrar os nossos comandos. Grande parte dos comandos são abreviaturas de palavras inglesas.

Comandos de uso geral

exit - para terminar a sessão de trabalho no Command Shell

> exit

help - para pedir ajuda sobre um comando.

> help nome_comando

Comandos sobre directorias

As directorias servem para agrupar os ficheiros por temas, permitindo uma melhor organização da informação.

dir - para ver o conteúdo das directorias. vai também mostrar o tamanho e o data da ultima mudança

> dir nome_directoria - mostra os ficheiros e outras directorias
> no interior da directoria indicada.
> dir - mostra os ficheiros da directoria corrente

cd - para mover entre directorias (cd de change directory).

> cd \ - vai para a raíz.

> cd .. - sobe um nível, vai para a directoria a cima.

> cd . - a própria directoria.

> cd nome_directoria - vai para a directoria indicada.

> cd - para indicar a directoria corrente

mkdir - para criar novas directorias (**mkdir** de **m**a**k**e **dir**ectory).

> mkdir nome_directoria

rmdir - para apagar directorias (**rmdir** de **r**e**m**ove **dir**ectory).

> rmdir nome_directoria - apaga apenas se a directoria
> estiver vazia.

nome_directoria: Caminho desde a raíz (\) até á directoria desejada. Para cada novo nível na árvore coloca-se uma nova \.

Comandos sobre ficheiros

copy - para copiar ficheiros

> copy nome_directoria_origem\nome_ficheiros nome_directoria_destino

move - para mover ficheiros

> move nome_dirertoria_origem\nome_ficheiros nome_direrctoria_destino\nome_ficheiros

del - para apagar ficheiros (del de delete).

> del nome_directoria\nome_ficheiros - apaga os ficheiros da directoria indicada.

> del nome_ficheiros - apaga os ficheiros da directoria corrente.

type - para ver o conteúdo de ficheiros.

> type nome_directoria\nome_ficheiros

> type ficheiro | more

Mostra no écran o contéudo do(s) ficheiro(s).

Comandos essenciais do more:
Space - ir para a próxima página
b - ir para a página anterior
q - quit (sair)

Metacaracteres (* e ?)

Os metacaracteres podem ser utilizados com qualquer um dos comandos de ficheiros, e facilitam bastante quando se pretende fazer a mesma operação sobre
ficheiros que têm algo em comum no seu nome.

O * representa uma cadeia de caracteres.

O ? representa um caracter.

Permissões de Ficheiros

O Windows tem um mecanismo que nos permite proteger os nossos ficheiros dos outros utilizadores. A esse mecanismo chama-se attributes de ficheiros.

Existem 4 attributes de ficheiros
A: Archive (ficheiro)
R: Read-Only (só ler)
H: Hidden (invisível)
S: System file (do sistema operativo. Não mexe!)

para mudar

> attrib {+,-}{A,R,H,S} nome_ficheiros

por exemplo:

> attrib -r prim.pas

Para ver as especificações existentes:

> attrib

O Ambiente de Trabalho

Para criar um programa numa linguagem de programação qualquer, existem quatro passos que têm sempre que acontecer:

1. Pensar no problema, de preferência com papel e lápis.
2. Escrever o programa, para isso é necessário um editor de texto.
3. Verificar se o código que se escreveu não tem erros, para isso é necessário um compilador da linguagem de programação usada.
4. Quando o programa estiver livre de erros, criar o ficheiro executável e correr o programa.

Em Windows, como se podem realizar os três últimos passos?

2 - Editar ficheiros

Existem alguns editores de texto em Windows. O notepad é o mais importante.
Podes usar qualquer um deles, e até podes gravar os ficheiros numa disquete e continuar a trabalhar com eles em casa.
Para escrever os programas em Pascal é melhor usar o IDE (integrated development environment) de Turbo Pascal. Use

TP <nome_ficheiro>

na pasta onde pretende escrever o programa.

3 e 4 - Compilar e executar programas

Para compilar/executar um programa devem chamar o compilador gcc na linha de comandos:

Compilar e tornar executável:

> tpc nome_ficheiro.pas

Correr o programa:

> nome_ficheiro

Aula Prática 1-b

Sumário

Exemplos práticos dos comandos básicos do Windows

Exemplos práticos dos comandos básicos do Windows

0. Login e abre uma janela "DOS Command"

em caso de erro de teclado: chcp 850

1. Comandos de uso geral

Comando **help**

```
> help dir
```

2. Comandos

Comando para mudar de disco

```
>  y:
```
(muda para o disco virtual que fica em Europa). O aluno sempre deve trabalhar em y:

Comando **cd**

```
> cd
```

Comando **dir**

```
> dir
```

Comando **cd**

```
> cd
```

Comando **mkdir**

```
> mkdir prog1
> cd prog1
```

Comando **edit** (em caso de erro de teclado: chcp 850)

```
> edit p1.pas
```

escreve lá texto, guarda o ficheiro e sai

Comando **type**

```
> dir
```

```
> type p1.pas
```

Comando **more**

```
> type p1.pas | more
```

Comando **copy**

```
> copy p1.pas p2.pas
> copy p1.pas p1a.pas
> dir
```

Comando **move**

```
> move p1.pas x.pas
> dir
```

Permissões de ficheiros. Comando **attrib**

```
> attrib
> attrib +r p1.pas
> attrib
```

Comando **del**

```
> dir
> del p*.pas
> dir
> attrib -r p1.pas
> del p*.pas
> dir
> del x.pas
> del *.*
> dir
```

Comando **rmdir**

```
> cd ..
> dir
> rmdir prog1
> dir
```

Ambiente de Trabalho

#Abrir o editor de texto.
#Escrever um texto livre.
#Guardar num ficheiro.
#Abrir um novo ficheiro.
#Escrever o programa Hello World.

```
PROGRAM HelloWorld;

begin
  writeln('Hello World');
end.
```

#Guardar num ficheiro hello.pas.

```
#Compilar e correr via linha de comandos.
```

Aula Prática 1-c

Sumário

A Internet

A Internet

A Internet é uma rede mundial de computadores que estão espalhados pelo mundo inteiro. Cada computador ligado à Internet utiliza software próprio para poder disponibilizar e/ou aceder a informação. A Internet é um meio através do qual se pode aceder a informação disponível em documentos ou ficheiros que estão contidos noutros computadores. A Internet pode ser comparada com uma infra-estrutura para suportar uma espécie de biblioteca gigantesca.

Os computadores ligados à Internet podem aceder aos seguintes serviços:

Correio electrónico (e-mail).
Permite receber e enviar mensagens para outras pessoas em qualquer ponto do mundo.

Telnet ou login remoto.
Permite que utilizes o teu computador para te ligares a um computador remoto (possivelmente noutro país) e usá-lo como se estivesses lá.

FTP (File Transfer Protocol).
Permite que o teu computador possa transferir ficheiros de/para outros computadores.

World Wide Web (WWW ou "Web").
Quando te ligas à Internet utilizando o Netscape ou outro browser qualquer (ex: Windows Explorer), estás a ver documentos na WWW. A WWW está baseada na noção de Hipertexto. Este é um sistema que tem links, uma espécie de apontadores para outros documentos na Web. Cada documento é identificado por um URL (Uniform Resource Locator) que não é mais do que um endereço único para um documento da Web.

Exemplos de URLs:

http://www.cnn.com é o endereço do canal Americano CNN.
http://www.publico.pt é o endereço do jornal Publico.
http://www.ualg.pt é o endereço da Universidade do Algarve.
http://diana.uceh.ualg.pt/IC/ é o endereço da cadeira de Introdução a Computação.
http://w3.ualg.pt/~pjotr/ic/index.html é o "*mirror*" do endereço em cima
http://www.sosmath.com/index.html para ajudar em matemática
http://www.physicscentral.com/lou/index.html física

Sempre que quiseres encontrar coisas na Internet, aconselho a utilizares o Altavista, cujo URL é http://www.altavista.com ou Google (URL: http://www.google.com. O Google permite que escrevas palavras e devolve-te um conjunto de URLs que têm informação relacionada com essas palavras. Experimenta.

Os documentos na Web estão feitos utilizando uma linguagem chamada HTML (HyperText Markup Language). O HTML tem um conjunto de comandos (tags) que permitem formatar texto, incluir imagens, e especificar links para outros documentos. Se quiseres ver o HTML que gerou o documento que estás a ver neste preciso momento, vai à opção View e dentro dessa opção escolhe Page Source.

Aula prática 2a

Sumário

- O compilador **tpc**
- Um exemplo passo a passo.
- Programas introdutórios em linguagem PASCAL.

O compilador

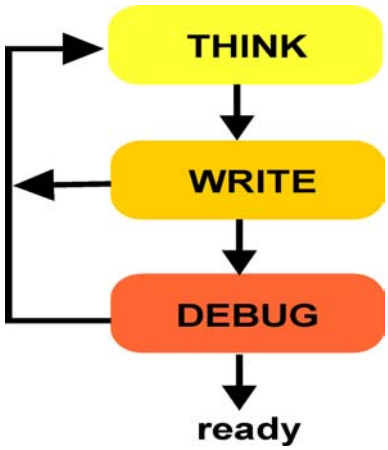
Nas nossas aulas vamos utilizar o compilador de linha de comando **tpc**. O **tpc** de Borland é um dos compiladores de PASCAL mais vendidos. Já não é o mais avançado (agora existe por exemplo Delphi ou Kylix) mas é muito bom para os nossos programas. Existem também compiladores de domínio livre (*public domain*), por exemplo o compilador de FreePascal (http://www.freepascal.org) . O que significa ser software de domínio livre? Significa que temos liberdade para:

1. Executar o software, qualquer que seja o nosso propósito
2. Estudar o modo como o software funciona e adaptá-lo às nossas necessidades
3. Distribuir cópias do software
4. Melhorar o software e distribuir esses melhoramentos para benefício da comunidade

O **tpc** suporta os standards modernos da linguagem PASCAL (como p/ex. o ANSI PASCAL) ao mesmo tempo que mantém a compatibilidade com os compiladores e estilos mais antigos.

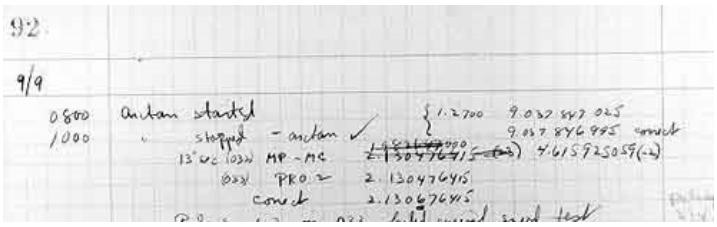
Software engineering

Crear programas consiste em vários etapas



1. **Pensar**. Estudar o problema.
2. **Escrever** um programa
3. Resolver os erros do programa. **Debugging**
4. Analisar o resultado
5. Em caso necesário, **Voltar** ao passo 3, 2 ou 1.

1. **Pensar**. Desenho um programa com papel e lápis. Ainda não toca o computador! Busca informação sobre o assunto. Chega a um algoritmo.



2. Escrever o programa. Use qualquer **editor**. Por exemplo notepad, edit, ou turbo.

3. **Debugging**. ("bug" = bicho, então debugging é debichomento). Eliminar erros do programa.
O compilador vai gerar "***compile-time errors***", por exemplo

```
type-mismatch error in line 34
```

Se o compilador nao encontre mais erros, vai gerar um ficheiro executável (`*.exe`) com o código binário.
Durante o correr do programa podem acontecer "***run-time errors***", por exemplo

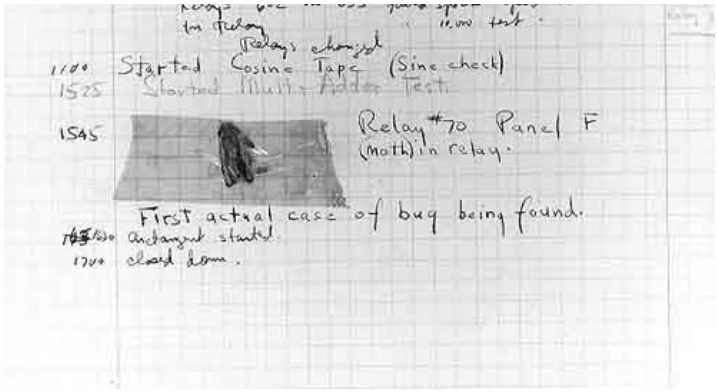
```
division by zero error
```

Ambas os tipos de erros devem ser eliminados!

4. Mesmo se o nosso programa passa o compilador sem gerar erros e corre sem gerar erros o resultado do programa pode estár mal. por exemplo, o "output" do programa pode ser

```
O raiz de 9 é 4
```

Temos de voltar ao passo 3, 2 ou 1.



The First Computer Bug

Grace Murray Hopper, working in a temporary World War I building at Harvard University on the Mark II computer, found the first computer bug beaten to death in the jaws of a relay. She glued it into the logbook of the computer and thereafter when the machine stops (frequently) they tell Howard Aiken that they are "debugging" the computer. The very first bug still exists in the National Museum of American History of the Smithsonian Institution. Edison had used the word bug and the concept of debugging previously but this was probably the first verification that the concept applied to computers.
(copied from http://www./firstcomputerbug.html)

Um exemplo passo a passo

Vamos então seguir, em quatro passos, um pequeno exemplo para criação do nosso primeiro programa.

1. Definir um espaço para o programa
2. Criar o programa
3. Compilar o programa
4. Executar o programa

Definir um espaço para o programa

É mais fácil mantermos o nosso espaço em disco organizado se criarmos uma directoria para cada programa no qual estejamos a trabalhar (excepção feita para programas ou projectos cujo código fonte esteja dividido em vários ficheiros). Neste caso vamos criar uma directoria chamada **prog1** para guardar o nosso primeiro programa.

mkdir prog1
cd prog1

Criar o programa

Um programa começa por ser um ficheiro de texto. Utiliza um editor de texto do teu agrado (p/ex. o **edit**, ou o **turbo (tp)**) para escrever esse texto a que é comum chamar-se código fonte. Consoante a escolha podes dar um dos seguintes comandos:

edit prog1.pas ou **tp prog1.pas**

e escrever o programa que se segue:

```
PROGRAM MyFirstProgram;

begin
  writeln('Hello World');
end.
```

Compilar o programa

O compilador pega no código fonte e converte-o num programa executável. Para compilares o teu código fonte usando o compilador **tpc** executa o comando:

tpc prog1.pas

O parâmetro **-o** diz ao compilador que o ficheiro executável deverá chamar-se prog1, enquanto o prog1.c no final do comando indica em que ficheiro se encontra o código fonte.

Executar o programa

Para correr o programa faz

prog1 ou **prog1.exe**

e a mensagem "Hello World" vai aparecer no ecrã.

Programas introdutórios em linguagem PASCAL

Agora podes fazer os programas que se seguem.

Aula prática 2 (continuação)

Agora que já fizeste o teu primeiro programa, utiliza um editor da tua preferência e um compilador para escrever e compilar os programas apresentados em baixo. Lembra-te de que cada programa fica num ficheiro separado, cada qual com o seu nome. Compara os programas, executa-os, observa e comenta os resultados.

Programa 1

Um programa com uma instrução:

```
PROGRAM SingleLine;

begin
  writeln('Este e o meu primeiro programa');
end.
```

Programa 2

Um programa com mais instruções:

```
PROGRAM MultiLine;

begin
  write('Este e ');
  write('o meu ');
  write('primeiro ');
  writeln('programa');
end.
```

Programa 3

Um programa mal estruturado, mas correrá sem erros. Experimente:

```
PROGRAM BadLayout; begin write('Este e '); write('o meu '); write('primeiro ');
writeln('programa'); end.
```

Programa 4

Encontre os erros no programa seguinte (tem 3). Use o compilador para eliminar os erros:

```
PROGRAM 3Erros;

begin
  write('Este e ');
  write('o meu ')
  write('primeiro ');
  writeln('programa');
end;
```

em caso de dúvidas, consulte os apontamentos da aula 4.

Programa 5

O seguinte program soma dois números inteiros, mas está mal estruturado. Faz um melhoramento do mesmo (indentação, rebaptizar os indentifiers, introduz comentário, etc).

```
PROGRAM program1;
var x10, a35, the_result_of_the_calculation_of_summing_two_variables; begin
write('Introduza um número: '); readln(x10');
write('Introduza um número: '); readln(a35);
the_result_of_the_calculation_of_summing_two_variables := x10 + a35;
writeln('A soma de ',x10,' e ',a35,' e
',the_result_of_the_calculation_of_summing_two_variables); end.
```

Programa 6

Elimina os erros do programa seguinte (*compile-time* erros e *run-time* erros!)

```
PROGRAM MyNameAndAge;

Var nome: string;
    start, end, idade: integer

begin
  write('O seu nome: ');
  readln(nome);
  write('o ano do seu nascimento: ')
  readln(start)
  write('O ano actual: ');
  readln(end);
  writelnn('Exmo. ',(end-start)/0,'  voce tem agora ',nome,' anos');
end.
```

Soluções

Introdução a Computação

Soluções da aula prática 2

Programa 4

Os erros

```
PROGRAM TresErros;              (* Indentifier não pode começar com um dígito *)

begin
  write('Este e ');
  write('o meu ');              (* faltava ; *)
  write('primeiro ');
  writeln('programa');
end.                            (* A última end do ficheiro é finalizada com . *)
```

Programa 5

```
PROGRAM program1;

var x10, a35, soma: real;

begin
  write('Introduza um número: ');
  readln(x10);
  write('Introduza um número: ');
  readln(a35);
  soma := x10 + a35;
  writeln('A soma de ',x10,' e ',a35,' e ',soma);
end.
```

Programa 6

```
PROGRAM MyNameAndAge;

Var nome: string;
    start, end, idade: integer              (* compile-time error: faltava ; *)

begin
  write('O seu nome: ');
  readln(nome);
  write('o ano do seu nascimento: ')        (* compile-time error: faltava ; *)
  readln(start)                             (* compile-time error: faltava ; *)
  write('O ano actual: ');
  readln(end);
  writelnn('Exmo. ',nome,'  voce tem agora ',(end-start),' anos');
      (* run-time error: division by zero *)
```

```
      (* erro no desenho do programa: trocar "(end-start)" com "nome" *)
end.
```

Aula prática 3

Sumário

- tamanho das variáveis
- inicializar
- texto formatado

Aviso geral

Save your programs! (guarda os seus programas).
Cada aluno tem uma conta na rede da universidade. Depois de fazer o *login*, o computador está ligado à rede e tem um disco rigido virtual (Y:) onde o aluno pode guardar os seus ficheiros. Ninguem pode mexer com os seus programas em Y:. Para trazer o trabalho a casa, pode também gravar os programas numa disquete (A:). Sem gravar os programas com regularidade, o aluno corre o risco de perder informação.

Jesus and Satan have an argument as to who is the better programmer. This goes on for a few hours until they agree to hold a contest with God as the judge. They set themselves before their computers and begin. They type furiously for several hours, lines of code streaming up the screen. Seconds before the end of the competition a bolt of lightning strikes, taking out the electricity. Moments later the power is restored and God announces that the contest is over. He asks Satan to show what he has come up with. Satan is visibly upset and cries I have nothing! I lost it all when the power went out. Very well then says God let us see if Jesus fared any better. Jesus enters a command and the screen comes to life in vivid display the voices of an angelic choir pour forth from the speakers. Satan is astonished. He stutters But how! I lost everything yet his program is intact! How did he do it?! God chuckles Jesus saves!

Tamanho das variáveis

Em PASCAL (e C também) existe uma instrução (`SizeOf`) que vai nos dizer o espaço uma variável ocupa na memória. Hoje vamos usar esta instrução para determinar os tamanhos de todos os variáveis que nos sabemos (veja aula 5).
A instrução `SizeOf` é um exemplo duma função. Ainde não sabemos o que é uma função, mas vamos dar aqui o uso geral de SizeOf:

```
SizeOf(nome)
```

retorne o tamanho da variável `nome`. Para mostrar o resultado, vamos usar `WriteLn` (aula 5).

```
WriteLn(SizeOf(nome));
```

1. Escreve um programa que

- declare variáveis de todos os tipos de variáveis nos sabemos.
- mostra o tamanho de cada variável no ecrãn, por exemplo:

```
integer: 2 bytes
real: 6 bytes
```

Inicialização

2. Muda o programa do trabalho 1 acima de forma que o programa vai mostrar o valor da cada variável, em vez de mostrar o seu tamanho. As variáveis têm todas valors igual a 0?
Agora faz uma inicialização de cada variável e experimenta outra vez.
Nunca assuma que o valor duma variável é 0.

Texto Formatado

O resultado do writeln(r) do programa acima fica feio. Por exemplo:

```
real: 6.93896007838148E003
```

Imagina receber a informação do saldo da sua conta no banco nesta forma!
Melhor seria

```
real: 6938.96
```

A instrução `WriteLn` pode gerar texto com formato predefinido. veja aula teórica 6.

3. Escreve um programa com variavéis `r`, `s`, e `i`, de tipo real, string e longint. Atribue valores: `r =` 63.9, `s =` o seu nome, e `i =` o numero do aluno. O programa deve mostrar esses tres variáveis no formato `s`: 30 lugares, `i`: 10 lugares, `r`: 10 lugares total e dois casos decimais depois o ponto, por exemplo:

```
Peter Stallinga     999999      63.90
```

Constantes

4. Escreve um programa que use variaveis do tipo *floating point* e uma constante igual a *e* (2.7....). O programa deve mostrar o valor de cada variável com 4 casas decimais.

Debugging

5. Elimina os erros do programa seguinte (compile-time erros, run-time erros e erros no desenho do programa). Não se preocupa com as instruções que não conhece; só elimina os erros que o compilador vai nos dizer:

```
FullOfErrors;

begin
  real: x,y,z;
  CONST PI = 1.6857;

  x := 3;
```

```
  y = 2*pi;
  1 := z;
  writelnn(x:0:3);
end.
```

---

soluções

Introdução a Computação

Soluções da aula prática 3

1.

```
PROGRAM TamanhoDasVariaveis;

Var bl: booelan;
    by: byte;
    i: integer;
    w: word;
    l: longint;
    r: real;
    d: double;
    e: extended;
    c: char;
    s: string;

begin
  WriteLn('boolean: ', SizeOf(bl), ' bytes');
  WriteLn('byte: ', SizeOf(bt), ' bytes');
  WriteLn('integer: ', SizeOf(i), ' bytes');
  WriteLn('word: ', SizeOf(w), ' bytes');
  WriteLn('longint: ', SizeOf(l), ' bytes');
  WriteLn('real: ', SizeOf(r), ' bytes');
  WriteLn('double: ', SizeOf(d), ' bytes');
  WriteLn('extended: ', SizeOf(e), ' bytes');
  WriteLn('char: ', SizeOf(c), ' bytes');
  WriteLn('string: ', SizeOf(s), ' bytes');
end.
```

resultado do programa:

```
boolean: 1 bytes
byte: 1 bytes
integer: 2 bytes
word: 2 bytes
longint: 4 bytes
real: 6 bytes
double: 8 bytes
extended: 10 bytes
char: 1 byte
string: 256 bytes
```

2a.

```
PROGRAM TamanhoDasVariaveis;

Var bl: booelan;
    by: byte;
    i: integer;
    w: word;
    l: longint;
    r: real;
    d: double;
    e: extended;
    c: char;
    s: string;

begin
  WriteLn('boolean: ', bl);
  WriteLn('byte: ', bt);
  WriteLn('integer: ',i);
  WriteLn('word: ', w);
  WriteLn('longint: ', l);
  WriteLn('real: ', r);
  WriteLn('double: ', d);
  WriteLn('extended: ', e);
  WriteLn('char: ', c);
  WriteLn('string: ', s);
end.
```

resultado do programa poderia ser:

```
boolean: TRUE
byte: 34
integer: -6589
word: 3256
longint: 0
real: 3.45927643E32
double: 5.111878091E103
extended: 9.9934512987E-204
char: %
string: BjHGgy^tTr4RFgjOI()0iOolooiGvTR4@#4Erd
BhHHGT&Yg%%90KJNhjJhHUY(7/&77676767767&77tyg6t
NBHGuyFIt675&%7687/8788hKJjho9YuhPUy
```

2b.

```
PROGRAM TamanhoDasVariaveis;

Var bl: booelan;
    by: byte;
    i: integer;
    w: word;
    l: longint;
    r: real;
    d: double;
    e: extended;
    c: char;
    s: string;

begin
  bl := FALSE;
  bt := 0;
  i := 0;
  w := 0;
  l := 0;
  r := 0.0;
  d := 0.0;
  e := 0.0;
  c := 'a';
  s := 'ola';
  WriteLn('boolean: ', bl);
  WriteLn('byte: ', bt);
  WriteLn('integer: ',i);
  WriteLn('word: ', w);
  WriteLn('longint: ', l);
  WriteLn('real: ', r);
  WriteLn('double: ', d);
  WriteLn('extended: ', e);
  WriteLn('char: ', c);
  WriteLn('string: ', s);
end.
```

resultado do programa será:

```
boolean: FALSE
byte: 0
integer: 0
word: 0
longint: 0
real: 0
double: 0
extended: 0
char: a
string: ola
```

3.

```
PROGRAM WriteFormatado;

Var r: real;
    i: longint;
    s: string;

begin
  r := 63.9;
  s := 'Peter Stallinga';
  i := 99999;
  WriteLn(s:30, i:10, r:10:2);
end.
```

resultado do programa:

```
               Peter Stallinga     99999      63.9
```

4.

```
PROGRAM Constante;

Var r1, r2: real;
Const E = 2.7;

begin
  r1 := 63.9*E;
  r2 := 1.0;
  WriteLn(r1:0:4);
  WriteLn(r2:0:4);
end.
```

5.

```
PROGRAM FullOfErrors;     (* faltava 'PROGRAM' *)

real: x,y,z;              (* o lugar de declarar as variaveis e antes 'begin' *)
                          (* a delaracao e: VAR nome: tipo; *)
CONST PI = 3.1415;        (* Pi nao e 1.6857 *)


begin
  x := 3.0;               (* lado esquerde: tipo real, lado direito deve ser igual *)
  y = 2.0*PI;             (* constantes com maiusculas *)
  z := 1;                 (* lado esquerdo: uma variavel, lado direito: expressao *)
  writeln(x:0:3);         (* writeln com um n *)
end.
```

Introdução a Computação

Aula prática 4

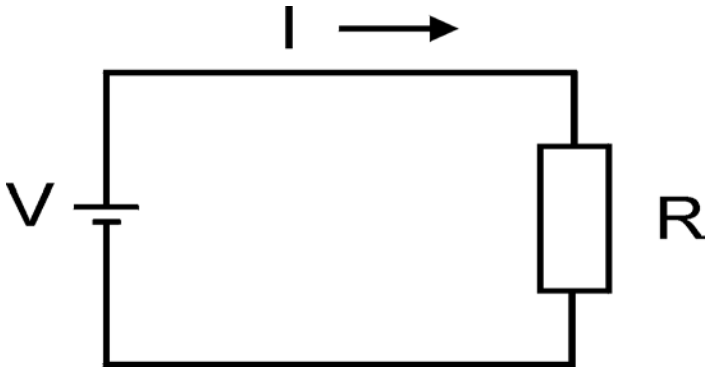
Sumário

- Calcular, atribuir
- estruturas `if ... then ... else`

---

1. Declarar variáveis, calcular valores e atribuição:

Como nos sabemos, o lei de Ohm é uma relação entre a corrente (*I*), a tensão (*V*) e a resistência (*R*), veja o circuito electrico abaixo:



Ohm's Law:

$$I = \frac{V}{R}$$

Escreve um programa que pede o utilizador valores de *V* e *R* e calcule a corrente *I*, por exemplo:

```
Qual o valor de V (volt)?
  12
Qual o valor de R (ohm)?
  1000
A corrente e 0.012
```

---

2. Muda o programa da pergunta 1 de forma que, quando o utilizador dá um valor da resistência negativo o programa vai gerar uma mensagem de erro, por exemplo:

```
Qual o valor de V (volt)?
  12
Qual o valor de R (ohm)?
  -1000
A resistencia nao pode ser negativa!
```

3.

Escreve um programa que calcule com números. O programa deve pedir o utilizador de escolher entre as opções 'multiplicar', 'addicionar', 'subtrair' e 'dividir', (Use a estrutura `if ... then ... else`)
Por exemplo:

```
numero 1: -1
numero 2: 3
Escolhe uma opcao:
1) addicionar
2) multiplicar
3) dividir
4) subtrair
 1
A soma e 2.0
```

4a. Números complexos

Números complexos são números especiais e muito usado no mundo de física, matemática e outras ciências. Os números são baseados na equação

$$i^2 = -1$$

Cada número complexo tem uma parte real e um parte imaginário. Então, um número complexo geral é de forma

$$z = a + b*i$$

onde a e b são números normais. Nota que o i não é uma variável, mas só um simbolo que represente um número imaginário.

Calcular com números complexos é fácil. Por exemplo, addicionar:

$$\begin{aligned} z_1 + z_2 &= (a_1 + b_1*i) + (a_2 + b_2*i) \\ &= (a_1 + a_2) + (b_1 + b_2)*i \end{aligned}$$

exemplo:

$$\begin{aligned} (1 - i) + (3 + 2i) &= (1 + 3) + (-1 + 2)i \\ &= 4 + i \end{aligned}$$

ou multiplicar:

$$\begin{aligned} z_1 * z_2 &= (a_1 + b_1*i) * (a_2 + b_2*i) \\ &= (a_1 * a_2) + (a_1* b_2)*i + (a_2 * b_1)*i + (b_1 * b_2)*i^2 \\ &= (a_1 * a_2 - b_1 * b_2) + (a_1* b_2 + a_2 * b_1)*\mathrm{i} \end{aligned}$$

exemplo:

$$\begin{aligned} (1 - i) * (3 + 2i) &= (1 * 3) + (1 * 2)*i + (-1 * 3)*i + (-1 * 2)i^2 \\ &= 3 + 2i - 3i + 2 \\ &= 5 - i \end{aligned}$$

Nota que o i não é uma variável, mas só um simbolo que represente um número imaginário.

Escreve um programa que pede o utilizador dois números complexos, multiplica os dois e mostra o resultado, por exemplo:

```
z1 = a1 + b1*i
Qual o valor de a1: 1
Qual o valor de b1: -1
z2 = a2 + b2*i
Qual o valor de a2: 3
Qual o valor de b2: 2
O resultado: 5.0 + -1.0i
```

4b.

Muda o programa da pergunta 4 da forma que o utilizador escolhe entre as opções 'mulitplicar' e 'addicionar' e o programa calcule um ou o outro.

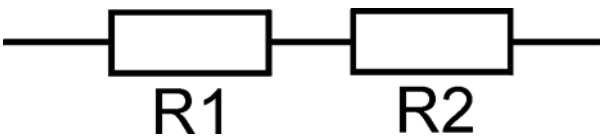
```
z1 = a1 + b1*i
Qual o valor de a1: 1
Qual o valor de b1: -1
z2 = a2 + b2*i
Qual o valor de a2: 3
Qual o valor de b2: 2
Escolhe uma opção:
1) addicionar
2) multiplicar
 1
A soma e 4.0 + 1.0i
```

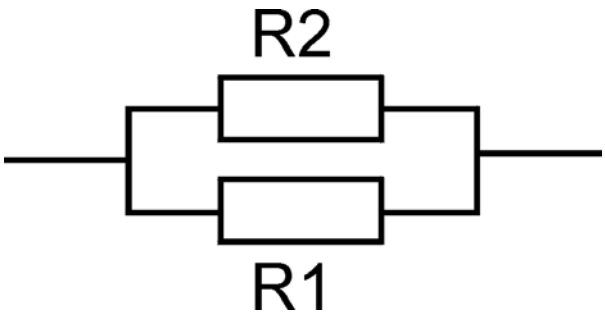
5.

Quem não gosta ou não percebe números complexos, pode escrever um programa que calcule a resistência equivalenta dum circuito com duas resistências em serie ou em paralel. O programa deve pedir o utilizador de escolher entre os dois

serial: $R = R_1 + R_2$

parallel: $R = R_1 * R_2 / (R_1 + R_2)$






```
Valor da resistencia 1: 1
valor da resistencia 2: 1
1) serie
2) paralel
 2
```

```
A resistencia equivalente e 0.500
```

---

soluções

Introdução a Computação

Soluções da aula prática 4

1.

```
PROGRAM Ohm;

Var v, r, i: real;

begin
  WriteLn('Qual o valor de V (volt)');
  ReadLn(v);
  WriteLn('Qual o valor de R (volt)');
  ReadLn(r);
  i := v/r;
  WriteLn('A corrente e ', i:0:3);
end.
```

resultado do programa:

```
Qual o valor de V (volt)?
  12
Qual o valor de R (ohm)?
  1000
A corrente e 0.012
```

2.

```
PROGRAM NegativeOhm;

Var v, r, i: real;

begin
  WriteLn('Qual o valor de V (volt)');
  ReadLn(v);
  WriteLn('Qual o valor de R (volt)');
  ReadLn(r);
  if (r<=0)
    writeln('A resistencia nao pode ser negativa!')
  else
    begin
      i := v/r;
      WriteLn('A corrente e ', i:0:3);
    end;
end.
```

resultado do programa:

```
Qual o valor de V (volt)?
  12
Qual o valor de R (ohm)?
  -1000
A resistencia nao pode ser negativa!
```

3.

```
PROGRAM Calculador;

Var a, b: real;
    opt: integer;

begin
  Write('numero 1: ');
  ReadLn(a);
  Write('numero 2: ');
  ReadLn(b);
  Writeln('Escolhe uma opcao: ');
  WriteLn('1) addicionar');
  WriteLn('2) multiplicar');
  WriteLn('3) dicidir');
  WriteLn('4) subtrair');
  readLn(opt);
  if (opt=1) then
    WriteLn('A soma e ', a + b:0:1)
  else
    if (opt=2) then
      WriteLn('O produto e ', a*b:0:1)
    else
      if (opt=3) then
        WriteLn('A divisao da ', a/b:0:1)
      else
        if (opt=4) then
          WriteLn('A diferenca e ', a-b:0:1)
        else
          WriteLn('Escolhe entre 1..4!');
end.
```

resultado do programa:

```
numero 1: -1
numero 2: 3
Escolhe uma opcao:
1) addicionar
2) multiplicar
3) dividir
4) subtrair
 1
A soma e 2.0
```

4a.

```
PROGRAM CalcularComplexo;

Var a1, b1, a2, b2, a3, b3: real;

begin
  WriteLn('z1 = a1 + b1*i');
  Write('Qual o valor de a1: ');
  ReadLn(a1);
  Write('Qual o valor de b1: ');
  ReadLn(b1);
  WriteLn('z2 = a2 + b2*i');
  Write('Qual o valor de a2: ');
  ReadLn(a2);
  Write('Qual o valor de b2: ');
  ReadLn(b2);
  a3 := a1 * a2 - b1 * b2;
  b3 := a1 * b2 + a2 * b1;
  WriteLn('O resultado: ',a3:0:1, ' + ', b3:0:1);
end.
```

```
z1 = a1 + b1*i
Qual o valor de a1: 1
Qual o valor de b1: -1
z2 = a2 + b2*i
Qual o valor de a2: 3
Qual o valor de b2: 2
O resultado: 5.0 + -1.0i
```

4b.

```
PROGRAM CalcularComplexoOpcao;

Var a1, b1, a2, b2, a3, b3: real;
    opt: integer;

begin
  WriteLn('z1 = a1 + b1*i');
  Write('Qual o valor de a1: ');
  ReadLn(a1);
  Write('Qual o valor de b1: ');
  ReadLn(b1);
  WriteLn('z2 = a2 + b2*i');
  Write('Qual o valor de a2: ');
  ReadLn(a2);
  Write('Qual o valor de b2: ');
  ReadLn(b2);
  Writeln('Escolhe uma opcao');
  Writeln('1) Addicionar');
  WriteLn('2) Multiplicar');
  ReadLn(opt);
  if (opt=1) then
```

```
    begin
      a3 := a1 + a2;
      b3 := b1 + b2;
      WriteLn('O resultado: ',a3:0:1, ' + ', b3:0:1);
    end
  else
    begin
      a3 := a1 * a2 - b1 * b2;
      b3 := a1 * b2 + a2 * b1;
      WriteLn('O resultado: ',a3:0:1, ' + ', b3:0:1);
    end;
end.
```

```
z1 = a1 + b1*i
Qual o valor de a1: 1
Qual o valor de b1: -1
z2 = a2 + b2*i
Qual o valor de a2: 3
Qual o valor de b2: 2
Escolhe uma opção:
1) addicionar
2) multiplicar
 2
A soma e 4.0 + 1.0i
```

5.

```
PROGRAM SerialParallel;

Var r, r1, r2: real;
    opt: integer;

begin
  Write('Valor da resistencia 1: ');
  ReadLn(r1);
  Write('Valor da resistencia 2: ');
  ReadLn(r2);
  Writeln('Escolhe uma opcao');
  Writeln('1) Serie');
  WriteLn('2) Paralel');
  ReadLn(opt);
  if (opt=1) then
    r := r1 + r2
  else
    r := r1*r2/(r1+r2);
  WriteLn('A resistencia equivalenta e ', r:0:3);
end.
```

Aula prática 5

Sumário

- Ciclos For
- Estruturas `Case ... Of`

---

1. Reescreva o programa do exercicio 3 da aula prática 4 usando a estrutura **`Case ... Of`**.

Um programa de calculo. O programa deve pedir ao utilizador para escolher entre as opções 'multiplicar', 'addicionar', 'subtrair' e 'dividir', por exemplo:

```
numero 1: -1
numero 2: 3
Escolhe uma opcao:
1) addicionar
2) multiplicar
3) dividir
4) subtrair
 1
A soma e 2.0
```

---

2. Álgebra Booleana; combinar condições

Escreva um programa que pede ao utilizador dois números. O programa deve mostrar
1) O texto 'Ambos negativos' se os números forem ambos negativos
2) O texto 'No mínimo um é negativo' se pelo menos um for negativo
3) O texto 'Só um é negativo' se um for negativo mas não os dois.
Use operadores AND, OR e XOR
Por exemplo

```
numero 1: -1
numero 2: 3
No minimo um e negativo
So um e negativo
```

---

3. Álgebra Booleana com inteiros

Na aula 9 aprendemos álgebra Boolean com variáveis do tipo inteiro. Por exemplo, o exercicio do mini-teste 9

43 AND 33 dá 33

Agora, faça os cálculos com papel e lápis:

expressão	resultado
25 AND 49	

37 OR 11	
39 XOR 17	
7 OR 14	

Agora escreva um programa que calcule o resultado das mesmas expressões. O programa deve pedir ao utilizador dois números. Depois o utilizador pode escolher entre as opções 'AND', 'OR', 'XOR' e 'NOT', e o programa deve calcular o resultado da operação. Use variáveis do tipo byte. (Lembre-se, um byte tem 8 bit e só pode armazenar valores positivos, veja aula 3).
Verifique os seus resultados obtidos na tabela acima usando o seu programa.

---

4. Ciclos for.

Ciclos For são usado para repetir coisas numa maneira contável (veja aula 10)

Escreva um programa que escreve no ecrã 'Estou muito feliz' 1000 vezes.

4b: Mude o programa da forma a pedir ao utilizador

1) o texto a mostrar no ecran (use uma variável do tipo string)
2) o número de vezes que quer mostrar o texto
Por exemplo

```
texto a mostrar: Benfica o glorioso
numero de vezes: 3
Benfica o glorioso
Benfica o glorioso
Benfica o glorioso
```

---

5. Faça um pograma que escreve a tabuada de um determinado número (O número vem do utilizador).

Por exemplo

```
Um numero: 8
1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
4 x 8 = 32
5 x 8 = 40
6 x 8 = 48
7 x 8 = 56
8 x 8 = 64
```

```
9 x 8 = 72
10 x 8 = 80
```

---

6. (difícil) Escreva um programa que pede ao utilizador um número (entre 0 e 255) e o programa mostrará o número em formato binário. (Sugestão: usa uma variável do tipo byte e usa as operações `x AND 128`, etc.) Exemplo

```
numero decimal: 33
binario: 00100001
```

---

soluções

Introdução a Computação

Soluções da aula prática 5

1.

```
PROGRAM Calculator;

Var a, b, c: real;
    opcao: integer;

begin
  Write('numero 1: ');
  ReadLn(a);
  Write('numero 2: ');
  ReadLn(b);
  WriteLn('Escolhe uma opcao:');
  WriteLn('1) addicionar');
  WriteLn('2) mulitplicar');
  WriteLn('3) dividir');
  WriteLn('4) subtrair');
  ReadLn(opcao);
  Case opcao of
    1: begin
         c := a + b;
         WriteLn('A soma e ',c:0:1);
       end;
    2: begin
         c := a * b;
         WriteLn('O produto e ',c:0:1);
       end;
    3: begin
         c := a / b;
         WriteLn('A divisao da ',c:0:1);
       end;
    4: begin
         c := a - b;
         WriteLn('A diferenca e ',c:0:1);
       end;
  end;
end.
```

```
numero 1: -1
numero 2: 3
Escolhe uma opcao:
1) addicionar
2) multiplicar
3) dividir
4) subtrair
```

```
 1
A soma e 2.0
```

2.

```
PROGRAM Combination;

Var a, b: integer;

begin
  Write('numero 1: ');
  ReadLn(a);
  Write('numero 2: ');
  ReadLn(b);
  if (a<0) AND (b<0) then
    WriteLn('Ambos negativos');
  if (a<0) OR (b<0) then
    WriteLn('No minimo um e negativo');
  if (a<0) XOR (b<0) then
    WriteLn('So um e negativo');
end.

numero 1: -1
numero 2: 3
No minimo um e negativo
So um e negativo
```

3.

expressão	resultado
25 AND 49	17
37 OR 11	47
39 XOR 17	54
7 OR 14	15

```
PROGRAM BooleanAlgebra;

Var a, b: byte;
    opcao: integer;

begin
  Write('numero a: ');
  ReadLn(a);
  Write('numero b: ');
  ReadLn(b);
  WriteLn('Escolhe uma opcao:');
  WriteLn('1) a AND b');
  WriteLn('2) a OR b');
  WriteLn('3) a XOR b');
  WriteLn('4) NOT a');
```

```
  ReadLn(opcao);
  Case opcao of
    1: WriteLn(a, ' AND ', b, ' = ', a AND b);
    2: WriteLn(a, ' OR ', b, ' = ', a OR b);
    3: WriteLn(a, ' XOR ', b, ' = ', a XOR b);
    4: WriteLn('NOT ', a, ' = ', NOT a);
  end;
end.
```

(Nota que o resultado do `NOT a` depende do tipo da variável)

4.

```
PROGRAM EstouFeliz;

Var i: integer;

begin
  for i := 1 to 1000 do
    WriteLn('Estou muito feliz');
end.
```

4b:

```
PROGRAM TextoAMostrar;

Var i, n: integer;
    s: string;

begin
  Write('texto a mostrar: ');
  ReadLn(s);
  Write('numero de vezes: ');
  ReadLn(n);
  for i := 1 to n do
    WriteLn(s);
end.

texto a mostrar: Benfica o glorioso
numero de vezes: 3
Benfica o glorioso
Benfica o glorioso
Benfica o glorioso
```

5.

```
PROGRAM Tabuada;

Var i, n: integer;

begin
  Write('Um numero: ');
  ReadLn(n);
```

```
  for i := 1 to 10 do
    WriteLn(i, ' x ', n, ' = ', i*n);
end.
```

```
Um numero: 8
1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
4 x 8 = 32
5 x 8 = 40
6 x 8 = 48
7 x 8 = 56
8 x 8 = 64
9 x 8 = 72
10 x 8 = 80
```

6.

```
PROGRAM Binario;

Var a: byte;

begin
  Write('numero decimal: ');
  ReadLn(a);
  Write('binario: ');
  if (a AND 128)>0 then Write('1') else Write('0');
   (* 128 = 10000000 *)
  if (a AND 64)>0 then Write('1') else Write('0');
   (* 64 = 01000000 *)
  if (a AND 32)>0 then Write('1') else Write('0');
   (* 32 = 00100000 *)
  if (a AND 16)>0 then Write('1') else Write('0');
   (* 16 = 00010000 *)
  if (a AND 8)>0 then Write('1') else Write('0');
   (* 8 = 00001000 *)
  if (a AND 4)>0 then Write('1') else Write('0');
   (* 4 = 00000100 *)
  if (a AND 2)>0 then Write('1') else Write('0');
   (* 2 = 00000010 *)
  if (a AND 1)>0 then WriteLn('1') else WriteLn('0');
   (* 1 = 00000001 *)
end.
```

```
numero decimal: 33
binario: 00100001
```

ou uma solução com um ciclo Repeat-Until

```
PROGRAM Binario;

Var a, mask: byte;

begin
  Write('numero decimal: ');
```

```
  ReadLn(a);
  Write('binario: ');
  mask := 128     (* 128 = 10000000 *)
  repeat
    if (a AND mask)>0 then Write('1') else Write('0');
    mask := mask DIV 2;
      (* 128 -> 64 -> 32 -> 16 -> 8 -> 4 -> 2 -> 1 -> 0 *)
      (* 10000000 -> 01000000 -> 00100000 -> 00010000 -> *)
      (* 00001000 -> 00000100 -> 00000010 -> 00000001 -> *)
      (* 00000000 *)
  until mask < 1;
end.
```

agora com um ciclo For para mostrar todos os códigos binários entre 0 e 255:

```
PROGRAM Binario;

Var a, mask: byte;

begin
  For a := 0 To 255 Do
    begin
      Write('numero decimal: ');
      Write('binario: ');
        (* nao precisa pedir ao utilizador um numero *)
        (* 'a' vem do ciclo *)
      mask := 128
      repeat
        if (a AND mask)>0 then Write('1') else Write('0');
        mask := mask DIV 2;
      until mask < 1;
      WriteLn;           (* introduz nova linha *)
    end;
end.
```

Introdução a Computação
Aula prática 6

Sumário

- Ciclos For, Repeat-Until e While-Do
- Procedimentos sem *input* nem *output*

1.

Estatísticas simples. Você é dado(a) um a um uma série de números inteiros. Os ditos números podem ser positivos ou negativos. Quando o número introduzido seja -999, a série termina, SEM incluir o dito número. Para além de mostrar cada vez no ecrã o número inserido, **no fim da série** você deverá escrever no ecrã os resultados dos seguintes cálculos:
a) o valor médio dos números na série
b) o número menor e o número maior na serie
c) quantos dos números na série são pares e quantos são impares.
d) O número de valores introduzidos.

Dica: defina uma variável adequada para guardar os valores parciais para cada uma destas operações

---

2. Escreva um programa, onde cada uma das opções do exercício Nº 1 desta TP seja um procedimento.

Isto é, a leitura da série até o fim (quando o número -999 é introduzido) constitui um procedimento.
Cada uma das tarefas indicadas nas letras a, b, c, e, d, devem ser também procedimentos individuais.
Desta forma, a secção principal do programa executa só os procedimentos:
ler_dados, media, menor_maior , par_impar, total_numeros.
Por favor respeite estes mesmos nomes para os procedimentos.

---

3. Escreva um programa que simula uma calculadora simples que implementa as seguintes funções:

+, -, *, / , raiz quadrada, valor absoluto, seno e coseno dum ângulo.
O programa deve pedir primeiro ao utilizador(a) qual operação deseja efectuar, ou seja o operador. Logo deve pedir os operandos.
Pedir confirmação para cada operando. Se o utilizador assim decidir, deve poder mudar o valor previamente introduzido num operando.
Defina alguma variável que permita conhecer qual é a opção escolhida pelo utilizador(a).
Feito os cálculos os resultados devem ser amostrados no ecrã.
Este programa continua a oferecer o menú com as diferentes opções indefinidamente após amostrar os resultados, até que uma opção que indique o fim deste ciclo seja premida pelo utilizador(a).

---

4. Números primos: são aqueles divisíveis por 1 e por eles próprios.

Fazer um programa que verifica se um número introduzido pelo utilizador(a) é ou não um número primo.

---

5. Enquanto a soma total duma série de números introduzidos pelo utilizador(a) não seja superior a 1000, continue a executar um programa que adiciona os números introduzidos pelo utilizador, e que mostra no ecrã o valor da adição parcial deles.

---

soluções

Introdução à Computação
Aula prática 6 - Respostas

---

1 e 2 . Estatísticas simples utilizando procedimentos com variáveis globais.

```
program calculosm;
          {----programa que calcula estatisticas num ciclo ----}
uses crt;

var
num, total, numeros, maior, menor, pares, impares: integer;
{-------------------------------------------------------------------------}
procedure inicializacao;
begin
   total:= 0; pares:= 0; impares:= 0;
   maior:= -9999; menor:= 9999;
   writeln('Este programa vai calcular valores num ciclo até input=-999');
end;
{-------------------------------------------------------------------------}
procedure ler_dados;
begin
   writeln('Entrar um número: '); readln(num);
end;
{-------------------------------------------------------------------------}
procedure maior_menor;
begin
     if (num < menor) then
        menor:= num;
     if (num > maior) then
        maior:= num;
end;
{-------------------------------------------------------------------------}
procedure media;
begin
   writeln('M,dia dos números introduzidos: ', total / numeros:0:2);
end;
{-------------------------------------------------------------------------}
procedure par_impar;
begin
     if ((num mod 2) = 0) then
         pares:= pares + 1
     else
         impares:= impares + 1;
end;
{-------------------------------------------------------------------------}
procedure total_numeros;
begin
        total:= total + num;
end;
{-------------------------------------------------------------------------}
procedure estatisticas;
begin
   writeln(' Estatisticas ');
   writeln(' ------------ ');
   writeln('Total de números introduzidos: ', numeros);
   writeln('Números pares: ',  pares);
   writeln('Número impares: ', impares);
   writeln('Número menor: ',   menor);
   writeln('Número maior: ',   maior);
   media;
   readln;
end;
{-------------------------------------------------------------------------}
begin       {secção principal do programa }
```

```
clrscr;
inicializacao;
ler_dados;

while (num <> -999) do       {ciclo do programa }
   begin
        numeros:= numeros + 1;
        total_numeros;
        maior_menor;
        par_impar;
        ler_dados;
     end;
     estatisticas;    {mostrar resultados }
end.
```

3.

```
PROGRAM Calculadora;
uses crt;

var op1, op2, c: real;
    opcao: integer;
{-------------------------------------}
PROCEDURE Somar;
begin
  c := op1 + op2;
  WriteLn(' ', op1:0:2 , ' + ', op2:0:2 , ' = ',c:0:2);
end;
{-------------------------------------------------------------}
PROCEDURE Dividir;
begin
  c := op1 / op2;
  WriteLn(' ', op1:0:2 ,' / ', op2:0:2 , ' = ',c:0:2);
end;
{-------------------------------------------------------------}
PROCEDURE Multiplicar;
begin
  c := op1 * op2;
  WriteLn( ' ', op1:0:2 , ' * ', op2:0:2 , ' = ',c:0:2);
end;
{-------------------------------------------------------------}
PROCEDURE Subtrair;
begin
  c := op1 - op2;
  WriteLn(' ', op1:0:2 , ' - ', op2:0:2 , ' = ',c:0:2);
end;
{-----------------------------------------------------------------------}
FUNCTION graus_radians(angulo: real): real;
         {funcao que devolve o valor em radians dum angulo em graus }
begin
     graus_radians:= (angulo * PI) / 180;
end;
{-----------------------------------------------------------------------}
PROCEDURE seno(angulo: real); {funcao trabalho com radians e nao graus}
var r : real;
begin
    r:= graus_radians(angulo); {chama funcao que converte graus em radianes}
    writeln('Seno de ',angulo:0:2, ' = ', sin(r):0:4);
end;
{-----------------------------------------------------------------------}
PROCEDURE coseno(angulo: real); {funcao trabalha com radians }
var r : real;
begin
   r:= graus_radians(angulo);
   writeln('coseno de ',angulo:0:2, ' = ', cos(r):0:4);
end;
{-----------------------------------------------------------------------}
PROCEDURE valor_absoluto( valor: real);
begin
    writeln('Valor absoluto de ',valor:0:2, ' = ', (abs(valor)):0:2);
```

```
end;
{--------------------------------------------------------------------}
PROCEDURE raiz_quadrada(valor: real);
begin
    writeln('Raiz quadrada de ', valor:0:2, ' = ',sqrt(valor):0:2);
end;
{--------------------------------------------------------------------}
PROCEDURE ler_operandos;
var OK :char;
begin
   OK:= 'x';
   repeat
       if (opcao < 5) then
           begin
              Write('favor entrar operando 1: ');    readln(op1);
              Write('favor entrar operando 2: ');    readln(op2);
           end
       else
           begin
              write('Favor entrar valor a converter: '); readln(op1);
           end;
       write(' Esta seguro(a) dos seus operandos? '); readln(OK);
   until ((OK = 's') or (OK = 'S'));
end;


end;
{--------------------------------------------------------------------}
begin              { programa principal }
{--------------------------------------------------------------------}
  Repeat
    clrscr;
    WriteLn('            Escolhe uma opcao:');
    WriteLn('(1) adicionar             (2) multiplicar');
    WriteLn('(3) dividir               (4) subtrair');
    WriteLn('(5) valor absoluto        (6) raiz quadrada');
    WriteLn('(7) seno dum angulo       (8) coseno dum angulo');
    Writeln('(0) sair');
    write('              Qual , a sua op‡Æo?: ');
    ReadLn(opcao);
    if (opcao <> 0) then
      begin
        ler_operandos;
        Case opcao of
              1: Somar;
              2: Multiplicar;
              3: Dividir;
              4: Subtrair;
              5: valor_absoluto(op1);
              6: raiz_quadrada(op1);
              7: seno(op1);
              8: coseno(op1);
        else
              writeln('Erro no input. Repeta s.f.f.');
        end;
      readln;
      end;
  until (opcao = 0);

  WriteLn('Obrigado e bom dia');
  readln;
end.
```

---

4. Números primos: são aqueles divisíveis por 1 e por eles próprios.

Um numero é divisível pelo outro número se o resto da divisão é zero. (Na aula teórica 7 foram explicados os operadores para divisões, Div e Mod.) Um número n é primo se não for divisível por todos os números entre 2 e (n-1). Com estes duas ideas vamos resolver o problema. Sabemos exactamento quantas vezes temos de executar o ciclo: Então, utilizamos o ciclo **For**.

```
PROGRAM DeterminePrimo;
Var i, n: integer;
   primo: boolean;
begin
  Write('numero: ');
  ReadLn(n);
  primo := TRUE;
  For i := 2 To n-1 Do
      if (n Mod i) = 0 then  (* a Mod b = 0 significa que a variavel a e divisivel pela variavel b *)
      primo := FALSE;
     (* fim do ciclo For *)
  if  (primo) then
      WriteLn(n, ' e primo')
  else
    WriteLn(n, ' nao e primo');
end.
```

Para os especialistas: Existem algorítmos muito mais inteligentes para determinar se um número é primo. O programa acima é o mais simples. É relativamente fácil melhorar a eficiência do algoritmo: se encontramos uma vez que o resultado do (n Mod i)=0, não precisamos de continuar com o ciclo até verificar todos os números entre 0 e (n-1) Não é ?. Imagine vamos determinar se o número 10000 é primo. Só calculando 10000 Mod 2 sabemos que o número não é primo e podemos acabar com os cálculos. Vamos implementar esta idea.
Temos duas condições para sair do ciclo
1) chegamos ao fim com os números (i = (n-1)) ou
2) encontrámos um resultado do Mod que deu 0. Vamos combinar estas duas condições e temos de usar um outro tipo do ciclo (com ciclos For não é possivel combinar condições):

```
PROGRAM MelhorPrimo;
Var i, n: integer;     primo: boolean;
begin
Write('numero: ');   ReadLn(n);
primo := TRUE;   i := 2;

While ((i <= (n-1)) AND (primo = TRUE)) Do
   begin
      if  (n Mod i) = 0 then
         primo := FALSE;
      i := i + 1;     end;
      if (primo) then
         WriteLn(n, ' e primo')
     else
        WriteLn(n, ' nao e primo');
    end;
end.
```

A notar que na condição " if (primo) ...", em vez de primo=TRUE podemos escrever só primo, porque esta já uma variável do tipo boleana. Assim o código acima fica mais legível.

4a. Agora vamos por tudo num (outro) ciclo for:

```
PROGRAM DeterminePrimo;
Var i, n: integer;
   primo: boolean;
begin
  WriteLn('Numeros primos ate 10000:');
  for n := 3 to 10000 do
    begin
```

```
      primo := TRUE;
      for i := 2 to n-1 do
        if (n Mod i) = 0 then
          primo := FALSE;
      (* aqui acaba o ciclo For i *)
      if primo then
        Write(n,'   ');
    (* aqui acaba o ciclo For n *)
    end;
end.
```

---

5. Enquanto a soma total duma série de números introduzidos pelo utilizador(a) não seja superior a 1000, continue a executar um programa que adiciona os números introduzidos pelo utilizador, e que mostra no ecrã o valor da adição parcial deles.

```
program chegar_1000;
uses crt;

var
total, num: real;
n : integer;
begin
  clrscr;
  total:= 0; n:= 0;
  write('entrar um numero s.f.f: '); readln(num);
  if (num > 1000) then
    begin
      total:= num;   {situacao especial para primeira vez, caso o primeiro número > 1000 }
      n:= 1;
    end
  else
  repeat                  {também podia ser implementado com o ciclo while ...do }
     total:= total + num;
     if (total <= 1000) then
        begin
           n:= n + 1;
           writeln('---- Total parcial: ', total:0:2);
           if (total < 1000) then
             begin
               write('numero: '); readln(num);
             end;
        end;
  until (total >= 1000);

  writeln('Chegamos ao numero ', total:0:2);
  writeln('Numero de valores introduzidos: ', n);
  readln;
end.
```

Introdução a Computação

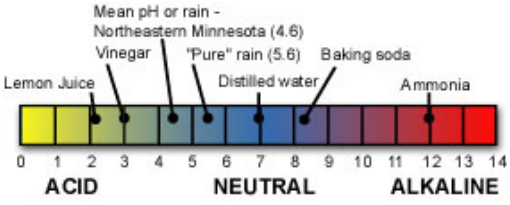
Aula prática 7

Sumário

- Funções e procedimentos sem e com parámetros
- Funções matemáticas implementadas em PASCAL
- números aleatórios

Nesta aula, usa funções e procedimentos onde é possível.

1a. Calcular o pH.



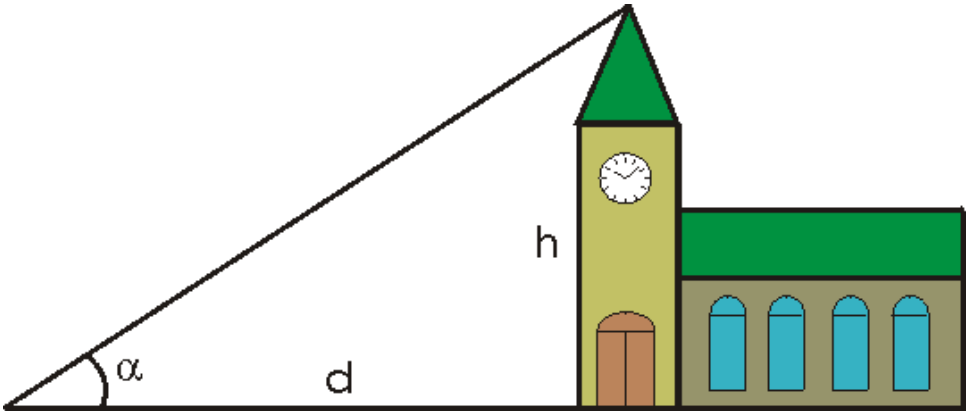
Como nos sabemos, o pH é um número que indica o acidez duma solução. Em mas pormenor, o pH diga a concentração dos iões de hidrogénio, $[H^+]$. Mais exacta, o pH é igual a $-^{10}\text{Log}\,[H^+]$. Com a unidade de concentração mol/l. Um pH muito baixo significa que a solução é muito ácido. Também, lembra a relação em agua: $[H^+][OH^-] = 10^{-14}\ \text{mol}^2\text{l}^{-2}$, então o $\text{pOH} = -^{10}\text{Log}([OH^-]) = -^{10}\text{Log}(10^{-14}/[H^+]) = 14 - \text{pH}$. Por isso, o pH da agua neutral, onde $[H^+] = [OH^-]$, é igual a 7.
Agora faça um programa que calcula o pH de uma solução. O utilizador pode escolher entre entrar a concentração dos iões de hidrogénio ou entrar a concentração dos iões de OH^-.
Por exemplo

```
Escolhe uma opcao:
1) [H+]
2) [OH-]
 1
Entra a concentracao de H+ em agua:
 1e-4
pH da solucao: 4.00
```

1b. Escreva um programa que faz o oposto. O utilizador entre o pH e o programa calculará as concentrações $[H^+]$ e $[OH^-]$. Por exemplo

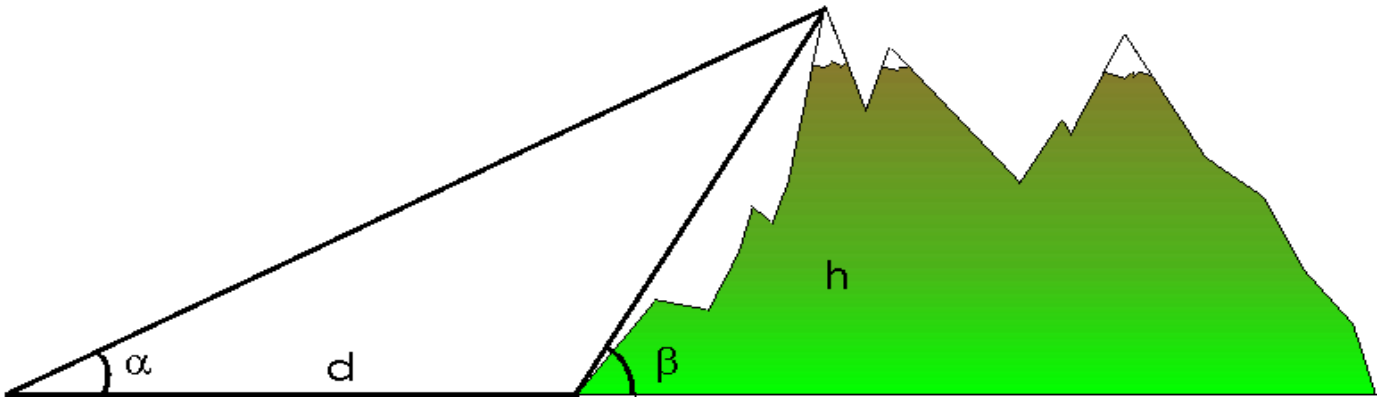
```
Diga o PH da solucao
 1.0
A concentracao [H+] = 1.0e-1 mol por litro
A concentracao [OH-] = 1.0e-13 mol por litro
```

2a. Calcular a altura de um edifício.



Determinar a altura de um objecto é nada fácil, mas com a ajuda de nosso computador será um pouco mais fácil. Imagine podemos só determinar a distância até o objecto e o ângulo de abertura do objecto. Com as regras de trigonometria é possível determinar a altura.
Escreva um programa que calcula a altura de um edifício (ou da serra) *h*. O utilizador deve entrar a distância até o objecto *d* e o ângulo de abertura do objecto, α.

```
Diga a distancia ate o objecto (m):
 100.0
Diga o angulo:
 15.0
A altura do objecto e 26.8 m.
```



2b. Programa de luxo: as vezes não é possível determinar a distância até o pé do objecto (imagine medir a altura de Mont Blanc). Ainde é possivel determinar a altura da serra se sabemos dois ângulos α e β e a distância entre os dois pontos de medição *d*.

```
Diga a distancia entre os dois pontos de medicao (m):
 8000.0
Diga o angulo 1:
 25.67
Diga o ângulo 2:
 67.41
A altura do objecto e 4807 m.
```

(a altura do Mont Blanc)

3. Mais um ciclo

Os números de Fibonacci são definidos da seguinta forma:

$f_1 = 1$

$f_2 = 1$

$f_n = f_{n-1} + f_{n-2}$

Faça um programa que mostra no ecrã os primeiros 30 números de Fibonacci. Usa um procedimento com nome `Fibonacci` que aceita um parámetro *n*, o número de números Fibonacci que o procedimento deve mostrar. O output deve ser igual a

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025
121393 196418 317811 514229 832040
```

---

4. Random numbers.

a. Escreva um programa que gera a média de 100 números aleatórios. Usa uma função com nome `Media100` que gera os valores e retorna esse valor. Corre o programa duas vezes. Tem uma diferença? Muda o programa da forma que dá um valor diferente cada vez.

b. Escreva um programa que simule atirar uma moeda no ar 1000 vezes. O programa deve contar quantas vezes sai "cara" e quantas vezes sai "coroa".

---

soluções

Introdução a Computação

Soluções da aula prática 7

1a.

```
PROGRAM CalulatepH;

var conc, pH: real;
    opcao: integer;

PROCEDURE ConcH;
begin
  WriteLn('Entra a concentracao de H+ em agua (mol/l)');
  ReadLn(conc);
  pH := -Ln(conc)/Ln(10);
  WriteLn('pH da solucao: ', pH:0:2);
end;

PROCEDURE ConcOH;
begin
  WriteLn('Entra a concentracao de OH- em agua (mol/l)');
  ReadLn(conc);
  pH := 14.0-Ln(conc)/Ln(10);
  WriteLn('pH da solucao: ', pH:0:2);
end;

begin
  WriteLn('Escolha uma opcao:');
  WriteLn('1) [H+]');
  WriteLn('2) [OH-]');
   (* nota a forma de chamar o procedimento: *)
  ReadLn(opcao);
  Case opcao of
    1: ConcH;
    2: ConcOH;
  end;
end.
```

```
Escolhe uma opcao:
1) [H+]
2) [OH-]
 1
Entra a concentracao de H+ em agua:
 1e-4
pH da solucao: 4.00
```

1b.

```
PROGRAM CalulateConc;
```

```
var pH: real;

FUNCTION CalcH: real;
begin
   (* Nota a forma de retornar um valor, atraves o nome da funcao: *)
  CalcH := Exp(-pH*Ln(10));
end;

FUNCTION CalcOH: real;
begin
   (* Nota a forma de retornar um valor, atraves o nome da funcao: *)
  CalcOH := 1.0e-14/CalcH;
end;

begin
   (* O programa comeca aqui *)
  WriteLn('Diga o pH da solucao:');
  ReadLn(pH);
   (* nota a forma de usar a informacao que vem da funcao *)
  WriteLn('A concentracao [H+] = ', CalcH, ' mol por litro');
  WriteLn('A concentracao [OH-] = ', CalcOH, ' mol por litro');
end.
```

```
Diga o PH da solucao
 1.0
A concentracao [H+] = 1.0e-1 mol por litro
A concentracao [OH-] = 1.0e-13 mol por litro
```

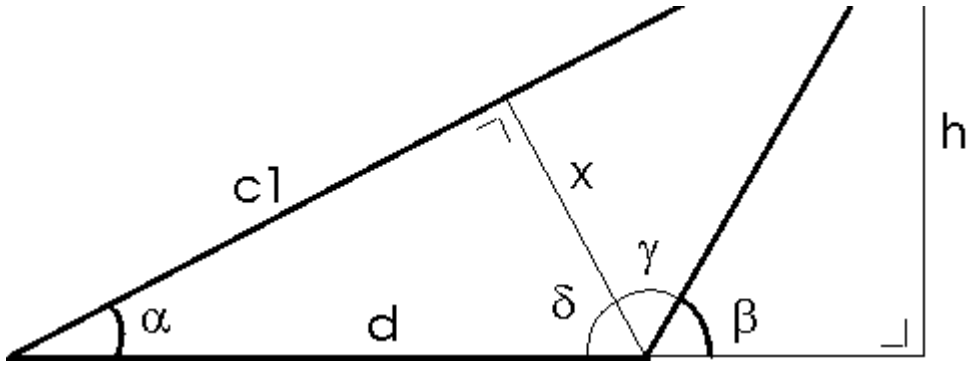
2a.

```
PROGRAM CalulateHeight;

var d, h, alfa: real;

begin
  WriteLn('Diga a distancia ate o objecto (m):');
  ReadLn(d);
  WriteLn('Diga o angulo:');
  ReadLn(alfa);
  h := d * Sin(Pi*alfa/180.0)/Cos(Pi*alfa/180.0);
  WriteLn('A altura do edificio e ', h ' m');
end.
```

```
Diga a distancia ate o objecto (m):
 100.0
Diga o angulo:
 15.0
A altura do objecto e 26.8 m.
```

2b. Tem *montes* de soluções. Aqui vem uma:

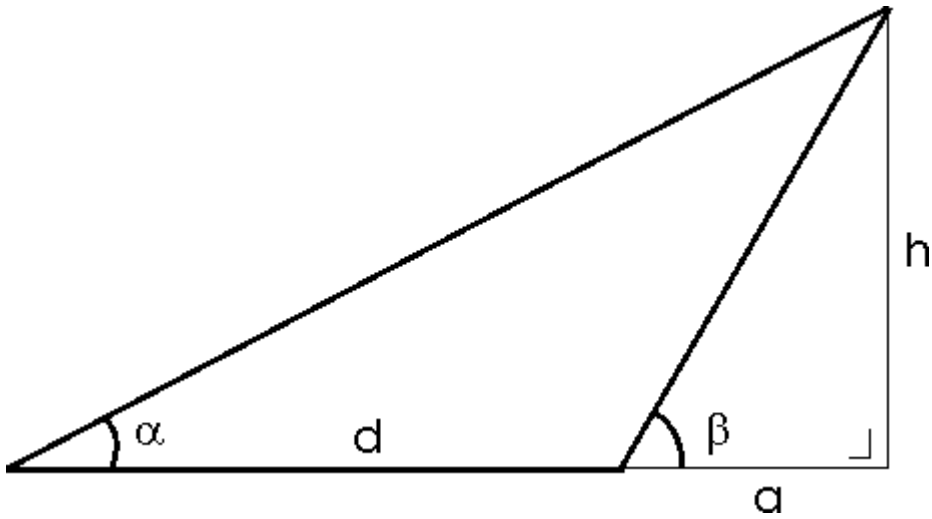
$$x = d \sin(\alpha)$$
$$c1 = d \cos(\alpha)$$
$$\gamma = 90^{o} + \alpha - \beta$$



$c2 = x \tan(\gamma)$
$h = (c1 + c2) \sin(\alpha)$

```
PROGRAM Mountains;

var alfa, beta, gama: real;
    d, x, c1, c2, h: real;
begin
  WriteLn('Diga a distancia entre os dois pontos de medicao (m)');
  ReadLn(d);
  WriteLn('Diga o angulo 1');
  ReadLn(alfa);
  WriteLn('Diga o angulo 2');
  ReadLn(beta);
  x := d * Sin(Pi*alfa/180.0);
  c1 := d * Cos(Pi*alfa/180.0);
  gama := 90 + alfa - beta;
  c2 := x * Sin(Pi*gama/180)/Cos(Pi*gama/180.0);
  h := (c1+c2) * Sin(Pi*alfa/180.0);
  WriteLn('A altura do objecto e ',h:0:0,' m.');
end.
```

Aqui vem mais uma:



$\tan(\alpha) = h/(d+a)$
$\tan(b) = h/a$

$h = d*\tan\alpha/(1-\tan\alpha/\tan\beta)$

```
PROGRAM Mountains;
var alfa, beta: real;
    d, a, h: real;
  FUNCTION Tan(ang: real): real;
  begin
    Tan := Sin(ang)/Cos(ang);
  end;
begin
  WriteLn('Diga a distancia entre os dois pontos de medicao (m)');
  ReadLn(d);
  WriteLn('Diga o angulo 1');
  ReadLn(alfa);
```

```
  WriteLn('Diga o angulo 2');
  ReadLn(beta);
  alfa := Pi*alfa/180.0;
  beta := Pi*beta/180.0;
  h := d * Tan(alfa)/(1-Tan(alfa)/Tan(beta));
  WriteLn('A altura do objecto e ',h:0:0,' m.');
end.
```

```
Diga a distancia entre os dois pontos de medicao (m):
 8000.0
Diga o angulo 1:
 25.67
Diga o ângulo 2:
 67.41
A altura do objecto e 4806 m.
```

3.

```
PROGRAM FibonacciSeries;

PROCEDURE Fibonacci(n: longint);
  (* o numero maximo sera 832040, entao so um longint chega: *)
Var f, fa, fb: longint;
begin
  fa := 1;
  fb := 1;
  WriteLn(fa);
  WriteLn(fb);
  for i := 3 to n do
    begin
      f := fa + fb;
      Write(f,' ');
      fa := fb;
      fb := f;
    end;
end;

begin
  Fibonacci(30);
end.
```

```
1 1 2 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229 832040
```

4a.

```
PROGRAM Media;

FUNCTION Media100;
Var i: integer;
    sum: real;
begin
  sum := 0.0;
```

```
  for i := 1 to 100 do
    sum := sum+Random;
  Media100 := sum/100.0;
end;

begin
    (* com a seguinta instrucao o programa vai correr cada vez
       diferente: *)
  Randomize;
    (* chama a funcao e mostra o resultado: *)
  WriteLn('Media de 100 numeros aleatorios: ', Media100);
end.
```

4b.

```
PROGRAM Moeda;

Var i: integer;
    cara, coroa: integer;

begin
  Randomize;
    (* nunca esquece inicializar as variaveis: *)
  cara := 0;
  coroa := 0;
  for i := 1 to 1000 do
    if (Random(2) = 1) then
      cara := cara + 1
    else
      coroa := coroa + 1;
  WriteLn('Cara: ', cara, ' vezes');
  WriteLn('Coroa: ', cara, ' vezes');
  if (cara=coroa) then
    WriteLn('Empate!')
  else
    if cara>coroa then
      WriteLn('Cara ganhou!');
    else
      WriteLn('Coroa ganhou!');
end.
```

```
Cara: 459 vezes
Coroa: 541 vezes
Coroa ganhou!
```

Aula prática 8

1a. Escreva um programa que gera 1000 números aleatórios (inteiros) entre −10 e 20 (inclusive). Usa uma função com nome `Aleatorio` que retorna um valor neste intervalo.

1b. Muda o programa da questão 1a da forma que a função tem dois parâmetros, `a` e `b`, para especificar os limites do intervalo.

1c. Muda o programa da questão 1b: Determine o número mínimo e máximo da série.

---

2. Faça um programa que determine o factorial de um número

$$n! = n \times (n-1) \times (n-2) \ldots \times 2 \times 1$$

$$n! = n \times (n-1)!$$
$$1! = 1$$

2a: com um ciclo.
2b: com uma função recursiva.

3a. Prediza os resultados dos programas a seguir. Qual programa usa a técnica de passagem por referência e qual usa a técnica de passagem por valor?

pogram:

```
PROGRAM Program4a1;

Var m: integer;

PROCEDURE Operations(n: integer);
begin
  n := n+1;
  writeln(2*n);
end;

begin
  m := 1;
  Operations(m);
  WriteLn(m);
end.
```

pogram:

```
PROGRAM Program4a2;

Var m: integer;

PROCEDURE Operations(Var n: integer);
begin
  n := n+1;
  writeln(2*n);
end;

begin
  m := 1;
  Operations(m);
  WriteLn(m);
end.
```

output: *output*:

3b. Verifica as suas respostas.

---

4. Acaba os trabalhos da aula prática 7.

Para os especialistas:

5a. Introduz uma variável local na função `Factorial` do programa do trabalho 2b. Quando chamamos a função `Factorial(5)`, quantas cópias da variável local existem no máximo? (veja aula teórica 16).

resposta:

5b. Verifica a sua resposta (por exemplo com uma variável contadora global).

5c. Mais difícil: Faça a mesma coisa com a função `Fibonacci` da aula teórica 16. Quantas variáveis existirão no máximo depois a chamada `Fibonacci(5)`?

resposta:

---

soluções

Soluções da aula prática 8

1a.

```
PROGRAM HundredRandom;

Var i: integer;

FUNCTION Aleatorio: integer;
begin
  Aleatorio := Random(31)-10;
    (* Random(31) returns a number between 0 and 30 (incl.) *)
    (* Random(31)-10 returns between -10 and 20 (incl.) *)
end;

begin
  for i := 1 to 1000 do
    WriteLn(Aleatorio);
end.
```

1b.

```
PROGRAM HundredRandom;

Var i: integer;

FUNCTION Aleatorio(a, b: integer): integer;
begin
  Aleatorio := Random(b-a+1)+a;
   (* numero minimo: a, numero maximo: b-a + a = b *)
end;

begin
  for i := 1 to 1000 do
    WriteLn(Aleatorio(-10, 20));
end.
```

1c.

```
PROGRAM MinMaxRandom;
  (* determine o numero minimo e maximo de uma serie de numeros
     aleatorios *)

Var i, n, maxn, minn: integer;

FUNCTION Aleatorio(a, b: integer): integer;
begin
  Aleatorio := Random(b-a+1)+a;
end;

begin
```

```
   (* initiliazar com numeros grandes: *)
  minn := 32000;
  maxn := -32000;
  for i := 1 to 1000 do
    begin
      n := Aleatorio(-10, 20);
      if n>maxn then maxn := n;
      if n<minn then minn := n;
    end;
  WriteLn('Min: ', minn);
  WriteLn('Max: ', maxn);
end.
```

2a. cópia da aula teórica 16.

```
PROGRAM TestFactorial;

FUNCTION Factorial(n: integer): integer;
 (* returns n! *)
var result: integer;
    i: integer;
begin
  result := 1;         (* initialize the variable *)
  for i := 1 to n do
    result := i*result;
  Factorial := result;  (* return result *)
end;

begin
  WriteLn(Factorial(5));
end.
```

2b.

```
PROGRAM TestFactorial;

FUNCTION Factorial(n: integer): integer;
 (* returns n! *)
begin
  if n=1 then
    Factorial := 1
  else
    Factorial := n*Factorial(n-1);
end;

begin
  WriteLn(Factorial(5));
end.
```

3a.

PASSAGEM POR VALOR:

pogram:

```
PROGRAM Program4a1;

Var m: integer;

PROCEDURE Operations(n: integer);
begin
  n := n+1;
  writeln(2*n);
end;

begin
  m := 1;
  Operations(m);
  WriteLn(m);
end.
```

output:

```
4
1
```

PASSAGEM POR REFERÊNCIA

pogram:

```
PROGRAM Program4a2;

Var m: integer;

PROCEDURE Operations(Var n: integer);
begin
  n := n+1;
  writeln(2*n);
end;

begin
  m := 1;
  Operations(m);
  WriteLn(m);
end.
```

output:

```
4
2
```

5a.

resposta: 5

5b.

```
PROGRAM TestFactorial;

Var globalcounter: integer;

FUNCTION Factorial(n: integer): integer;
Var m: integer; (* local variable *)
begin
   (* uma copia da variavel local m sera criada *)
   (* vamos contar isso: *)
  globalcounter := globalcounter + 1;
  WriteLn('m existe ', globalcounter, ' vezes');
  if n=1 then
    Factorial := 1
  else
    Factorial := n*Factorial(n-1);
   (* a copia da variavel local m sera anihilada *)
   (* vamos contar isso: *)
  globalcounter := globalcounter - 1;
  WriteLn('m existe ', globalcounter, ' vezes');
end;

begin
  globalcounter := 0;
  WriteLn(Factorial(5));
end.
```

output:

```
m existe 1 vezes
m existe 2 vezes
m existe 3 vezes
m existe 4 vezes
m existe 5 vezes
m existe 4 vezes
m existe 3 vezes
m existe 2 vezes
m existe 1 vezes
m existe 0 vezes
120
```

5b.

```
PROGRAM TestFibonacci;

Var globalcounter: integer;

FUNCTION Fibonacci(n: integer): integer;
Var m: integer; (* local variable *)
begin
    (* uma copia da variavel local m sera criada *)
    (* vamos contar isso: *)
  globalcounter := globalcounter + 1;
  WriteLn('m existe ', globalcounter, ' vezes');
  if (n=1) OR (n=2) then
     Fibonacci := 1
  else
     Fibonacci := Fibonacci(n-1) + Fibonacci(n-2);
    (* a copia da variavel local m sera anihilada *)
    (* vamos contar isso: *)
  globalcounter := globalcounter - 1;
  WriteLn('m existe ', globalcounter, ' vezes');
end;

begin
  globalcounter := 0;
  WriteLn(Fibonacci(5));
end.
```

output:

```
m existe 1 vezes
m existe 2 vezes
m existe 3 vezes
m existe 2 vezes
m existe 3 vezes
m existe 2 vezes
m existe 1 vezes
m existe 2 vezes
m existe 3 vezes
m existe 2 vezes
m existe 3 vezes
m existe 4 vezes
```

```
m existe 3 vezes
m existe 4 vezes
m existe 3 vezes
m existe 2 vezes
m existe 1 vezes
m existe 0 vezes
5
```

Aula prática 9

---

1. Escreva um programa que calcule o saldo na conta bancária no fim de cada ano. O programa deve pedir ao utilizador
   1) O capital no início.
   2) A taxa de juros.
   3) O ano de começar e o ano de acabar a tabela.

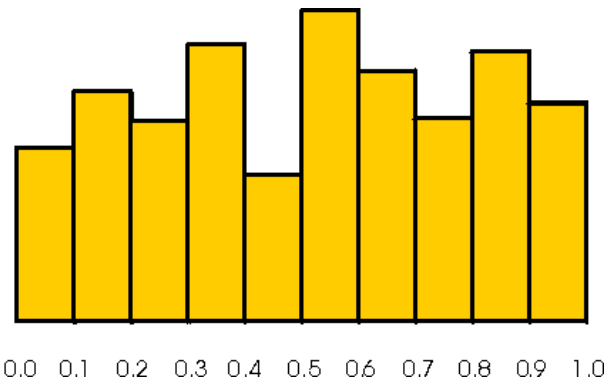
---

2. Declara num programa um record para guardar a informação de uma data. Tem os seguintes campos para guardar
1) dia da semana ('domingo' até 'sabado').
2) dia do mês.
3) mês.
4) ano.

Agora escreva um program que vai pedir ao utilizador os valores para cada campo e depois o programa deve calcular o número do dia do ano (por exemplo, dia 3 de fevereiro é dia 34 do ano).

---

3. Escreva um programa que vai gerar 1000 números aleatórios entre 0.0 e 1.0. O programa deve contar quantos números caiam nos dez intervalos 0..0.1, 0.1..0.2, 0.2..0.3, 0.3..0.4, 0.4..0.5, 0.5..0.6, 0.6..0.7, 0.7..0.8, 0.8..0.9, e 0.9..1.0.
Sugestões: Usa um array de dez elementos para contar. Usa a função `Trunc` (aula 14) para determinar no qual intervalo um número caiu.



---

4. Escreva um progama que declara um array de 100 números. Vamos guardar lá os primeiros 100 números primos.
Então vamos usar um array de tipo _____.
Escreva o resto do programa para calcular os números primos.

---

soluções

Soluções da aula prática 9

1.

```
PROGRAM Banco;

Var taxa: real;
    capital: real;
    i, ano1, ano2: integer;

begin
  Write('Capital : ');
  ReadLn(capital);
  Write('Taxa de juros: ');
  ReadLn(taxa);
  Write('ano comeco: ');
  ReadLn(ano1);
  Write('ano fim: ');
  ReadLn(ano2);
  WriteLn(i,'  ',capital:0:2);
  for i := ano1+1 to ano2 do
    begin
      capital := (1+taxa/100.0)*capital;
      WriteLn(i,'  ',capital:0:2);
    end;
end.
```

2.

```
Faz parte do trabalho pratico
```

3.

```
PROGRAM ContaAleat;

Var bins: array[1..10] of integer;
    i: integer;
    binnr: integer;
    aleat: real;

begin
   (* inicializar o array: *)
  for binnr := 1 to 10 do
    bins[binnr] := 0;
   (* gerar 1000 numeros: *)
```

```
  for i := 1 to 1000 do
    begin
        (* gerar um numero: *)
      aleat := Random;
        (* determinar o numero da caixa: *)
      binnr := Trunc(10*aleat)+1;
        (* aumentar o conteudo da caixa: *)
      bins[binnr] := bins[binnr] + 1;
    end;
   (* mostrar o resultado: *)
  for binnr := 1 to 10 do
    begin
      write((binnr-1)/10.0:0:1,'..',binnr/10.0:0:1,'   : ');
      writeln(bins[binnr]);
    end;
end.
```

possível output:

```
0.0..0.1   : 100
0.1..0.2   : 99
0.2..0.3   : 101
0.3..0.4   : 97
0.4..0.5   : 90
0.5..0.6   : 108
0.6..0.7   : 103
0.7..0.8   : 90
0.8..0.9   : 110
0.9..1.0   : 102
```

4.

```
PROGRAM Primos100;

Var primos: array[1..100] of integer;
    numero: integer;
    nprimos: integer;

FUNCTION IsPrimo(n: integer): boolean;
 (* retorna TRUE se n e numero primo *)
Var i: integer;
begin
  IsPrimo := TRUE;
  For i := 2 To n-1 Do
    if (n Mod i) = 0 then  (* a Mod b = 0 significa a e divisivel pelo b *)
      IsPrimo := FALSE;
end;

begin
  (* Nao sabemos quantos vezes temos de executar o ciclo *)
  (* no minimo uma vez: entao ciclo repeat-until *)
  (* mas tambem facilmente possivel com ciclo while-do *)
  (* Ao primeiro inicializar os parametros de controlo: *)
  nprimos := 0;    (* ainda nao temos nenhum primo *)
  numero := 1;     (* vamos comecar com determinar se 1 e primo *)
```

```
  repeat
    If IsPrimo(numero) then
      begin
         (* se numeor for primo, faz a administracao: *)
        nprimos := nprimos + 1;
        primos[nprimos] := numero;
      end;
    numero := numero + 1;
  until (nprimos=100);
    (* mostrar o resultado: *)
  for nprimos := 1 to 100 do
    Write('primo',nprimos,'=',primos[nprimos],' ');
end.
```

output (recortado):

```
primo1=1 primo2=2 primo3=3 primo4=5 primo5=7 primo6=11 primo7=13 ... primo100=523
```

Aula prática 10

---

1. Escreva um programa que pede ao utilizador 10 números. O programa deve ordenar os números e mostrar a lista ordenada.

2.

- Define um array (100 elementos) de records com campos `numero` (integer) e `div3` (boolean).
- Enche o array com números aleatórios entre 14 e 114. (guardar no campo `numero`)
- Determine quais os records contêm valores divisíveis por 3. Guarda isto no campo `div3` de cada record.
- Determine quantos valores foram divisível por 3.

Usa funções e procedimentos onde possível. Define novos tipos de variáveis!

---

3. Escreva um programa que guarda a informação dos seus amigos. As informações são nome e número telefónico. Usa um procedimento para inicializar o array com os dados (que não vêem do utilizador). O programa deve pedir um nome ao utilizador e como resposta dá o respectivo número telefónico. Exemplo do programa a funcionar:

```
Diz um nome:
alex
O numero do alex e 289123456
```

---

4. Declare

1 um word `w`
2 um integer `i`
3 um apontador ao um word `wp`
4 um apontador ao um integer `ip`

Faça uma atribuição –1 ao `i` e 1 ao `w`

Faça uma atribuição do endereço de `w` ao `wp` e o endereço de `i` ao `ip`.
Mostra o conteúdo dos endereços `wp` e `ip`.

Faça o contrário: uma atribuição do endereço de `i` ao `wp` e o endereço de `w` ao `ip`.
Mostra o conteúdo dos endereços `wp` e `ip`.

---

5. Determine quanto tempo leva executar os programas `TestSpeed` da aula teórica 20.

s e s

---

soluções

Soluções da aula prática 10

1.

```
PROGRAM Orden;

CONST N = 10;

Var ra: array[1..N] of real;
    i: integer;
    changed: boolean;
    temp: real;

begin
   (* pedir 10 numeros: *)
  for i := 1 to N do
    begin
      WriteLn('Numero ',i,':');
      ReadLn(ra[i]);
    end;
  changed := TRUE;
   (* continue ate nao houve mais mudancas: *)
  while changed do
    begin
      changed := FALSE;
      for i := 1 to N-1 do
          (* se ra[i] maior que ra[i+1] troca os dois *)
          (* e asinala que houve mudanca: *)
        if ra[i]>ra[i+1] then
          begin
            temp := ra[i];
            ra[i] := ra[i+1];
            ra[i+1] := temp;
            changed := TRUE;
          end;
    end;
  for i := 1 to N do
    Writeln(i,'  ',ra[i]:0:1);
end.
```

2.

```
PROGRAM Recs;

type rec =
  record
    numero: integer;
```

```
    div3: boolean;
  end;
Var ra: array[1..100] of rec;
    i: integer;
    totdiv3: integer;

begin
  for i := 1 to 100 do
    ra[i].numero := Random(101)+14;
  for i := 1 to 100 do
    if ((ra[i].numero MOD 3) = 0) then
      ra[i].div3 := TRUE
    else
      ra[i].div3 := FALSE;
  totdiv3 := 0;
  for i := 1 to 100 do
    if (ra[i].div3=TRUE)
      then totdiv3 := totdiv3+1;
  WriteLn(totdiv3);
end.
```

3.

```
PROGRAM Recs;

type rec =
  record
    nome: string;
    numero: longint;
  end;
Var ra: array[1..100] of rec;
    s: string;
    i: integer;

PROCEDURE Initialize;
begin
  ra[1].nome := 'Alex';
  ra[1].numero := 289123456;
  ra[2].nome := 'Sandra';
  ra[2].numero := 289234567;
  ra[3].nome := 'Joao';
  ra[3].numero := 289345678;
  ra[4].nome := 'Carla';
  ra[4].numero := 289456789;
  ra[5].nome := 'Hugo';
  ra[5].numero := 289567890;
end;

begin
  Initialize;
  Writeln('Diz um nome:');
  ReadLn(s);
  for i := 1 to 5 do
    if (ra[i].nome = s) then
```

```
      WriteLn('O numero do ', s, ' e ', ra[i].numero);
end.
```

4a.

```
PROGRAM Recs;

Var w: word;
    i: integer;
    wp: ^word;
    ip: ^integer;

begin
  i := -1;
  w := 1;
  wp := Addr(w);
  ip := Addr(i);
  WriteLn('ip: ', ip^);
  WriteLn('wp: ', wp^);
end.
```

output:

```
ip: -1
wp: 1
```

4b.

```
PROGRAM Recs;

Var w: word;
    i: integer;
    wp: ^word;
    ip: ^integer;

begin
  i := -1;
  w := 1;
  ip := Addr(w);
  wp := Addr(i);
  WriteLn('ip: ', ip^);
  WriteLn('wp: ', wp^);
end.
```

output:

```
ip: 1
wp: 65535
```

O programa do lado direito mostra que um integer com valor -1 é igual a 1111111111111111 binário, o que é igual a 65535 se for um word.
O conteúdo do um edereço depende do tipo de informação que fica lá.

5.

Veja aula teórica 20.

Aula prática 10

Esta aula consta de duas partes. A primeira está destinada ao alunos(as) que fazem esta disciplina pela primeira vez. A segunda parte está destinada aos repetentes que já fizeram os exercicios da primeira parte anteriormente. Contudo, os alunos(as) novos que desejem, podem fazer os exercicios da segunda parte em substituição da primeira.
Recomendação geral: *Fazer sempre o algoritmo de cada exercício com lápis, papel, bonecos e desenhos. Só quando tiver claro o que vai fazer, é que escreve as instrucções em Pascal.*

Parte A

---

1. Escreva um programa que pede ao utilizador 10 números. O programa deve ordenar os números e mostrar a lista ordenada.

2.

- Defina um array (capaz de conter 100 elementos) de *records* com os seguintes campos:
  a) `numero` de tipo inteior e `b) div3` de tipo boleano.
- Preencher este array com números aleatórios entre 14 e 114 e guardar no campo `numero`.
- Determine um a um todos registos cujos valores no campo *numero* são divisíveis por 3. Caso isto seja verdadeiro, armazenar esta resposta no campo `div3` do registo correspondente.
- No fim revise o array todo e calcule quantos valores são divisíveis por 3.
  Utilizar funções e procedimentos onde quer que seja possível. Defina novos tipos de variáveis!

---

3. Escreva um programa que guarda informações sobre os seus amigos. As informações são *nome* e *número telefónico*. Utilice um procedimento para inicializar o array com os dados (que não vêem do utilizador).
Posteriormente, o programa deve ser capaz de: pedido um nome ao utilizador e responder devolvendo o respectivo número telefónico. Exemplo do programa a funcionar:

```
Diga um nome: cheila
O numero de telefone da cheila é 289.693459
```

---

4. Declare

1 uma variável do tipo word `w`
2 uma variável do tipo integer `i`
3 um apontador ao um word `wp`
4 um apontador ao um integer `ip`

Atribuir os valores –1 ao `i` e 1 ao `w`

Agora faça uma atribuição do endereço de `w` ao `wp` e o endereço de `i` ao `ip`.
Mostra o conteúdo dos endereços `wp` e `ip`.

Seguidamente faça o contrário: atribuir o endereço de `i` ao `wp` e o endereço do `w` ao `ip`.
Mostrar o conteúdo dos endereços `em wp` e `ip` e aquele das variáveis apontadas pelos mesmos endereços.

5. Determine quanto tempo leva executar os programas `TestSpeed` da aula teórica 20.

s e s

B. Destinada aos alunos(as) repitentes.

---

6. Escreva um programa que faz o desenho dum triângulo de Pascal com oito (8) linhas. Como se sabe, este triângulo tem muitas propriedades interessantes para as matemáticas e as estatísticas. Nele encontramos números primos, números triangulares, a serie de Fibonacci, combinatória, etc.

Na imagem seguinte pode-se ver o exemplo do triângulo de Pascal que queremos construir:

							1							
						1		**1**						
					1		**2**		**1**					
				1		**3**		**3**		**1**				
			1		**4**		**6**		**4**		**1**			
		1		**5**		**10**		**10**		**5**		**1**		
	1		**6**		**15**		**20**		**15**		**6**		**1**	
1		**7**		**21**		**35**		**35**		**21**		**7**		**1**

O triângulo de Pascal é construído com um algoritmos simples. Será o mesmo utilizado por você para fazer na linguagem do mesmo nome do matemático francês:

1. A primeira linha tem um ‘1’ no centro. (o centro da maior linha, ou seja a última.)
2. Nas restantes linhas o valor de cada célula (ou coluna) corresponde **a soma dos valores nas diagonais da linha precedente** . Por exemplo, a linha 3 tem ‘1’ na coluna 6 e ‘2’ na coluna 8. Por tanto na coluna 7 da linha 4 corresponde a soma de 1 + 2 = 3.
   Isto é: linha[n+1, x] = linha[n, x-1] + linha[n, x+1].
3. Para calcular a soma das colunas na diagonal da linha precedente, imagine sempre que as posições em branco correspondem ao valor zero. De facto você pode trabalhar com uma matriz assim(zeros ou valores), e só no momento de imprimir pode converter os zeros em espaços em branco.
4. O número de colunas em cada linha está relacionado com o número de linhas que desejamos preencher. Você vai descobrir facilmente. Uma vez calculado isso, você sabe onde é que será inserido o primeiro ‘1’ da primeira linha.
5. O primeiro valor inserido em cada linha é sempre ‘1’ e é inserido na coluna anterior (na esquerda) respecto a posição onde começou a linha precedente, excepto a primeira linha claro. (0 + 1 = 1). Por isto, é fundamental saber sempre onde foi inserido o primeiro valor da linha anterior.

Mais dicas:
Criar uma função que recebe como argumentos una linha e uma coluna, e devolve a soma das diagonais da linha precedente. Pode ser muito útil.
Pode trabalhar só com duas linhas: actual e precedente e imprimir linha a linha. Porque depois de tudo, o único necessário saber para preencheer e imprimir uma linha são os valores que tive a linha precedente.

Alternativamente pode utilizar a matriz completa tal como no gráfico e imprimir no fim, após preenchida a matriz toda. Em quaisquer caso, não esqueça de inicializar os arrays utilizados.

Para saber mais sobre o triângulo de Pascal ver por exemplo em:
http://mathforum.org/dr.math/faq/faq.pascal.triangle.html

7.

Escreva um programa que guarda a informação sobre 10 elementos a sua escolha da **tabela periódica de elementos** e que permita a sua consulta. (actualmente esta tabela tem 103 elementos como sabe.)
Para informação e obtenção de dados para o seu exemplo, ver os sites:
http://www.cdcc.sc.usp.br/quimica/tabelaperiodica/tabelaperiodica1.htm
ou também:
http://www.brasil.terravista.pt/Albufeira/1895/

Para fazer, crie um array de registos ("*record")* em Pascal onde vai armazenar toda a informação interessante para cada elemento: Número, Nome, Símbolo, camadas, massa, ponto ebulição, ponto de fusão, nome do(s) descobridor(es), etc. Escolha pelo menos **cinco** informações distintas sobre cada elemento. Lembre-se que cada tipo de dado distinto merece também um **tipo** distinto em Pascal.
Uma vez carregada a tabela, faz uma função que **quando é dado o número do elemento,** a função **devolve a posição** (indice) na matriz onde está situado o elemento. Isto pela sua vez permite a um outro procedimento mostrar os dados armazenados no correspondente registo.

---

soluções

Soluções: Aula prática 10

Parte B.

---

6. Escreva um programa que faz o desenho dum triângulo de Pascal com oito (8) linhas. Como se sabe, este triângulo tem muitas propriedades interessantes para as matemáticas e as estatísticas. Nele encontramos números primos, números triangulares, a serie de Fibonacci, combinatória, etc.

```
{-----------------------------------------------------}
program Tpascal;
{programa que constroi o triângulo de Pascal com 8 linhas. Pode ser adaptado
a qualquer tamanho. P.Serendero }
uses crt;
const max_col = 15;
      max_lin = 8;
var centro: word;
   a : array[1..max_lin, 1..max_col] of word;
{-------------------------------------------------------------------}
procedure limpar_array;
{procedimento que inicializa todo o arranjo com zeros }
var i,j : word;
begin
   for  i:= 1 to max_lin do
        for j:= 1 to max_col do
            a[i, j]:= 0;
end;
{-----------------------------------------------------------------------}
function SomaDiag(lin, col: word):word;
{esta função calcula o valor a inserir numa posição, que corresponde a
soma dos valores existentes nas diagonais da linha precedente }
begin
   if (col+1 > max_col) then
      SomaDiag:= a[lin-1, col-1]
   else
      SomaDiag:= a[lin-1, col-1] + a[lin-1, col+1];
end;
{-------------------------------------------------------------------}
procedure imprimir;
{imprime o triangulo de Pascal. Cada posiçãoo que tem 0 , impressa como
um espaço livre. }
var lin, col, i : word;
begin
  clrscr;
  for i:= 1 to 3 do writeln;

  for lin:= 1 to max_lin do
     begin
```

```
        write('            ');
        for col:= 1 to max_col do
           if (a[lin, col] = 0) then
                write('  ')
           else write(a[lin, col]);
        writeln;
     end;
end;
{-----------------------------------------------------------------}
function centro_linha: word;
{inserir um 1 no centro do array correspondente a primeira linha }
begin
    centro_linha:= (max_col div 2) + 1;
end;
{-----------------------------------------------------------------}
procedure constroi_triangulo(centro: integer);
{sen optimiza‡äes. Verifica todas as posições para construção do triângulo}
var lin, col, inicio : word;
begin
    a[1][centro]:= 1;          {no centro da primeira linha so um 1 }
    for lin:= 2 to max_lin do   {algoritmo para as restantes linhas }
      begin
         centro:= centro - 1;
         for col:= centro to max_col do
            a[lin, col]:= SomaDiag(lin, col);
      end;

end;
{-----------------------------------------------------------------}
begin
     limpar_array;
     centro:= centro_linha;
     constroi_triangulo(centro);
     imprimir;
     readln;
end.
```

7.

Escreva um programa que guarda a informação sobre 10 elementos a sua escolha da **tabela periódica de elementos** e que permita a sua consulta. (actualmente esta tabela tem 103 elementos como sabe.)
Para fazer, crie um array de registos (“*record”)* em Pascal onde vai armazenar toda a informação interessante para cada elemento: Número, Nome, Símbolo, camadas, massa, ponto ebulição, ponto de fusão, nome do(s) descobridor(es), etc. Escolha pelo menos **cinco** informações distintas sobre cada elemento. Lembre-se que cada tipo de dado distinto merece também um **tipo** distinto em Pascal.
Uma vez carregada a tabela, faz uma função que **quando é dado o número do elemento,** a função **devolve a posição** (indice) na matriz onde está situado o elemento. Isto pela sua vez permite a um outro procedimento mostrar os dados armazenados no correspondente registo.

```
{-----------------------------------------------------------------------}
program tabelap;
Uses Crt;
```

```
type elemento = record
              numero     : integer;
              nome       : string[25];
              simbolo    : string[5];
              camadas    : real;
              massa      : real;
            end;

var  tabela:  array[1..10] of elemento;
    i : word; r: char;
{---------------------------------------}
procedure procura_mostra(n: integer);
begin
    writeln('      Elemento seleccionado: ');
    writeln('numero : ',tabela[n].numero);
    writeln('nome   : ',tabela[n].nome);
    writeln('simbolo: ',tabela[n].simbolo);
    writeln('camadas: ',tabela[n].camadas:0:4);
    writeln('massa  : ',tabela[n].massa:0:4);
    readln;
end;
{-------------------------------------------------------------}
procedure carrega_elemento(n:integer);
begin
    writeln('Favor entrar os dados dum elemento: ');
    write('numero : '); readln(tabela[n].numero);
    write('nome   : '); readln(tabela[n].nome);
    write('simbolo: '); readln(tabela[n].simbolo);
    write('camadas: '); readln(tabela[n].camadas);
    write('massa  : '); readln(tabela[n].massa);
end;
{-------------------------------------------------------------}
begin
   clrscr;
   for i:=1 to 10 do
      carrega_elemento(i);
   readln;
   clrscr;
   r:= 'x';

   while (r <> 'n') do
     begin
         write('Qual elemento quer ver? '); readln(i);
         procura_mostra(i);
         writeln; write('Deseja continuar? '); readln(r);
     end;
   writeln('vocˆ decidiu terminar. Adeus.');
   readln;
end.
```

Aula prática 11

Once upon a midnight dreary, while I pondered, weak and weary
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
" 'Tis some visitor," I muttered, "tapping at my chamber door -
Only this, and nothing more."

The Raven, Edgar Allan Poe, 1845.

1a. Neste aula vamos escrever um programa que vai ler um ficheiro. Como exemplo vamos ler um ficheiro que contem o poema *The Raven* de Edgar Allan Poe (um excerto, a primeira parte, fica no início desta página) e vamos contar quantas vezes cada letra aparece no texto.

- Só conta as letras normais ('A' .. 'Z')
- Não distingue entre 'A' e 'a', etc. Usa a função UpCase para converter 'a' para 'A', etc.
- Põe de lado todos os outros caracteres.

Funções e procedimentos úteis (além das funções da aula teórica 21):

`Ord(c)`
: Retorna o "número" do caracter c, ou seja o código ASCII do caracter. Por exemplo: `Ord('A')` é igual a `65`, `Ord('Z')` é igual a `90`. Veja tabela no fim da página.

`Chr(n)`
: É o inverso da função `Ord`. `Chr` retorna o caracter com código ASCII `n`. Por exemplo: `Chr(65)` é igual a `'A'`, `Chr(97)` é igual a `'a'`.

`EoF(f)`
: Retorna TRUE se estamos a ler no fim do ficheiro `f`. `f` é uma variável do tipo `text`. Veja aula 21.

`UpCase(c)`
: Retorna a letra maiúscula do caracter c. Por exemplo: `UpCase('f')` é igual a `'F'`. `UpCase('G')` é igual a `'G'`.

Aqui encontra-se o ficheiro `theraven.txt` com o texto inteiro do *The Raven* de Edgar Allan Poe. Guarda este ficheiro no seu disco (Y:). Carrega o botão direito do rato em cima da palavra azul 'aqui' e depois: Em Microsoft Internet Explorer "Save Target As...". Em Netscape Navigator "Save Link As...".

1b. Muda o programa da 1a de forma que vai guardar a informação no ficheiro 'contas.txt'.

1c. Qual letra aparece o mais frequente? (escreve um programa). Isto segue a regra 'ETAOIN SHRDLU' (em inglês)

ASCII table and description

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

---

soluções

Soluções da aula prática 11

1a.

```
PROGRAM EdgarAlanPoe;

Var f: text;
    c: char;
    ci: integer;
    cont: array[65..90] of integer;
    i: integer;

begin
  for i := 65 to 90 do cont[i] := 0;
  Assign(f, 'theraven.txt');
  Reset(f);
  While NOT Eof(f) do
    begin
      Read(f, c);
      c := UpCase(c);
      ci := Ord(c);
      if (ci>=65) AND (ci<=90) then
        cont[ci] := cont[ci] + 1;
    end;
  Close(f);
  for i := 65 to 90 do
    writeln(Chr(i),'  ',cont[i]);
end.
```

output (ecrã)

```
A  339
B  94
C  71
D  194
E  618
F  94
G  122
H  290
I  318
J  2
K  32
L  225
M  158
N  374
O  370
P  95
Q  9
R  336
```

```
S  278
T  437
U  121
V  66
W  79
```

1b.

```
PROGRAM EdgarAlanPoeF;

Var f: text;
    c: char;
    ci: integer;
    cont: array[65..90] of integer;
    i: integer;

begin
  for i := 65 to 90 do cont[i] := 0;
  Assign(f, 'theraven.txt');
  Reset(f);
  While NOT Eof(f) do
    begin
      Read(f, c);
      c := UpCase(c);
      ci := Ord(c);
      if (ci>=65) AND (ci<=90) then
        cont[ci] := cont[ci] + 1;
    end;
  Close(f);
  Assign(f,'contas.txt');
  Rewrite(f);
  for i := 65 to 90 do
    writeln(f, Chr(i), '  ', cont[i]);
  Close(f);
end.
```

1c.

```
PROGRAM EdgarAlanPoeC;

Var f: text;
    c: char;
    ci: integer;
    cont: array[65..90] of integer;
    i: integer;
    max: integer;
    maxi: integer;

begin
  for i := 65 to 90 do cont[i] := 0;
  Assign(f, 'theraven.txt');
```

```
  Reset(f);
  While NOT Eof(f) do
    begin
      Read(f, c);
      c := UpCase(c);
      ci := Ord(c);
      if (ci>=65) AND (ci<=90) then
        cont[ci] := cont[ci] + 1;
    end;
  Close(f);
  max := 0;
  for i := 65 to 90 do
    if cont[i]>max then
       begin
         max := cont[i];
         maxi := i;
       end;
  WriteLn('Max: ', Chr(maxi), '  ', cont[maxi]);
end.
```

output (ecrã)

```
Max: E  618
```

Aula prática 12

---

1. Faz um programa que determine quanto tempo leva dobrar o capital numa conta do banco. O utilizador deve dar os dados relevantes (por exemplo a taxa de juros).

---

2. Declare um array de 1000 elementos. Enche o array com números aleatórios e escreve o código para ordenar o array com o segundo algoritmo da aula teórica 22.

---

3. O programa que se segue deveria calcular o factorial e o somatório de um número introduzido pelo utilizador, no entanto tem alguns erros. Assinale e corrija os erros do programa para que realize o que é pretendido.

```
Program Factorial;

Var num, somatorio, factorial: integer;

begin
  Writeln('Indique um numero inteiro');
  ReadLn(num);
  while (num>0) do
    begin
      factorial := factorial * num;
      somatorio := somatorio + num;
    end;
  writeln('Factorial ', factorial, 'Somatorio ',
          factorial);
end.
```

---

4. Define um novo tipo de variável para guardar um coordenado ou vector (x, y, z). Depois escreve uma função que recebe um coordenado e devolve o comprimento do vector (a distância até o origem).

---

soluções

Soluções da aula prática 12

1.

```
PROGRAM DobrarCapital;

Var cap, juros: real;
    ano: integer;

begin
  Writeln('Taxa de juros (%): ');
  ReadLn(juros);
  ano := 0;
  repeat
    cap := cap + cap*(juros/100.0);
    ano := ano+1;
  until cap>=2.0;
  writeln('numero de anos: ', ano);
end.
```

2.

```
PROGRAM Ordenar1000;

Const N = 1000;
Var ra: array[1..N] of real;
    i, j: integer;
    min: real;
    jmin: integer;
    temp: real;

begin
  for i := 1 to N do
    ra[i] := Random;
  for i := 1 to N-1 do
    begin
      min := 2.0; { com certeza maior que numero maximo do array }
      jmin := i;
        { procura minimo no resto do array }
      for j := i to N do
        if ra[j]<min then
          begin
            min := ra[j];
            jmin := j;
          end;
        { troca o numero minimo com ra[i] }
      temp := ra[i];
```

```
      ra[i] := ra[jmin];
      ra[jmin] := temp;
    end;
   { mostrar resultado }
  for i := 1 to N do
    writeln(i:4,'  ',ra[i]:0:6);
  readln;
end.
```

3.

```
Program Factorial;

Var num, somatorio, factorial: integer;

begin
  Writeln('Indique um numero inteiro');
  ReadLn(num);
  factorial := 1;
  somatorio := 0;
  while (num>0) do
    begin
      factorial := factorial * num;
      somatorio := somatorio + num;
      num := num - 1;
    end;
  writeln('Factorial ', factorial, 'Somatorio ',
          somatorio);
end.
```

4.

```
PROGRAM Distance;

type coordenate = record
  x, y, z: real;
end;

FUNCTION VectorLength(co: coordenate): real;
begin
  VectorLength := Sqrt(Sqr(co.x) + Sqr(co.y) + Sqr(co.z));
end;

begin
  { codigo principal }
end.
```
