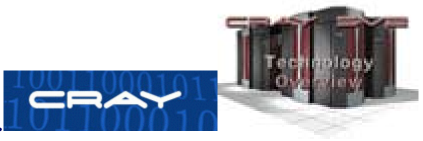
Lecture 2: Computers

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**.

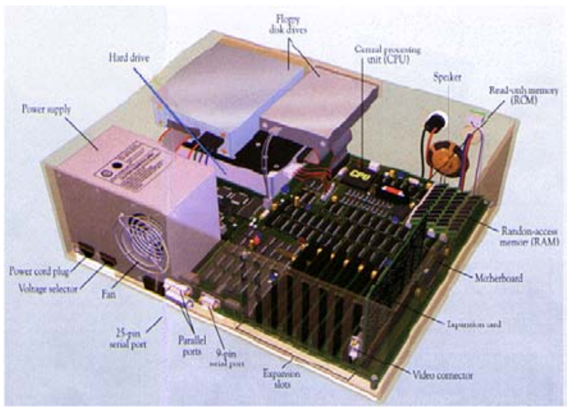
	Examples: IBM PC and compatibles with a microprocessor like the Intel Pentium IV, 1.5 Ghz. Portable computers are a portable variant of microcomputers Apple Macintosh encorporating the Motorola processor family.
Processors	Many home appliances, like washing machines and ovens, contain a small processor 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

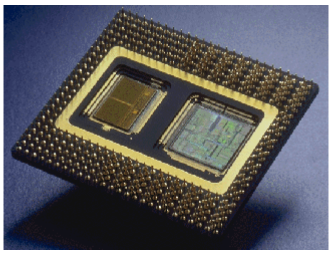
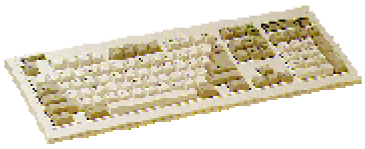
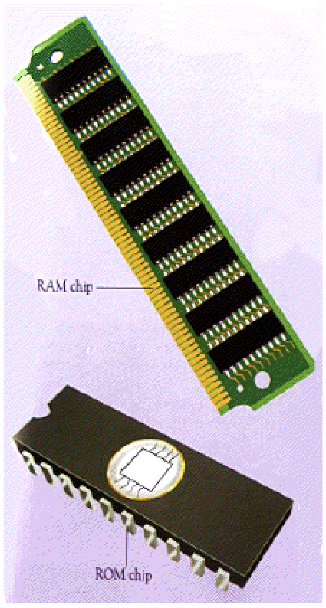
Two important things to distinguish are hardware and software. Hardware is everything that you can feel and touch. Sofware is the programs that are running on the hardware. Example are given below.

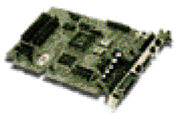
Every working computer consists at least of the following:
1) A processor
2) Memory to store the program
3) An output device
4) A program running
Most computers also have
5) An input device, to either change the program running, enter new data to be processed, or control the processes running.



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

Table: hardware elements

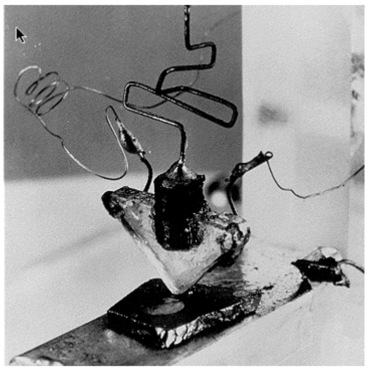
Mouse		input device	to control the processes of the computer
CPU		processor	Central Processing Unit is what is doing all the work. Calculating, controlling data flow, etc.
Joystick		input device	input data to a game
Keyboard		input device	to give instructions to the computer or enter data to be processed
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

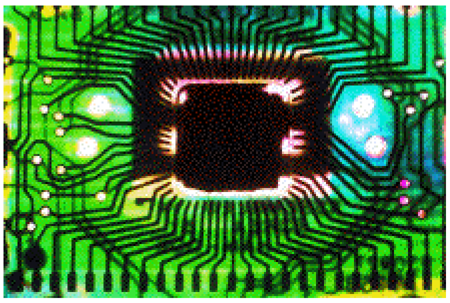
At the lowest level we can determine the level of **Physics**. Electrons (and holes) are responsible for the electrical conduction of the material. The materials used in computers is called semiconductors, meaning that they have a resistance in-between metals (like copper and gold) and isolators (like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

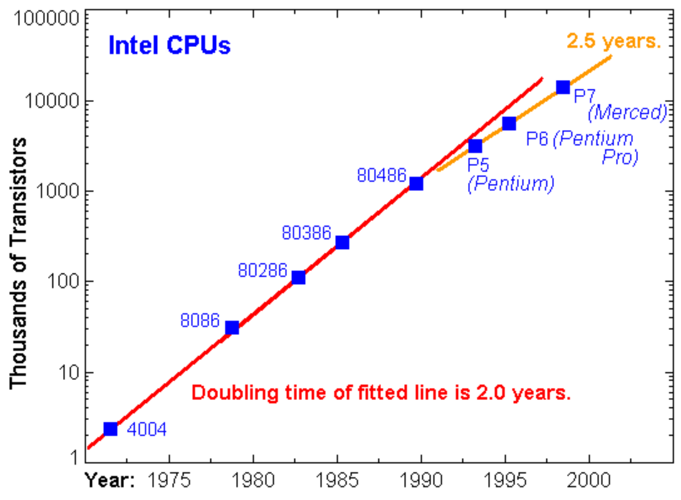
At the next level we find **Electronics**. Electronics connect materials with different properties and the resulting electronic components show remarkable and usefull behavior. Note the diode which conducts current only in one direction, or the transistor, whose conductivity is controlled by an external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

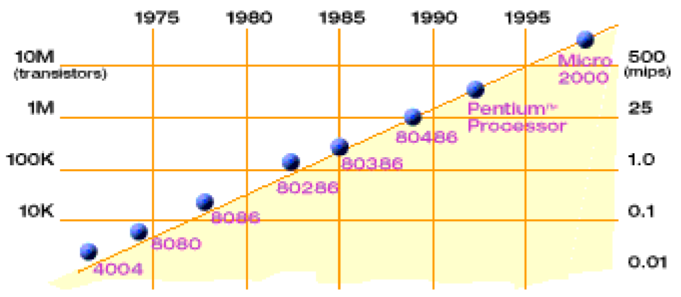


The picture shows the first transistor, as invented at Bell Labs in 1947.

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

```
  00110111000111111100011110100001111
```

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!

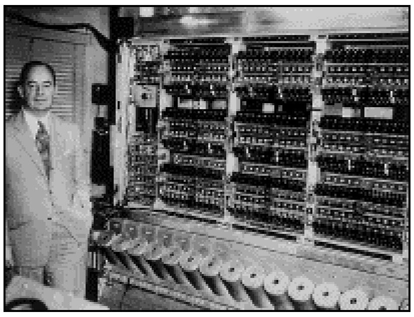
Peter Stallinga, Universidade do Algarve, 6 fevereiro 2002

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

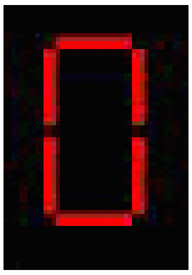
These ideas can be mixed at our wish. For example, PASCAL uses the ideas 'TRUE' and 'FALSE' for logical calculations, while C uses '1' and '0'.

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 hexadecinal 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.

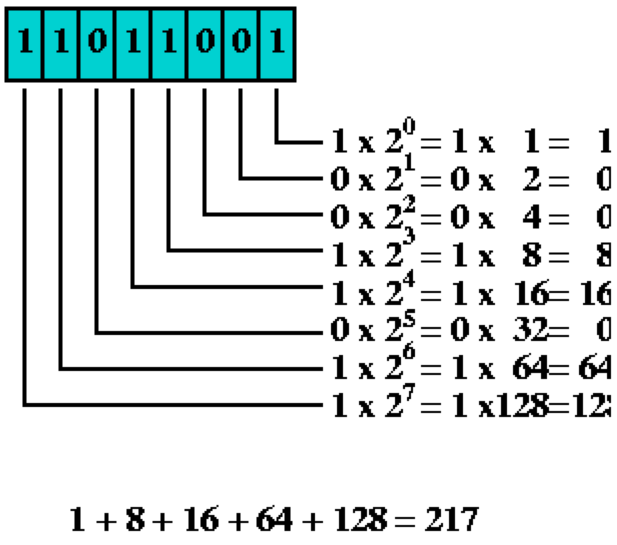
BYTE

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

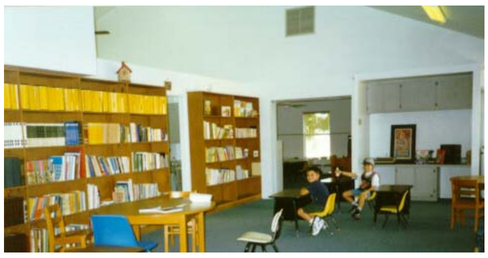
The 1.7 GHz specifies the speed of the CPU (the central processing unit). 1.7 GHz thus means that it can do 1.7 million simple instructions per second.
(Since most commands given to the processor take more than one simple instruction, the actual number of commands per second is lower. For example an 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

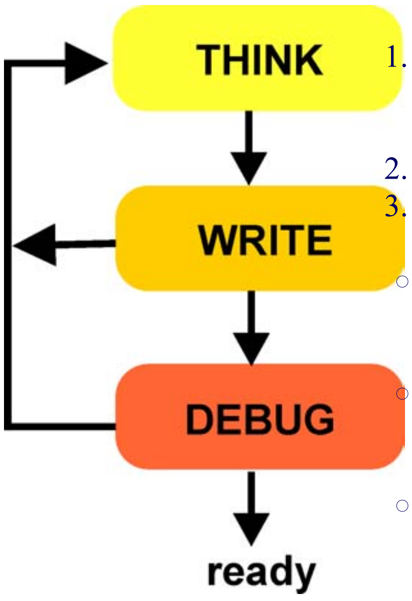
Lecture 4: Introduction to Programming

Any program is nothing more than a set of instructions for the computer. The computer will execute the command one after the other, in principle in the order as they are written (apart from so-called branching instructions (if, if..else, switch) that we will see later). It will do nothing more and nothing less than what we tell it to do.

Moreover, we have to give (write) the instruction with a lot of care. The computer understands only the thing we taught it to understand

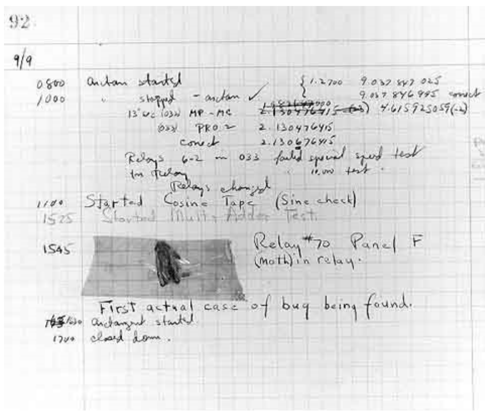
Software engineering

Creating programs always consists of the following steps:



1. Think! Study and analyze the problem. Collect information. Come up with a possible solution. Don't touch that keyboard yet! Use pen and paper.
2. Write a program. Use whatever editor to enter your program
3. Eliminate the errors from the program. This is called "debugging". There are several types of errors:
   - compile-time errors. Writing errors (For example when we write prinntf instead of printf). Easy to elliminate; the compiler is going to help us by telling us something like "type-mismatch error in line 34"
   - programmatic errors. For example forgetting to initialize (setting to a certain value) our variables. This can also cause so-called "run-time errors", for example "division by zero error".
   - logical errors (for example, we do not know that the Pythagoras rule is $a^2 + b^2 = c^2$). The program will run without generating errors, but the result will not be what we wanted.
4. Analyze the result. Is this what you wanted? Maybe the program wrote "The square-root of 9 is 4" Clearly not what we wanted.
5. If necessary, go to step 3, 2 or even 1.

Spending some more time on point 1 can often save a lot of time in the other steps.



92

9/9

0800 Anctan started
1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" UC (033) MP - MC ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay 11.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

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)

The C programming language

C was invented in Bell Labs in 1971-1973. It was an evolution of the language B, which in turn was based on BCPL. In 1983 the language was standardized and that became the official version. It is probably the most used programming language in the world.
The evolution of C went hand-in-hand with the evolution of the UNIX operating system which we are going to use in our lectures (in the form of Linux, which is a graphical variant of UNIX). In fact, UNIX itself was written in C.

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).

A very simple C program:

```
#include <stdio.h>

main()
{
  printf("Hello World\n");
}
```

Let's take a look at this program.

- Every line that starts with a hash # are instructions for the compiler or linker rather than command that will be executed at run-time. #include <stdio.h> means that the library stdio ("standard input-output") has to be fetched by the compiler in order to understand what is coming. We will use the command printf which is part of the stdio library.
- After this we can write our own procedures and functions (see the lecture on modular programming).
- "main" is a special function. The first instruction of this function is always the first instruction to be executed. For the first couple of weeks we will only write instructions in this function.
- Therefore, the instruction `printf("HellowWorld\n")` is executed first and since it is the only instruction this also finishes the program. The instruction writes the text `Hello World` on the screen and goes to the next line (\n).

Reserved keywords in ANSI C

auto	float	static
break	for	struct
case	goto	switch
char	if	typedef
continue	int	union
default	long	unsigned

`do`	`register`	`void`
`double`	`return`	`while`
`else`	`short`	
`extern`	`sizeof()`	

Note that there is only one function defined in the C language, namely `sizeof()`. All the other functions are described in so-called libraries. For example, printf can be found in the libary stdio. We therefore have to put the compiler directive `#include <stdio.h>` in the beginning of our code.

Identifiers

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

- They should start with a letter; "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 C. They are called **reserved characters**.

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

- Identifiers cannot be equal to **reserved keywords** (and also better to avoid predefined functions) of C, such as words like "int" or "main". Note that identifiers like "int1", "int_" or "Int" are allowed, although it is advised to avoid such confusing names. Note: In many programming environments, 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 variable is used for storing interest rates, call it, for example "interest" and not "variable1". Although it is not an error to give it the name "variable1", 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 case sensitive, "i" is not 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:

```
#include <stdio.h> main() {printf("Hello world!\n")}
```
and
```
#include <stdio.h>

main()
{
  printf("Hello world!\n")
}
```
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**" 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 C 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 C comments are placed after `//` on a single line, or between `/*` and `*/` for multi-line comment.
- Use **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 FileReadLn(). This will make the program more readable, more efficient and shorter.

```
          main(l
      ,a,n,d)char**a;{
  for(d=atoi(a[1])/10*80-
 atoi(a[2])/5-596;n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAfHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
 H E L L O,    W O R L D! "
   [l++-3];)for(;n-->64;)
      putchar(!d+++33^
          l&1);}
```
The program above shows an example of how NOT to program. Do you manage to predict what the program does? Don't worry, neither do the specialists. If you want to know what will be the output of this program, click here.
(program copied from http://www.ioccc.org/)

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, 14 outubro 2002

Lecture 5: Variables

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 C there are many types of basic variables. The exact implementation of the types of variables depends a little on the compiler. For the Borland C compiler we have:

group	*variable type*	*range*	*size it occupies in memory*
II	unsigned char	0 .. 255	8 bit = 1 byte
II	char	-128 .. 127	8 bit = 1 byte
II	int	–32 768 .. 32 767	16 bit = 2 bytes
II	unsigned int	0 .. 65 535	16 bit = 2 bytes
II	long	–2 147 483 648 .. 2 147 483 647	32 bit = 4 bytes
II	unsigned long	0 .. 4 294 967 295	32 bit = 4 bytes
III	float	–3.4E–38 .. 3.4E38	32 bit = 4 bytes
III	double	–1.7E–324 .. 1.7E308	64 bit = 8 bytes
III	long double	–3.4E–4932 .. 1.1E4932	80 bit = 10 bytes
IV	char	#0 .. #255 ASCII	8 bit = 1 byte

Notes:

- The convention of writing exponents in the scientific notation: 2.9E–39 means 2.9 x 10^{-39},

1.7E308 means 1.7 x 10^{308}, etc.
- In C there is no variable type for boolean information (bit). Instead, where boolean information is needed, this is implemented by the use of integers, where "false" is equal to 0 and all the other values are "true".
- char is used (or can be used) both for ASCII information (text) as for small short integer numbers (0..255 or -128..127)

I **Boolean** is used to store and manipulate information of the type true-or-false or yes/no. As we have seen in lecture 3, this is one bit of information. In C this type of variable does not exist. Instead it is emulated with int's, where 'false' is equal to 0 and any other number is equalto 'true'.

FIVE, NO, SIX DINOSAURS COMING.
HOW CAN YOU POSSIBLY TELL THAT BY LISTENING TO THE GROUND?
EASY, CONAHONTY COUNT FOOTSTEPS AND DIVIDE BY 4.
9-17

II **int**, **unsigned int**, and **long** and **char** and **unsigned char** 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. **Unsigned char** and **int** only store positive numbers, while **int** and **long** can store both positive and negative numbers, at the cost of limited maximum value. The **char** occupies the least memory, only 8 bits, but the range of values it can take is therefore very limited, only 256 different values. If we need to store larger numbers, we have to use **int**. If we want to store even larger numbers and want positive and negative numbers we use **long**. Note: since a **char** has 8 bits, it can store 2^8 = 256 different numbers: 0 .. 255 or -128..127. The same calculation we can make for the 16-bit units **int** and **unsigned int**: 2^{16} = 65536, numbers from 0 .. 65535 for **unsigned int**, and -32768 .. 32767 for **int**. Remember the calculations of lecture 3.

III **Float**, **double** and **long double** 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 **float**, it occupies only 4 bytes, at the cost of smaller precision in our calculations. The best is **long double**, 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 type is used for storing text. **Char** is used for a single character of ASCII code. Since char can be used to store information of the type integer numbers as well as ASCII characters, we have to be very careful to not make mistakes.

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 C

Variables: Declaration!

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 C we declare variables by writing the name of the variable preceded by the type of the variable. For example

```
  int i;
```

The place to do that is in the beginning of the function (main), before the first real instructions For example:

```
main()
{
  int i;
  float f;
  long li;

  instruction1;
  instruction2;
}
```

Everytime when the program runs and an instruction like `int i;` is encountered, space is reserved in memory and the address of this space is remembered for later reference when we need to store information in this box or retrieve a value from it. This space is released at the end of the function (or program in case of `main`).
For the experts: Variables are placed on the stack and stay there until the scope of the variables has expired.

If we want to define more variables of the same type, we can do that on a single line with the variables separated by commas, although it is nor prohibited to use several lines to define different variables of the same type:

```
main()
{
  int i, j, k;
  float f;
  float g;
  int n;

  instruction1;
  instruction2;
}
```

Forgetting to declare variables will cause a compiler error.

Problems with variables I: Type mixing

Always be careful when using diferent types of values in calculations. Consider the following code

```
main()
```

```
{
  int i, j;
  float f;

  i = 1;
  j = 3;
  f = i/j;
  printf("%f", f);
}
```

What will be the value of f at the end? in other words, what will be the output of the program?
NOT 0.3333 as we might expect. Instead the value of `f` will be 0.0000. This is because the division calculation is done with int's and in integer calculation 1 divided by 3 is 0. This is then attributed to `f`.
To avoid this we can

 * force the type of the value in the calculations. Thisis called "type casting". For example:

```
    f = (float) i/ (float) j;
```

 * Where we are calculating with constants we can force the type by including the seemingly redundant floating point part:

```
    f = 1.0 / 3.0;
```

 instead of

```
    f = 1 / 3;
```

Problems with variables II: Overflow

Consider the following program

```
main()
{
  int i, j, k;

  i = 20000;
  j = 20000;
  k = i + j;
  printf("%d", k);
}
```

Again, what will be the output of the program?. Naively we might think 40000, but `k` is of the type int and int's only go up to 32767. What is done with the rest? What happens is that the overflowing part "reenters at the bottom of the range". In this case we will get -25536. An overflow can occur when we add two numbers with limited range. An underflow can occur when we substract 2 numbers.
This is easier shown with a byte-sized variable (a byte has 8 bits and a range from 0 to 255 and is called `unsigned char` in C):

```
i      130 =    10000010
j      130 =    10000010
i+j   (260)=   100000100
```

What happens is that the 9th bit in the above equation is ignored and the result will be binary `00000100` which is equal to 4 in the decimal system. Therefore, for byte calculation 130 + 130 = 4.

What can be done to prevent this

- Always use variable types of sizes large enough to store the results of any possible outcome of the calculations. In the above example we should use (at least) int's. In the first example we should use

long int's.
- Some languages have the option to check for this at run time. This is called "range checking", which will also check for "array index out of bounds" as we will discuss later in the lecture about arrays.

Problems with variables III: Initialization

Declaring a variable does not assign a value to the variable, it only reserves space for it in memory!

```
main()
{
  int day;

  printf("Today is day %d\n", day);
}
```

In the program above, the output *might* be

```
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 there, 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".

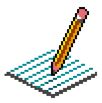
Note: In many programming languages it is possible to assign a value to a variable at the moment of declaration. To do that in C we can use

```
  int day = 0;
```

---

printf

The instruction `printf` is one of the most useful instructions in C and is used for for showing information on the screen: text, numbers, values of variables, etc. Nearly every program has somewhere a `printf` statement. We have already seen some example above and skipped the discussion about how this functions works until now.



`printf` is part of the standard-input-output library (`stdio.h`). The output is formatted, which means that we can specify how exactly the information should be shown (how much space it should take onthe screen, how many significant digits should be shown for floats, etc.). The format is always the first argument of the function. The next argument(s) are the information to be shown, separated by commas.

```
  printf("%d", i);
```

will show the value of the integer `i`

```
  3
```

for every piece of information we have to specify the format:

```
  printf("%d %d", i, j);
  3 5
```

other format specifiers that are interesting for us:

%d	integer with sign
%i	integer with sign
%u	integer without sign
%x	hexadecimal
%f	float (31.2000) for example %10.3f shows the float with 10 spaces on screen and 3 decimal cases
%e	float with scientific notation (3.12E1)
%c	ASCII character
%s	string

Note that we can also directly write text in the format:

```
printf("Hello%d World", i);
```

which will give as output (when i=3)

```
Hello3 World
```

There are special characters that we can use to control the position of the output. There are written with an \. The most important is:

\n	goto to the beginning of the next line

For example

```
printf("Hello\n%d World", i);
```

will give as output (when i=3)

```
Hello
3 World
```

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, 22 October 2002

Lecture 6: Assignment, `scanf` and calculations

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 C 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 float):

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

In reality, for C the last example was correct. C knows what we want and helps us by converting the integer 1 to the real 1.0. In any case, it is bad practice to mix floats with ints and we'd better write: `a = 1.0;`

The symbol `=` should be distinguished 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 C). 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 from the mathematical equation. 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'.

The symbol = should also be distinguished from the comparison symbol =. For example `a=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.

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 takes the value of b")

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
`main()`		
`{`		
`  float a;`		
`  float b;`		
`  float c = 4.3;`		
		`undefined`
`  a = 1.0;`	`undefined`	`undefined`
`  b = c;`	`1.0`	`4.3`
`  a = a + b + c;`	`1.0`	`4.3`
`}`	`9.6`	

Formatted input: `scanf`

In the previous lectures we have learned how to show the results of our calculations on the screen. With `printf` 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.

C: With `scanf` we can get information from the keyboard. They can read values and characters

scanf

from the keyboard and directly store them into specified variables. The general form the instructions take is

```
scanf("format
decription(s)",
&var1, &var2, ...);
```

Note the symbol `&`. This means "the address of ..." We have to give the address of the variable(s) to the function `scanf`. As we will see later, giving the address of a variable is equal to "passing by reference" and this allows for changes of the value of the variable by the function. For the moment it suffices to say that we have to give the address of the variables in `scanf`. In the analogy of the boxes we could see it as giving the box to the function who then fills it with a value.

The format descriptions are the same as for the output instruction `printf`. For example `%d` for int's, etc. Note that the variables to be read do not have to be of the same type.

As an example

```
// lines starting with these // are cooment
// always when we use input or output
// we have to include the standard-I/O library:
#include <stdio.h>

main()
{

// don't forget to declare the variables we will use
  int n1, n2: integer;

  printf("Please enter two numbers separated by a space\n");
  scanf("%d", &n1);
  scanf("%d", &n2);
  printf("n1 = %d  n2 = %d", n1, n2);
}
```

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
```

Operations, operators and operands

After learning how to assign values above, 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 calculations, need two *operands* and are called binary operators. In C 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 above. 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.

In C There also exists unary operators which need only one operand:

operator	operation
`++`	increment
`--`	decrement
`!`	negate (boolean algebra)
`~`	invert all bits of this int
`-`	negate number
`&`	address of ..

For example

```
i++;
--k;
```

Increases the value of i and decreases the value of k by 1 respectively. The operator can be placed before or after the operator. The difference is the moment the variable is changing value, before execution of the rest of the instruction (`++i`) or after (`i++`). For example

`i = 1;`
`j = i++;`
 	
at the end
i is equal to 2
j is equal to 1
	`i = 1;`
	`j = ++i;`
	
	at the end
	i is equal to 2
	j is equal to 2

The ! and ~ operators we will see later in the lecture on Boolean algebra.
The get-address operator `&` will be discussed in the lecture on pointers.
The unary - operator returns the negated number (floating point or integer).

Combinations of calculation and assignment.

C has thte peculiar and confusing possibility to use a calculation and assigment at the same time. For example:

```
    c += a;
```

which is equivalent to

```
    c = c + a;
```

Any binary operator can be used in this form (`*=`, `/=`, `+=`, `-=`).
This is confusing and as such should be avoided. On the other hand, it can prevent errors. For example:

```
  matrixelements[i+2*k-offset][n+m-f] = matrixelements[i+2*k-offset][n+m+f] +
matrixelements[i+2*l+offset][n+m-f]
```

It is easy to make a typing mistake here. As a matter of fact, there is a mistake in there. Did you spot it?.
Simpler and more readable would be:

```
  matrixelements[i+2*k-offset][n+m-f] += matrixelements[i+2*l+offset][n+m-f]
```

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 works a little different. 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:

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

In C exist two operators / and % that reproduce these results. Example:

expression	result
`7 / 3`	2
`7 % 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

```
main()
 /* an example floating point calculation */
{
  double x, y, sum, diff, divis;

  printf("Give the value of the first variable x:\n");
  scanf("%f", &x);
  printf("Give the value of the second variable y:\n");
  scanf("%f", &y);
  sum = x + y;
  printf("The sum of %f and %f is %f\n", x, y, sum);
  diff = x - y;
  printf("The difference between %f and %f is %f\n", x, y, diff);
  divis = x / y;
  printf("%f divided by %f is %f\n", x, y, divis);
}
```

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
```

```
main()
 /* an example integer calculation */
 /* note the different format specifiers in printf and scanf */
{
  int x, y, sum, diff, divis, divr;

  printf("Give the value of the first variable x:\n");
  scanf("%d", &x);
  printf("Give the value of the second variable y:\n");
  scanf("%d", &y);
  sum = x + y;
  printf("The sum of %d and %d is %d\n", x, y, sum);
  diff = x - y;
  printf("The difference between %d and %d is %d\n", x, y, diff);
  divis = x / y;
  divr = x % y;
  printf("%d divided by %d is %d plus %d/%d\n", x, y, divis, divr, y);
}
```

will produce, when running:

```
Give the value of the first variable x:
13
Give the value of the second variable y:
5
```

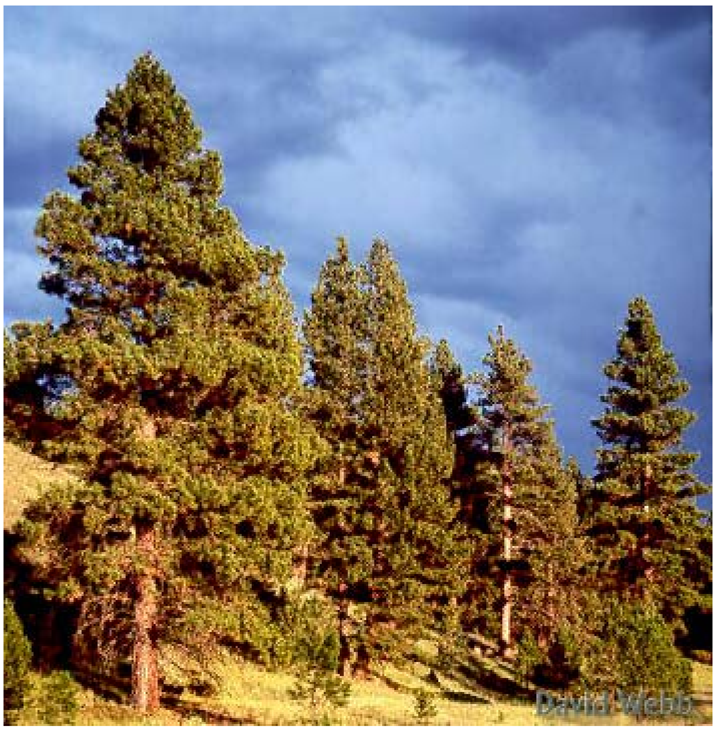
```
The sum of 13 and 5 is 18
The difference between 13 and 5 is 8
13 divided by 5 is 2 plus 3/5
```

Quick test:

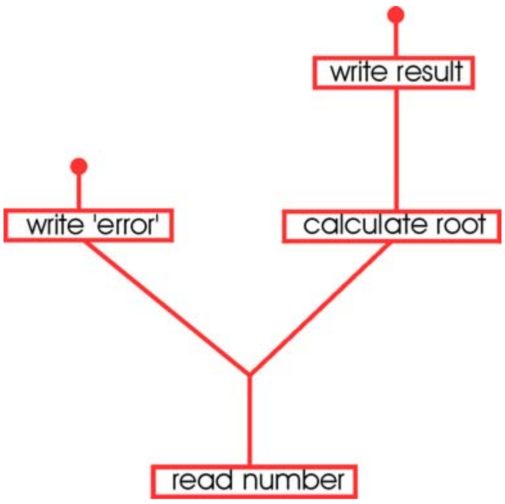
To test your knowledge of what you have learned in this lesson, click here for an on-line test. Note that this is **NOT** the form the final test takes!

Peter Stallinga. Universidade do Algarve, 10 October 2002

Lecture 7: Branching I (`if ...` , `if ... 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 ...

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

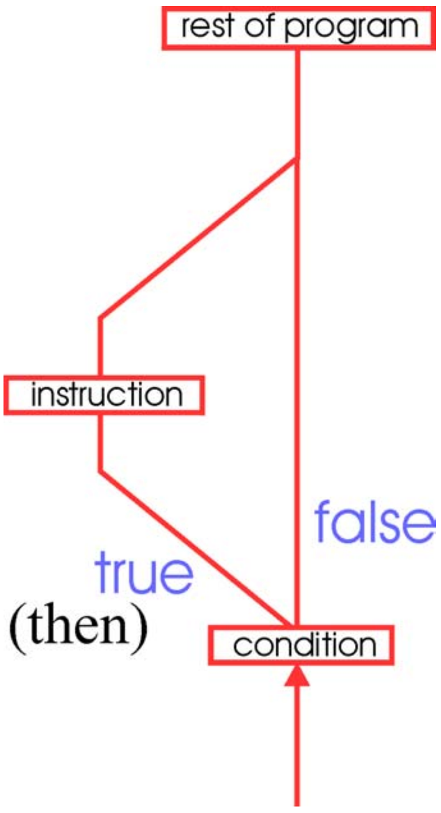
```
if condition
  instruction;
```

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 `int`, the following is correct:

```
if b instruction;
```

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

```
if (x == y) instruction;
if (x < y) 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 { ... } combination, so that for the if statement they appear as one.

```
if (a == b)
  {
    instruction1;
    instruction2;
  }
```

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)
  {
    instruction1;
    instruction2;
  }
instruction3;
instruction4;
```

to be executed:

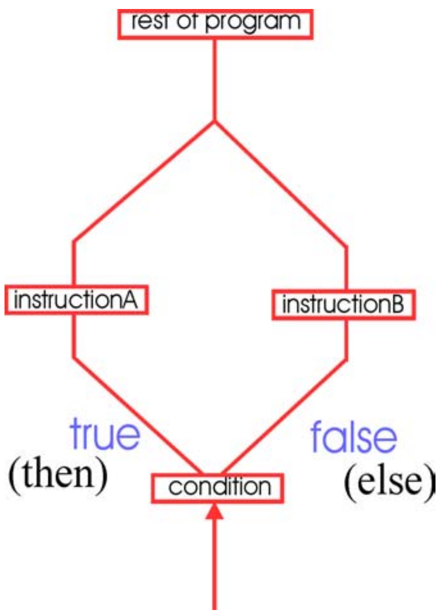
(a equal to b)	(a not equal to b)
`instruction1` `instruction2` `instruction3` `instruction4`	`instruction3` `instruction4`

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 … else …

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

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



example:

```
if (a==b)
  {
    instruction1;
    instruction2;
  }
else
  {
    instruction3;
    instruction4;
  }
instruction5;
instruction6;
```

to be executed:

(a equal to b)	(a not equal to b)
instruction1 instruction2 instruction5 instruction6	instruction3 instruction4 instruction5 instruction6

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

```
#include <stdio.h>

main()
{
   double x;
   double root;

  printf("Give a number");
  scanf("%f", &x);
  if (x<0)
    printf("Negative numbers are not allowed!\n");
  else
    {
      root = sqrt(x);
      printf("The square-root of %0.4f is %0.4f\n", x, root);
    }
  printf("Have a nice day\n");
}
```

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 7: ... 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, 10 October 2002

Lecture 8: Branching II (`switch`) / 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 7) we have seen how we can control the flow of the program by the branching instructions `if ...` and `if ... 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)
    if (x>-2)
      printf("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) && (x>-2))
    printf("Error");
```

which means 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 && condition2;
  if (condition3)
    instruction;
```

logical operators in C

&&	and
||	or

Another important logical operation is XOR, although it is not implemented for Boolean calculation in C. `a XOR b` means "a or b true, but not both!". We will use it later for integer logical operations. Others that are not implemented include NAND and NOR which are only important at the electronic level and are not very useful at the programming level.
Finally, there is the boolean negator !. It means taking the opposite; if `a` is false (0), !`a` is true (1), and vice verse. Whereas `&&` and `||` need two operands (for example `a || b`), ! needs only one (!`a`).
With these 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	C
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)
    printf("TRUE");
  else
    printf("FALSE");
```

```
(a>0)                    TRUE
(b>0)                    FALSE
(a>0) && (b>0)           FALSE
(a>0) || (b>0)           TRUE
!(a>0)                   FALSE
(! (a>0)) || (b>0)       FALSE
(2==b) || (!(2==b))      TRUE
```

Boolean algebra for integers

We can also apply boolean algebra to complete numbers (char, int, long int, 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". To avoid confusion, the symbols for operations on numbers are differents from the ones given above, namely "&" for "AND", "|" for "OR", "^" for "XOR" and "~" for "NOT"

operation	C

AND	&
OR	|
XOR	^
NOT	~

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 =	0 0 1 1 0 0 0 1	
24 =	0 0 0 1 1 0 0 0	
49 | 24 =	0 0 1 1 1 0 0 1	= 57
49 & 24 =	0 0 0 1 0 0 0 0	= 16
49 ^ 24 =	0 0 1 0 1 0 0 1	= 41
~49 =	1 1 0 0 1 1 1 0	= 206
~24 =	1 1 1 0 0 1 1 1	= 231

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

In the table on the left, also the other operations with the numbers 49 and 24 are given.

Note the bit-wise inverter ~ depends on the size of the variable:
For a byte (unsigned char) ~49 = ~(00110001) = (11001110) = 128 + 64 + 8 + 4 + 2 = 206, while for an unsigned int ~49 = ~(0000000000110001) = (1111111111001110) = 65486.

Multiple branching: `switch`

In the previous lecture (see aula 7) we have seen how to use the `if ...` 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 `switch` instruction.
Imagine the following program that asks from what year the student is:

```
main()
 // outputs the lectures to follow on basis of year
{

  int ano;

  printf("From what year are you?\n");
  scanf("%d", &ano);
  if (ano==1)
    printf("primeiro ano: MAT-1, CALC-1\n");
  else
```

```
      if (ano==2)
        printf("segundo ano: INF, LIN-ALG\n");
      else
        if (ano==3)
          printf("terceiro ano: ELEC, FYS\n");
        else
          if (ano==4)
            printf("quarto ano: QUI, MAT-2\n");
          else
            if (ano==5)
              printf("quinto ano: PROJECTO\n");
            else
              printf(">5: AINDA NAO ACABOU?\n");
  }
```

(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
```

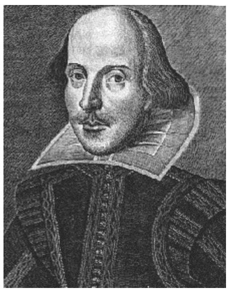
The structure is not very readable, though. To make it better, we can use `switch`. Whereas in "`if (condition) instruction1;`" the condition is necessarily of the boolean type (true or false), in `switch` we can use any type of variable that has discreet values (in contrast to floating point variables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

```
switch (expression)
{
  case value1: instruction1;
         break;
  case value2: instruction2;
         break;
           |
  case valueN: instructionN;
         break;
  default: instructionE;
}
```

The `expression` needs to result in a value of any countable type (for example: `int`, `long int`, but also `char`. Not, for example: `float`). 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, but cannot contain expressions or variables; they have to be of constant type.
The special reserved word "default" means that it will execute this instruction if the `expression` doesn't result in any of the values `value1` .. `valueN`.
The `break` statement forces the program to jump to the end of the loop (the switch-loop in this case).
As an example the above program rewritten to make use of `switch`:

```
  main()
  // same program as above, but with switch statement
 {
   int ano;

   printf("From what year are you?\n");
   scanf("%d", &ano);
   swicth (ano)
    {
     case 1: printf("primeiro ano: MAT-1, CALC-1\n");
             break;
     case 2: printf("segundo ano: INF, LIN-ALG\n");
             break;
     case 3: printf("terceiro ano: ELEC, FYS\n");
             break;
     case 4: printf("quarto ano: QUI, MAT-2\n");
             break;
     case 5: printf("quinto ano: PROJECTO\n")
             break;
     default: printf(">5: AINDA NAO ACABOU?\n");
    }
 }
```

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



Lecture 8: `(2==b) || (! (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, 28 October 2002

Lecture 9: Loops I: `for`

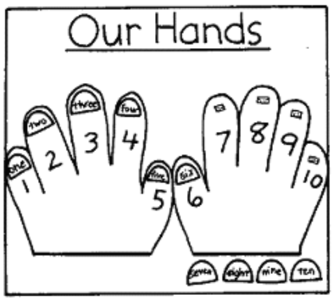
Loops are for repeating part of the code a number of times. In C there exist three types of loops,

- for
- while
- do ... while

There is no difference between the first two types of loops, only a matter of legibility of the program. The do .. while loop differs from the other two by the fact that the condition of exiting the loop is checked at the end, while for the other two (for and while) it is checked in the beginning of the loop.
Todays lecture is about the for loop.

For loop

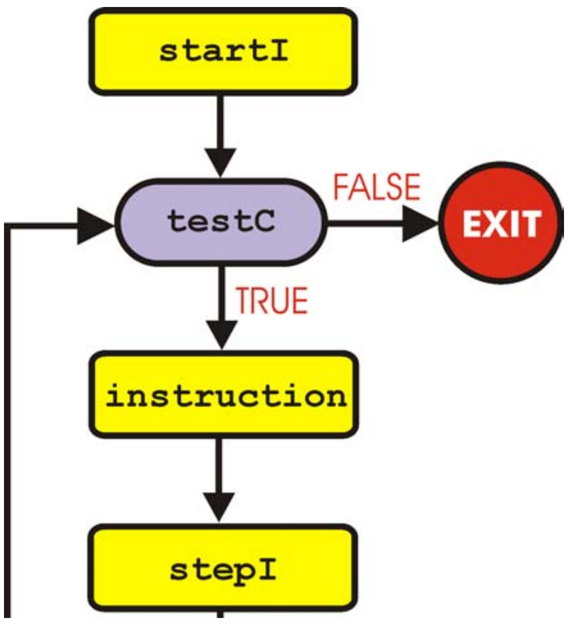
The most common loop in C is the for loop. This loop, in principle, is used to execute things a predetermined number of times in a **countable** way. This in contrast to loops that will run while a certain condition is true, as we will learn in the next lecture.



The general structure of the for loop is

```
for (startI; testC; stepI)
   instruction;
```

with `startI` and `stepI` general instructions and `testC` a Boolean condition. This will repeat the instructions `instruction` and `stepI` until the test condition results in "false".



The `instruction` is repeated a number of times, determined by the control instructions `startI` and `stepI` and the test condition `testC`. The control instructions `startI` and `stepI` can be any instruction or even combination of instructions (separated by commas).
The `startI` instruction is only executed at the beginning of the loop and only executed once. Immediately after that, the condition `testC` is evaluated. If this results is "false" (0), the loop is immediately exited. **Hence it is possible with the for loop that the `instruction` is not even executed a single time** (contrary to the `do .. while` loop as we will see in the next lecture). If the result of the condition is "true" (!=0), the instructions `instruction` and `stepI` are executed.

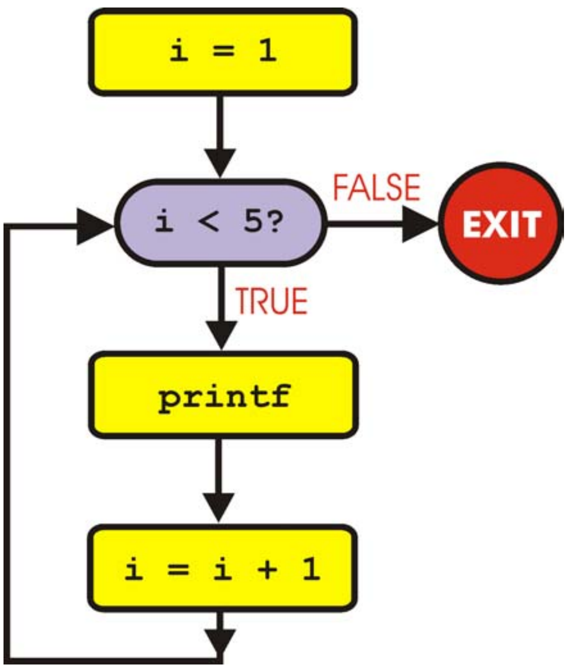
Since the loop is designed for doing things in a **countable** way, it is advised to use control variables of the integer type as in the next example. Do not forget to declare the variable (see lecture 5).

program code

```
main()
// for-loop example;
{
  int i;

  for (i=1; i<5; i++)
    printf("Ola\n");
}
```

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 smaller than 5
3) if not so: EXIT LOOP immediately. Else
4) execute `printf("Ola\n");`
5) add 1 to `i`
6) go to step 2)

Multiple instructions

Just like with the `if ... else ...` structure, we can also group instructions together with `{..}` in loops:

program code

```
for (i = 1; i<5; i++)
 {
    printf("Ola\n");
    printf("It is a nice day\n");
 }
```

output

```
Ola
It is a nice day
Ola
It is a nice day
Ola
It is a nice day
Ola
It is a nice day
```

Compare this with

program code

```
for (i = 1; i<5; i++)
  printf("Ola\n");
  printf("It is a nice day\n");
```

output

```
Ola
Ola
Ola
Ola
It is a nice day
```

Use of the loop variable

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

Good code:

program code

```
for (i=1; i<5; i++)
  printf("%d Ola\n", i);
```

output

```
1 Ola
2 Ola
3 Ola
4 Ola
```

Bad code:

program code

```
for (i=1; i<5; i++)
  {
    printf("%d Ola\n", i);
     i = i + 1;
  }
```

output

```
1 Ola
3 Ola
```

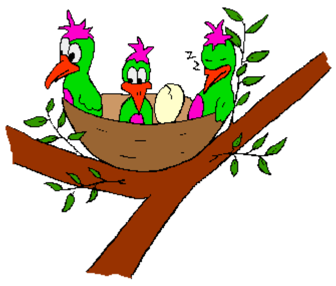
The program on the right is an example of bad code. Such style of programming, although at some times it will save space and maybe executing time, makes your program unstructured and very difficult to understand for others! If you want to achieve things like in the program on the right, use other loops,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

program code

```
for (i=1; i<5; i++)
  printf("%d Ola", 2*i-1);
```

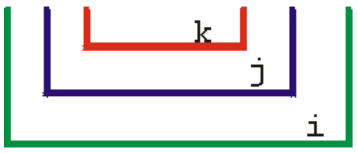
output

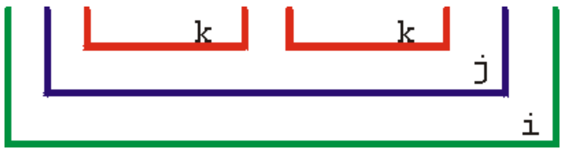
```
1 Ola
3 Ola
```



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

```
main()
// three nested loops
{
 int i, j, k;

 for (i=1; i<=2; i++)
   for (j=1; j<=2; j++)
     for (k=1; k<=2; k++)
```

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
```

program code

```
main()
// three nested loops
{
 int i, j, k;

 for (i=1; i<=2; i++)
   for (j=1; j<=2; j++)
     {
```

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
```

```
      printf("i=%d j=%d k=%d\n",
i, j, k);
}
```

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

```
      for (k=1; k<=2; k++)
        printf("i=%d j=%d k=%d\n",
i, j, k);
      for (k=1; k<=2; k++)
        printf("i=%d j=%d k=%d\n",
i, j, k);
    }
}
```

```
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
```

An example. Fibonacci.

Fibonacci numbers are numbers that follow the following rules:

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

The program below implements this. The user is asked to give the number of Fibonacci numbers to calculate (n) and then, in a for-loop, the next Fibonacci number is calculated, until there are enough numbers.

```
main()
/* program to calculate the first n Fibonacci numbers
   using the algorithm
   F(1) = 1, F(2) = 1 and F(i) = F(i-1) + F(i-2) */
{
  int i, n, fi, fi1, fi2;

  // first ask the user for the number of Fibonacci numbers to calculate
  printf("How many Fibonacci numbers do you want to see?\n");
  scanf("%d", &n);
  fi1 = 1;               // variables to store F(i-1) and F(i-2)
  fi2 = 1;               // initialize them to their starting value
  printf("1 1 ");        // print the first 2 Fibonacci numbers
  // we are going to do something in a countable way, so we will use
  // the for-loop structure:
  for (i=3; i<=n; i++)   // print the rest
    {
      fi = fi1 + fi2;    // calculate the next number
      fi2 = fi1;         // calculate the new F(i-1) and F(i-2)
      fi1 = fi;
      printf("%d ", fi); // print the result
    }
}
```

Later we will learn a much more elegant way to calculate Fibonacci numbers using recursive programming.

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 October 2002

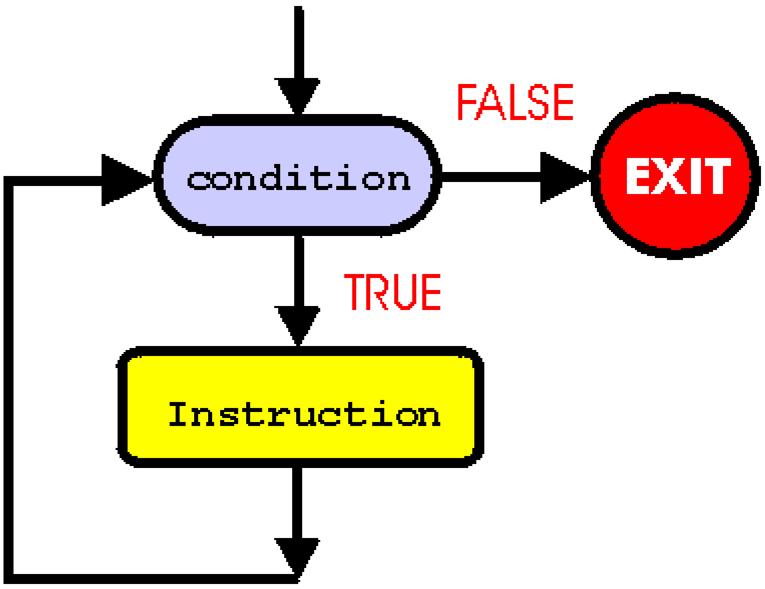
Lecture 10: Loops II: `while ...` and `do ... while`

`while ...`

The loops in this lecture, while and do-while 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 and do-while loops.
As we will show, the difference between the while and the do-while loops is the moment the condition is tested.

The general format of the while loop is

```
while (condition)
  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 does not attribute a starting value to any variable like in the for-loop. So, we have to do this ourselves.
Example:

program code	*output*

```
main()
// while example
{
  float x;

  x = 0.0;
  while (x<=1.0)
    {
      printf("%0.1f\n", x);
```
	```
	0.0
	0.1
	0.2
	0.3
	0.4
	0.5
	0.6
	0.7
	0.8
	```

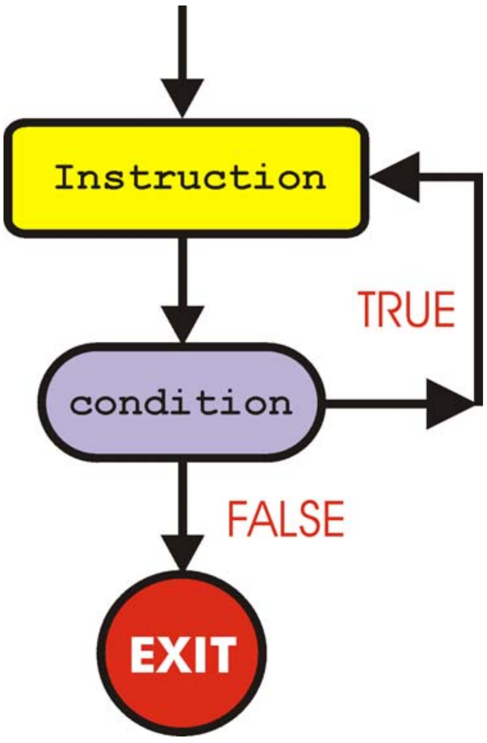
```
        x = x + 0.1;
    }
}
```

```
0.9
1.0
```

do ... while

The do-while structure is very similar to the while structure. Theonly difference is that now the condition is checked at the end of the instructions to be repeated. The general format is

```
do
 Instruction;
while (condition);
```



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

program code

```
x = 100;
do
  {
    printf("Ajax\n");
    x = x + 1;
  }
while (x<=10);
```

output

```
Ajax
```

program code

```
x = 100;
while (x<=10)
  {
    printf("Ajax\n");
    x = x + 1;
  }
```

output

program code

```
for (i=100; i<=10; i++)
  printf("Ajax\n");
```

output

for, while e do-while compared

Effectively, the for and while loops are exactly the same. Both have a condition that is checked in the beginning of the loop and a step instruction. The only difference is that the for-loop already includes a starting instruction. If we precede the while-loop with a starting instruction, the two loops are equal:

for	while

```
for (startI; testC; stepI)
  instruction;
```
	```
	startI;
	while (testC)
	  {
	    instruction;
	    stepI;
	  }
	```

```
for (i=1; i<=10; i++)
  printf("%d", i);
```
	```
	i=1;
	while (i<=10)
	  {
	    printf("%d", i);
	    i++;
	  }
	```

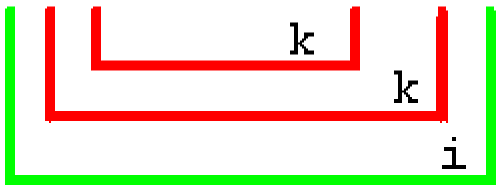
Only for the sake of readability of the program will we use the for-loop for repeating things in a countable way. For all other types of loop will we use while and do-while. In the do-while loop the instructions are at least executed once. Always take these considerations into account when deciding for the type of loop to use.

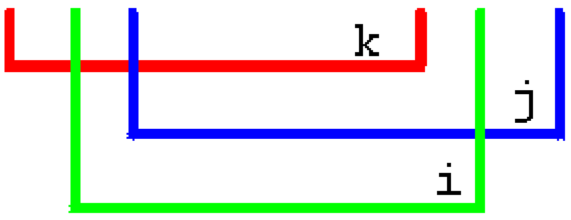
Nesting II

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; i<=10; i++)
  {
    for (i=1; i<=20; i++)
      printf("%d", i);
  }
```

Two loops with equal control variable `i`.

```
for  (i=1; i<=10; i++)
  {
    x = i;
    do
      x = x+0.1;
  }
while (x<20);
```

Two loops not well nested.
Good indentation of your program will always avoid such errors.

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, 4 November 2002

Lecture 11: Modular Programming I: Functions

Until now we have only used the things that were supplied by C. We have learned how to write loops (`for`, `while`, `do-while`), how to have input and output (`scanf`, and `printf`), how to control the flow of the program (`if`, `if-else`, `switch`), how to declare variables, how to assign values to constants (=) and how to calculate, even up to complicated Boolean algebra, but we have never invented anything NEW. With Functions we can do exactly that.
We already met a couple of functions which are standard in C, namely `printf`, e `scanf`. Now we will define our own functions.

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. Moreover, functions allow for easy copying of parts of the code for other programs.

input/output

Modules (functions) can have input and/or output. In this case, input means accepting parameters, while output means returning a value. Note that, regardless of the fact if the functions accepts input parameters, the function always has parenthesis, both in the declaration as in the function call, as we will see later. This is to distinguish functions from variables.

examples	without input	with input
without output	`abort()`	`srand()`
with output	`rand()`	`sqrt()`

`abort()` stops the program, `rand()` returns a random value, `srand()` initializes the random-number generator, `sqrt()` returns the square root of the argument.
This is the main difference between functions we know from mathematics and functions from C. In mathematics, every function has input (the argument) and output (the function value). For example
$f(x) = x^2 - 1$
with x the input and $f(x)$ the output. While in C, we can have functions with and without input and output. Therefore, the more genaral name for functions is "procedures", "(sub)routines", or "modules".

Functions are like programs-within-programs. They

- must have a name. The same rules for identifiers applies to the names of functions. See aula 5.
- can have variables. These have to be declared in the function and are only accessible by the function.
- must have a `{` and `}` combination indicating the start and end of the function.
- (can) have instructions.

A prototype of the declaration of a function:

```
type functionname(type <parameters>)
{
  type <variables>;

  instructions;
}
```

output (type, void, return)

The **type** of the value the function will return has to be specified at the declaration of the function. For example

```
  int countnumber();
```

will mean that the function will return a value of the type int.

Somewhere in the function we **must** return a value to the calling program. We do this with the **return** statement

```
  return (<value>);
```

for example

```
  return (3);
```

When want that the function doesn't return anything, we can specify this with the word **void**:

```
  void showresult();
```

will not return anything. Therefore, functions of type void don't need the return statement.

Note: in some compilers the type declaration of the value to return is optional. In some compilers, the absence of the type declaration signifies that the function is of type `int`, while in others it signifies `void`. To add to the confusion, omitting a return statement is in some compilers an error, while in others it is not.
Therefore: stick to the conventions of standard ANSI C above and write compiler-independent programs!

`main()`

In fact, `main` is nothing more than a normal function, with the only difference that the program always starts with the first instruction of this function main. Otherwise it follows the same rules of functions described above. Technically speaking, we either have to specify the type as void or return something at the end of our program.

```
void main()
{
 ...
}
```

```
int main()
{
 ...
 return (0);
}
```

placement

The place to declare our own fucntions is **before** the function `main`. If we declare them after the function main we will not be able to use them inside main because the compiler does not know them yet when it arrives at compiling main. A function can only call other functions that have been declared (in some way) before.

Calling

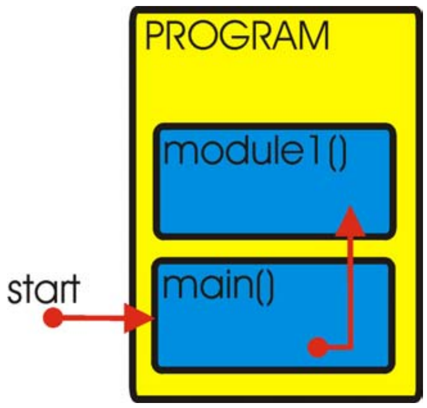
After declaring a new function, we can use it in the main program. This is called **calling** the function. We do this by writing the name of the function. As a complete example of a program with a single module (function) without input parameters or output (void):

program code

```
#include <stdio.h>

void module1()
{
  int y;

  printf("Now I am entering Procedure Module1\n");
  printf("Give a value for y\n");
  scanf("%d", &y);
  printf("%d", 3*y);
// Type of function is void.
// No need to return anything.
}

void main()
{
// The program starts at the first instruction
// of function main:
  printf("Starting the program\n");
// now our function will be called:
  module1();
  printf("Ending the program\n");
}
```

output

```
Starting the program
Now I am entering Procedure Module1
Give a value for y:
 4
12
Ending the program
```



A program calling one of its functions

As seen above, at the end of the function, the program continues with the instruction immediately after the function call. In this case, after `module1()` has finished, it will execute `printf("Ending the program\n");`

Functions calling functions

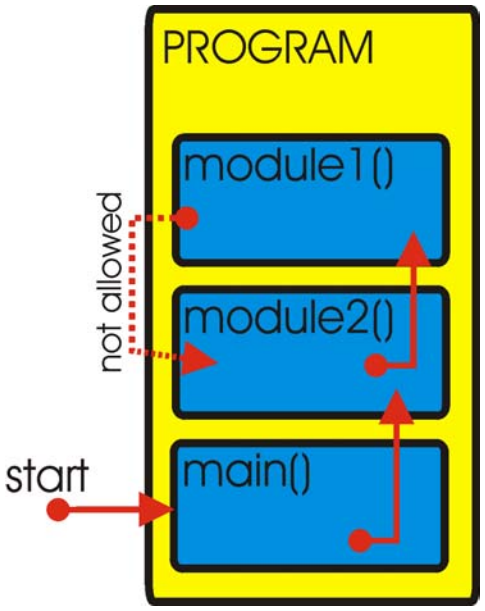
Functions can also be called by other procedures or functions. Under normal circumstances, in C these procedures have to be declared after the procedure to be called, though. Take a good look at the following program and the output it generates when run:

program code

```
# include <stdio.h>

void module1()
{
  printf("    inside module 1\n");
  printf("    Hello World\n");
  printf("    leaving module 1\n");
}


void module2()
{
  printf("  inside module 2\n");
  printf("  calling module 1\n");
  module1();
  printf("  back in module 2\n");
  printf("  leaving module 2\n");
}

void main()
{
  printf("starting the program\n");
  printf("calling module 2\n");
  module2();
  printf("back in main\n");
  printf("ending program");
}
```

output

```
starting the program
calling module 2
  inside module 2
  calling module 1
    inside module 1
    Hello World
    leaving module 1
  back in module 2
  leaving module 2
back in main
ending program
```



A program calling one of its functions which, in turn, is calling one other function

Note that forward function calls, like in the above example a call from `module1()` to `module2()`, are not allowed under normal circumstances. main can call `module1()` and `module2()`, `module2()` can call `module1()` and `module1()` can call nothing.

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 November 2002

Lecture 12: Modular Programming II: functions with input and output

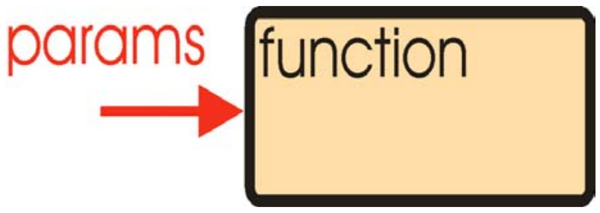
In the previous lecture (aula 11) we have seen functions that are not accepting parameters and are not returning values. These are simple functions. Now we are going to look at functions that are accepting parameters and functions that are producing return values.

Parameters to pass to functions

We can pass parameters to functions. The functions can then work with these parameters. Inside the function, the parameters work like normal variables. In C the parameters the functions expects are put after the name of the functions inside parenthesis. For a function that doesn't have output (or in other words, "returns type void"):

A function with input (but without output):

```
 void
functionname(parameter_list)
{
 <local variable
declarations>

  instructions;
}
```



The variables on the parameter list are declared in the same way as the normal variables of a program or functions, namely we have to specify the type of each parameter. Inside the functions we can use the parameter as if it were a normal variable. We can calculate with it, use it in conditions, and even change its value. It doesn't have to be initialized, though, because the initialzation comes from the calling program.

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

```
#include <stdio.h>

void write_square(float r)
{
  float y;

  y = r*r;
  printf("The square of %f is %f", r, y);
}
```

output

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

```
void main()
{
  float x;

  x = 4;
  write_square(x);
  write_square(3.0);
}
```

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

Another example, that uses two parameters:

program code

```
#include <stdio.h>

void write_sum(int i1, i2)
/* write the sum of i1 and i2 */
{
 int j;

  j = i1+i2;
  printf("The sum of %d  and %d is %d',
    i1, i2, j);
}

void main()
{
  int x, y;

  x = 4;
  y = 5;
  write_sum(x, y);
  write_sum(3, 4);
}
```

output

```
The sum of 4 and 5 is 9
The sum of 3 and 4 is 7
```

Note that we have to pass to the function the type of information that is expected. In this case, the function expects two integers, so we should pass two integers (x and y).
Finally, an example with a parameter list of mixed types. Variables to be declared in the parameter list are separated by a comma ,

program code

```
#include <stdio.h>

void write_N_times(float r, int n);
  /* Will write n times the real r */
{
  int i;

  for (i= 1; i<=n; i++)
    printf("%10.3f\n", r);
}

void main()
{
```

output

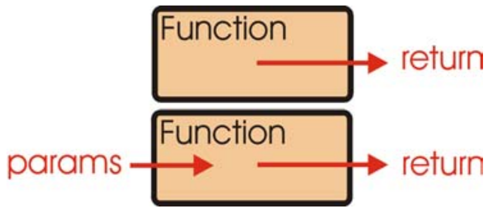
```
3.000
3.000
3.000
3.000
```

```
  write_N_times(3.0, 4);
}
```

Functions with output

Functions can return an output value. The **type** of the returning value has to be specified at the moment of declaring the function before the name of the function

```
type FunctionName(parameter_list)
{
 <variable_list>

  instructions;
 }
```



Functions with output and with input parameters

Somewhere in the instructions we have to specify a **returning value**. We do this with the reserved word `return`

```
return (value);
```

Obviously, the value has to be of the same type as the type of the declaration of the function.
Note that the return instruction immediately exits from the function and the instructions after return will not be executed.

```
double square(double r)
  /* will return the square of of the parameter r */
{
  r = r*r;
  return (r);
}
```

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, for example

```
printf("%f", square(3.0));
```

A full example:

program code	*output*
`/* example with parameters */`	`The square of 4.0 is 16.0`

```
#include <stdio.h>                                       The square of 3.0 is 9.0

double square(double r)
  (* will return the square of of the parameter r *)
{
  r = r*r;
  return(r);
}

void main()
{
  double x, y;
  x = 4.0;
  y = square(x);
  printf("The square of %f is %f\n", x, y);
  printf("The square of %f is %f\n", 3.0, square(3.0));
}
```

What is happening in the instruction `y = square(x)` is the following:

1. The value of the expression inside parenthesis is calculated. In this case this is simple. It is the value of `x`, namely 4.0;
2. This value (4.0) is passed to the function `square()`. Inside the function:
3. A temporary variable with name `r` is created.
4. The value passed to the function is attributed to this variable `r`. Effectively an instruction `r=4.0` has happened.
5. `r=r*r;` The new value of `r` is calculated. r now has the value 16.0.
6. `return(r);` The value of the expression inside parenthesis (16) is passed back to the calling instruction (`y=square(x);`); where it will be used further.
7. In this case, the value returned (square(4.0) which is 16.0) will be attributed to `y`. The instruction `y=square(x);` effectively becomes `y=16.0;`
8. The next line (`printf`...) shows the values of the variables x and y. (**Note that the value of `x` has remained unchanged** and is still 4.0. later, in the lecture on "passing by value / passing by reference" will we see that it is not necessarily like that.)

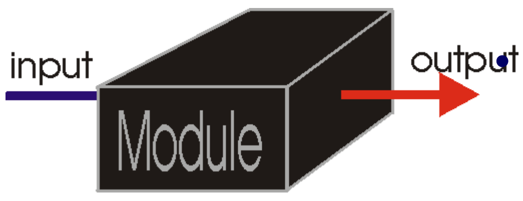
Note that in C we don't have to use the value that is coming back from the function. We could, for example write in our function `main()`

```
  square(3.0);
```

This construction is very confusing and should be avoided when possible in structured programs; a value returned by a function should, in principle, be used on the receiving side.

Why?

Now the big question is "why?". Why write 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 three 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 function.

- For the same reason, we can easily copy parts of other programs or libraries for our purpose (as long as we will only use parameters and local variables, as we will show later). In the ideal case we will just "link" those functions 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). An interesting side effect of this is that we can distribute our functions in a precompiled way, so that the source code remains safe with us.
- With 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, 8 November 2002

Lecture 13: 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

```
  float a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;
  float average;
```

and in the program code:

```
  average = (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10.0;
```

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:

```
  int n;

  scanf("%d", &n);
  switch (n)
   {
     case 1: average = a1;
             break;
     case 2: average = (a1 + a2) / 2.0;
             break;
     case 3: average = (a1 + a2 + a3) / 3.0;
             break;
          |
     case 10: average = (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10.0;
    }
```

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:

type name[numelement] ;

name is the identifier of the array, just like a name for other variables.
numelement defines the number of elements in the array. The elements run from 0 to

numelement-1. This is a little confusing for beginning programmers, who are used to indices from 1 to n. **`type`** is any variable type, for instance float or int.

Examples:

```
   float account[100];
```
This might be used to store the information of 100 bankaccounts.

```
   long int prime[10];
```
This might be used to store the first 10 prime numbers..

```
   int propinas[2000];
```
This might be used to store some simple information of the status of the students with numbers between 0 and 1999, for instance if they paid their tuition fees or not.

Use of an array

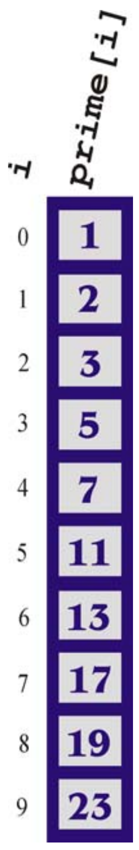
Inside the program we can use the elements of an array.

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 float - of element 20 of the array with name `account`.

```
   prime[8]
```
is the value - of type long int - of element 8 of the array `prime`. The figure on the right might represent this array, so element number 8 would be equal to 19. Of course, our program has to fill this array in some way before the array really contains the prime numbers.



```
   propinas[1055]
```
is 1 or 0 (type int). 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=0; i<20; i++)
    printf("%f\n",account[i]);
```

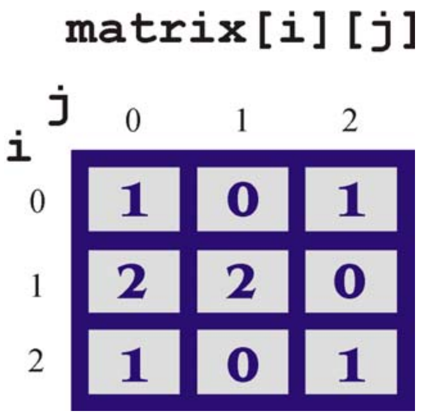
Multiple arrays

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):

type name[numindex1][numindex2];

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]

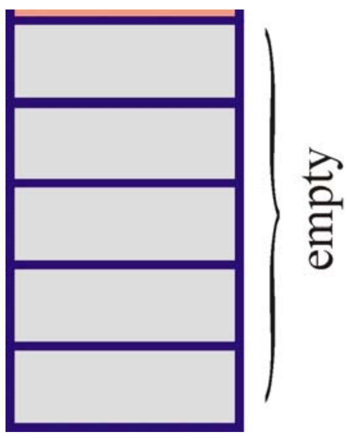


As an example: to write the matrix of the figure on the left we might do the following in a complete program. Note that the array consists of 9 (3x3) elements of type integer.

```
void main()
{
  int  matrix[3][3];

  matrix[0][0] = 1;
  matrix[0][1] = 0;
  matrix[0][2] = 1;
  matrix[1][0] = 2;
  matrix[1][1] = 2;
  matrix[1][2] = 0;
  matrix[2][0] = 1;
  matrix[2][1] = 0;
  matrix[2][2] = 1;
  for (i=0; i<3; i++)
    {
      for (j=0; j<3; j++)
        printf("%d ", matrix[i][j]);
      printf("\n");
    }
}
```

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 less than the total number of elements in the array. If we use a too large index, the results of our program can be very odd. This is best illustrated in an example. The following program defines an array `r` of 4 integers running from `r[0]` to `r[3]`, 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[4]`? If `r[4]` had existed, it would have occupied the place that is now taken by `a`, and an assignment to `r[4]` 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[4]` there anyway, thereby **overwriting the value of `a`**.



```
main()
{
  int a;
  int r[4];

  a = 0;
  printf("a=%d\n", a);
  r[4] = 1;
  printf("a=%d\n", a);
}
```

The output of the program will probably be

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

Some programming languages can check for this at run-rime. This is called range-checking and when the program tries to use a wrong index, a message 'range-check error' or 'array index out of bounds' will be displayed. The disadvantage of doing this is that the program becomes slower and the compiled program will occupy more space in memory and on disk.

Note: it depends on the exact implementation of the language/compiler. With some languages, it will overwrite the variable declared *before* the array as in the example above, while in other languages it will overwrite the variable *after* the array. You can find out by declaring the variables

```
    int a;
    int r[4];
    int b;
```

and see if `r[4]` overwrites `a` or `b`.

"Hurray! I know everything about arrays."

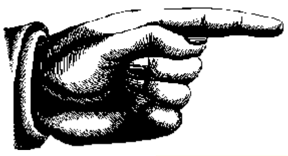
Quick Test

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

Peter Stallinga. Universidade do Algarve, 12 November 2002

Lecture 14: 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.

Declaration

To declare a pointer we can use the following syntax:

```
type *name;
```

With `name` the name of the pointer variable and `type` the type of information the pointer name points to. This can be any type we have learned, from simple floats or ints to more complicated types we will learn later.
Examples:

```
  int *p;
```

Now `p` is a pointer that points to information of the type int.

```
  float *f;
```

Now `f` is a pointer that points to information of the type float. The value of `f` itself is not of type float, because `f` is a pointer and its value is an address. Only the *contents* of the memory address that `f` is pointing to contains information of type float.

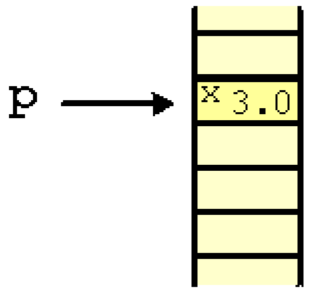
Pointer operations

There are two pointer operations in C.

`&x` returns address of variable **`x`**
`*p` is what pointer **`p`** points to (contents of address p)

Let's declare a variable of type float x and a pointer to floats p:

```
  float x;
  float *p;
```



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

```
  p = &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 can do this with

```
  printf("%f", x);
```

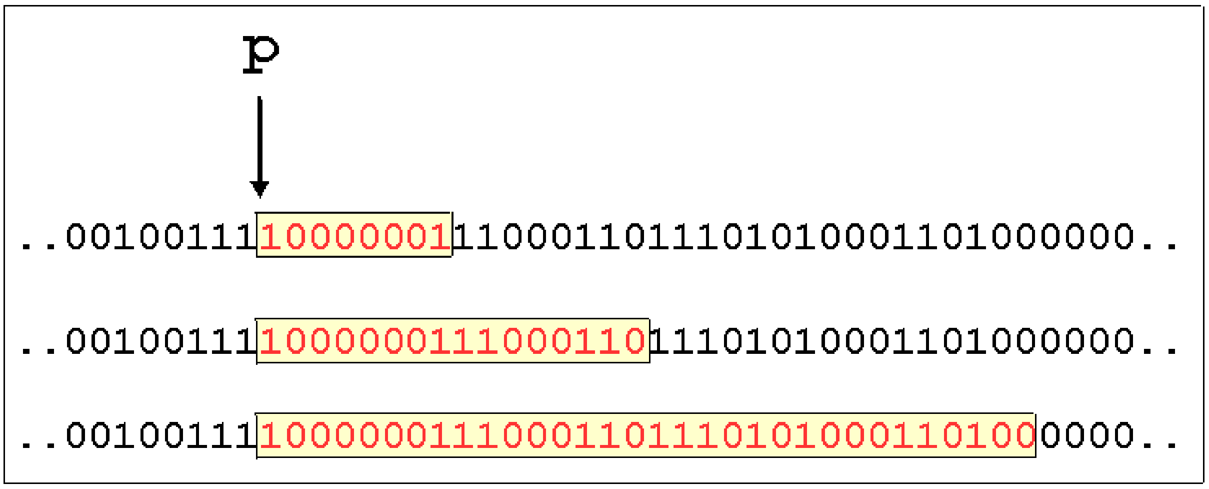
or via the pointer

```
  printf("%f", *p);
```

Type specification

Why is it important that we specify the type the pointer points to? This is best shown in an example

Picture the following situation. The figure below shows a part of the memory and its contents. A pointer `p` points to a place in this memory. If `p` is pointing to a byte (`unsigned char`) as shown in the top line, the contents of the address `p` are `*`p =129 (binary: 10000001), while with `p` pointing to the same place in memory, but pointing to an `unsigned int*`p = 25473 (binary: 0110001110000001), or `p` pointing to a `long int` (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.

Example

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:

```
void main()
{
    /* declare a word (unsigned int) and  a pointer to word: */
```

```
  unsigned int *wordptr;
  unsigned int w;

    /* assign a value to the word */
  w = 25473;

    /* let a 'pointer to word' point to our word */
  wordptr = &w;
    /* show the contents of the memory wordptr points to */
  printf("%d", *wordptr);
}
```

output:

```
 25473
```

Now let's see a more complicated example. We are going to create a pointer of type 'pointing to a byte (**unsigned char**)', and let it point to our **unsigned int**:

```
void main()
{
    /* declare a word (unsigned int) and  a pointer to word: */
  unsigned char *charptr;
  unsigned int w;

    /* assign a value to the word */
  w = 25473;

    /* let a 'pointer to char' point to our word */
  charptr = &w;
    /* show the contents of the memory wordptr points to */
  printf("%d", *charptr);
}
```

output:

```
 129
```

(output for Borland C++ version 3.1 for MS-DOS. On other computers or versions the output might be different)
This shows that we have to be careful what our pointer points to. The value depends on the type!

Pointers and arrays in C

In C, all arrays are pointers. What this means is that when we declare an array

```
  int a[10];
```

this will
 - reserve 10x2 bytes in memory
 - assign the address of this memory to `a`
Therefore,
 `a` is of type 'pointer to int' and the value of `a` is an address.
 `*a` is the contents of address `a`. In this address resides the first element of `a`, namely `a[0]`.
Therefore, the two forms, `*a` and `a[0]` are completely interchangeable and any of the two can be used at any time. We will see this later, when we discuss strings (arrays of `char`).
We have to remember this when we pass information to a function; when we pass an array, we pass the **address** of the array, rather than all the elements of the array. With arrays in C we always use the technique of passing by reference (see lecture 15).

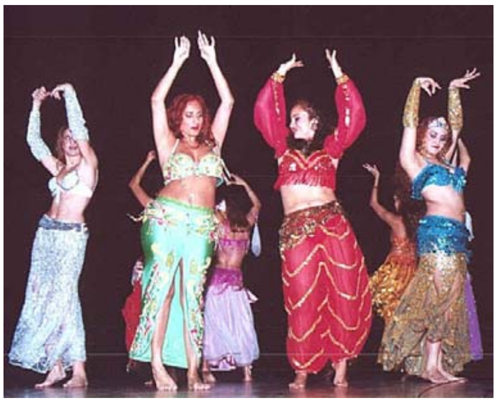
Initialization

Initialization of a variable is even more important for pointers. Without initialization, a pointer points to a random part of memory, where important programs might be running. These programs and the computer can crash when we write in this place. The following program might crash the computer. It defines a pointer and doesn't assign an address to it. The value of the pointer (the address) is therefore unpredictable. The program then writes a value in this random address.

```
void main()
{
  int *p;

  *p = 0;
}
```

Why?

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



- Speed
- Flexibility

Speed: Imagine you want to write a progam that moves a lot of information around. In the conventional way, this would mean copying a lot of bytes from one part of the memory to another part. Take for example the sorting of an array with name.
(a pointer is just 4 bytes in Intel computers).

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. 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.

Quick Test

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

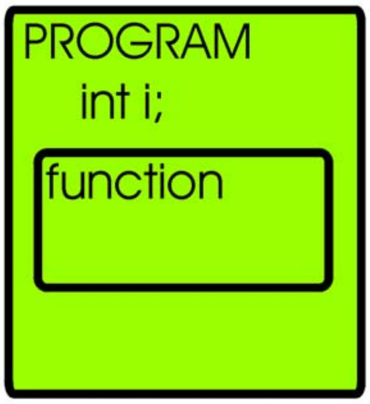
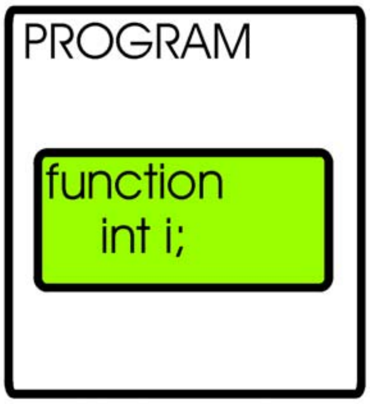
Peter Stallinga. Universidade do Algarve, 13 November 2002

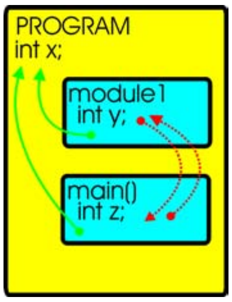
Lecture 15: ... of variables and functions

This lecture:

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

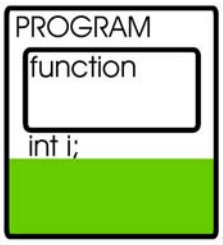
Scope of variables: Global or local

	
Global variables are variables that can be used everywhere in the program	Local variables are only defined inside the function where they were declared.



With this new information, it is much more clear which variable we can use when. Inside functions we can use the global **and** the local variables but functions cannot use eachothers variables. The function main cannot use the variable of the function module1, `int y`. Module1 cannot use the variable of function main, `int z`. Both functions can use the global variable `x`, however.

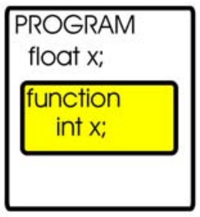
Warning: try to avoid the use of global variables in functions as much as possible. The reason why is simple. If we want to copy the function for another program this will be more difficult, becuase the new program probably will not have the same global variables. Using only local variables in functions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



One more rule, typically for C 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 function, this function cannot use the variable, even if the variable is global.
Let's take a look at some examples. First a program of lecture 13:

```
#include <stdio.h>

 /* declare global variables x and y */
double x, y;

double square(double r)
{
 /* declare local variable localr */
  double localr;

 /* using the local variable localr */
  localr = r*r;
 /* using the global variable x */
  x = localr;
  return(x);
}

void main()
{
  x = 4.0;
  y = square(x);
}
```

Priority



Variable x is a local and a global variable. Inside the function, 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 function. 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 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 function will have no effect on the original value of the variable used in calling the function. As an example, to make this more clear. Assume we have a function 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 function:

```
void write_square(double p)
{
  p = p*p;
  printf("%f", p);
}
```

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

```
void main()
{
  double x;

  x = 2.0;
  write_square(x);
  printf("%f", x);
}
```

After returning from the function, the value of x hasn't changed. The total output of the program above will therefore be

```
4.0
2.0
```

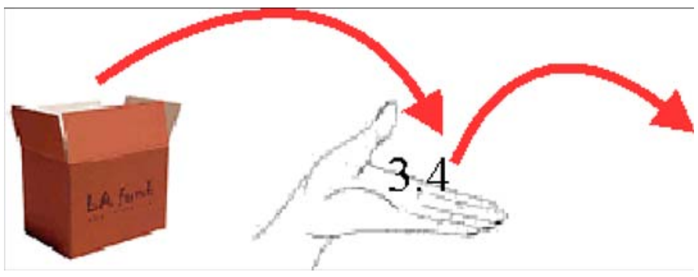
Passing by reference:
On the other hand, if we do want to change the value of the variable used in calling the function, we can do this by passing the address of the variable to the function. All changes to the contents of this address are therefore permanent

```
void write_square(double *p)
 /* p is a pointer-to-double */
{
    /* change the contents of address p: */
  *p = *p * *p;
    /* show the contents of address p: */
  printf("%f", *p);
}

void main()
{
  double x;

  x = 2.0;
  /* now we have to pass the address (&) of the variable x: */
  write_square(&x);
  printf("%f", x);
}
```

If we now run the program, the output will be

```
4.0
4.0
```

because the value of `x` has changed simultaneously with the the contents of address 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 funtion which can then use and change the value in the box, while passing by value is equivalent to opening the box, copying the value and handing only that value over to the function. 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.

`scanf()` revisited

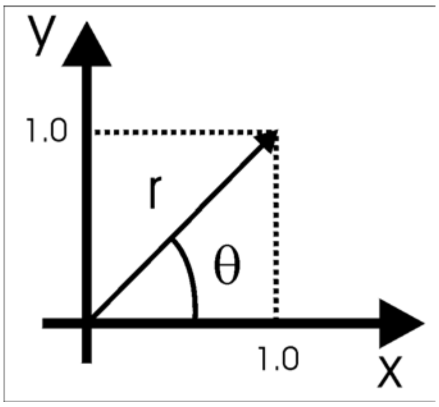
With this in mind, we can take a look again at how we used the function `scanf()` for getting input from the user. Remember that we always had to use the form

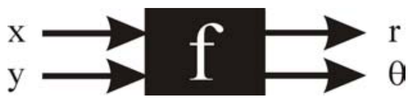
```
  scanf("%d", &i);
```

Or, in other words, we always had to give the address of (&) the variable (`i`) to `scanf`. Now it makes sense. We want to change the value of `i` with the function `scanf`. Therefore, we have to use the technique of passing by reference. Therefore, we have to give the address of `i`, rather than the value of `i` to `scanf`.

Example: Carthesian to polar coordinates

As we know, functions can only return a single value. What if we want to return more than one value to the calling part of the program? Take for example the conversion from Carthesian to polar coordinates. As input we have a coordinate (x, y) and as output we have a Carthesian coordinate (r, θ). These are two values, one value for the radius (r) and one for the angle (θ). The following solution shows how we can pass this information without using global variables:





```
#include <stdio.h>
#include <math.h>

void convert_to_polar(float x, float y, float *r, float
*theta)
/*********************************************************
 * function to convert carthesian coordinate (x,y) to    *
 * polar coordinate (r,theta)                            *
 * parameters:                                           *
 *    x,y: floats, passed by value                       *
 *    r, theta: (pointers to) floats, passed by reference *
 *              (changes are permanent)                  *
 *********************************************************/
{
  *r = sqrt(x*x + y*y);
  *theta = atan(y/x);
}

void main()
{
  float r1, theta1;

  convert_to_polar(1.0, 1.0, &r1, &theta1);
  printf("polar coordinate is (%f, %f)\n", r1, theta1)
}
```

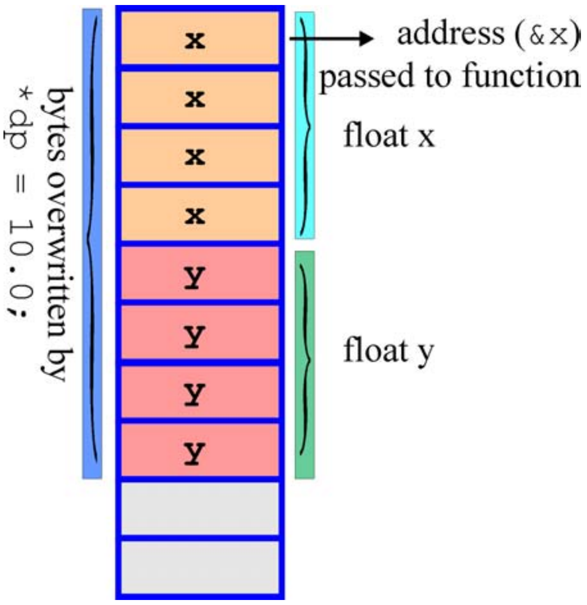
output:

```
polar coordinate is (1.4142, 0.78539)
```

Type mixing

When passing information to functions we have to take even more care that we don't do any mixing of types, especially when we use pointers. As a (bad) example:

```
void double10(double *dp)
{
  *dp = 10.0;
}

void main()
{
  float y=10.0;
  float x=10.0;

  double10(&x);
  printf("%f %f\n", x, y);
}
```



A double occupies 8 bytes. Assigning a value to the contents of a pointer-to-double will therefore write in 8 consecutive bytes of memory. The pointer passed to the function is of type pointer-to-float, however. We have (by declaring the variable `x` of type `float`) only reserved 4 bytes of space in memory. The instruction `*dp = 10;` will now write in these 4 bytes **and** the 4 bytes next to it (that don't belong to `x` but to `y`)
The output of the program above (Borland C++ 3.01 for MS-DOS):

```
0.000000 2.562500
```

Always avoid mixing of type:

Give to the function what the function wants

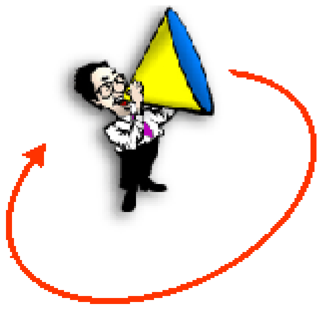
Quick Test

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

Peter Stallinga. Universidade do Algarve, 19 November 2002

Lecture 16: Recursive programming

Recursive



A function 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:

```
int factorial(int n)
 /* returns n! */
{
  int i, result;

  result = 1;         /* initialize the variable */
  for (i=1; i<=n; i++)
    result = i*result;
  return(result);
}
```

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

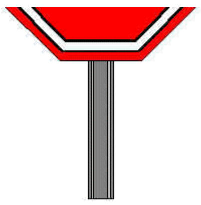
A much more interesting solution is via defining the function factorial in terms of itself, just as we have learned in school,

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

Let's do exactly that:

```
int factorial(int n)
 /* should return n! */
{
    /* the value to be returned is expressed in terms of itself: */
  factorial = n*factorial(n-1);
}
```

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:

```
int factorial(int n)
 /* returns n! */
{
  if (n==1)
    return(1);
  else
    return(n*Factorial(n-1));
}
```

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:

```
int fibonacci(int n)
{
  if ((n==1) || (n==2))
    return(1);
  else
    return(fibonacci(n-2) + fibonacci(n-1));
}
```

Variables

Variables that are declared inside recursive functions are all local. Moreover, everytime the function is called, a new instance of the variable is created that exists until the function 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:

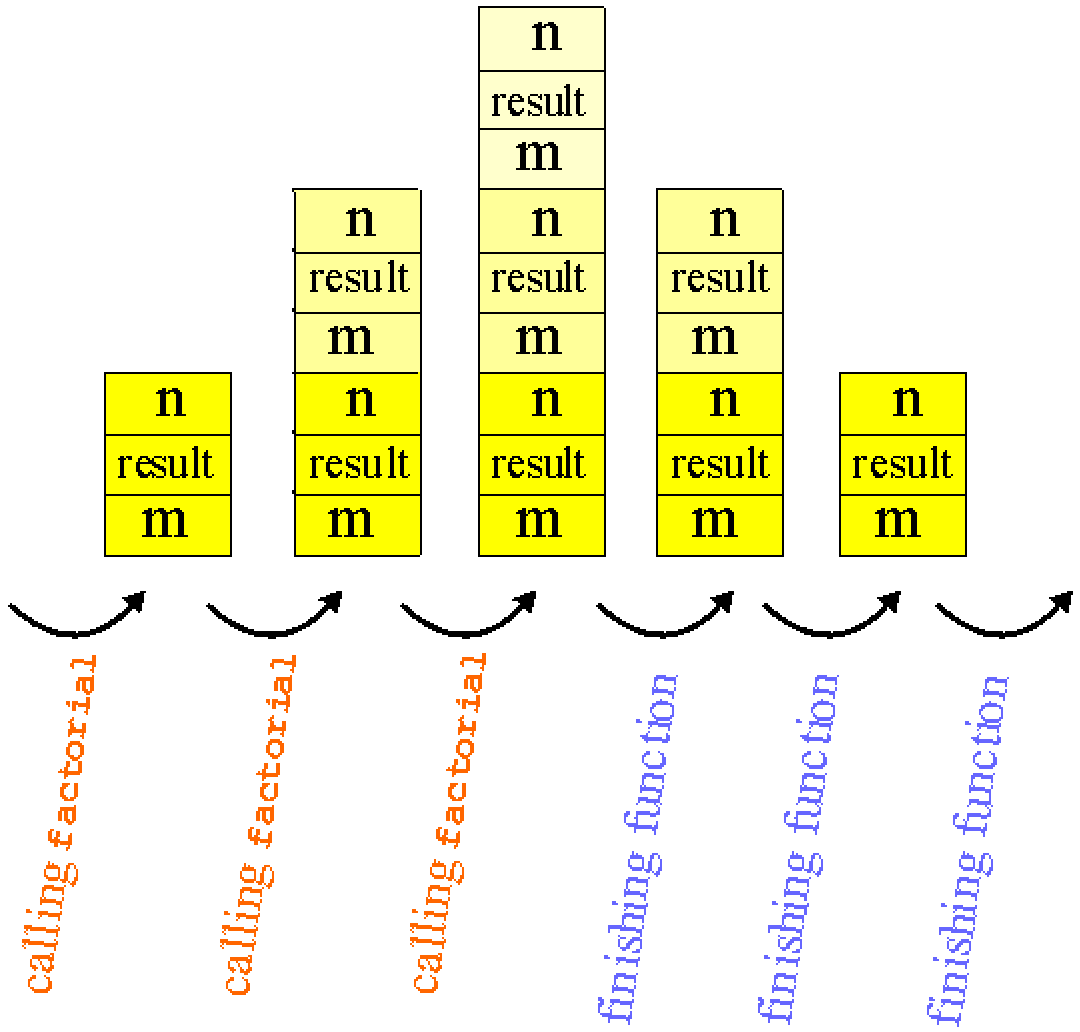
```
int factorial(int n)
{
  int m, result;

  m = 2*n;
  printf("%d ", m);
  if (n==1) then
    result = 1;
  else
    result = n*factorial(n-1);
  printf("%d ", m);
  return(result);
}
```

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

instruction	variables
`factorial(3) is called`	
`variables n, m and result are created`	`m=* r=* n=*`
`3 is assigned to n (from the function call)` `2*3 is assigned to m`	`m=6 r=* n=3`
`factorial(2) is called`	`m=6 r=* n=3`
`variables n, m and result are created` `(new ones; different from variables above!)`	`m=* r=* n=* m=6 r=* n=3`
`2 is assigned to n (from the function call)` `2*2 is assigned to m`	`m=4 r=* n=2 m=6 r=* n=3`
`factorial(1) is called`	`m=4 r=* n=2 m=6 r=* n=3`
`variables n, m and result are created` `(different than the ones above)`	`m=* r=* n=* m=4 r=* n=2 m=6 r=* n=3`
`1 is assigned to n (from the function call)` `2*1 is assined to m`	`m=2 r=* n=1 m=4 r=* n=2 m=6 r=* n=3`
`... we reach the stop condition and result = 1;`	`m=2 r=1 n=1 m=4 r=* n=2 m=6 r=* n=3`
`The value of m is writen: 2`	`m=2 r=1 n=1 m=4 r=* n=2 m=6 r=* n=3`
`The value of result (1) is returned to the calling instruction` `we exit factorial(1), the last n, m and result are destroyed`	`m=4 r=* n=2 m=6 r=* n=3`
`result is calculated: n*factorial(n-1) becomes` `n*'value just returned' -> 2*1 = 2`	`m=4 r=2 n=2 m=6 r=* n=3`
`The value of m is written: 4`	`m=4 r=2 n=2 m=6 r=* n=3`
`The value of result (2) is returned to the calling instruction` `we exit factorial(2), the last n, m and result are destroyed`	`m=6 r=* n=3`
`result is calculated: n*factorial(n-1) becomes` `n*'value just returned' -> 3*2 = 6`	`m=6 r=6 n=3`
`The value of m is written: 6`	`m=6 r=6 n=3`
`The value of result (6) is returned to the calling instruction` `we exit factorial(3), the variables n, m and result are destroyed`	

```
the value returned from factorial(3) is 6
```

```
 r means variable 'result'
* means undetermined value
```



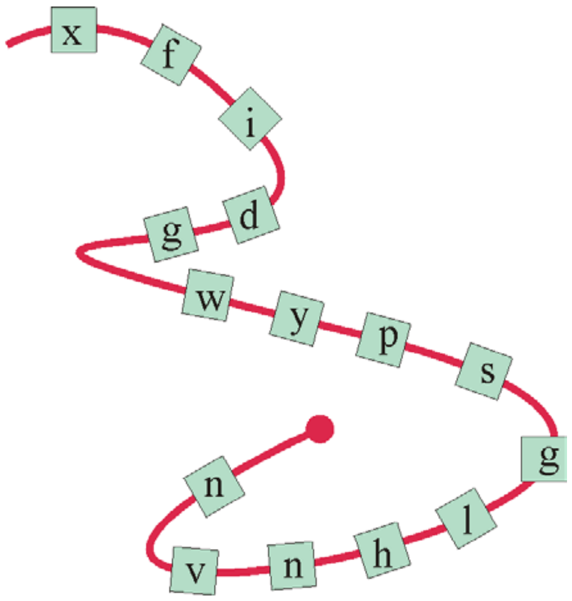
What we learn from this is that the variables are not static objects in memory, pointing to certain addresses, but instead, the variables are created at the time we run the program. Each time we enter the function, new ones are created that live until we exit the function. Moreover, as can be seen in the example above, the last variables created are the ones used locally.
(This only applies to languages that have dynamic variables, like C or 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, 22 November 2002

Lecture 17: strings



string

A string is an array of `char`

We will now see the implications of this simple definition.

As an example: declaring a string of 100 characters:

```
  char n[100];
```

Naively we might now think that assigning a value to this variable could be done with

```
  n = "Benfica";
```

but remember that `n` is a pointer to char (see lecture 15) and the right side of the = isn't (it is a string constant). The way to do it would be to assign a value to each element of the array:

```
  n[0] = 'B';
  n[1] = 'e';
  n[2] = 'n';
  n[3] = 'f';
  n[4] = 'i';
  n[5] = 'c';
  n[6] = 'a';
  n[7] = '\0';
```

With on the left side of each = an element of the array (char) and on the right side a character constant. Therefore the assignements are correct. (Note the `'\0'` character at the end, which signifies end-of-string.) This, however is very clumsy. For this reason and in general to facilitate working with strings, there exist many instructions to help working with strings. These are defined in the library `string.h` and when we want to use them we have to "include" the library in the beginning of the program with

```
  #include <string.h>
```

`string.h`

The following functions of `string.h` are interesting for us

`strcpy`	copy contents of one string to another
`strcat`	adds one string to another
`strcmp`	compares two strings
`strlen`	returns the length (number of characters, excluding \0) of the string
`strstr`	look for position of one string in another string

`strcpy`

short for "**str**ing **c**o**py**". copies contents of one string to another.

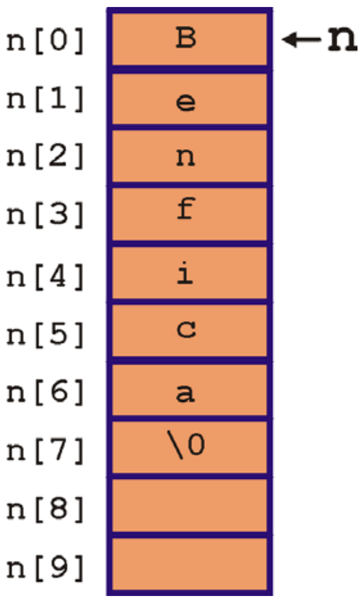
function definition: `int *strcpy(char *s1, const char *s2)`
copies contents of `s2` to `s1`.

This is a function that takes two parameters, two strings (array of char, therefore pointer to char) `s1` and `s2` and copies contents of `s2` to `s1`. The changes to `s1` are permanent. The function returns an `int` indicating the success of the operation. For the moment we can ignore this result, to not further complicate matters. Also we will ignore the word `const` in front of the second parameter declaration. An example:

```
  char n[10];
  strcpy(n, "Benfica");
  printf("%s\n", n);
```

output

```
  Benfica
```



Effectively, what the function `strcpy` is doing is copying the characters of string s2 to s1 until the end-of-string character is encountered:

```
do
{
  *s1 = *s2;          // copy a character; copy contents of
                      //   address s2 to address s1
  s1++;               // make pointer s1 point to next character
  s2++;               // make pointer s2 point to next character
}
while (*s2 != '\0');  // repeat until end-of-string encountered
```

`strcat`

short for "**str**ing con**cat**enate". adds string to other string

function definition: `char *strcat(char *s1, const char *s2)`
adds `s2` at end of `s1`.

Two parameters, `s1` and `s2` are pointers to char (arrays of char). Again, like above, we will ignore the value (char pointer) that the function returns.
Example:

```
  char n[30];
  strcpy(n, "Benfica");
  strcat(n, " o glorioso");
  printf("%s\n", n);
```

output

```
  Benfica o glorioso
```

Effectively, the `strcat` function is equivalent to the following code:

```
while (*s1 != '\0')  // look for
  s1++;              //   end-of-string
```

```
strcpy(s1, s2);       // now copy the string s2 at place of the new
                      //   address s1 that points to \0
```

strcmp

short for "**str**ing **comp**are". compares two strings

function definition: `int strcmp(const char *s1, const char *s2)`
compares `s1` with `s2`. If equal returns 0, if alphabetically `s1`<`s2` returns a negative number, or returns a positive number if `s1`>`s2`.

Example:

```
  char n[30], m[30];
  int cmp;

  strcpy(n, "Benfica");
  strcpy(m, "Sporting");
  cmp = strcmp(n, m);
  printf("%s\n", n);
  if (cmp==0)
    printf("strings are equal");
  else
    if (cmp>0)
      printf("%s is after %s", n, m);
    else
      printf("%s is before %s", n, m);
```

output

```
  Benfica is before Sporting
```

The function `strcmp` could be implemented by something like the following

```
  // possible implementation of strcmp
while ((*s1==*s2) && (*s1!='\0'))  // still equal and not end-of-string?
{
  s1++;    // move pointer s1 one place
  s2++;    // move pointer s2 one place
}
return((int) *s1 - *s2);  // return difference between contents of s1 and s2
```

strlen

short for "**str**ing **len**gth". Returns the number of characters in the string (not counting \0)

function definition: `int strlen(const char *s1)`
Returns the number of characters in the string (not counting \0)

Example:

```
  char n[30];
  strcpy(n, "Benfica");
  printf("%s has %d characters", n, strlen(n));
```

output

```
  Benfica has 7 characters
```

The function `strlen` could be implemented by something like the following

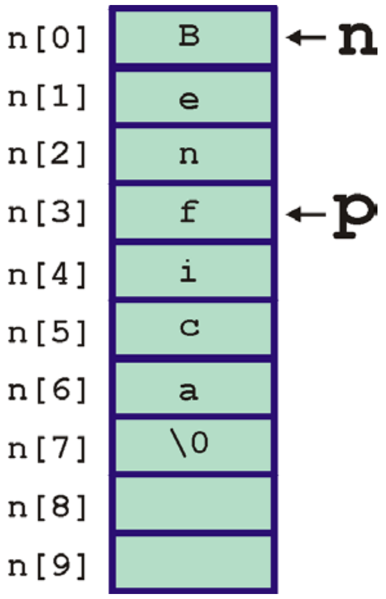
```
cnt=0;
while (*s1 != '\0')
{
  cnt++;
  s1++;
}
```

```
return(cnt);
```

strstr

short for "**str**ing **str**ing". Returns the position of a string in another string.

function definition: `char *strstr(const char *s1, const char *s2)`
Returns a pointer (char array) to the position of text `s2` in text `s1`.



Example:

```
  char n[30];
  char *p;
  strcpy(n, "Benfica");

  p = strstr(n, "fic");
  printf("%s", p);
```

output

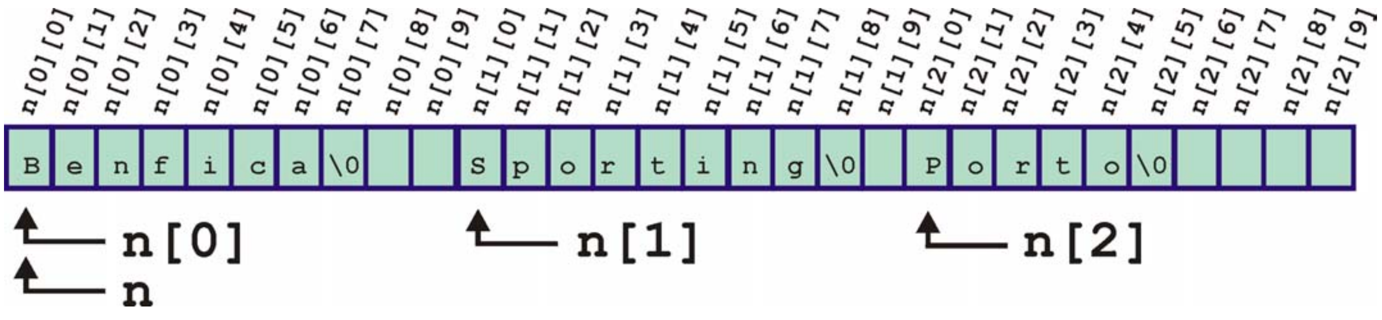
```
  fica
```

Arrays of strings

Arrays of strings are therefore arrays of arrays of char:
For instance, if we want to store the names of three footbal teams of Portugal, we might do this with

```
  char n[3][10];
```

The picture below shows how the names can be stored in this array of strings.



n[0][0] n[0][1] n[0][2] n[0][3] n[0][4] n[0][5] n[0][6] n[0][7] n[0][8] n[0][9]
n[1][0] n[1][1] n[1][2] n[1][3] n[1][4] n[1][5] n[1][6] n[1][7] n[1][8] n[1][9]
n[2][0] n[2][1] n[2][2] n[2][3] n[2][4] n[2][5] n[2][6] n[2][7] n[2][8] n[2][9]
B e n f i c a \0
S p o r t i n g \0
P o r t o \0
n[0]
n
n[1]
n[2]

```
for (i=0; i<3; i++)
  printf("%s\n", n[i]);
```

output:

```
 Benfica
 Sporting
 Porto
```

Quick Test

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

Lecture 18: struct

struct

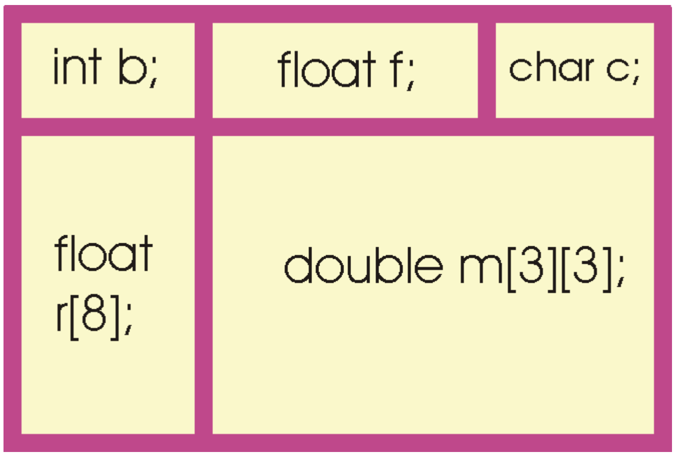
In the lecture 13 we learned how an array can store variables of the same type in a nicely ordered, indexed way, like in a file cabinet with in each drawer the same type of information. If we want to group variables together that are not of the same type, we can do this in a **struct**.

*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 **struct** is used for storing variables of **different type**, integers, real, or whatever, all together in the same box.*

A struct is a set of variables of mixed type.

Declaring a struct



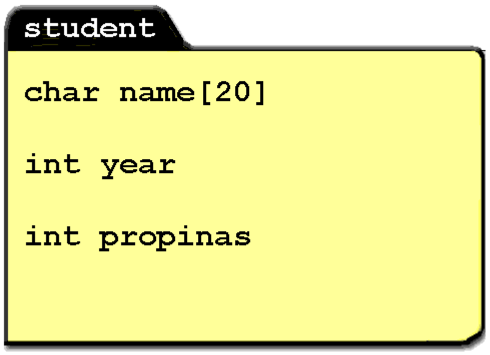
A visualization of a struct, namely a motley set of variables. Each variable inside a struct is called a field. Here we have 5 fields: an int (b), a float (f), a char (c), a single array of floats (r) and a double array of doubles (m).

To declare a struct we do the following

```
struct {
    type1 item1;
    type2 item2;
      |
    typeN itemN;
} name;
```

with
name: the name for the variable holding the struct.
item1..itemN: the name for the various fields in the struct. These have the same rules as the other identifiers for variables, functions, etc. Note that we can put as many fields in the struct as we want, with any combination of types.
type1..typeN: the type of the fields of the struct. This can be any type of variable that we know, including structs!

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



```
struct {
  char name[20];
  int year;
  int propinas;
} student;
```

This struct can only contain the information of a single student. Later we will see how to make an array of structs.

Using a struct

To access a struct we use the format

name.field

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

```
  strcpy(student.name, "Peter Stallinga");
  student.year = 2002;
  student.propinas = 1;
```

for the explanation of the function `strcpy`, see the lecture on strings.
Another example:

```
  struct {
    float x;
    float y;
} coordinate;

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:

```
float coordinate[2];

coordinate[0] = 1.0;
coordinate[1] = 0.0;
```

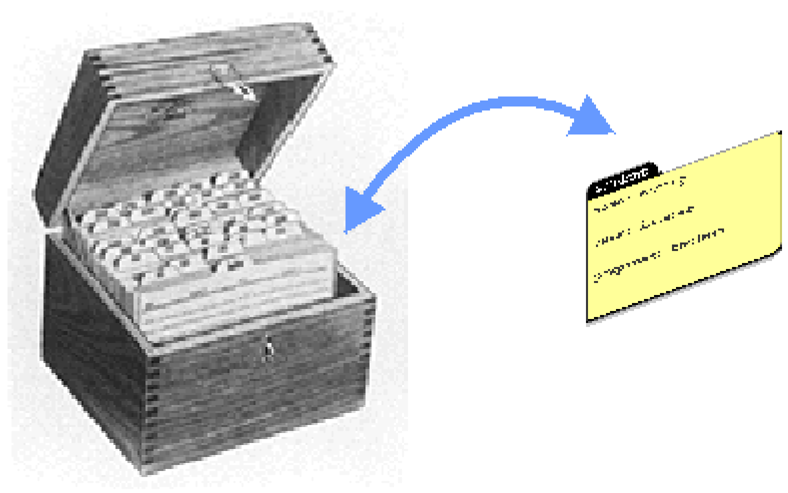
but, the first version, with the struct is more logical.
Another example:

```
struct {
  char rua[20];
  int numero;
  int andar;
  char porta;
} address;

strcpy(address.rua, "Rua Santo Antonio");
address.numero = 34;
address.andar = 3;
address.porta = 'E';

printf("%s %d", address.rua, address.numero);
printf("%d %c", address.andar, address.porta);
```

```
Rua Santo Antonio 34
3 E
```

Arrays of structs, structs of arrays

The above examples use simple structs. With the information of the lecture on arrays (lecture 13), however, we know how to build an array that can store the information of many equal objects of any type, even structs. Let's build an array of 2000 students:
(Question: How many bytes does this variable occupy in memory? Answer at the end)



```
struct {
  char name[20];
  int year;
  int propinas;
} students[2000];
```

```
  i = 1055;
  strcpy(students[i].name, "Peter Stallinga");
  students[i].year = 2002;
  students[i].propinas = 1;
```

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

Later we will see that we can much more easily declare variables of type array of structs (see lecture 19).

Quick Test

To test your knowledge of what you have learned in this lesson, click here for an on-line test.
Answer to the question in the text: 2000 x (20+2+2) = 48000 bytes.

Peter Stallinga. Universidade do Algarve, 28 November 2002

Lecture 19: Defining new types and structs

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 variable types can be done with the word typedef.

```
  typedef description
typename;
```

with `typename` the name we want to give to the type and `decription` any type of variable we have learned until now, including arrays, pointers and all the simple variable types.

Examples:

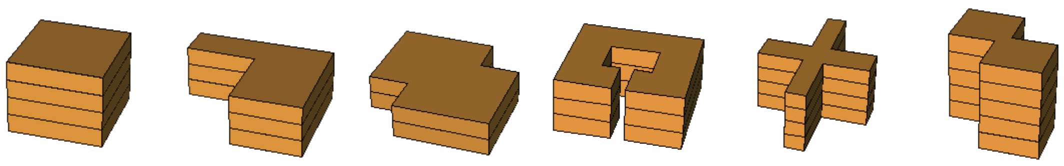
```
typedef float real;
```

This is useful for people that are used to programming in PASCAL. After writing the line above, we can use variables of type real, just like in PASCAL.Variables of 'type' real will be translated by the compiler into variables of type float.

```
typedef float floatarray[10];
```

Defines a new type of variable. The new type is called floatarray and a variable of this type will be equal to an array of 10 floats.

Note that a 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:

```
typename varname;
```

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:
after the declaration of the new type `typedef float real;` we can use

```
 real x;
```

This looks already much more like PASCAL (*in PASCAL it would be "`Var x: real`"*)

```
 floatarray ra;
```

And in the code we can use this array:

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

This is completely equivalent with

```
 float ra[10];
 ra[1] = 2.68;
```

More examples

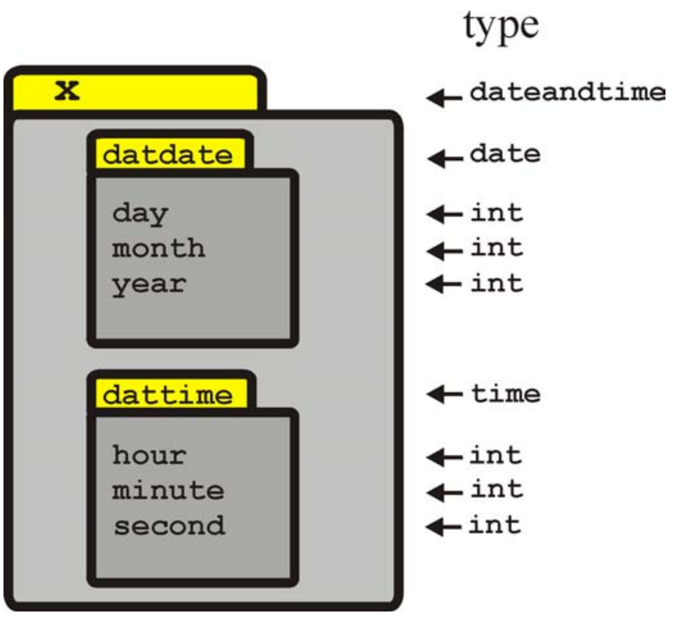
```
/* example with typedef */

/* definig a new type of variable: */
typedef int ra[6];
/* declare two global variables: */
ra x;
int y[7];

int AreEqual(ra r)
  /* Note that the definition can also be used for parameters */
{
  if (r[0]==r[1])
    return(1);
  else
    return(0);
}

void main()
{
  x[1] = 1;
  x[2] = 0;
  if AreEqual(x)
    printf("First two elements are equal");
  else
    printf("First two elements are different");
  y[1] = 1;
  y[2] = 0;
   /* The following instruction is bad code, because the type of value we pass to
      the function is different than the type of value the function expects: *\
  AreEqual(y);
}
```

```
#include <stdio.h>

typedef struct {
  hour, minute, second: integer;
} time;
```

```
void showtime(time t);
  /* Will show the time in format h:m:s */
{
  printf("%d:%d.%d\n", t.hour, t.minute, t.second);
}

void main()
{
 time atime;

 atime.hour = 23;
 atime.minute = 16;
 atime.second = 9;
 showtime(atime);
}
```

struct of structs:

```
typedef struct {
   int day, month, year;
} date;

typedef struct {
   int hour, minute, second;
} time;

typedef struct {
    time dattime;
    date datdate;
} dateandtime;

dateandtime x;

x.dattime.hour = 1;
```



x is a variable of type dateandtime which is a struct containing two fields. One field is dattime which is of type time. The other field is of type date. One of the fields of the struct time is hour which is an int.

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 November 2002

Lecture 20: 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).

Keywords

The following keywords are related to file access:

FILE	**feof()**
fopen()	**fputs()**
fclose()	**fgets()**
fscanf()	**fputc()**
fprintf()	**fgetc()**

These are defined in the library <stdio.h>, which we should therefore include in the beginning of our program.

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 a so-called "handle" to it. This is a variable that stores information about the status of the file. In C we can declare a text file in the following way:

```
FILE *filehandle;
```

With `filehandle` the variable that will store the pointer to the information. 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 other variables.
Example:

```
  FILE *f;
```

This makes `f` a pointer to a file handle.
Without ever having to worry about it (no need to remember this), the actual variable type FILE is

```
typedef struct{
  short          level;
  unsigned       flags;
  char           fd;
  unsigned char  hold;
  short          bsize;
  unsigned char *buffer, *curp;
  unsigned       istemp;
  short         token;
} FILE;
```

For more information, look in your compilers help.

Opening a file

Inside the program we open the file with the <stdio.h> function fopen()

`fopen()`

short for "**f**ile **open**". opens a file.

function definition: **`FILE *fopen(const char *filename, const char *mode)`**
Opens a file. returns 0 if unsuccesful.

`fopen()` returns a pointer to a FILE (which we can assign to our variable of type FILE-pointer above).
`fopen()` takes two parameters, both strings (pointers to char). The first one is the name of the file and the second one is the way it should be opened, for reading ("r"), for writing ("w"), or for appending ("a").
Examples:
to open a file named "OLA.TXT" for reading we use

```
  FILE *f;
  f = fopen("OLA.TXT", "r");
```

to open a file named "output.asc" for writing we can use

```
  FILE *f;
  f = fopen("output.asc", "w");
```

Reading and Writing

Reading from file and wring to file is done in exactly the same way as reading and writing to screen, with the only modification that we will use `fscanf()` instead of `scanf()` for reading and `fprintf()` instead of `printf()` for writing. The first parameter of these function calls have to be our file handle (for example `f`)

`fscanf()`

short for "**f**ile **scan f**ornatted". input from file

function definition: **`int fscanf(FILE *stream, const char *format, ....)`**
gets fromatted input from file

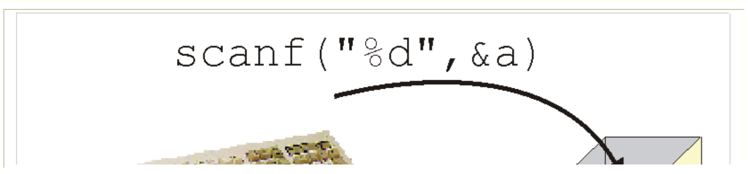
`fprintf()`

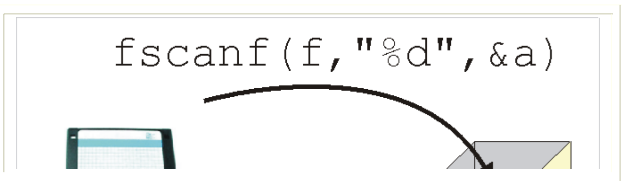
short for "**f**ile **print f**ormatted". Formatted output to file

function definition: **`int fprintf(FILE *stream, const char *format, ...)`**
outputs to a file. returns number of characters written, EOF if unsuccesful.

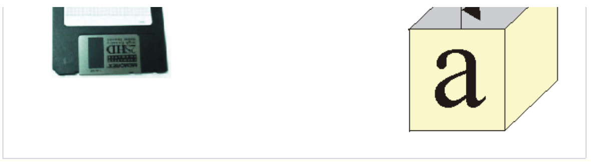
Note that we can only use the `fscanf` instructions for files that have previously been opened for input ("r") and `fprintf` is only to be used for files opened for output ("w"). Examples:

```
  fprintf(f, "%f", r);
  fscanf(f, "%d", opcao);
```

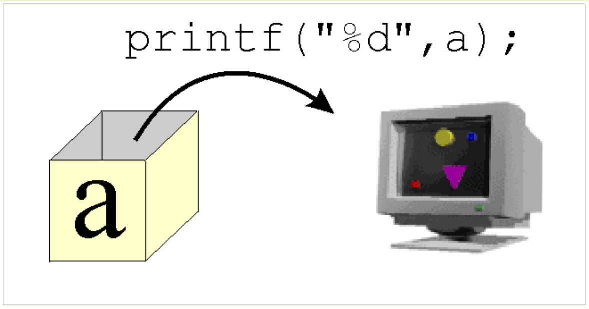




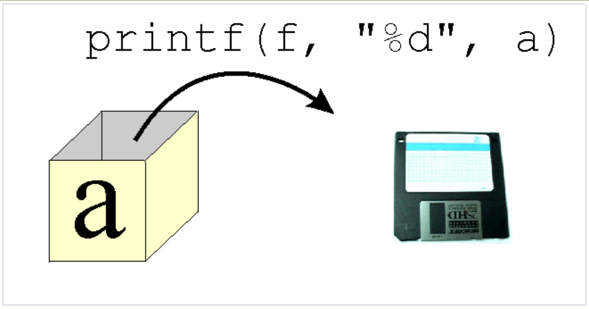
Read a variable with the keyboard



Read a variable from file



Write a variable to screen



Write a variable to 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 (the "buffer" will not be emptied). To close the file we use `fclose()`.

`fclose()`

short for "**f**ile **close**". Closes a file.

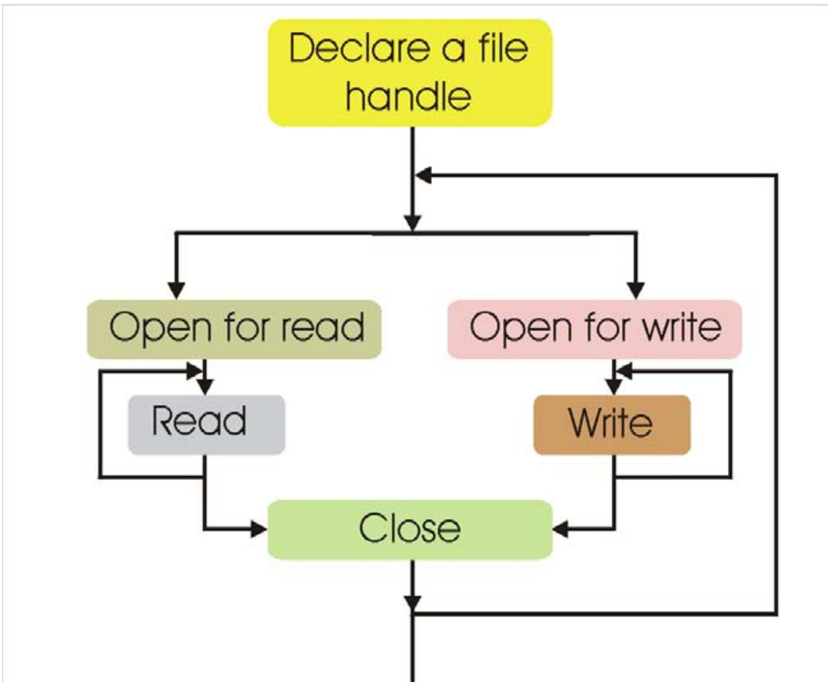
function definition: **`int fclose(FILE *stream)`**
closes a file. returns 0 if succesful.

for example:

```
fclose(f);
```

Summary

File input and output consist of the following steps:



End-of-file testing

The following instruction can be useful

feof(filehandle): returns true if we are reading at the end of the file.
eof stands for end-of-file

Example:

```
  while (!eof(f))
  {
    fscanf("%s", 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*
/* With File Output */	Name of File:	1 Hello
#include <stdio.h>	*TEST.TXT*	2 Hello
		3 Hello
FILE *f;		4 Hello
char s[100];		5 Hello
int i;		6 Hello
		7 Hello
void main()		8 Hello
{		9 Hello
printf("Name of File:");		10 Hello
scanf("%s", s);		
f = fopen(s, "w");		
for (i=1; i<=10; i++)		
fprintf(f, "%d Hello", i);		
fclose(f);		
}		

code	*screen*	*file TEST.TXT* ***before*** *running the program*
/* With File Input */	Name of File:	1 Hello
#include <stdio.h>	*TEST.TXT*	2 Hello
	1	3 Hello
FILE *f;	Hello	4 Hello
char s[100];	2	5 Hello
int i;	Hello	6 Hello
	3	7 Hello
void main()	Hello	8 Hello
{	4	9 Hello
printf("Name of File:");	Hello	10 Hello
scanf("%s", s);	5	
f = fopen(s, "r");	Hello	
while (!feof(f))	6	
{	Hello	
fscanf(f, "%s", s);	7	
printf("%s\n", s);	Hello	
}	8	
fclose(f);		

`}`	`Hello`
	`9`
	`Hello`
	`10`
	`Hello`

The output is probably not what we would have liked. Maybe we should use the function `fgets()` instead to read strings. See below.

Other file functions

Other useful file functions are

`fgets()`

short for "**f**ile **get s**tring". Reads a string from file.

function definition: **`char *fgets(char s1, int n, FILE *stream)`**
reads a string from file until eol (end of line), eof (end of file) or maximum of n-1 characters are read

`fgetc()`

short for "**f**ile **get c**har". Reads a character from file.

function definition: **`int fgetc(FILE *stream)`**
reads a single character from file. Returns value of character if successful.

`fputs()`

short for "**f**ile **put s**tring". Writes a string to file.

function definition: **`int fputs(const char s1, FILE *stream)`**
writes a string to file. Returns 0 if succesful.

`fputc()`

short for "**f**ile **pu c**har". Writes a character to file.

function definition: **`int fputc(char *s1, FILE *stream)`**
Writes a single character to file. returns value of character if succesful.

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 December 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 C

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 C we write comment

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

Quick Test 5: Variables

1. The use of `\n` in `printf`

- is used for output to the printer.
- signifies the end-of-string.
- puts the cursor on the beginning of the next line.
- is used for specifying the format of `int`.

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

- `pointer`
- `int`, or `long int`
- `float` or `double`
- string

3. The range of an `int` 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. In C, 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
- float
- double
- long double

Quick Test 6: Assignment, Input and Math

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 C will produce
```
8.3000
```

- ○ `printf(8.3000);`
- ○ `printf(6*r,4);`
- ○ `printf(r:6:4);`
- ○ `printf("%6.4f", r);`

3. What is wrong with the following program?

```
main()
{
  float x;
  double c = 1.0;

  x*x = 2*c;
}
```

- ○ 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 expressions

4. What is the output of the next program

```
main()
{
  double x;
  double C = 1.0;

  x = C + 1.0;
  x = 2;
  x = x + 3.0;
  printf("%4.1f", x);
}
```

- ○ `7.0`
- ○ `5.0`
- ○ `3.0`
- ○ `1.0`

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

- ○ 1.5
- ○ 16
- ○ 16.5
- ○ 1

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

- ○ 1.5
- ○ 16
- ○ 16.5
- ○ 1

7. What is priority of the operators +, * and (..)

- ○ first + then * then (..)
- ○ first * then + then (..)
- ○ first (..) then * then +
- ○ first (..) then + then *

8. 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 7: if ... else ...

1. Which is not a valid comparison in C:

- ○ `(a = b)`
- ○ `(a != b)`
- ○ `(a == b)`
- ○ `(a > b)`

2. The proper syntax for simple branching is

○
```
if (condition) then
   instruction;
```
○
```
case condition
   instruction;
```
○
```
if (condition)
   instruction;
```
○
```
case condition of
   instruction;
```

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

```
main()
{
  int a, b, c, d;

  a = 5; b = 3; c = 99; d = 5;
  if (a>6) printf("A");
  if (a>b) printf("B");
  if (b==c)
    {
      printf("C");
      printf("D");
    }
  if (b!=c) printf("E"); else printf("F");
  if (a>=c) printf("G"); else printf("H");
  if (a<=d)
    {
      printf("I");
      printf("J");
    }
}
```

Check   I give up

4. What will be the output of the following program?

```
main()
{
  int a=1;

  if (a>0)
    printf("The value of a ");
    printf("is larger than zero");
  else
    printf("The value of a ");
    printf("is less than zero");
}
```

- ○ `The value of a is larger than zero`
- ○ `The value of a is less than zero`
- ○ The structure of the program is wrong. We should group instructions with {..}
- ○ Depends on the value of `a`.

Quick Test 8: Boolean Algebra / `switch`

1. What will be the output of the following program?

```
main()
{
  double a, b;
  double c = 10.0;

  a = 9.0;  b = 2.0*c;
  if ((a>0) || (b>0))
    printf("Fixe!");
  else
    printf(" Uma pena");
}
```

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

2. What is wrong with the following program

```
main()
{
  double a;
  double c = 2;

  a = 3.0;
  switch (a+1.0)
   {
    case 1: printf("Fixe!\n");
        break;
    case c:
         printf("Cool!\n");
         printf("Ingles");
        break;
    case 3: printf("Super!");
        break;
    default: printf("Language?\n");
   }
}
```

○ a) `switch` cannot contain expressions (`a+1.0`)
○ b) `switch` cannot have expressions of type float (`a+1.0`)
○ c) In the structure `switch` we cannot use variables (`case c:`)
○ d) Both b) and c)

3a. What will be the result of the Boolean calculation `(41 | 35)`?

Check Help More help I give up

3b. What will be the result of the Boolean calculation `(41 & 6)`?

Check Help More help I give up

3c. What will be the result of the Boolean calculation `(41 && 6)`?

4.

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

is an example of

○ an expression
○ a condition
○ an assignment
○ an operation

Check Help More help I give up

Quick Test 9/10: Loops

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

- ◯ for
- ◯ while
- ◯ do-while
- ◯ 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-while
- ◯ 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 and do-while?

- ◯ `while` is for integer numbers, `do-while` is for variables of floating point type.
- ◯ `do-while` is for integer numbers, `while` is for variables of floating point type.
- ◯ In loops of type `do-while` the condition is checked in the beginning, whereas in loops of type `while` the condition is checked at the end.
- ◯ In loops of type `while` the condition is checked in the beginning, whereas in loops of type `do-while` the condition is checked at the end.

5. What is wrong with the following code?

```
x = 0.0;
while (x<10.0)
  {
    y = x*x;
    z = x*y;
    printf("The square of %f is %f", x,
y);
    printf("The cube of %f is %f", x, z);
  }
```

- ◯ This loop will never finish
- ◯ We should use a loop of do-while instead.
- ◯ We should use a for-loop instead.
- ◯ The condition cannot contain variables of type float.

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-while
- ◯ Other structure

Quick Test 11,12: Modular Programming

1. What word in C indicates that the function will not return anything?

Check   Correct Answer

2. What are the advantages of writing modules

1:
2:

Correct Answer

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

```
#include<stdio.h>
void write_N(float r, int n)
{
  printf("%f" ,r);
}

void main()
{
  float x;
  x = 10.0;
}
```

- ○ 10.0
- ○ r
- ○ x
- ○ This program doesn't have output. We forgot to CALL the function!

4. Why we don't have to initialize a parameter?

- ○ A parameter cannot change its value.
- ○ The initialization comes from the function call.
- ○ Parameters are automatically set to 0.
- ○ Not true! Parameters are like normal variables and have to be initialized in the beginning of the function.

Quick Test 14: Pointers

1. How to declare a pointer to an int?

- `int *a;`
- `int a*;`
- `int &a;`
- `int a&;`

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

- Depends on the type of `x`.
- `p = *x;`
- `p = &x;`
- `p = ^x;`

3. Assume `b` is a variable of type "pointer to int";
How to put a value of 0 in `b`?

Check   Correct Answer

4. What happens if we forget to initialize a pointer?

- The result will be 0.
- The compiler will warn us.
- The program will crash.
- The pointer is automatically initialized.

5. What is wrong in the following code?

```
double *p;
int i;
p = &i;
*p = 10.0;
```

- We mixed `*` with `&`.
- `*p = 10.0;` will overwrite other variables or code.
- `p = &i;` will generate an error.
- Nothing is wrong here.

6. What are the advantages of pointers

1:
2:

Correct Answer

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

```
float a;

void proc1(float b)
{
  float c;
  int d = 10;

  c = b+ (float) d;
  printf("%f", c);
}

float proc2(float *e)
{
  float f = 20.0;

  return(*e+f);
}

float g;

void main()
{
  float h;

  a = 10.0;
  proc1(a);
  proc2(&a);
}
```

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

2. Consider the program below

```
int x;

void show(int *a)
{
  printf("%d ", *a);
  *a = *a + 1;
}

void main()
{
  x = 0;
  printf("%d ", x);
  show(&x);
  printf("%d ", x);
}
```

The procedure is called using the technique of
- ○ Passing by value
- ○ Passing by reference

And, hence the output will be

Correct Answer

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

```
include<stdio.h>
/* double names */
int x;

void show()
{
  int x;

  x = 1;
  x = x*x;
  printf("%d ", x);
}

void main()
{
  x = 0;
  show;
  printf("%d ", x);
}
```

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

○ 0 1
○ 1 1

Quick Test 16: Recursive Programming

Consider the following program:

```c
#include <stdio.h>
float a;

float XfuncN(float x, int n)
{
  float c;

  c = 0.0;
  if (n==0)
    return(1.0)
  else
    return(x*XfuncN(x, n-1));
}

void main()
{
  printf("%0.1f", XfuncN(3.0, 3));
}
```

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:

```
#include <stdio.h>
#include <string.h>

int func(char *p)
{
  int c;

  c = 0;
  if (*p == '\0')
    return(0);
  else
   {
     p++;
     return(1 + func(p));
   }
}

void main()
{
  char a[20];

  strcpy(a, "Ola");
  printf("%d", func(a));
}
```

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 18: Arrays and Structs

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

- ○ An array can store only countable things, whereas a struct can store anything.
- ○ A struct can only store countable things, whereas an array can store anything.
- ○ Arrays are for combining variables of different type, structs store variables of the same type.
- ○ Structs are for combining variables of different type, arrays store variables of the same type.

```
float maximum(float a, b)
{
  float max;

  if (a>b)
    max = a;
  else
    max = b;
 .....
}
```

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

- ○ Nothing, this is done automatically.
- ○ `return (max);`
- ○ `maximum = max;`
- ○ This function has no output and will not return anything

```
struct {
  struct {
    float z[10];
    int i[3];
  } x;
  struct {
    float r;
    double p;
  } y;
} a[10];
```

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

○
```
struct {
          int number;
          char name[20];
          int year;
   } a;
```
○
```
struct {
          int number;
          char name[20];
          int year;
   } a[1000];
```
○
```
struct {
      int number[1000];
      char name[1000][20];
      int year[1000];
  } a;
```
○
```
struct {
      int number[1000];
      char name[1000][20];
      int year[1000];
  } a[1000];
```

Quick Test 19: typedef

1. What is `typedef` 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.

```
typedef float b;
 ......
b = 3.1;
```

2. Why doesn't the above code work?

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

```
typedef struct {
   struct {
      float x[10];
      int y[3];
    } ri;
    struct {
      float v;
      double w;
    } rd;
  end;
} mystructs[10];

mystructs b;
```

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?

```
typedef float reals[10];

void WriteIt(reals r)
{
  printf("%f\n", r[1]);
}

void main()
{
  int x[10];

  x[1] = 3;
  WriteIt(x);
}
```

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