Curso de C

Tim Buchalka's Learn Programming Academy

Autores: Daniel Ceron

Índice

Índice de figuras

Índice de tablas

1. Introducción

C is an efficient, portable, powerful, and flexible programming language, oriented to programmers.

Advantages of C programmes: small, fast, reliable, easy to learn and understand.

1.1. Fundamentals of a programme

The basic operations of a computer form the computer's instruction set. To solve a problem using a computer, you must provide a solution by sending instructions to the instructions set. So, a computer programme sends the instructions to solve a certain problem to the instructions set.

The approach or method used to solve the problem is an algorithm. Then if there is a programme that tests if a number is odd or even.

- The statements that solve the problem are the program.
- The methods used to verify if the number is even or odd is the algorithm.

The instructions to implement the algorithm are expressed in a partircular computer language, such as Java, C++ or C.

CPU (Central Processing Unit) is the brain of the computer, where most of computing work is done and the instructions are executed.

The RAM (Random Access Memory) stores the data of the programme while it's running.

The hardware drive is the permanent storages, that stores files that contain programme source code, even while computer is off.

An OS (Operating System) controls the operation of the computer: input, output, computer resources, execution of programmes. It makes it more convenient to use computers. E.g.: Windows, iOS...

A fetch or Execute Cycle fetches an instruction from memory using registers and executes it in a loop. A 1 GHz CPU can do this loop 1 billion times a second. This cycle can be seen below.

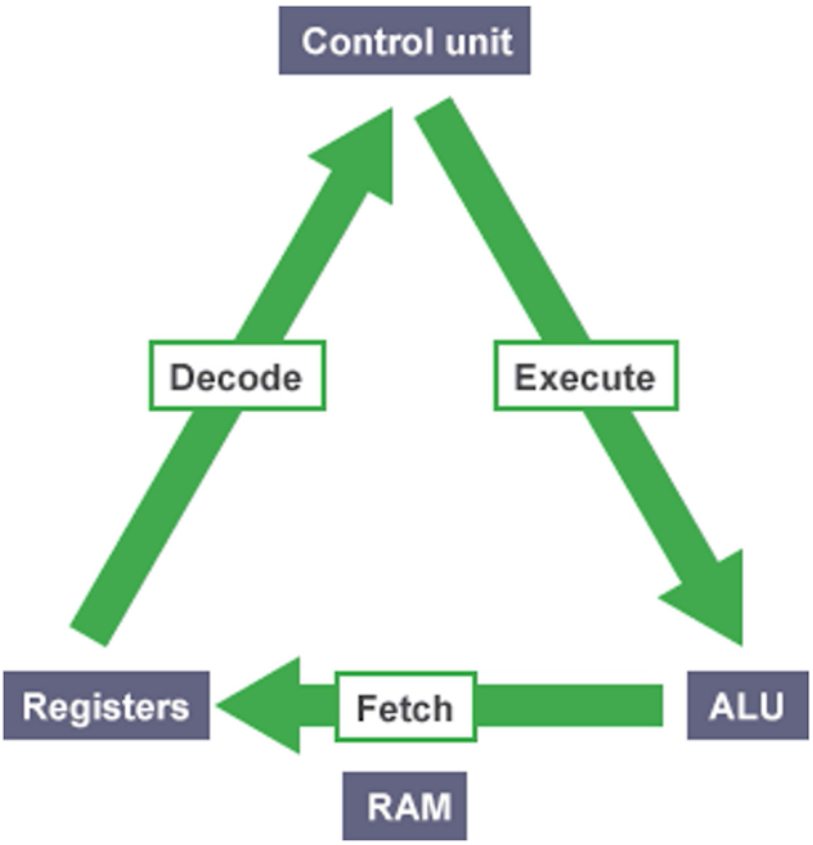


Figura 1.1: Execute cycle

At the lowest level, a programme is merely interacting with the CPU.

Higher-level programming languages make it easier to write programmes. They describe actions in a more abstract form. The instructions of these programmes are like problem-solving steps. They allow the programmer not to worry about the particular steps followed by CPU.

A compiler translates the high-level languages into the machine languages instructions. They also check if a programme has a valid syntax. If there is a syntax error, it finds it and report back to the programmer.

The act of writing a C programme can be broken down into multiple steps:

1. **Define programme objectives:** Understand the requirements of the programme and get a clear idea of what you want the programme to acomplish.
2. **Design the programme:** Decide how to meet the requirements, define the user interface and organise the programme.
3. **Write the code:** Translate the design in the syntax of C. A text editor is used to create a source code.
4. **Compile:** Translate it into machine code
5. **Run the programme:** An executable file is a programme that can be run.

6. **Test and debug the programme:** Check if the programme works as supposed, finding and fixing errors.
7. **Maintain and modify the programme:** Fix new bugs and add new features.

The above steps can be jumped around and repeated.

Many new programmers ignore the first two steps, which is a mistake. I should develop the habit of planning before coding, which long-term saves a lot of time.

You always want to work in small steps and test constantly a few lines.

1.2. Overview

C is a general purpose, imperative (states what to do instead of asking questions) computer programming language that supports structured programming. This is in contrast to languages like MATLAB which are created for specific applications.

C is a modern language: it has most basic control structures and features of modern languages, is designed for top-down planning, is organised around functions (modular design) and is very reliable and readable.

C is widely used and is the preferred language for producing word processing programmes, compilers, etc. It is popular for programming embedded systems.

C programmes are easy to modify and adapt to new models of languages. It is a subset of C++ with object-oriented programming tools, which means that knowing C means that you know much of C++.

C provides constructs that efficiently map to typical machine instructions. It provides low-level access to memory (it has many low-level capabilities) and requires minimal run-time support.

C evolved from a previous programming language, B, adding data typing and other features. Is available for most computers and independent of hardware, being versatile.

There are many variations of C, which means that a Standard C version had to be created. A programme written in Standard C without hardware-dependent assumptions will run correctly on any platform with a standard C compiler. Non-standard C programmes may run only on certain platforms or compilers.

C89 (created in 1989) is the basis of most C codes. C99 is a revised standard of C that refines and expands its capabilities, but it has not been widely adopted. C11 (2011) is the current standard, with additional optional features, and the classes are based on this standard.

1.3. Language features

C produces compact and efficient programmes and is one of the most important programming languages. The main features of C are:

- Efficient.

- Flexible.
- Powerful and Flexible.
- Programmer Oriented.

C is efficient because it takes advantage of the capabilities of current computers, its programmes are compact and fast and programmers can fine-tune their programmes for maximum speed or most efficient use of memory.

C is portable because its programmes written in one system can be run in other systems with little or no modificacion. Besides, compilers are available for many computer architectures.

The Linux kernel (the brain of the CPU, contains the interfaces with hardware, manages memory...), compilers and interpreters for other languages are usually written in C, C is used for solving physics and engineering problems... This speaks for itself for the power of the C language.

C is flexible since it is used for developing every kind of programme, is a basis for other languages, such as C++... However, there is a trade-off between flexibility and performance, since a high flexibility can cause bugs or the developers not to understand certain concepts.

C is easy to learn because of its compactness, being ideal as a first language to learn.

C fulfils the needs of programmers, since it gives access to hardware and enables to manipulate individual bits in memory. It contains a large selection of operators which allows to express yourself succinctly. This may be a danger, since you can make mistakes impossible to make in other languages. C have a large library of useful C functions that deal with common needs of most probrammers.

Other features of C are that provides low level features generally provided by lower level languages, the fact that programmes can be manipulated using bits and pointers play a big role in C.

The main disadvantages of C are that the programmer has an extra responsability for the freedom that he has and its conciseness may turn a C programme difficult to understand sometimes.

1.4. Creating a C programme

There are four fundamental tasks in the creation of a C programme: editing, compiling, linking, and executing.

Editing is the process of creating and modifying your source code. A simple editor like a notepad can be used, but it is better to use an IDE (Integrated Development Environment), with several facilities for the programmers. The IDE used for the course is CodeBlocks.

Compiling converts the code into machine languages and detects and reports syntax errors. Is a two-stage process: preprocessing (the code may be modified or added) and the actual compilation that generates the object code. The compiler examines each programme statement and checks it to ensure that it is consistent with the syntax and semantics of the language. Can recognise structural errors (dead code that is not being used), does not find logic errors, and the typical errors reported are syntactic errors (unbalanced parentheses in an expression) or semantic errors (use of a variable not defined).

After the errors are fixed, the compiler will translate it into assembly language and then into actual machine instructions. The output of the compiler is an object code and is stored in files called object files (same name as source file with a .obj or a .o extension).

The standard command to compile a C program will be cc (or gcc for the GNU compiler). The typical instruction is **cc -c myprog.c**. If omit -c, the programme will automatically be linked as well. In CodeBlocks there is an option in the menu to compile.

The purpose of the linking is to get the programme in a final form for execution by the computer. Usually occurs automatically when compiling, but sometimes can be a separated command. There are one or several .obj from compiling and you turn them into an executable. Also detect and reports errors if a part of the programme is missions or a non-existent library component is referenced. It includes libraries into the executable.

A failure in linking means that you have to go back and edit the source code. Success will produce an executable file; in Windows, it will be a .exe. Many IDEs have a buid option to compile and link the programme in a single operation to obtain the executable.

A programme of significant size will consist of several source code files. The programme is much easier to manage by breaking it up in smaller source files: it is cohesive and makes the development and maintenance much easier. The set of source files will usually be integrated under a project name, which usually refers to the whole programme.

The execution can be performed by double clicking on the executable. Each of the statements is executed sequentially. If data is needed, the programme temporarily suspends execution so that the input can be entered. The results (output) are displayed in a window called the console. The execution can generate a wide variety of errors: wrong outputs, doing nothing or crashing the computer. If there is an error, it is necessary to go back and reanalyse the programme, which is called debugging.

The entire process of compiling, linking and executing the programme must be repeated until the desired results are obtained.

2. Installing Required Software

2.1. Overview

For Windows, the necessary software to install are a C compiler (Cygwin for this course) and an IDE (CodeLite).

2.2. Installing the C compiler

The components to install are: a GNU gcc compiler, make and ddb debugger from cygwin.com, which simulates a unix environment.

After selecting them in the installer and installing them, you must go to the directory of installation of Cygwin, go to bin, and copy the path. Then, you go to "This PC", right-click, and select properties. Then you go to advanced system settings, select environment variables, go to the list of system variables, and select path. A new window is opened, where you click on new and

paste the path previously copied.

To check that it has been done correctly, you open the command window and type çygcheck -c cygwin". It has to appear the package cygwin, the version, and that the status is ok. Then you type "gcc –version.ªnd the version of the gcc compiler has to appear.

2.3. Installing CodeLite

CodeLite can be downloaded from codlite.org.

2.4. Configuring CodeLite

The compiler has to be added manually to CodeLite by clicking in Settings -> Build Settings -> + (add an existing compiler). Then you have to go to the folder of cygwin64 installed, go to bin folder and add it. Then you hace to select the C compiler (gcc.exe), the assembler name (as.exe), the linker (gcc.exe), the Shared Object Linker (gcc.exe -shared -fPIC), the archive (ar.exe rcu), the make (make.exe -j16, it should be correct) and gdb (gdb.exe).

To create a new workspace, you have to click in new workspace and select a C++ workspace. The folder in which the workspace will be created is important that does not have any space, & and special characters that can create problems. When creating a new workspace, the tick of Çreate the workspace under a separate directory"must be activated.

The workspace is where the C projects are to be created. The projects are created by clicking at the workspace and select New -> New Project. Then the categories of the new project must be filled:

- **Category:** The normal programmes are Console. A GUI, a Library and other types of programmes are other options, but usually it will be Console programmes.
- **Type:** Usually it will be a Simple executable (gcc).
- **Compiler:** Select the name of the compiler previously added (Cygwin for me).
- **Debugger:** GNU gdb debugger is the only choice.
- **Build System:** Select CodeLite MakeFile Generator -UNIX.

The red ball that appears in a programme open means that there are non-saved changes.

The tool icon that appears under the Workspace and Explorer tags is for changing the project settings.

To compile a programme, you have to go to Build -> Build Project.

Ctrl+S is the shortcut to saving a programme.

To run a programme, you have to go to Build -> Run.

In the window below of the CodeLite application, the two most important tabs are build (compiler, says if an error occurs when compiling) and output (where the results of a run are shown).

2.5. Creating a default CodeLite Project Template

It is important to select in the project settings -> Compiler options the option of Enable C99 features to access some features of this version of C. and in Linker options the option of static.

To avoid the process of selecting the project settings each time that a new workspace is created, you can click on the project folder and select Save as template. You select a name (without spaces or special characters), the category of User templates and you can add a description if desired.

Then, when creating a new project, in the category you select User templates and then in type the name of the template that you saved previosuly. The new project will have all the settings of the project used for template.

When you have several projects, the active project of the workspace is highlighted in **bold** letter. You make a project active by right-click -> Make Active or by double-click the folder.

Another option that appears when right-click a folder is to rename a project.

2.6. Using the Command Line Interface

The IDE is no necessary, you can use a text editor, a terminal window or an installer C compiler to run and compile C programmes.

You can open a terminal window of cygwin by typing cygwin of Windows and selecting the option that appears.

The main directory of this terminal is C:\cygwin64\home\daer.

A text in Notepad can be saved as a C programme by saving it as AllFiles and add to the name at the end a .c.

By typing ls in the terminal window, you can see what files are in your current directory. The current directory can be changed by cd and a new folder can be created by using mkdir. The command more prints what is written in the C file and clear clears the window.

To see where the gcc compiler is you can write which gcc and to see the version you can write gcc –version. To compile a file in your current directory you type gcc and the name of the .c file. If the compilation is correct, the prompt will come back inmediately with no messages. Now a new executable has been created since there is no compile errors, usually called a.exe. If you type ./a or ./a.exe (if the name is different, change a for the name of the executable), the programme will be executed. If there is a compile error, the command window will point the line of the error.

The name of the exe generated by the compiler can be changed by typing gcc (.c name).c -o and the name of the executable that you want.

With gcc –help you can see the instructions of all possible commands.

2.7. Using a Web-based C compiler

There is the possibility of using online C compilers to create, compile and run C programmes. The most famous is OnlineGDB Beta. It is usually the first option that appears in Google when searching online C compiler. It has the option of write, run, debug and save programmes.

Other option is the Online C Compiler from Tutorialspoint. It has more limited options but it may have some performance problems. Another C compiler is the one of Programiz.

Finally, CS50 IDE is an online IDE that uses Gitlab (needed for its use) to save the programmes.

These online compilers may not support all the functions of the C language, but there is a nice option if you don't have access to a C compiler in the computer.

3. Starting to write code

3.1. Exploring the CodeLite environment

CodeLite is a lightweight IDE with minimal features and minimal graphical interface components compared to other IDEs such as Visual Studio.

In Settings -> Preferences several options for the configuration of CodeLite can be selected. In Folding -> Show Folding Margin, the brackets are highlighted and you can hide and show the code that is contained into the brackets, which can be useful if you have a lot of code lines.

The colours of the programmes mark what are the words used. For the Dark palette of colours:

- Orange for header files and the strings for a printf function.
- Purple for some of the different data types and for the return option.
- Yellow for numbers.
- Blue for typical functions such as printf and for the name of the programme.
- White for the #include options and other normal words.

In Settings -> Colours and fonts these colours can be edited.

In the lower part of the workspace, between the Build and Output tags, there are options to search and replace text of the programme.

In the right side of the workspace, there are a toolbar with the options usually performed by shortcuts such as copy, paste, cut, undo, redo...

In view you can select which windows you want to see at each time in the workspace and hide those that you do not want to see at a certain time. The toolbar can be put at the top, the left of the bottom in this tag in MainToolBarOrientation.

The tag of search allows you to search for certain words in the file, replace them, go to a certain line, search in various files...

In Workspace you can open, close or reload workspaces, create a new project, select the active project, among other features.

In Build -> Clean Workspace you can delete all the intermediate files: .o, libraries...

In Plugins there are several plugins to use, the only important usually are some for code completion.

In Help you can find answers to some questions.

The other options have already been covered or will be covered in future sections.

3.2. Creating the first C programme

The first thing to do when starting a new C programme is to include a header file from other C library, which is used by the command **#include** and add the header file. The standard header file is <stdio.h>.

After that, the next thing that must be entered is the main function, the entry point of the programme. Forst a type of data must be written (**int** for example) and then the name of the function with their inputs between parenthesis: (input). If the function has no input, you can leave the parenthesis () or write (**void**).

If a parenthesis, square bracket [] or a curly brace opened has no closure, it will be highlighted red.

After the inputs, a curly brace must be opened to indicate that between them is contained the function code.

int before a variable declares a variable as integer.

printf prints something in the command window. It must end with a \n.

scanf ask the user to enter a data and then stores into a variable.

After each statement, a semicolon; must be written.

3.3. Building and compiling our first programme

The Clean option for a programme cleans all intermediate files generated during compilation or execution: .o, libraries... It reuslts in a clean workspace that must be recompiled again. It can be useful is a compilation is generating problems in programmes with a lot of intermediate objects generated.

The difference between build and rebuild is that build compiles every programme in the workspace and rebuild only those that have changed since the previous build.

Build basically compiles and links the programme in the same step, instead of having to compile first and then link it.

The Workspace view only shows the .c and the executables of the workspace. To see all the files

in it, you must click and the folder drawing in the right side and select the folder of the workspace in which you are working, showing the json, .o, makefile...

You can use the Stop Build option if a build is taking too long and you want it to stop.

3.4. Compiler errors

A compiler error is an error that breaks the rules of the programming language of the compiler. They can be syntax errors or semantic errors.

A syntax error is when something is wrong with the structure of the code: missing semicolon, a parenthesis or bracket not closed, a quotation mark not closed, a forgotten # behind the include, a bad syntax of the header file stdio.h...

A semantic errors is something wrong with the meaning of a statement. For example, if a is an integer and b is a string, it does not make sense to state c = a + b;, put a string after a return.

Some of the compiler errors are not very descriptive, but at least the line of the error is clearly marked and with practice it becomes easy to identify the compiler error. If there is several errors, it returns all the errors founded.

When having several errors, start by fixing the first one since it may be causing another errors in following lines.

3.5. Compiler warnings

A warning is a notification of the compiler about an found issue with the code that could lead to a potential problem, even if it is not an error.

It is only a warning because the compiler is still able to generate correct machine code and an object file, but it should not be ignored.

An example of a warning occurs when you define a variable and try to print it or do an operation with it but has not been initialised with a value. The code can be generated, but when the programme is run is very likely that it will fail because it can print a non-existent error.

Another example is an unused variable, a variable, declared, initialised but never used in the code. The programme will not likely fail, but it is dead code that uses memory and resources in a useless way.

3.6. Linker errors

A linker error is caused when there is a problem linking all object files together to create an executable. Typically, this means that a library or an object file is missing.

It usually appears ündefined reference to"when this type of error occurs.

3.7. Runtime errors

These errors occur when the programme is executing: divide by zero, file not found, out of memory... and can cause the programme to crash.

3.8. Logic errors

These are errors in the code that cause the programme to run incorrectly. These are mistakes made by the programmer. For example, if you want to make a programme in which you introduce the age of a person and decide if he can vote, you have to put >=18, no>18, which can make the programme to return a non correct solution if you introduce 18.

3.9. Structure of a C programme

The main structure of a C programme are the following:

- In the first lines, the inclusion of libraries and external programmes is done by the command **include**.
- Then the main function of the C programme is initialised. Is the entry point of the programme and the contents are included between .
- After that, the contents of the main function are included.
- If necessary, after the main function, the auxiliary functions used will be written below the end of the main function.

The readability of the programme is important for correcting errors, debugging, and for other users to understand the programme.

An important note is the uppercase and lowercase letter are different in C, so A is not the same variable as a.

4. Basic concepts

4.1. Comments

Comments are used in a programme to document it and enhance its readability. There are to remind you or someone else reading your code what the programme or a specific line is doing. The comments are ignored by the compiler. The comments are useful when you return to a programme several months later and you do not remember what the programme does.

There are two ways to add comments into a C programme:

- Using the characters / and *. It is used for multi-line comments. These types of comments have to be terminated. The opening of the comment is marked with /* and the end with */. All characters between these are treated as part of the comment.

- Using //. The comment goes from the // to the end of the line. It does not have to be closed. It is used for single-line comments.

An example in the use of these two types of comments is the following:

```c
/* This programme adds two integer values
 * and displays the results */

#include <stdio.h>

int main(void)
{
    // Declare variables
    int value1, value2, sum;

    // Assign values and calculate their sum
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;

    // Display the result
    printf("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

Figura 4.1: Comments use example

If there are many comments, the readability of a programme may decrease instead of improve, so the comments need to be intelligently used. It is a good habit to insert comments statements into the programme as it is being written: it eases to document the programme when you have the logic in your head, and it helps with the debugging.

Using meaningful names help to self-document the code without using too many comments.

When you want to comment a part of the code, you can select those lines and use Ctrl+Shift+7 or go to Edit -> Comment and select Comment selection.

4.2. Preprocessor

A preprocessor is a unique feature of C that is not found in other programmes. It allows process to be easier to develop, read, modify and port to different computer systems. The preprocessor is a part previous to the compilation of the programme that recognises special statements and analyses them. It gives an instruction to your compiler to do something before compiling the source code.

The commands of the preprocessor are usually at the beginning of the programme, but they could be anywhere in it.

Preprocessor statements are identified by the pressence of a pund sign #, which must be the first non-space character on the line. The directive **#include** is an example of it.

The preprocessor is used to create constants (a data that never can change in the code) and macros with the **#define** statement, build your own library files with the **#include** statement and make more powerful programms with the statements **#ifdef**, **#endif**, **#else** and **#ifndef**.

4.3. The include statement

The statement **#include** is an example of a preprocessor directive. The example on the initial programmes were #include <stdio.h>. It is similar to the import statement in Java.

In the previous example, the compiler is instructed to include in the programme the contents of the file with the name stdio.h, which is called a header file because it is usually included at the head of a programme. The extension of these header files is.h.

The header files define information about some of the functions provided by the files. stdio.h is the standard C library header and provides functionality for displaying output, among many other things. We need to include this file in a programme when using the printf() function. stdio is short for standard input/output. stdio.h contains the information that the compiler needs to understand what printf() and other functions that deal with input and output mean.

Header files specify information that the compiler uses to integrate any predefined function within a programme. You will be creating your own header files for use in your programmes.

The header files should always be written in lowercase. There are two ways to include the header files:

- Using angle brackets: **#include <stdio.h>**. This tells the preprocessor to look for the file in one or more standard system directories.
- Using double quotes: **#include "stdio.h"**. This tells the preprocessor to first look in the current directory.

Every C compiler that conforms to the C11 standard will have a set of standard header files supplied.

You should use #**ifndef** and #**define** to ptrotect against multiple inclusions of a header file, so that the preprocessor does not enter the header file on multiple occasions unnecessarily.

The typical syntax is: a header, definition of some types, structures... and the functions that it includes.

Executable code normally goes into a source code file, not a header file.

4.4. Display output

printf() is a standard library function that outputs information to the command line based on what appears between its parenthesis. The line of use of the function must end in a semicolon.

This is probably the most common function used in C. It provides an easy and convenient mean to display programme results, since it can print variables and results of computations. It can also be used for debugging.

4.5. Reading input from the terminal

It is very useful to ask the user to enter data into a programme via the terminal. The C library contains several input functions, being **scanf()** the most general of them since it can read a variety of formats.

scanf() reads the input from the standard input stream stdin and scans that input according to the format provided. Format can be a simple constant strings, but you can specify %s, %d, %c, and %f to read strings, integer, character or floats.

scanf() is similar to **printf()** since both use a cnotrol string followed by a list of argument. The control string indicates the destination data types for the input stream of characters. However, **prinf()** uses variable names, constants and expressions as its argument list, but **scanf()** uses pointers to variables (although you don't need to know anything about pointers to use it).

There are three rules about **scanf()**:

- It returns the number of items that it successfully reads.
- If it is used to read a value for one of the basic variable types discussed, precede the variable name with an &.
- If it is used to read a string into a character array, don't use &.

scanf() uses whitespace (newlines, tabs, spaces) to decide how to devide the input into separate fields.

When a programme uses **scanf()** to gather input from the keyboard, it waits for you to input text. To read the input introduced, you must press Enter.

scanf() expects input in the same format as fou provided. If you gave the function %s and %d, it will expect an string followed by a number, not any other combination.

To read a double variable, the specified type of the function **scanf()** is %lf.

5. Variables and data types

5.1. Overview

The programme needs to store the instructions of the programme and the data it acts upon while the computer is executing it. This information is stored in memory (RAM), whose contents are lost when the computer is turned off. Hard drives store persistent data.

RAM can be think of as an ordered sequence of boxes. The box is full when it represents a 1 and empty when represents a 0. Each box represents a binary digit, 0 or 1 (true and false) and each box is called a bit.

Bits are grouped into sets of eight (byte). Each byte has been labeled with a number, which is called the address of the byte. The address of a byte uniquely references that byte in the memory of the computer.

The true power of programmes is the maniputation of data, so we need to understand the data types you can use to create and name variables.

Constants are types of data that do not change and retain their values throughout the life of the programmes.

Variables are types of data that may change or be assigned values as the programme runs. The variables are the names you give to computer memory locarions which are used to store values in a computer programme.

The rules for naming variable sin C are: all names must begin with a letter or underscore (_) and can be followed by any combination of letters (uppercase or lowercase), underscores or the digits 0-9. Invalid names of variables are: temp$value ($ is not a valid character), my flag (spaces are not allowed), 3Jason (it cannot start by a number), int (reserved word).

As has already been said, use meaningful names for the variables, which eases the readability and helps the debug and documentation phases.

A data type represents a type of the data which you can process using your programme: integers, floats, doubles...

Primitive data types are types that are not objects and store all sorts of data. Everything is a primitive data type in C.

Declaring a variable is when you specify the type of variable followed by the variable name. It specifies to the compiler how a particular variable will be used. For example **int** is used to declare the basic integer variable. First comes int, then the chosen name of the variable and then a semicolon.

The structure is: **type-specifier variable-name;**. You can declare more than one variable in one line by separating their names by commas.

C requires all programme variables to be declared before being used in a programme. The declaration creates variables but not provides values for them, which it is done by using the = operator. For example: **int x;** declares a integer variable named x and **x = 112;** gives it the value 112.

Initialise a variable means to assign it a starting or initial value. This can be done as part of the declaration by following the variable name with the = operator and the value you want. For example: **int x = 21;** declares a integer variable x and assign it the value of 21. You can do it to every variable declared in the line: **int y = 32, z = 14;** or to only some of them: **int x, z = 94;**. In the last example, only z in initialised. It is best to avoid putting initialised and non-initialised variables in the same declaration statement.

After initialise a variable, you can modify its value.

5.2. Basic Data types

The different types of data are used for storing kind of data: integers, nonintegral numerical values, characters. Some examples of basic data types in C are: int, float, double, char and _Bool.

The difference between the types is in the amount of memory the occupy and the range of values they can hold. This depend on the computer which is running.

5.2.1. int

A type **int** variable can be used to contain integral values only, without any decimal place. A minus preceding the data type and variable indicates a negative value. It can be positive, negative or zero. If an integer is preceded by a 0x (lowercase or uppercase) the value is expressed in hexadecimal (base 16) notation. Example: **int rgbColor = 0xFFEF0D**.

No embedded space are allowed between the digits. Values larger than 999 cannot be expressed using commas: 12,000 must be written as 12000.

5.2.2. float

A variable of type **float** can be used for storing floating-point numbers (values containing decimal places).

Floating-point constants can be expressed in scientific notation: **1.7e4** is the same as 17000.0.

5.2.3. double

It is the same type as float, but with roughly twice the precision. It is used when the range provided by a float variable is not enough. It can store twice as many significant digits and most computers represent them by using 64 bits.

All floating point constants are taken as doubles by the C compiler, to express a float constant there must be an f or F at the end of the number: **12.5f**

5.2.4. _Bool

It can be used to store just the values 0 or 1. It is used for indicating binary choices: yes/no, on/off, true/false...

Are used in programmes that need to indicate a Boolean condition. 0 is used to indicate a false value and 1 indicates a true value. You can give them the values 1/0 and true/false and it is the same.

Another way for declaring boolean variables is to include the head file <**stdbool.h**> and then you can define boolean variables as **bool**. For example: **bool myBoolean = true;** declares a boolean variable called myBoolean with value 1.

5.2.5. Other data types

C offers many other integer types, that vary in the range of values and in whether negative numbers can be used. C offers three adjective keywords to modify basic integer type: short, long and unsigned. Short can be used to save memory when only small numbers are needed. Long enables you to use more storage than int and express larger integer values. The type long long may use more storage than long.

This specifiers can be applied to doubles. A long double constant iw written as a floating constant with the letter I or L inmediately followintg: 1.234e+7L.

The unsigned int is used for variables that have only positive values. Its accuracy is extended.

The signed keyword is used with any of the signed types to make your intent explicit: short, short int, signed short, signed short int are all names for the same type.

5.3. Enums and chars

5.3.1. Enums

Enum is a data type that allows a programmer to define avariable and specify the valid values that can be stores in that variable. For example, you can create a variable named myColor and it can only contains one of the primary colors: red, yellow, blue and no other values.

The first thing is to define the enum type and give it a name: first the keyword enum, then the name of the variable and finally the list of valid identifiers between curly braces. Example: **enum PrimaryColor {red, yellow, blue}**.

To declare a variable to be of type enum PrimaryColor, you use the enumerated type name, followed by the enumerated type name and the variable list. Example: **enum PrimaryColor myColor, gregsColor** makes that the variables myColor and gregsColor can only be of the types of PrimaryColor: red, yellow and blue.

The compiler actually treats enumeration identifiers as integer constant starting in 0. For example if there is an enum month with the list of months, ans a variable of this type is equal to February, the variable will be identifed as 1. You can assign different values to the values. **enum direction {up, down, left = 10, right}** has 4 values up (0 because it is the first), down (1

because it is the second), left (10 because it is specified) and right (11 because it is after the left value).

5.3.2. Char

It represents a single character such as the letter 'a', the digit '6' or a semicolon ';'. Character use single quotes to be defined. They can be declared to be unsigned to explicity tell the compiler that a variable is a signed quantity. There are different to strings.

For declaring it, you must type **char broiled;** and to assign it, there must always be single quotes: **broiled = 'T';** is valid while **broiled = T;** it is not valid, since it will think that T is a variable. **broiled = "T";** will define a string. It can be assigned as a numerical code of the character.

Escape characters

C contains special characters that represent actions: backspacing, going to the next line, making the terminal bell ring... They can be represented by using special symbol sequences called escape sequences.

For example, \ **n** represents a new line. If you define **char x = '\ n'** you define that the command of new line can be represented as an x. The following is a list of escape sequences:

Sequence	Meaning
\a	Alert (ANSI C).
\b	Backspace.
\f	Form feed.
\n	Newline.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\\	Backslash (\).
\'	Single quote (').
\"	Double quote (").
\?	Question mark (?).
\0oo	Octal value. (o represents an octal digit.)
\xhh	Hexadecimal value. (h represents a hexadecimal digit.)

Figura 5.1: Escape sequences in C

5.4. Challenge Enum

The programme requires to create an enum type named Company with valid values: GOOGLE, FACEBOOK, XEROX, YAHOO, EBAY, MICROSOFT and the programme has to create three variables of the above enum type with values XEROX, GOOGLE and EBAY. The programme should display as output the value of the three variables with each variable separated by a new line. The correct output is the variables XEROS, GOOGLE and EBAY to be displayed as: 2 0 4

```c
#include <stdio.h>
#include <stdbool.h>

int main()
{
    // Define the enum data type
    enum company {GOOGLE, FACEBOOK, XEROX, YAHOO, EBAY, MICROSOFT};

    // Define the enum variables
    enum company xerox = XEROX;
    enum company google = GOOGLE;
    enum company ebay = EBAY;

    // Define the character of new line
    char x = '\n';

    // Print the three companies separated by new lines
    printf("The three values of the companies are: %c %d %c %d %c %d", x, xerox, x, google, x, ebay);

    return 0;
}
```

Figura 5.2: Challenge Enum solution

The programme that solves the challenge is:

5.5. Format specifiers

They are used when displaying variables as output. They specify the type of data of the variable to be displayed.

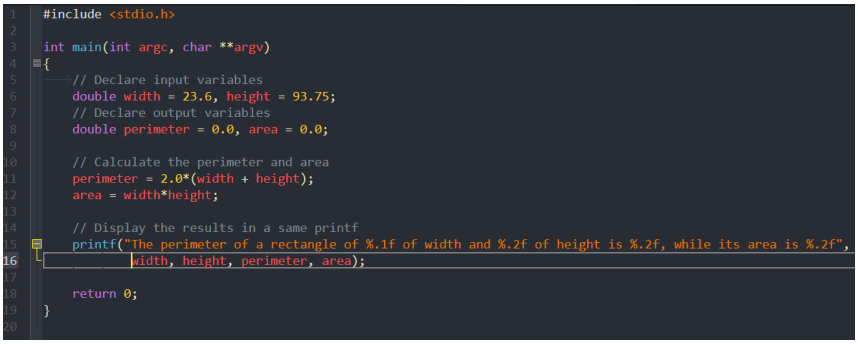
The **printf()** function is the function used to display the variables. Its first argument is the string to be displayed and after that there are optional arguments. Often, along with the character string, you want the value of certain programme variables. The output of the printf function that will be displayed is the character string, and inside it you can include format specifiers to point to the variables you want to display.

The character % inside the string is a special character recognised by the **printf()** function. The character that follows the % specifies the type of value to be displayed. After the string, the following arguments of the **printf()** function are the variables that correspond to the format specifier.

The following commands are used for the different types of variables:

- %i is used to print a integer variable, or boolean variable (0 or 1).
- %d is used to print a integer variable, or boolean variable (0 or 1).
- %f is used to print a float variable.
- %e is used to print a double variable.
- %lg is used to print a double variable of different type, without the final decimals number.
- %c is used to print a character variable.

When printing numbers, you can specify the number of decimal numbers that will the printed number will be rounded to by adding a .3 if you want it rounded to three decimals, .5 if you want it to five...



```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Declare input variables
    double width = 23.6, height = 93.75;
    // Declare output variables
    double perimeter = 0.0, area = 0.0;

    // Calculate the perimeter and area
    perimeter = 2.0*(width + height);
    area = width*height;

    // Display the results in a same printf
    printf("The perimeter of a rectangle of %.1f of width and %.2f of height is %.2f, while its area is %.2f",
            width, height, perimeter, area);

    return 0;
}
```

Figura 5.3: Rectangle challenge solution

5.6. Challenge: Print the rectangle area

The programme should display the perimeter and area of the rectangle.

It should create 4 double variables: one for width, another for height, another for perimeter and another for area. They should display the variables in the correct format in one print statement. Perimeter and area need to be initialised with 0.0 value and the other two with any values desired.

The solution is the following:

5.7. Command line arguments

There are times when a programme is developed that requires the user to enter a small amount of information at the terminal. There are two ways of handling it: request the data from the user or supply the information at the time the programme is executed.

The main function is the entry point of the programme. When is called by the runtime system, two arguments are passed to the function:

- **argc**, the first argument, is an integer that specifies the number of arguments typed on the command line.
- **argv**, the second argument is an array of character pointers (strings).

An argument is how you can pass data to a function. Basically, you first specify how many arguments the function has and then an array with their values.

The name of the programme is always the first argument of argv, so if not specified otherwise, argc = 1 and argv will have only the name of the programme.

One way to edit this is by adding arguments through the command line option, which can be done in CodeLite by right-clicking on the project, select settings, go the General and there is an option called program arguments in which you can introduce the arguments desired. The other option is to ask the user to introduce the data.

6. Operators

6.1. Overview

Operators are functions that use a symbolic name and perform mathematical or logical functions. Are predefined in C, like in most languages, and tend to be combined with the infix style (5 + 8, prefix style would be 5 8 +).

A logical or Boolean operator is an operator that returns a Boolean result based on the Boolean result of one or two other expressions.

An arithmetic operator is a mathematical function that takes two operand and performa calculation on them.

Other operators include assignment (=), relational (<, >,!=), bitwise (≪, ≫,).

6.1.1. Expressions and statements

Statements are the basic programme steps of C, most statements are constructed from expressions.

An expression is a combination of operators (what an operator operates on, its arguments) and operands (constants, variables or combination of the two types) that always has a value.

Statements are the building blocks of a programme. A programmes is a series of statements with special syntax ending with a semicolon (simple statements) that results in a complete instruction to the computer. There are several types of statements:

- Declaration statement: **int Daniel;**.
- Assingnment statement: **Daniel = 5;**.
- Function call statement: **printf("Daniel");**.
- Structure statement: **while(Daniel¡20) Daniel = Daniel + 1;**.
- Return statement: **return 0;**.

C considers any expression to be a statement if it is ended with a semicolon (expression statements).

6.1.2. Compound statements

Two or more statements grouped together by enclosing them in braces (block).

Example: **int index = 0; while (index¡10) { printf("hello"); index = index + 1; }**

6.2. Basic operators

In this Subsection, four operators will be discussed: arithmetical, logical, assignment and relational.

A logical or Boolean operator is an operator that returns a Boolean result based on the Boolean result of one or two other expressions.

An arithmetic operator is a mathematical function that takes two operand and performa calculation on them.

An assignment operator set variables equal to values: assigns the value of the expression at its right to the variable at its left.

A relational operator will compare variables against each other.

In the following table the arithmetic operators in C are displayed:

Tabla 6.1: Arithmetic operators in C

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Substracts two operands	A - B = 10
*	Multiplies both operands	A*B = 200
%	Modulus operator and remainder after an integer division	B %A = 0
++	Increment the integer value by one	A++ = 11
–	Decrement the integer value by one	A– = 9

The logical operators are the following:

- **&&** is the logical AND operator. If both operands are non-zero, the condition becomes true. Example: **A&&B** is false.
- || is the logical OR operator. If any of the two operands is non-zero, the condition becomes true. Example: **A||B** is true.
- **!** is the logical NOT operator. It is used to reverse the logical state of its operand. If a condition is true, this operator will make it false. Example: **!(A&&B)** is true.

The assingment operators are the following:

- = is the simple assignment operator. Example: C = A + B will assign the value of A + B to C.
- += is the add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand. Example: C += A is equivalent to C = C + A.
- -= is the subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. Example: C -= A is equivalent to C = C - A.
- *= is the multiply AND assignment operator. It multiplies the right operand by the left operand and assigns the result to the left operand. Example: C *= A is equivalent to C = C*A.

- /= is the divide AND assignment operator. It divides the right operand by the left operand and assigns the result to the left operand. Example: C /= A is equivalent to C = C / A.
- %= is the modulus AND assignment operator. It takes modulus using two operands and assign the result to the left operand. Example: C %= A is equivalent to C = C %A.

The shift and bitwise operators can also be assigned like the above operators by adding the = to the right. Those operators will be seen in depth in future lectures.

The relational operators compare the values of two variables:

- == Checks if the values of two operands are equal or not. If yes, the condition becomes true.
- != Checks if the values of two operands are equal or not. If yes, the condition becomes false.
- < Checks if the value of the left operand is greater than the value of right operand. If yes, the condition becomes true.
- > Checks if the value of the left operand is less than the value of right operand. If yes, the condition becomes true.
- < Checks if the value of the left operand is greater than or equal to the value of right operand. If yes, the condition becomes true.
- > Checks if the value of the left operand is less than or equal to the value of right operand. If yes, the condition becomes true.

6.3. Challenge: COnvert minutes to years and days

This programme must convert the number of minutes to days and years. It should ask the user to enter the number of minutes via the terminal and display as output the minutes and then its equivalent in years and days. The programme should create variables to store: minutes (int), minutes in year (double), years (double), days(double).

The solution is:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Define variables
    int minutes;
    double minutesinyear = 60.0*24.0*365.0;
    double years, days;

    // Ask the user to input the number of minutes for the conversion
    printf("Enter the number of minutes to convert \n");
    scanf("%d", &minutes);

    // Convert the minutes received into years and days
    days = minutes/(24.0*60.0);
    years = minutes/minutesinyear;

    // Return the results
    printf("%d minutes are equivalent to %2.2f days and %4.4f years", minutes, days, years);
    return 0;
}
```

Figura 6.1: Solution to the Challenge of converting minutes

6.4. Bitwise operators

Look similar to the logical operators but are quite different. It operate on the bits in integer values. They are not used in the common programme. One major use are to test and set individual bits in an integer variables, that can be used to store data that involve one of two choices.

For example, you could use a single integer variable to store several characteristics of a person: one for whether the person is female or male, three to specify if can speak French, German and Italian; and another if the salary is $50000 or more, and in five bits there are a substantial set of data recorded.

6.4.1. Binary numbers

A binary number is a number that includes only ones and zeroes. The number could be of any lenght: 1, 0, 1010, 00101001, 10100101010101000101010011110 10...

To obtain its decimal value, for each digit you multiply it by its position value and add up all products to get the final result. In general, the posision values are the powers of two. Example: the number 01101001 can be obtained as: (128*0 + 64*1 + 32*1 + 16*0 + 8*1 + 4*0 + 2*0 +1 = 105.

6.4.2. Bitwise operators

The bitwise operators are the following:

- & Binary AND operator copies a bit to the result if it exists in both operands.
- | Binary OR operator copies a bit if it exists in either operand.
- ∧ Binary XOR operator copies the bit if it exists in one operator but not both.
- Binary Ones Complement operator is unary and flips all the bits of a number.
- ≪ Binary Left Shift operator. The left operands value is moved to the left by the number of bits specified by the right operand.
- ≫ Binary Right Shift operator. The left operands value is moved to the right by the number of bits specified by the right operand.

6.5. Cast and Sizeof operators

Conversion of data between different types can happen automatically (implicit conversion) by the language or explicit by the programme. To effective develop C programme, you must understand the rules used for the implicit conversion of floating-point and integer values in C.

Normally, the types should not be changed, but there are occasions when it is useful. C is flexible, it gives freedom, but it must not be abused.

Truncated means that the data have been changed to a less precise type and Promoted means that the data have been changed to a more precise type.

When a floating-point value is assigned to an integer variable in C, the decimal portion of the number gets truncated. Assigning an integer to a floating variable does not change the value of the number.

When performing integer arithmetihic:

- If two operands in an expression are integers, any decimal portion resulting from a division operation is discarded, even if the result is assigned to a floating variable.
- If an operand is an integer and the other a float, the operation is performed as a floating point operation.

6.5.1. Cast operator

It is usually much better to perform an explicit conversion through the cast operation. A cast consists on preceding the quantity with the name of the desired type in parentheses. The parenthesis and type name together is the cast operator.

It has a higher precedence than all the arithmetic operator except unary minus and unary plus.

Example: **(int)21.51 +(int)26.99** is equivalent to 21 + 26 = 47.

6.5.2. Sizeof operator

This operator is used to find out how many bytes are occupied in memory by a given type by this operator. Its argument can be a variable, an array name, the name of a basic data type, the name of a derived data type or an expression.

6.5.3. sizeof(int)

returns the number of bytes occupied by a variable of type int.

It is usually used how much many memory to allocate when using pointers.

It is also used to avoid having to calculate and hard-code sizes into your programme.

6.5.4. Other operator

When an asterisk * is used with only one argument it represents a pointer to that variable: *a represents a pointer to the variable a.

? is the ternary operator, an operator used for comparisons. If condition is true ? then value X, otherwise value Y.

They will be explained in detail when studying pointers.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int intsize, charsize, longsize;
    int longlongsize, doublesize, longdoublesize;

    intsize = sizeof(int);
    charsize =sizeof(char);
    longsize = sizeof(long);
    longlongsize = sizeof(long long);
    doublesize = sizeof(double);
    longdoublesize = sizeof(long double);


    printf("The different data types size are the following: \n");
    printf("int type: %d \n", intsize);
    printf("char type: %d \n", charsize);
    printf("long type: %d \n", longsize);
    printf("long long type: %d \n", longlongsize);
    printf("double type: %d \n", doublesize);
    printf("long double type: %d \n", longdoublesize);

    return 0;
}
```

Figura 6.2: Challenge of sizeof solution

6.6. Print the byte size of the basic data types

The challenge is to create a C programme that displays the byte size of basic data types supported in C. The output will vary depending on the system that runs the programme.

The types must be: int, char, long, long long, double, long double. It must use the sizeof operator.

The solution is:

6.7. Operator precedence

Operator precedence decides the order of evaluation in an operation with more than one operator. Certain operators have higher order of preference than others.

C has unambiguous rules for choosing what to do first. For example the multilication has a higher precedence than the addition, so it is performed first.

You can decide what to execute first; whatever is enclosed in parentheses is executed first.

If two operators have the same precedence level, associativity rules are applied. If ther share an operand, they are executed according to the order in which they occur in the statement. For most operators the order is from left to right.

Operators == and != have the same precedence. The associativity of both parameters is left to right. 1 == 2 != 3 does first the 1==2 operation and then 2!=3. It will return: 1==2 results into 0 (false) then (0!=3) executes resulting into 1 (true).

The next table includes the assignment operators precedence (highest to lowest) and the associativity for each operator:

Category	Operator	Assocativity
Postfix	(),[],->, ., ++, –	Left to right
Unary	+!, -!, , ++, –, (type), *, &, sizeof	Right to left
Multiplicative	*, /, %	Left to right
Additive	+, -	Left to right
Shift	$\ll, \gg$	Left to right
Relational	$<, <=, >, >=$	Left to right
Equality	==, !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	$\wedge$	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Bitwise OR	||	Left to right
Conditional	?:	Right to left
Assignment	=, +=, *=, /=, %=, $\gg=$, $\ll=$, &=,$\wedge=$,|=	Right to left
Comma	,	Left to Right

Tabla 6.2: Operators precedence and associativity rules

The best way of avoiding errors of associativity and precedence is using parenthesis when there are any doubts.

7. Control Flow

7.1. Overview

Control flow deals with conditional statements and loops. The statements of a programme are generally executed from top to bottom, in order. Control flow statements break up the flow of execution by employing decision-making, looping and branching; enabling your programme to conditionally execute particular blocks of code. The three types of control flow statements are:

- **Decision-making statements:** Decides which part of the code runs. if-then, if-then-else, switch, goto.
- **Looping statements:** Repeat a part of the code a number of times or until a condition is met. for, while, do-while.
- **Branching statements:** Alterate a decision-making or looping statement. break, continue, return.

7.1.1. Decision-making

Structures that requires that the programmer specify one or more conditions to be evaluated or tested by the programme. If a condition is true, a statement or statements are executed. If it is false, other statements are executed.

The types of if statements are:

- **if statement:** A boolean expression followed by one or more statements.
- **if...else statement:** An if statement followed by an optional else statement, executed if the boolean expression is false.
- **nested if:** You can use one if or else if statements inside another if or else if statements.

7.1.2. Repeating code

Structures that execute a block of code multiple times. A loop becomes infinite if a condition never becomes false. The for loop is traditionally used for this purpose.

The types of loops in C are:

- **while loop:** Repeats a statement or statements while a given condition is true. It tests the condition before executing the loop body.
- **for loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- **do while loop:** Similar to a while statement, except it tests the condition at the end of the loop body. The main difference to a while is that is guarantees that the loop is entered at least one time.
- **nested loops:** One or more loops inside another while, for, do while loop.

7.2. If statements

As it has been said, is the general decision-makinf in the form of an if statement. The syntax is: **if(expression) programme statement**

It is used to estipulate execution of a programme statement/s based upon specified conditions. The curly brackets are required for compound statemens inside the if block, but are not mandatory if there is only one statement inside the if condition. If there are two or more statements, they are mandatory.

In the if(expression) there are not semicolons at the end, but they must be in the statements of inside it.

The else statement gives more flexibility by adding a second block of code if a boolean expression is false:

if(expression) Statement 1 else Statement 2

The next statment under the if or the else if they do not have curly brackets must be separated by one blank line to not have any warnings.

The expression must have a relational (with the operators $<, <=, \gg=$) or equality (with the operators $==, !=$) statement, a single $=$ never can appear, since it is an assignment, not an expression.

If there is more than one condition or cases for the if, the else if expression can be used. The structure is:

if(expression 1) Statement 1 else if (expression 2) Statement 2 else Statement 3

A nested if-else statement means that you can use one if or else if inside another if or else if. The structure is:

if(expression 1) { if (expression 2) Statement 1 else Statement 2 } else Statement 3

The conditional operator or ternary statement is a unique operator. Most operators are unary or binary operators (one or two operands), but the ternary operator takes three operands. The two symbols used to denote this operatr are the question mark ? and the colon :. The first operand is placed before the ?, the second between the ? and the : and the third after the :. Structure: **Condition ? expression 1 : expression 2**. The first operand is the boolean condition, the expression 1 is the expression evaluated if true and the expression 2 is the expression evaluated if false. It is a short form to make an if-else statement in which the statements of both conditions are a single statements. It cannot be used if one of the conditions has multiple assignment.

Example: **x = y ¿7 ? 25 : 50** results in x being set to 25 if y is greater than 7 or to 50 otherwise. Is the same as:

if(y>7) x = 25; else x = 50;

7.3. Challenge Amount of pay

This function should create the weekly pay. The program should ask the user the number of houts worked in a week via the keyboard. The programme should display as output the gross pay, the taxes and the net pay. The following assumptions hould be made:

- Basic pay rate: $12.00 per hour.
- Overtime (in excess of 40 hours) = time and a half pay rate.
- Tax rate: 15 % of the first $300, 20 % of the next $150 and 25 % the rest.

The solution is:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Declare variables
    double hours, extrahours, income, taxes, netincome;
    double payrate = 12.00, taxrate1 = 0.15, taxrate2 = 0.2, taxrate3 = 0.25;

    // Enter the number of hours worked
    printf("Please, enter the number of hours: \n");
    scanf("%lg", &hours);

    // Calculate the extra hours
    if (hours > 40.0)
        extrahours = hours - 40.0;
    else
        extrahours = 0.0;

    // Calculate the total income
    income = payrate*(hours - extrahours) + 1.5*payrate*extrahours;

    // Calculate taxes
    if (income > 300)
        if (income > 450)
            taxes = (income - 450)*taxrate3 + 150*taxrate2 + 300*taxrate1;
        else
            taxes = (income - 300)*taxrate2 + 300*taxrate1;
    else
        taxes = income*taxrate1;

    // Obtain final net income
    netincome = income - taxes;

    printf("For %lg hours worked, the gross salary is: $%lg \n", hours, income);
    printf("Among the %lg hours worked, the number of extra hours is : %lg \n", hours, extrahours);
    printf("And the resulting net salary after pay $%lg taxes is: $%lg \n", taxes, netincome);
    return 0;
```

Figura 7.1: Solution to the Challenge of Amount of pay

7.4. Switch statement

It is a similar operator to the if-else if, that is specially useful when you want to choose one of several alternatives. It is used when the value of a variable is successively compared against different values, being more convenient and efficient for these cases. You can only use it when comparing the values of a variable to a series of values.

The structure is:

switch (expression)

{

case value 1:

programme statement

break;

...

case value n:

programme statement

break;

default:

programme statement

...

break

}

The default case is the else of this statements, if any case studied fulfils the condition, this is the case run.

The expression enclosed within parentheses is successively compared against the values: value 1, value 2, value 3... value n. The cases must be simple constants or constant expressions.

If a case is found whose value is equal to the value of the expression, the statements that follow the case are executed. When more than one statement is included, they do not have to be enclosed within braces.

The break statement signals the end of a particular case and causes the execution of the switch statement to terminate. They must be included at the end of every case; otherwise the switch will keep going to the next cases.

This is a useful statement for the enum variables. See the SwitchExample of the Workspace for an example of how to do a switch with an enum.

7.4.1. goto statement

The goto statement is available in C. It has two parts, the goto and a label name. Label is named following the same convention used in naming a variable. It is not liked because it makes hard to read the code and follow where it is going. Example: **goto part2;**, with part2 being a label of a certain statement.

A case where it may be useful is when having a lot of loops and you want to jump them for any reason, but it is not recommendable to use it.

7.5. For loop

The loops are a way to repeat a block of code when you need to use the same code repeatedly. The number of time that a loop is repeated can be controlled by a count (repeating the statement block a number of times: counter controlled loop) ot can depend on when a condition is met (for example, until the user enters 'quit').

You typically use of the for loop is to execute a blovk os statements a given number of times.

For example, if you want to display the numbers from 1 to 10, instead of writing ten **prinf()** you would use a for loop.

The syntax of the for loop is:

for(int count = 1 ; count <= 10 ; ++count) { **statement** }

As the if statements, if the loop is a single stamement you do not need to use curly brackets, but if there are several statements they are mandatory.

The for loop operation is controlled by what appears between the parentheses that follow the keyword for, separated by semicolons. Theh action that you want to repeat is the block contained between the curly braces.

The general pattern for the for loop is; **for(StartingCondition ; ContinuationCondition; ActionPerIteration)**.

The starting condition typically (but not always) sets an initial value to a loop control variable. Is typically a counter os some kind that tracks how often the loop has been repeated. It can also declare and initialise several variables of the same type with declarations separated by commas, variables that will be local to the loop and will not exist once the loop ends.

The continuation condition is a logical expression evaluated to true or false that as long as its true, the loop continues. Typically, checks the value of the loop control variable. You can write any logical or arithmethic expression as long as you know what you are doing. THe continuation condition is tested at the beginning of the loop, not at the end. This means that the loop statement will not be executed if the continuation condition is false.

The action per iteration is executed at the end of each loop interation. It is usually an increment or decrement of one or more loop control variables. Several variables may be modifies, just need to use commas to separate.

An example of a loop of several variables:

for (int i=1, j=2; i<5, ++i, j = j +2

printf(" %5d", i·j

The output will be a single line with values 2, 8, 18, 32 and 50.

You have no obligation to put any parameter in the for loop statement: **for(;;)**, which creates an infinite loop. This is sometimes useful for monitoring data or listening for connections.

7.6. While and Do-While

These are mechanisms to repeat a set of statements continuously until a specified logical expression evaluates to true.

The general syntax for the while loop is as follows:

while (expression)

{

statement

}

As before, the curly brackets are optional if there is only one statement and mandatory if there are more than one.

The condition for continuation of the while loop is tested at the start of the loop, so if the expression starts false the loop will not be executed. If the loop condition starts true, the condition must be modified in the inside of the loop.

The do-while is a loop where the body is executed for the first time unconditionally. The condition is at the bottom (post-test loop), so it is always guaranteed to be executed at least once.

The general syntax for the do while loop is as follows:

do

{

statement

}**while (condition)**

TO decide which loop to use, first decide whether you need a pre- or post-test loop. Usually it will be a pre-test loop (for or while), which also makes it easier to read if the condition is at the beginning.

Between a for or a while, you really can do everything that one do with the other. Sometimes, for is a little faster, but there are no differences.

One way of use a while as a for loop is initialise the variable outside the while and include update statements at the end of the loop.

7.7. Nested loops

Sometimes you may want to place one loop inside another. The structure will be:

for(StartingCondition1 ; ContinuationCondition1; ActionPerIteration1) {

statement1

for(StartingCondition2 ; ContinuationCondition2; ActionPerIteration2)

{

statment 2

}

statement posterior (optional)

}

7.7.1. Continue statement

Sometimes a situation arises where you do not want to end a loop but you want to skip the current iteration. This is done by the continue statement inside the body of a loop.

An advantage of using continue is that it can sometimes eliminate nesting or additional blocks of code and can enhance readability when the statements are long or deeply nested.

It is better not to use it if it can worsen the code complexity or readability.

The continue statement is followed by a semicolon.

7.7.2. Break statement

The break keyword inside of a loop will cause the loop to end its execution and the programme to get out of the loop. If the break statement is inside a nested loop, it only affects the innermost loop that contains it.

The break statement is followed by a semicolon.

It is often used to leave a loop when there are two separate reasons to leave.

Break is also used in switch to get out of the switch, the functioning is very similar.

7.8. Challenge Guess the Number

The objective is to create the Guess the NUmber programme. Your programme will generate a random number from 0 to 20 and ask the user to guess it. User should only be able to enter numbers from 0 to 20. Then the programme will indicate to the user if each guess is too high or too low. The player wins the game if they can guess the number within five tries.

To generate a random number, the libraries stflib.h and time.h need to be included. Then a time variable must be created: **time_t t;** and the random number generator must be initialised: **srand((unsigned) time(&t));**. Finally, you get the random number and store it in a variable: **int randomNumber = rand() %21;**.

The solution is:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv)
{
    // Declare variables
    time_t t;
    int randomNumber = 0 , numbertries = 1;
    int enterNumber = 0;

    // Initialise the random number generator
    srand((unsigned) time(&t));

    // Get the random number
    randomNumber = rand()%21;

    printf("%d \n", randomNumber);

    while (numbertries < 6)
    {
        printf("Enter an integer number between 0 to 20: ");
        scanf("%d", &enterNumber);

        printf("%d \n", enterNumber);

        if (enterNumber > 20)
            {
                printf("The introduced number is not between 0 to 20, try again \n");
                continue;
            }
        else if (enterNumber < 0)
            {
                printf("The introduced number is not between 0 to 20, try again \n");
                continue;
```

Figura 7.2: Solution Challenge Guess the Number 1

```c
                continue;
            }
        else
            {
                if (enterNumber == randomNumber)
                    {
                        printf("Congratulations, you guessed it \n");
                        break;
                    }
                else if (enterNumber > randomNumber)
                    {
                        printf("Wrong, your guess was too high \n");
                        printf("You have %i guesses remaining \n", 5 - numbertries);
                    }
                else
                    {
                        printf("Wrong, your guess was too low \n");
                        printf("You have %i guesses remaining \n", 5 - numbertries);
                    }
            numbertries += 1;
            }
    }

    if (numbertries == 6)
        {printf("Sorry, you are out of tries \n");}
    else
        {printf("You won!!! \n");}

    return 0;
}
```

Figura 7.3: Solution Challenge Guess the Number 2

8. Arrays

8.1. Creating and using arrays

Arrays are a type of data that allows to store different data values of a specified type in a single variable. Arrays allows to group variables together under a single name, without needing separated variables for each item of data.

An array is a fixed (once created, its size cannot be modified, important limitation) number of data items that are all of the same type (you cannot mix data types, for example integers and doubles).

The data items in an array are called elements.

Declaring an array is similar to a normal variable that contains a single value, but you need a number between square brackets [] following the name. Example: **long numbers[10];** means that it is an array of size 10 called numbers with values of type long.

Each of the data items stored in an array is accessed by the same name. If you want a particular element, you use an index value between square brackets following the array name.

Index values are sequential integers that start from zero. Index values for elements in an array of size 10 would be 0-9. Arrays are zero-based; 0 is the index value for the first array element, and the last array element number will be its size minus 1.

It is a very common mistake to assume that arrays start from one (as happens in MATLAB). This is called the off-by-one error. To access the fourth value in an array, use the expression arrayName[3].

You can also specify an index for an array element by an expression in the square brackets following the array name. The expression must result in an integer value that corresponds to one of the possible index values.

It is very common to use a loop to access each element in an array:

for i=0; i<10; ++i

printf("Number is %d", numbers[i]);

If you use an expression or a variable for an index value that is outside the range of the array, your programme may crash, or the array can contain garbage data. This is referred to as an out-of-bounds error. The problem of this error is that the compiler cannot check this, so the programme will still compile. It is very important to ensure that the arrays remain in its bounds.

A value can be stored in an element of an array simply by specifying the element of the array on the left side of an equal sign: **grades[100] = 95;**.

8.2. Initialising an array

You will want to assign initial values for the elements of your array most of the time. Definining initial values for array elements makes it easier to detect when things go wrong.

Just as you can assign initial values to variables when declared, you can also assign initial values to an array's elements. To initialise the values of an array, provide the values in a list enclosed in curly braces and with the values separated by commas. Example: **integers[5] = 0,1,2,3,4;**.

It is not necessary to completely initialise an entire array. If fewer values are specified, only the first elements are initialised, and the remaining values are set to zero. Example: **float sample_data[500] = 100.0, 300.0, 500.5;** initialises the first three values of the float to 100.0, 300.0 and 500.5 and the remaining 497 elements to zero.

C99 added a feature called designated initializers. It allows to pick and choose which elements are initialised by enclosing an element number in a pair of brackets. The example above would be: **float sample_data[500] = [2] = 500.5, [1] = 300.0, [0] = 100.0;**.

C does not provide any shortcut mechanisms to initialise the array elements to the same number. If it were desired to initially set all 500 values of **sample_data** to 1, all 500 would have to be explicitly assigned. To solve this problem, normally the array values are initialised inside the programme using a loop.

8.3. Challenge Prime Numbers

The programme should find all prime numbers from 3-100. There will be no input to the programme and the input will be each prime separated by a space on a single line. You can hard-code the first two prime numbers in the primers array. You should utilise loops to only find prime numbers up to 100 and a loop to print out the primes array.

The criteria that can be used to identify a prime number is that a number is considered prime if it not evenly divisible by any other previous prime numbers.

The solution is the following:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Define variables
    int primenumbers[50] = {2,3};
    int Nprimes = 2;
    int module = 0;
    int i = 0, j = 0, k = 0;


    // First loop to go from numbers 5 to 100 (2 and 3 initialised, 4 is even)
    // Increments from 2 to 2 because an even number is never prime
    for (i = 5; i<=100; i = i + 2)
    {
        for (j = 0; j < Nprimes - 1; j++)
        {
            // Divides by the previous prime numbers
            module = i%primenumbers[j];
            if (module == 0)
                {break;}
        }
```

Figura 8.1: Solution Challenge Prime Number 1

```
23
24          // If module == 0, is not a prime number
25          if (module > 0)
26          {
27              Nprimes = Nprimes + 1;
28              primenumbers[Nprimes - 1] = i;
29          }
30          else
31              {continue;}
32      }
33
34      printf("\nThe prime numbers between 1 to 100 are:");
35
36      for (k = 0; k < Nprimes; k++)
37      {
38          printf("%d, ", primenumbers[k]);
39      }
40
41      return 0;
42  }
43
```

Figura 8.2: Solution Challenge Prime Number 2

8.4. Multidimensional arrays

Until now, all arrays seen have ben lineal arrays, with single dimensions. However, C allows arrays of any dimension to be defined. Two dimension arrays are the most common. It represents a matrix with rows and columns, and this is the most natural application. An example of a two-dimensional array declaration would be: **int matrix[4][5];**. This declares the array matrix to be a two-dimensional array consisting of 4 rows (first bracket for rows) and 5 columns (second bracket for columns) for a total of 20 elements.

To initialise a two-dimensional array, it is done in the same manner of a one-dimensional. When listing elements for initialisation, the values are listed by row. The difference is that you put the initial values of each row between braces and then enclose all the rows between braces. Example:

int numbers[3][4] = {

10,20,30,40, // Values for the first row

15,25,35,45, // Values for the second row

47,48,49,50 // Values for the third row

}

Commas are required after each brace that closes off a row, except in the case of the final row. The use of the inner pair of braces is optional, but should be used for readability.

Multidimensional arrays are also zero-based; the first row and column will be [0][0]. They also do not require that the entire array is initialised; you can initialise only certain rows and columns, and remaining values will be set to 0. In this case, the inner pairs of braces are required to force the correct initialisation.

Subscripts can be used in the initialisation list in a similar manner to single-dimensional arrays: **int matrix [4] [3] = [0] [0] = 1, [1] [1] = 5, [2] [2] = 9**. Unspecified elements are set to zero by default.

Everything mentioned so far about two-dimensional arrays can be generalised to three-dimensional arrays and further. Example: **int box[10][20][30]**. Typically, to initialise and handle a three-dimensional array, you need three nested loops, four for four-dimensional arrays, and so on. Each loop iterates over one dimension. For multidimensional arrays, the process of initialisation can be extended a lot, with several levels of nested brackets.

8.5. Challenge Weather

The programme will find the total rainfall for each year, the average year rainfall, and the average rainfall for each month. The input will be a 2D array with hard-coded values for rainfall amounts for the past 5 years. The array should have 5 rows and 12 columns, with rows being years and columns being months. Rainfall amounts can be floating point numbers.

The output shall be the total rainfall for each year, the yearly average and the monthly averages, in that order.

The solution to the challenge is:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Initialise variables
    float rainfall_data[5][12] = {
        {100.0, 120.0, 135.0, 150.0, 130.0, 120.0, 60.0, 40.0, 70.0, 120.0, 115.0, 100.0},
        {110.0, 110.0, 145.0, 170.0, 110.0, 90.0, 55.0, 44.0, 73.0, 100.0, 101.0, 110.0},
        {80.0, 104.0, 176.0, 178.0, 100.0, 71.0, 34.0, 22.0, 61.0, 102.0, 94.0, 103.0},
        {105.0, 127.0, 114.0, 180.0, 104.0, 101.0, 42.0, 31.0, 79.0, 108.0, 97.0, 108.0},
        {75.0, 121.0, 129.0, 166.0, 87.0, 95.0, 46.0, 51.0, 81.0, 111.0, 111.0, 109.0}
    };
    float year_total[5] = {0.0,0.0,0.0,0.0,0.0};
    float month_average[12] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
    float year_average = 0.0;
    int i = 0,j = 0;

    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 12; j++)
        {
            year_total[i] = year_total[i] + rainfall_data[i][j];
            month_average[j] = month_average[j] + rainfall_data[i][j]/5;
        }
        year_average = year_average + year_total[i]/5;
    }

```

Figura 8.3: Solution Challenge Weather 1

```
    printf("The year total rainfall for each year are: %.1f, %.1f, %.1f, %.1f and %.1f \n",
    year_total[0], year_total[1], year_total[2], year_total[3], year_total[4]);

    printf("The yearly rainfall average is: %.1f \n", year_average);

    printf("The monthly averages for each month are the following: \n");
    printf("January: %.1f, \n", month_average[0]);
    printf("February: %.1f, \n", month_average[1]);
    printf("March: %.1f, \n", month_average[2]);
    printf("April: %.1f, \n", month_average[3]);
    printf("May: %.1f, \n", month_average[4]);
    printf("June: %.1f, \n", month_average[5]);
    printf("July: %.1f, \n", month_average[6]);
    printf("August: %.1f, \n", month_average[7]);
    printf("September: %.1f, \n", month_average[8]);
    printf("October: %.1f, \n", month_average[9]);
    printf("November: %.1f, \n", month_average[10]);
    printf("December: %.1f, \n", month_average[11]);
    return 0;
}
```

Figura 8.4: Solution Challenge Weather 2

8.6. Variable length arrays

So far, the sizes of an array have been specified using a number. The term variable does not mean that you can modify the length of the array after creating it, it will keep the same size after creation, but the size will be defined by using a variable. You can not initialise a VLA in declaration.

Example:

int n = 5;

float a[n];}

The example creates a float variable of a length that depends on the value of the variable n, that for this case is 5.

9. Functions

9.1. Basics

A function is a self-contained unit of programme code designed to accomplish a particular task. Syntax rules define its structure and how to use it. A function in C is the same as a subroutine or procedure in other programming languages.

Some functions cause an action to take place, for example: **printf()**, that causes the data to be printed on the screen. Other functions find a value for a programme to use, for example: **strlen()** tells a programmeme how long a certain string is.

Its advantages are because ease the test, debug, and maintain of a programme, tasks much more difficult in single large main functions. The tasks can be divided into several independent subtasks, reducing complexity and separating functions for each subtask. It also reduces duplication of code, saving time when writing, testing, and debugging code. If you have to do a certain task several times in a programme, you only need to write a function once and use it when needed. Besides,

it helps with readability, resulting in better organised programmes and easier to read and change. They also allow parts of the programme to be developed, tested, and debugged independently, reducing the overall development time. Functions can also be used for other different programmes. Many programmers think of functions as a black box, with information that goes in (input) and information that goes out (output). Using this black-box thinking helps to concentrate on the programme overall design rather than the details.

The arguments of the functions are in parentheses after the function name. For example, for **printf()** the first argument is a string and the following the variables or expressions to display.

The information received from a function can be received in two ways: through one of the function arguments (the output takes the value of one of the inputs, **scanf()** for example) or as a return value.

To call functions, they must be implemented first, by using libraries or using user-defined functions, functions written by the programmer.

Always use descriptive function names to make it clear what the programme does and how it is organised. If you make the functions general enough, you can reuse them in other programmes.

The main function is the starting point of the programme and all C programmes must have a main. The main function can also receive input (command line arguments) and returning data (error code) as optional features.

9.2. Defining functions

When creating a function, the function header is specified as the first line of the function definition followed by a starting curly brace. The executable code of the function is in between the starting and the ending curly braces, and is called the function body.

The function header defines the name of the function, their parameters (the number and types of values passed to the function when is called), and the type for the value that the function returns. The function body contains the statements executed when the function is called and have access to any values passed as arguments to the functions. The structure would be:

Return_type Function_name(Parameters)

{

Statements

}

The arguments of the function must be separated by commas.

The first line of a function definition tells the compiler (in order from left to right) three things about the function: the type of value it returns, its name, and the arguments it takes. Choosing meaningful function names is as important as choosing meaningful variable names, greatly affecting the programme readability.

The keyword void implies that the function does not return data.

Statements in the function body can be absent, but the braces must be present. If there are

no statements in the body of a function, the return type must be void, and the function will not do anything. Defining an empty function is often useful during testing phase, allowing to run the programme with only selected functions, then adding the detail for the function bodies step by step, testing at each stage, until the whole thing is implemented and fully tested.

The name of a function can be any legal name: not a reserved word (int, double, main...), not the same name as other functions of the programme or of a standard library function (this would prevent you from using the library function).

A legal name has the same form as that of a variable: sequence of letters and digits, the first character must be a letter, and an underline character counts as a letter. The name should be meaningful and relevant to the function.

There are three appraches to define function names with more than one word:

- Separate each word with an underline character: function_name.
- Capitalise the first letter of each word: FunctionName.
- Capitalise words after the first: functionName.

A function prototype is a statement that defines a function. It defines its name, its return type value, and the type of each parameter. Provides all the external specifications for the function. You write a prototype exactly as a function header, but adding a semicolon at the end. Example: **void printMessage(void);**. A function prototype enables the compiler to generate the appropriate instructions at each point where you call the function and to check if you are using the function correctly in each invocation.

When you include a standard header file in a programme, the header file adds the function prototypes for that library to the programme. The header file stdio.h contains function prototypes for **printf()**, among others. They usually appear at the beginning of a source file before the implementation of any functions in a header file. Allows any of the functions in the file to call any function regardless of where you have placed the implementation of the functions. Parameter names do not have to be those used in the fnuction definition.

It is good practice to always include declarations for all the functions in a programme source file, regardless of where are called, helping to keep the programmes consistent in design and preventing any error from occurring if you choose to call a function from another part of the programme.

Example: **double Average(double data:values[], size_t, count);**.

9.3. Arguments and parameters

A parameter is a variable in a function declaration and function definition/implementation. The declaration is the function prototype and the definition is the actual code.

When a function is called, the arguments are the data you pass into the functions' parameters, the actual value of a variable that gets passed to the function. Function parameters are defined within the function header, and they are placeholders for the arguments that need to be specified when the function is called.

The parameters for a function are a list of parameter names with their types, each parameter separated by a comma. The entire list of parameters is enclosed between the parentheses that follow the function name. A function can have no parameters, in which case you should put void between the parentheses.

Parameters provide the means to pass data to a function; they are data passed from the calling function to the function that is called. The names of the parameters are local to the function; they will assume the values of the arguments that are passed when the function is called. The body of the function should use these parameters in its implementation. A function body may have additional locally defined variables that are needed for the function implementation.

When passing an array as an argument to the function, you must also pass an additional argument specifying the size of the array. The function has no means of knowing how many elements there are in the array.

Example: When **printf()** is called, you always supply one or more values as arguments, the first value being the format string and the remaining any variables to display.

Parameters greatly increase the usefulness and flexibility of a function, since it allows to operate in different ways according to the input data that receive.

It is a good idea to add comments before each of the function definitions to explain what the function does and how the arguments are to be used.

Example: **void multiplyTwoNumbers(int x, int y)**

int result = x*y

This is a flexible function that receives the two arguments that you want as the parameters x and y and provides different results for each combination of the parameters provided.

9.4. Returning data from functions

You might now always want to have the results of your calculations displayed. Functions can return data using specific syntax. The first thing to write in a function is the return type, which specifies the type of the value returned by the function.

You can specify any legal type of value to be returned, including enumeration types and pointers. The return type can also be of type void, which means that no value is returned. Using void helps the readability of a function. The return statement provides the means of exiting from a function.

The return statement provides the means of exiting from a function. Using **return;** implies that the return type is void and does not return a value; it only exits the function. The more general form of the return statement is: **return expression;**, where the expression must be of the return type specified of the function, that must be of a different type from void.

The value that is returned to the calling programme is the value that results when expression is evaluated and must be of the specified type of the function.

A function that has statements in the function body but does not return a value must have the return type as void. An error message will appear if you compile a programme that contains a

function with a void return type that tries to return a value.

A function that does not have a void return type must return a value of the specified return type. An error message will appear if return type is different from the specified one.

If expression results in a value that is a different type from the return type in the function header, the compiler will insert a conversion from the type of expression to the one required. If conversion is not possible, the compiler will produce an error message.

There can be more than one return statement in a function. Each return statement must supply a value that is convertible to the type specified in the function header for the return value.

9.4.1. Invoking a function

You call a function using the function name followed by the argument to the function between parentheses. When you call the function, the values of the arguments that you specify in the call will be assigned to the parameters in the function.

When the function executes, the computation proceeds using the values you supplied as arguments. The arguments you specify when you call a function should agree in type, number, and sequence with the parameters in the function header.

If the function is used as the right side of an assignment statement, the return value supplied by the function will be substituted for the function. Example: **int x myFunctionCall();** will assign the return value of myFunctionCall (that must be an integer) to the variable x.

The calling function does not have to recognise or process the value returned from a called function. It is up to you how to use any values returned from function calls.

9.5. Local and Global Variables

Local variables are variables defined inside a function. They are automatically created each time the function is called. Their values are local to the function and cannot be accessed from outside the function. If an initial value is given to a variable inside a function, that initial value is assigned to the variable each time the fnuction is called.

The auto keyword can be used to be more precise in defining local variables, but it is not necessary as the compiler adds this by default.

Local variables are also applicable to any code where the variable is created in a block (loops, if statements).

Global variables are the opposite; they can be accessed by any function in the programme. A global variable has the lifetime of the programme. Global variables are declared outside of any function and do not belong to any particular function. Any function in the programme can change the value of a global variable.

If there is a local variable declared in a function with the same name as a global variable, then, within that function, the local variable will mask the global variable. Therefore, for this function, the global variable is not accessible and it is prevented to access it normally.

An example of creation of local ang global variables and where can I use them can be seen in the following Figure:

Figura 9.1: Example of local ang global variables

In general, global variables are bad and should be avoided. They promote coupling between functions (dependencies), make it hard to find the location of a bug in a programme, and also make it hard to fix it. Instead, use parameters in functions; if there is a lot of data, use a struct.

9.6. Challenge Write functions

The programme shall have three functions. One to find the greatest common divisor of two non-negative integer values and return the result (takes two ints as parameters and returns an int). Another to calculate the absolute value of a number (takes as parameter a float and return a float) and test it with ints and floats. And a final one to comupte the square root of a number. If anegative argument is passed, a message will be displayed an -1.0 should be returned. You should use the function of absolute value as implemented in the previous step.

The function of the greatest common divisor is:

```c
int MCD(int x, int y)
{
    int z = 1; // Maximum common divisor
    int remainder_x, remainder_y; // Remainders of divisions
    int i; // Loop variable

    if (x <=0 || y<=0)
    {
        printf("One of the numbers is negative or zero, MCD not calculated right \n");
        z = 0;
        return z;
    }

    if (x < y)
    {
        for (i = 2; i <= x; i++)
        {
            // Calculate the remainders for each value of i
            remainder_x = x%i;
            remainder_y = y%i;
            if (remainder_x == 0 && remainder_y == 0)
                {
                    // If both remainders are equal, i is the new MCD
                    z = i;
                }
        }
    }
```

Figura 9.2: Greatest common divisor function 1

```c
    else if (x > y)
    {
        for (i = 2; i <= y; i++)
        {
            // Calculate the remainders for each value of i
            remainder_x = x%i;
            remainder_y = y%i;
            if (remainder_x == 0 && remainder_y == 0)
                {
                    // If both remainders are equal, i is the new MCD
                    z = i;
                }
        }
    }
    else
        // If both numbers are equal, the MCD is their own value
        z = x;

    return z;
}
```

Figura 9.3: Greatest common divisor function 2

The function for the absolute value is:

```
80  float AbsoluteValue(float x)
81  {
82      float abs;
83
84      if (x < 0)
85      {
86          abs = -x;
87      }
88      else
89      {
90          abs = x;
91      }
92      return abs;
93  }
```

Figura 9.4: Absolute value function

The function for the square root is:

```
95  float SquareRoot(float x)
96  {
97      const float epsilon = .000001;
98      float guess = 1.0;
99
100     if (x < 0)
101     {
102         printf("Negative argument to SquareRoot. \n");
103         return -1.0;
104     }
105
106     while (AbsoluteValue(guess*guess -x) >= epsilon)
107     {
108         guess = (x / guess + guess) / 2.0;
109     }
110
111     return guess;
112
113 }
114
```

Figura 9.5: Square root function

9.7. Challenge Tic Tac Toe

The programme should be a tic-tac-toe. It is a game played on 3x3 grid, the game played by two players, who take turns. You must create an array to represent the board. Can be of type char and consist of 10 elements (do not use zero). Each element represents a coordinate on the board that the user can select.

Some functions that you should create are: checkforWin (checks if a player has won the game or if it is a draw), drawboard (redraws the board for each player turn) and markBoard(sets the char array with a selection and check for an invalid selection).

The solution is in the Workspace of the C course (too long to be put in here).

10. Character strings

10.1. Overview

A string is a word enclosure between quotation marks .

The data type char contains a single character. To assign a single character to a char variable, the character is enclosed within a pair of single quotation marks ". There is a distinction between single quotation marks " and double quotation marks . They are used to create two different types of constants in C.

Every time a message has been displayed using the **printf()** function, you have defined the message as a constant string. To display a quotation mark in a string, it must be preceded by \. Equally, to print \, you have to put two consecutive \.

The string in memory occupies one bit per character, with the spaces being empty bits and ended with a null character \0, a special character added to the end of each string to mark where it ends. A string is always terminated by a null character, so in reality the length of a string is always one greater than the number of characters in the string.

The null character is not the same as NULL, a symbol that represents a memory address that does not reference anything.

10.2. Defining a string

C has no special variable for strings. This means that there are no special operators in the language for processing strings. The standard library provides an extensive range of functions for handling strings.

Strings in C are stored in an array of type char. Characters in a string are stored in adjacent memory cells, one character per cell. To declare a string in C, simply use the char type and the brackets to indicate the size. Example: **char myString[20]**. This variable can accommodate a string that contains up to 19 characters (one element is for the termination character).

When you specify the size of an array that you intend to use to store a string, it must be at least one greater than the number of characters that you want to store, since the compiler automatically adds the null character at the end of every string constant.

You can initialise a string variable when you declare it. Example: **char word[] = 'H','e','l','l','o'** initialises a char string of 6 characters (5 plus the null character that will be later added by the compiler) and the statement reserves space in memory for 6 characters.

You can specify the size of a string explicitly, but it is important to consider the extra null

character. If the size is too small, the compiler does not put a null character at the end and does not complain about it, which may cause errors, so it is better not to declare the size of the array if possible and let the compiler decide. Also, you can initialise only a part of the char array, and the rest will be empty.

Since you cannot assign arrays in C, you cannot assign a string either. The following example is an error:

char s[100];

s = "hello";

You are preforming an assingment operation and you cannot assign one array of characters to another array of characters like this. You have to use a special function **strncpy()** to assign a value to a char array after it has been declared or initialised. The other option is to assign each character individually, but that could take a long time for large arrays.

When you want to refer to a string stored in an array, you just use the array name by itself. To display a string as output using the **printf()** function, you do the following: **printf("\n The message is: %s", message)** The %s is the format specifier for outputting a null-terminated string. The function assumes that the string to display has a null character to end it, so it may cause problems if the definition is wrong and the null character has not been assigned.

To input a string through the keyborad, use the **scanf()** function using the %s format specifier. Example: **scanf(" %s", input);**. Note that there is no need to use the & on a string. This function will only read until the first space appears; to read the complete strings there are other functions to use.

You cannot directly test two strings to see if they are equal with a statement such as **if (string1 == string2)**. The equality operator can only be applied to simple variable types, so it does not work on structures and arrays.

To determine if two strings are equal, you must explicitly compare the two character strings character by character. There is an easier way, which will be discussed later by using the **strcmp()** function.

10.3. Constant strings

There are strings that cannot be modified. For example, the name pi for the π number instead of including the value everytime.

The preprocessor lets you define constants: **#define TAXRATE 0.015**. When you programme is compiled, the value 0.015 will be substituted wherever you have used TAXRATE, which is called compile-time substitution. A defined name is not a variable, you cannot assign a value to it. The define statement has a spaceial syntax, there is no equal sign, and there is no semicolon at the end.

The deinfe statements can appear anywhere in a programme. There is no local define. Most programmers define them at the beginning of the programme or inside an include file. The define statements make the programmes more portable. It also can be used for character and string constants. Example: **#define BEEP '\a**

C90 added a second way to create symbolic constants, the keyword const. Example: **const int MONTHS = 12;**, MONTHS is a symbolic constant for 12, it can be displayed and used in calculations but cannot be changed (it is not a variable).

const is more flexible than using define, it lets to declare a type and allows better control over which part of a programme can use the constant.

Another way to create the symbolic constants are the enums variables.

Initialising a char array and declaring it as a constant is a good way of handling standard messages. Example: **const char message[] = "The end of the world is here"**. Because you declare the message as const, it is protected from being modified explicitly within the programme, any attempt to do so will result in an error message from the compiler. This technique for defining standard messages is useful if they are used in many places within a programme, preventing accidental modification of such constant in other parts of the programme.

10.4. Challenge Understanding char arrays

You must write a function to count the number of characters in a string (length). Cannot use standard library functions, the function should take a character arrays as a parameter and should return an int (the lenght).

Also, you must write a function to concatenate two character strings, without using standard functions. The function should take 3 parameters: char result[], const char str1[] and const char str2[]. It can return void.

Finally, you must write a function that determines if two strings are equal. Cannot use strcmp library function and function should take two const char arrays as aprameters and return a Boolean of true if they are equal and false otherwise.

The function to count the number of characters of a string is:

```
36  int length(const char a[])
37  {
38      int i = 0;
39      while (a[i] != '\0') // Do until reaching the null character
40      {
41          i = i + 1;
42      }
43      return i;
44  }
45
```

Figura 10.1: Length function

The function to concatenate two strings is

```c
bool EqualStrings(const char string1[], const char string2[])
{
    int length1, length2, i;
    char index1, index2;
    bool Equal;

    length1 = length(string1);
    length2 = length(string2);

    if (length1 == length2)
    {
        for (i = 0; i <= length1; i++)
        {
            index1 = string1[i];
            index2 = string2[i];

            // Compare each string and continues if they are equal
            if (index1 == index2)
            {
                Equal = 1;
            }
            else // If one index is different, break the for loop
            {
                Equal = 0;
                break;
            }
```

Figura 10.3: String comparison function 1

```c
            }
        }
    }
    else // If lengths are not the same, the strings are not equal
    {
        Equal = 0;
    }

    return Equal;
}
```

Figura 10.4: String comparison function 2

```c
void concat(char result[], const char string1[], const char string2[])
{
    int length1, length2, i, j, index;

    length1 = length(string1);
    length2 = length(string2);

    for (i = 0; i<length1; i++)
        {
            result[i] = string1[i];
        }
    for (j = 0; j<length2; j++)
    {
        index = j + length1;
        result[index] = string2[j];
    }

    // Add null character
    result[i+j] = '\0';

    printf(" \nThe combination between \"%s\" and \"%s\" is \"%s\" ", string1, string2, result);
}
```

Figura 10.2: Concatenate function

The function to compare two strings is:

10.5. Common string functions

C provides many functions specifically designed to work with string. The following are:

- **strlen():** It gets the length of a string.
- **strcpy()** and **strncpy():** Copy one character string to another.
- **strcat()** and **strncat():** Combine two characters strings together (concatenation).
- **strcmp()** and **strncmp():** Determine if two character strings are equal.

The C library supplies these string.handling function prototypes in the string.h header file.

10.5.1. strlen()

It finds the length of a string and returns it as a size_t. The string.h header file is required. This function does change the string, so the function header does not use const in declaring the fomal parameter string. Example: **size_t = strlen(string1);**

10.5.2. strcpy() and strncpy()

You can use these functions to copy a string to an existing string. This function is useful since you cannot assign strings directly in C. Is the equivalent of the assignment operator for strings of characters. The first argument of the function is the string od destination and the second the string that you want to copy. Example: **strcpy(destination, "This is the source"** will copy "This is the sourceïnto the array destination.

strcpy() does not check if the string copied fits in the destination string, so a safer way to copy is **strncpy()** that fnuctions exactly the same but adds a thrid argument, the maximum number of characters to copy. Example: **strcpy(destination, "This is the source", 10** will copy the first 10 characters of the array into destination.

10.5.3. strcat() and strncat()

This functions take two strings for arguments: a copy of the second string is tacked onto the end of the first, this combined version becomes the new first string and the second sring is not altered. It returns the value of the first argument. Example: **strcpy(dest, src))** returns in dest the combination of the previous dest and src.

As happened with **strcpy()**, **strcat()** does not check whether the second string will fit in the first array. If you fail to allocate enough space for the first array, you will run into problems as excess characters overflow into adjacent memory locations.

strncat() takes a third argument indication the maximum number of characters to add. For example: **strcpy(dest, src, 15))** returns in dest the combination of the previous dest and only the 15 additional characters (or the number of characters until the null character, what comes first) of src, so it is safer.

10.5.4. strcmp() and strncmp()

The strings cannot be compared using ==, they are compared using these functions. It does not compare arrays, so it can be used to compare strings stored in arrays of different sizes, and it does not compare characters.

strcmp() returns 0 if the two string arguments are the same and nonzero, it returns <0 to indicate that str1 is less than str2 and it returns >0 to indicate that str1 is more than str2.

strcmp() compares strings until it finds corresponding characters that differ, which could take the search to the end of one of the strings. **strncmp()** compares the strings until they differ or until it has compared a number of characters specified by a third argument. This is useful, for example, when you want to search for things that begin for a certain words. If you want to search things that begin with astro, you could limit the search to the first five characters.

10.6. Challenge Common string functions

Write a programme that displays a string in reverse order. It should read input from the keyboard and use the strlen string function. Another function should sort the strings in an array using a bubble sort. It need to use the strcmp and strcpy functions.

The solution is the following:

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

int main(int argc, char **argv)
{
    // Reverse a string
    char string[] = "Daniel";
    char reverse[100];
    int i;

    long length = strlen(string);

    for(i = 0; i < length; i++)
    {
        reverse[length - i] = string[i];
    }

    printf("The reverse of \"%s\" is \"%s\"", string, reverse);

    // Order some strings
    int number, j, k, l;
    char arrays[25][50], temp[25];

    printf("\n Introduce the number of strings to order:");
    scanf("%i", &number);

    printf("\n Introduce the %i strings to order (maximum 50 characters):", number);
```

Figura 10.5: Challenge Common string functions 1

10.7. Searching, tokenising and analysing strings

All the functions seen in this Subsection are included in the header file string.h. Searching a string is finding a single character or a substring in a string. Functions: **strchr()** and **strstr()**.

```
    for (j = 0; j<number; j++)
    {
        printf("\n Array %i: ", j);
        scanf("%s", arrays[j]);
    }

    for (k=1; k<=number;k++)
    {
        for (j=0;j<=number-k;j++)
        {
            if(strcmp(arrays[j], arrays[j+1])>0)
            {
            strncpy(temp, arrays[j], sizeof(temp)-1);
            strncpy(arrays[j], arrays[j+1], sizeof(arrays[j])-1);
            strncpy(arrays[j+1], temp, sizeof(arrays[j+1])-1);
            }
        }
    }

    printf("The sorted strings are:");

    for (l = 0; l<number;l++)
    {
        printf("%s \n", arrays[l]);
    }

    return 0;
}
```

Figura 10.6: Challenge Common string functions 2

Tokenising a string is breaking a sequence of characters bounded by delimiters (space, comma, period) into different words. Function: **strtok()**. Other functions are for analysing strings: **islower, issuper, isalpha...**

The pointer will be seen in depth in the next Section, but a quick peek may be useful to understand some of these functions. C provides a remarkably useful type of variable called a pointer, a variable that stores an address. Its value is the address of another location in memory that can contain a value. The **scanf()** function uses addresses by the symbol & before the variable, which marks that it is a pointer.

Example: **int Number = 25;**

int *pNumber = &Number;

Above, we declare a variable Number with the value of 25 and a pointer, pNumber, which contains the address of Number. The asterisk is used to declare a pointer. To get the value of the variable pNumber, you can use the asterisk to dereference the pointer: ***pNumber = 25;**. * is the dereference operator, and its effect is to access the data stores at the address specified by a pointer.

The value of &Number is the address where NUmber is located and it is used to initialise pNumber in the second statement.

Many of the string functions return pointers.

10.7.1. Searching a string for a character

The function **strchr()** searches a given string for a specified character. The first argument is the string to be searched (which will be the address of a char array) and the second is the character that you are looking for. The function will search the string starting at the beginning and return a pointer to the first position in the string where the character is found, the address of this position in memory. Is of type char*, described as pointer to char.

To store the returned value, you must create a variable that can store the address of a character. If the character is not found, the function returns a special value NULL, hte equivalento of 0 for a pointer and represents a pointer that does not point to anything.

Example:

char str[] = "The quick brown fox"; The string to be searched

char ch = 'q'; The character we are looking for

char *pGot_char = NULL; Pointer initialised to NULL

pGot_char = strchr(str,ch); Stores address where ch is found.

The first argument is the address of the first location to be searched, the second the character sought (it expects a type int, so it converts the type char to int) and the output will point to the value "The quick brown fox".

The **strstr()** function is probably the most useful of all searching functions. It searches one string for the first occurrence of a substring and returns a pointer for the position, if no match, returns NULL. The first argument is the string that is to be searched, and the second the substring you are looking for.

Example:

char text[] = .Every dog has its day"; The string to be searched

char word[] = "dog"; The substring we are looking for

char *pFound = NULL; Pointer initialised to NULL

pFound = strstr(text,word); Stores address where word is found.

Since the string dog appears starting at the seventh character of text, pFound will be set to the address text+6 ("dog has its day"). The search is case-sensitive, so "Dog"will not be found.

10.7.2. Tokenizing a string

A token is a sequence of characters within a string bounded by delimiters. A delimiter can be anything, but should be unique to the string: spaces, commas, periods, etc. Tokenising is breaking a sentence into words, and to do that the **strtok()** function is used. It requires two arguments: a string to be tokenised and a string containing all possible delimiter characters.

Example:

char str[80] = "Hello how are you - my name is - Daniel";

const char s[2] = " ";

char *token;

token = strtok(str,s);

This will result into the first word: "Hello how are you". If you enter it in a loop while until finding NULL, it will tokenise the entire string.

10.7.3. Analysing strings

These are functions to obtain a characteristic of a character. The argument to each of these functions is the character to be tested. They return a nonzero value of type int if the character is within the set that is being tested for. They convert to true and false, so you can use them as Boolean values.

- **islower():** Tests for lowercase letter.
- **issupper():** Tests for uppercase letter.
- **isalpha():** Tests for lowercase or uppercase letter.
- **isalnum():** Tests for lowercase or uppercase letter or a digit.
- **iscntrl():** Tests for control character.
- **isprint():** Tests for any printing character including space.
- **isgraph():** Tests for any printing character excluding space.
- **isdigit():** Tests for a decimal digit: 0 to 9.
- **isxdigit():** Tests for a hexadecimal digit: 0 to 9, A to F, a to f.
- **isblank():** Tests for standard blank characters: space, \t.
- **isspace():** Tests for whitespace characters: space, \t, \n, \v, \r, \f.
- **ispunct():** Test for printing character for which isspace and isalnum return false.

10.8. Converting strings

It is very common to convert character case to all uppercase or all lowercase. **toupper()** function converts from lowercase to uppercase and **toloer()** does the contrary. Both functions return either the converted character or the same character for characters that are already in the correct case or are not convertible such as punctuation characters. You have to use them in a loop to analyse and convert each character until reaching the null character.

You can use them in combination with **strstr()** to search in a string without considering if the word is in uppercase or lowercase.

To convert strings to numbers, the following functions for the stdlib.h header file are used:

- **atof:** A value of type double that is produced from the string argument. Infinity as a double value is recognised from the strings INF or INFINITY and not a number from the string NAN, all of them in lowercase or uppercase.
- **atoi:** A value of type int that is produced from the string argument.
- **atol:** A value of type long that is produced from the string argument.
- **atoll:** A value of type long long that is produced from the string argument.
- **strtod:** A value of type double is produced from the initial part of the string specified by the first argument. The second argument is a point to a variable, ptr, of type char* in which the function will store the address of the first character following the substring that was converted to the double value. If no string was found that could be converted to type double, the variable ptr will contain the address passed as the first argument.
- **strtof():** Equal to **strtod**, but produces a value of type float.
- **strtold():** Equal to **strtod**, but produces a value of type long double.

11. Debugging

11.1. Configuring the Debugger in CodeLite

To configure the settings of the debugger: Settings -> GDB settings.

If you want to do so for a concrete project: right-click on the project, settings, debugger and in select debugger path include the cygwin64/bin/gdb.exe.

If that does not work (as it was my case) it is because a general bug of the compiler, so the solution is to install the MinGW debugger. For that, go to winlib.com to the downloads page and download the .zip wherever you want. I extracted it on the disk C. There is a folder named mingw64 that has a bin folder with a gdb executable that will be the debugger used.

To debug a programme, you go to Debugger and select Start/Continue Debugger. The programme will run and stop in the debugging points that you select. To select a line to set a debug point, you can click at the right of the line number or right click in the line and select Add breakpoint.

You always have to set the path to the debugger in every new project that you create in CodeLite. In the tab that is just top of the projects of the workspace, the option debug must be selected.

11.2. Debugging

Debugging is the process of finding and fixing errors in a programme, usually logic errors, but can also include compiler and/or syntax errors. For syntax errors, debugging is understand what the compiler is telling you. You must always focus on fixing the first problem of the programme because the following errors may be caused by the previous ones.

Debugging can range in complexity from fixing simple errors to collecting large amounts of data for analysis. The ability to debug is an essential skill (problem solving) that can save you tremendous amounts of time and money.

The maintenance phase is the most expensive phase of the software life cycle, is the phase in which new features are added, and new bugs are found and fixed.

Bugs are unavoidable when programming.

The common problems related to debugging are logic errors (the programme is well written but there is something that makes your code now work as intended), syntax errors (the programme is not well written: semicolon missing for example), memory corruption (there is no address defined for a pointer for example), performance/scalability (the programme works well, but the scale of data that are running makes it run very slow), lack of cohesion (there is features not used or a function or main programme does many things, being better to use functions), and tight coupling (many dependencies among the functions of the code).

The debugging process starts with understanding the problem (sit down with a tester, understand requirements, etc.). Once found, you must reproduce the problem, which sometimes is difficult, as it can be intermittent or only happen in rare circumstances. Parallel processes or threading problems also makes harder to debug the programme. After that, your target must be

simplify the problem, by divide and conquer strategies or isolating the source: remove parts of the original test case, comment out code, turn a large programme into a lot of small programmes...

After that, the origin of the problem must be identified in the code, using debugging tools if necessary. There you can check what is happening in the code while running, watching the evolution of the variables, and detecting anomalies. Finally, you must solve the problem, which becomes easier with experience and practice. Sometimes, it includes a redesign or refactoring of the code. And finally, a lot of tests to check that the programme works for all possible cases are done.

Some useful techniques and tools are, for example, tracing and using print statements, giving the output values of variables at certain points of a programme, which shows the execution flow and can help isolate the error. Debuggers are other tools adequate for debugging, as has already been commented on. Other option is the use a log file. Finally, there are monitoring softwares that run-time analysis of memory usage, network traffic, thread, and object information.

The exception handling helps a lot to identify catastrophic errors, but C has no features related to this. Static analysers analyse the source code for a specific set of known problems. They are semantic checkers; they do not analyse syntax. They can detect things like uninitialised variables, memory leaks, unreachable code, deadlocks, or race conditions. Test suites run a set of comprehensive end-to-end system tests. Finally, analysing the call stack and memory dump helps to understand what and where happened when the programme crashed.

Preventing errors is better than debugging them. It is useful to write high-quality code, following good design principles and good programming practices. Unit tests are automatically executed when compiling. They help to avoid regression and find errors in new code before it is delivered. This is called Test Driven Development; you write the code in order to pass the tests. Besides, providing good documentation and proper planning reduces the number of errors committed. Finally, working in steps and constantly testing after each step helps reduce the errors.

11.3. Understanding the Call Stack

A stack trace or call stack is generated whenever your application crashes due to a fatal error. Shows a list of the function calls that lead to the error: includes the filenames and line numbers of the code that cause the error. At the top of the stacks there is the last call that caused the error, and at the bottom there is the first call that started the chain of calls that caused the error. You need to find the call in your application that causes the crash. A programme can also dump the stack trace. You can access the call stack via the CodeLite menu when using the debugger.

After setting a breakpoint and run the debugger, when the programme stops at the right bottom the debugging menu appears. There, one of the options is Call Stack. There, there is a series of functions, depending on where the programme has stopped, the file where are located, the line in which are stopped and the memory address related. It shows the order of the functions that are being called in the line in which you are stopped.

11.4. CodeLite Debugger

Before using the debugger, the project settings -> Debugger, and the compiler setting must be configured correctly.

A breakpoint is something you set in the programme to temporarily stop its execution in a certain line and access the information that the programme has at that point: variable values, for example. To use the debugger, there must be at least one breakpoint set.

The debugger options are the following:

- **Locals:** Local variables at the breakpoint location.
- **Watches:** They cna be put on variables and they will pop up if that variable changes its value.
- **Output:** To print information while the programme is at the breakpoint.
- **Threads:** To look at the threads followed by the programme.
- **Call Stack:** See above.
- **Memory:** To see information about the memory locations.
- **Breakpoints:** To see the breakpoints of the programme.

The most important ones are local and breakpoints, they will be the most used.

Also, at the top of the programme, there is the debugger toolbar. The play is for continuing the programme, the stop is for stop the execution, the two vertical lines are for pausing the debugging, the two cycled lines are for restart the debug, the > is for show the current line, the right arrow is for step into a function in that line, the down arrow is for pass to the next line, and the up arrow is for stepping out of a function to the previous call.

11.5. Common C mistakes

11.5.1. Missplacing a semicolon

For example:

if (j == 100);

j = 0;

There are usually committed in loops and ifs, putting a semicolon after the if, while, for, etc. In the above example, the value of j will be always set to 0 due to the misplaced semicolon after the closing parenthesis. Semicolon is syntactically valid (it represents the null statement) and therefore no error is produced by the compiler.

11.5.2. Confusing the = operator with the == operator

They are usually made inside an if, while or do statement. This is perfectly valid, assigning the value at the right to the variable at the left, but it will not do what you want, it will always be a true condition since you are assigning it. Example:

if (a = 2)

printf(”Your turn \n)

It will always print the statement, since you are assingning the value 2 to the variable a.

11.5.3. Omitting prototype declarations

result = squareRoot(2);

If squareRoot is defined later in the programme or in another file and is not explicitly declared otherwise, the compiler assumes that functions return an int. Therefore, it is always safest to include a prototype declaration for all functions that you call (explicitly yourself or implicitly by including the correct header file in your programme.

11.5.4. Failing to include the header file that includes the definition of a function

double answer = sqrt(value1);

If this programme does not include the math.h header file, it will generate an error since sqrt() is undefined.

11.5.5. Confusing a character constant and a character string

text = ’a’;

assigns a single character to text, while

text = .a”;

assigns a pointer to the character string a to text.

In the first case, text is normally declared to be a char variable, while in the second it should be declared to be of type pointer to char.

11.5.6. Using the wrong bound for an array

int a[100], i, sum = 0;

for (i = 1, i<=100, ++i);

sum += a[i];

This code will return an error, since the valid subscripts of an array range from 0 to the number of elements minus one. The preceding loop is incorrect because the last valid subscript of a is 99 and not 100. Also, probably intended to start with the first element of the array, therefore i should be initially set to 0. When i = 100, there will be bound error.

A similar mistake is forgetting to reserve an extra location in an array for the terminating null character of a string. When declaring character arrays, they need to be large enough to contain the

terminating null character. The character string hello would require six locations in a character array if you wanted to store a null at the end.

11.5.7. Confusing the operator -> with the operator . when referencing structure members

The operator . is used for structure variables and -> is used for structure pointer variables.

11.5.8. Omitting the ampersand before nonpointer variables in a scanf() call

int number;

scanf(" %i", number);

All arguments that appear after the format string in a **scanf()** call must be pointers. The ampersand # indicates that the input is the pointer that points to the variable that follows, in the case of the above example, to number.

11.5.9. Using a pointer variable before it is initialised

char *char_pointer;

***char_pointer = 'X';**

You can only apply the indirection operation * to a pointer variable after setting the variable point somewhere. char_pointer is never set pointing to anything, so the assignment is not meaningful.

11.5.10. Omitting the break statement at the end of a case in a switch

If a break is not included at the end of a case, then execution continues into the next case.

11.5.11. Inserting a semicolon at the end of a preprocessor definition

Usually happens because it becomes a matter of habit to end all statements with semicolons.

#define END_OF_DATA 999;

leads to a syntax error if used in an expression such as

if value == END_OF_DATA

because the compiler will se this statement after preprocessing

if value == 999;

11.5.12. Omitting a closing parenthesis or quatoation mark in an statement

The use of embedded parentheses to separate each portion of an equation makes for a more readable line of code. However, there is always the possibility of missing a closing parenthesis or adding one too many. Closing a quotation mark for a string may also cause errors in functions such as **printf()**.

Both of these will generate a compiler error, but sometimes the error will be identified as coming on a different line, depending on whether the compiler uses a parenthesis or quotation mark on a subsequent line to complete the expression which moves the missing character to a place later in the programme.

11.6. Understanding compiler errors

Sometimes the compiler errors and warning are deceiving or you cannot understand what is trying to tell you. You need to understand compiler errors in order to fix them.

The compiler makes decisions about how to translate the code that the programmer has not written in the code. Is convenient because the programmes can be written more succinctly (only expert programmers take advantage of this feature). You should use an option for the compiler to notify all cases where there are implicit decisions, this option is -Wall, which is activated by default in CodeLine.

The compiler shows two types of problems:

- **Errors:** A condition that prevents the creation of a final program. No executable is obtained until the errors have been corrected. The first errors are the most reliable because the translation is finished but there are some errors that may derive from the previous ones. Fix the first errors and then compile again to see if other errors also disappear.
- **Warning:** Messages that the compiler shows about special situations in which an anomaly has been detected but fatal errors have not been found. The final executable programme may be obtained with any number of warnings, but is recommendable to eliminate all to not cause any problems in the future.

Compile always under the -Wall option and do not consider the programme correct until all warnings have been eliminated.

11.6.1. Variable undeclared (first use in the function)

This is one of the most common and easier warnings to detect. The symbol shown at the beginning of the message is used, but has not been declared.

11.6.2. Implicit declaration of function

This warning appears when the compiler finds a function used in the code, but no previous information has been given about it. You need to declare a function prototype.

11.6.3. Control reaches end of non-void function

This warning appears when a function has been defined as returning a result but no return statement has been included to return this result. Either the function is incorrectly defined or the statement is missing.

11.6.4. Unused variable

This warning is printed by the compiler when a variable is declared but is not used in the code. The message disappears if the declaration is removed.

11.6.5. Undefined reference to

Appears when there is a function invoked in the code that has not been defined anywhere. The compiler tells us that there is a reference to a function with no definition. Check which function is missing and make sure that its definition is compiled.

11.6.6. Conflicting types for

Two definitions of a function prototype have been found. One is the prototype of the function, and the other is the definition with the function body. The information in both places is not identical and a conflict has been detected. The compiler shows in which line the conflict appears and the previous definition that caused the contradiction.

11.6.7. Runtime errors

The execution of C programmes may terminate abruptly (crash) when a run-time error is detected. C programmes only print the succinct message Segmentation fault. Usually results in a core file, depending on the signal that has been thrown). You can analyse the core file and the call stack to find the cause.

11.6.8. Others

If you do not know the cause of an error and you cannot understand the compiler message, a useful way to know the cause of an error is to Google it.

12. Pointer basics

12.1. Overview

Pointers are very similar to the concept of indirection in real life. Suppose that you need to buy a new ink cartridge for your printer. All purchases are handled by the purchasing department,

so you call Joe in purchasing and ask him to order it for you. Joe then calls the local supply store to order the cartridge. You are not ordering the cartridge directly for the supply store yourself (indirection).

In programming, indirection is the ability to reference something using a name, reference, or container instead of the value itself. As example is variable names, you do not use the value of the memory address directly, you call a variable that references it.

The most common form of indirection is the act of manipulating a value through its memory address. A pointer provides an indirect means of accessing the value of a particular data item. A pointer is a variable whose value is a memory address. Its value is the address of another location in memory that can contain a value.

There are good reasons why it makes sense to use pointers in C. They are one of the most powerful tools available in the C language, but also one of the most confusing concepts of C. The compiler must know the type of data stored in the variable to which it points. You need to know how much memory is occupied or how to handle the contents of the memory to which it points. Each pointer will be associated with a specific type of variable. It can be used only to point to a variable of that type: a pointer of type pointer to int can point only to variables of type int, and a pointer of type pointer to float can point only to variables of type float.

Why use pointers in C? Firstly, accessing data by means of only variables is very limiting, with pointers you can access any location (you can treat any position of memory as a variable) and perform arithmetic with pointers. Pointers in C make it easier to use arrays and strings. Besides, pointers allow to refer to the same space in memory form multiple locations. This means that you can update memory in one location and the change can be seen from another location in your programme, which can also save space by being able to share components in your data structures. Furthermore, pointers allow functions to modify data passed to them as variables, which is called pass by reference: passing arguments to function in a way they can be changed by function, this change being reflected outside the function, which means that the use of global variables can be eliminated. Finally, pointers can also be used to optimise a programme to run faster or use less memory.

Pointers allow to get multiple values from the function. A function can return only one value, but by passing arguments as pointers we can get more than one value from the pointer. With pointers, dynamic memory can be created according to the programme use, saving memory from static (compile time) declarations. Pointers allow to design and develop complex data structures like a stack, queue or linked list. Pointers provide direct memory access, which is very efficient.

12.2. Defining pointers

A pointer is a variable that points to a memory address or contains a memory address as its value.

Pointers are not declared as normal variables. It is not enough to say that a variable is a pointer, you have to specify the kind of variable to which the pointer points. Different types of variables take up different amounts of storage. Some pointer operations require knowledge of that storage size.

You need to use an asterisk to define a variable:

int *pnumber;

declares a pointer that points to a memory address in which there is a variable of type int. The type of the variable with the name pnumber is int*, and it can store the address of any variable of type int. **char *pc** defines a pointer to a character variable, **float *pf, *pg** defines two pointers to float variables.

You can leave a space between the asterisk and the pointer name but is optional. Programmers usually use the space in a declaration and omit it when dereferencing a variable.

The value of a pointer is an address, represented internally as aun unsigned integer on most systems. You should not think of a pointer as an integer type, things you can do with integers you cannot do with pointers, and vice versa, for example, you can multiply one integer by another, bout you cannot multiply one pointer by another.

A pointer is a new type of variable, not an integer type. The format specifier for pointers is %p.

The previous declarations creates the variable but does not initialise it. A pointer is dangerous when not initialised, so you should always initialise a pointer when you declare it.

You can initialise a pointer so that it does not point to anything:

int *pnumber = NULL;

NULL is a constant defined in the standard library. Is the equivalent of zero for a pointer, is a value that is guaranteed not to point to any location in memory. This means that it implicitly prevents accidental memory overwriting by using a pointer that does not point to anything specific. To use NULL, you should add an include directive for stddef.h to your source file.

If you want to initialise your variable with the address of a variable you have already declared, you use the operator &, just like in the **scanf()** function (logical, since this function uses pointers as arguments). Example:

int number = 99;

int *pnumber = &number;

This makes that the initial value of pnumber is the address of the variable number. Declaration of number must precede the declaration of the pointer that stores its address. The compiler must already have an allocated space and thus an address for number to use it to initialise pnumber.

Be careful, there is nothing special about the declaration of a pointer, you can declare regular variables and pointers in the same statement. Example:

int *p, q;

The above declares a pointer p of type *int and a variable q of type int. A common mistake is to think that both p and q are pointers. It is a good idea to use names beginning with p as pointer names, to be easily identified.

12.3. Accessing pointers

Accessing a pointer is to access the value of the address that the pointer contains. For that, you use the indirection operator *, also referred to as the dereference operator because you use it

to dereference a pointer.

The pointer variable contains the address of the variable number. For example:

int number = 15;

int *pointer = &number;

int result = 0;

result = *pointer + 5;

*pointer would evaluate to the value stored at the address contained in the pointer, 15 so result would be 15 + 5 = 20.

The * is also the symbol for multiplication. Depending on whether the asterisk appears, the compiler will interpret it as an indirection operator, as a multiplication sign, or as part of a type specification. The context determines what it means in any instance.

To output the address of a variable, use the output format specifier %p. This will output a pointer value as a memory address in hexadecimal form.

Pointers occupy 8 bits and the addresses have 16 hexadecimal digits. If a machine is 64 bits, the pointer will be represented as a 64-bit value and if only occupies 32 bits, there will be displayed by 32 bits.

Remember, a pointer itself has an address, just like any other variable, so the %p is the conversion specifier to display an address. You use the & operator ot referende the address that a variable occupies. You can use (void*) to prevent a possible warning from the compiler, because the %p spacification expects the value to be some kind of pointer type, but the type of &pnumber is pointer to pointer to int:

printf("pnumber address is %p\n", (void*)&pnumber);

You use the sizeof operator to obtain the number of bytes a pointer occupies. You may get a compiler warning when using sizeof this way, since is an implementation-defined integer type. To prevent it, you could cast the argument to ypt int like this:

printf("pnumber size: %d bytes \n", (int)sizeof(pnumber));

12.4. Challenge Pointer Basics

Write a programme that creates an integer variable with a hard coded value. Assign that variable address to a pointer address. Display as output the address of the pointer, the value of the pointer and the value of what the pointer is pointing to.

The solution to the challenge is:

```c
#include <stdio.h>
#include <stddef.h>

int main(int argc, char **argv)
{
    // Define variables
    int variable = 10;      // Integer to which the pointer will point
    int *pvariable = NULL;  // Pointer of point to int type

    // Assign the pointer to variable
    pvariable = &variable;

    // Display the pointer characteristics
    printf("The address of the variable is: %p \n", &variable);
    printf("The address of the pointer is: %p \n", &pvariable);
    printf("The value of the pointer is: %p \n", pvariable);
    printf("The value of what the pointer is pointing is: %i \n", *pvariable);

    return 0;
}
```

Figura 12.1: Pointer basics challenge solution

13. Utilising pointers

13.1. Overview

C offers several basic operations you can perform on pointers: assigning an address to a pointer (& operator), dereferencing (* operator), take a pointer address (&) and perform pointer arithmetic (see following Section).

You can also find the difference between two pointers, you do this for two pointers to elements that are in the same array to find out how far apart the elements are. You can use the relational operators to compare the values of two pointers of the same type. There are two types of addition and subtraction: you can add (or subtract) a pointer to another to get an integer or add an integer to a pointer and get a pointer.

Be careful when incrementing or decrementing pointers and causing an array out-of-bounds error. The computer does not keep track of whether a pointer still points to an array element.

The value referenced by a pointer can be used in an arithmetic expression. If a variable is defined to be pointer to integer then it is evaluated using the rules of integer arithmetic.

Example: **pnumber = &number; *pnumber += 25;** increments the value of the number variable by 25. * indicates that you are accessing the contents to which the variable pnumber is pointing to.

If a pointer points to a variable x, that pointer has been defined to be a pointer to the same data type as is x. The use of *pointer in an expression is identical to the use of x in the same expression.

A variable defined as pointer to int can store the address of any variable of type int. A pointer can contain the address of any variable of the appropiate type, so you can use one pointer variable to change the values of many different variables as long as they are of a type compatible with the pointer type.

When we have used **scanf()** to input values, we have used & operator to obtain the address of a variable that is to store the input (second argument). When you have a pointer that already contains an address, you can use the pointer name as an argument for **scanf()**. Example:

int *pvalue = & value;

scanf(" %d", pvalue)

There is one very important rule: do not dereference a uninitialised pointer.

int *pt // an unitialised pointer

***pt = 5** // a terrible error

Second line means that the value 5 should be stores in the location to which pt points. pt has a random value, there is no knowing where the 5 will be placed. It might go somewhere harmless, it might overwrite data or code or it might cause the programme to crash.

Creating a pointer only allocates memory to store the pointer itself, not to store data. Before using a pointer, it should be assigned to memory location: you must assign the address of an existing variable to the pointer, or you can use the **malloc()** function to allocate memory first.

When declaring a pointer that does not point to anything, it must be initialised to NULL, the equivalent to 0 for pointers. Because of this, if you want to test whether a pointer is NULL you can do this: **if(!pvalue)** or you can do it explicitly by using == NULL.

You want to check for NULL before dereference a pointer, often when pointers are passed to functions.

13.2. Pointers and const

When we use the const modifier on a variable or an array it tells the compiler that the contents of the variable/array will not be changed by the programme.

With pointers, there are two things to consider when using the const modifier: whether the pointer will be changed and whether the value that the pointer points will be changed.

You can use the const keyword when you declare a pointer to indicate that the value pointed to must not be changed. Example:

long value = 9999L;

const long *pvalue = &value; // defines a pointer to a constant

This last statement declares the value pointed to by pvalue to be const. The compiler will check for any statements that attempt to modify the value pointed to by pvalue and flag such statements as an error. You can still modify the variable value (you have only applied const to the pointer). You could change value: **value = 7777L** would not result in an error.

The pointer itself is not constant, so you can still change what it points to:

long number = 8888L;

pvalue = &number; // Changes the address in pvalue

This will change the address stored in pvalue to point to number. You still cannot use the pointer to change the value that is stored, but you can change the address stored in the pointer as much as you like. Using the pointer to change the value pointed to is not allowed, even after you have changed the address stored in the pointer.

You might also want to ensure that the address stored in a pointer cannot be changed. You can do this by using the const keyword in the declaration of the pointer:

int count = 43;

int *const pcount = &count; // Defines a constant pointer

The above ensures that a pointer always points to the same thing, indicating that the address sotred must not be changed. Compiler will check that you do not try to chagne what the pointer points elsewhere in your code. You can still change the value that pcount points to: ***pcount = 345;** will be correct.

You can create a constant pointer that points to a value that is also constant:

int item = 25;

const int *const pitem = &item

pitem is a constant pointer to a constant so everything is fixed: you cannot change the address stored in pitem nor use pitem to modify what it points to. You can still change the value of item directly, if you want to make everything not change, you could specify item as const as well.

13.3. Void pointers

The type void means absence of any type. A pointer of type void* can contain the address of a data item of any type. Void* is often used as a parameter type or return value type with functions that deal with data in a type independent way. Any kind of pointer can be passed around as a value of type void*.

The void pointer does not know what type of object it is pointing to, so it cannot be dereferenced directly. The void pointer must first be explicitly cast to another pointer type before being referenced.

The address of a variable of type int can be stores in a pointer variable of type void*. When you want to access the integer value at the address stores in the void* pointer, you must first cast the pointer to type int.

13.4. Pointers and arrays

An array is a collection of objects of the same type that you can refer to using a single name.

You can use a pointer to hold the address of different variables of the same type at different times.

Array and pointers seem quite different, but they are very closely related and can sometimes be used interchangeably. One of the most common uses of pointers in C is as pointers to arrays. The main reasons for using pointers to arrays are notational convenience and programme efficiency. Pointers to arrays generally result in code that uses less memory and executes faster.

If you have an array of 100 integers:

int values[100];

You can define a pointer to access the integers contained in the array.

int *valuesPtr;

When you define a pointer to point the elements of an array, you do not designate the pointer as type pointer to array, you designate it to the type of element that is contained in the array.

To set valuesPtr to the first element in the values array you write:

valuesPtr = values;

The address operator is not used, the C compiler treats the appearance of an array name without a subscrip as a pointer to the array. Specifying values without a subscript has the effect of producing a pointer to the first element of values.

An equivalent way of producing a pointer to the start of values is to apply the address operator to the first element of the array:

valuesPtr = &values[0];

Either one is fine and is a matter of programmer preference.

The two expressions **ar[i]** and ***(ar+i)** are equivalent in meaning. Both work if ar is the name of an array and if ar is a pointer variable. Using an expression such as ar++ only works if ar is a pointer variable.

13.5. Passing pointers to a function

There are a few different ways you can pass data to a function: by value or by reference. C passes it by value, but it simulates pass by reference using address.

Pass by value is when a function copies the actual value of an argument into the formal parameter of the function. Changes made to the parameter inside the function have no effect on the argument. C uses pass by value, so the code within a function cannot alter the arguments used to call it.

You can pass a pointer as an argument to a function and you can also have a function return a pointer as its result.

Pass by reference copies the address of an argument into the formal parameter. The address is used to access the actual argument used in the call and the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other

value. You need to declare the function parameters as pointer types. Changes inside the function will be reflected outside the function as well, unlike calls by value.

You can communicate two kinds of information to a function: the value of x and the function must be declared with the same type as x:

function1(x);

int function1(int num)

Or you can transmit the address of x and requires the function definition to include a pointer to the correct type:

function2(&x);

int function2(int *ptr)

You can qualify a function parameter using the const keyword. Indicates that the function will treat the argument that is passed for this parameter as a constant. It is only useful when the parameter is a pointer. You apply the const keyword to a parameter that is a pointer to specify that the function will not change the value to which the argument points. Example:

bool SendMessage(const char* pmessage)

The type of parameter, pmessage, is a pointer to a const char. It is the char value that is constant, not its address. You could specify the pointer itself as const too, but this makes little sense because the address is passed by value. You cannot change the original pointer in the calling function.

The compiler knows that an argument that is a pointer to constant data will be safe. If you pass a pointer to constant data as an argument for a parameter, then the parameter must be used like in the above.

Returning a pointer from a function is a particularly powerful capability. It provides a way for you to return not just a single value, but a whole set of values.

You would have to declare a function returning a pointer:

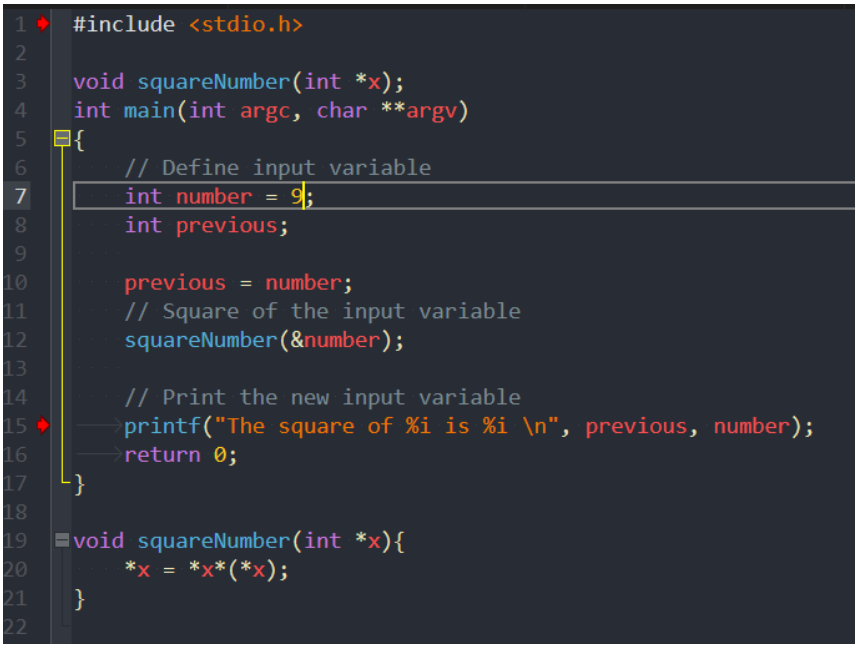
int* myFunction()

Be careful, there are specific hazards related to returning a pointer. Use local variables to avoid interfering with the variable that the arguments points to.

13.6. Challenge Pointers as parameters

This challenge is focussed to learn how to pass by reference. Write a function that squares a number by itself. The function should define as a parameter an int pointer.

The solution is:



```c
#include <stdio.h>

void squareNumber(int *x);
int main(int argc, char **argv)
{
    // Define input variable
    int number = 9;
    int previous;

    previous = number;
    // Square of the input variable
    squareNumber(&number);

    // Print the new input variable
    printf("The square of %i is %i \n", previous, number);
    return 0;
}

void squareNumber(int *x){
    *x = *x*(*x);
}
```

Figura 13.1: Pointers as Parameters challenge solution

14. Pointer arithmetic

14.1. Overview

The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array. A pointer can be used to access the first value of an array. To reference values[3] from a pointer that points to values[0], you can add 3 to the pointer (valuesPtr, for example) and the apply the indirection operator: ***(valuesPtr + 3)**. Generally, the expression *(valuePtr + i) can be used to access the value contained in values[i].

To set valuesPtr to point to the second element of the values array, you can apply the address operator to values[1] and assign the result to valuesPtr:

valuesPtr = &values[1]

Or, if valuesPtr points to values[0], you can set it to point to values[1] by simply adding 1 to the value of valuesPtr:

valuesPtr += 1;

This is a perfectly valid expression in C and can be used for pointers to any data type.

The increment and decrement operators ++ and – are particularly useful, the increment operator has the same effect as adding one to the pointer:

++valuesPtr

sets valuesPtr pointing to the next position in values array (values[1]), while

–valuesPtr

sets valuesPtr pointing to the previous position, assuming that valuesPtr was not pointing to the beginning of the values array.

To pass an array to a function, you simply specify the name of the array. To prodecue a pointer to an array, you need only specify the name of the array. This explains why you are able to change the elements of an array from within a function. The array can be declared to be a pointer. If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array, this reflects more correctly the use of the array by the function. If you are using the argument as a pointer to the array, declare it to be of type pointer.

Applying ++ or – to an array is not correct. Also, you cannot multiply arrays or sum pointers directions.

Functions that process arrays actually use pointers as arguments. You have a choice between array notation and pointer notation for writing array-processing functions. Using array notation makes it more obvious that the function is working with arrays. Other programmers might be more accustomed to working with pointers and find the pointer notation more natural. It is closer to machine language and, with some compilers, leads to more efficient code.

14.2. Pointers and arrays example

In PointersArraysExample there is another example of the relation between pointers and arrays.

14.3. Pointers and strigs

One of the most common applications of using a pointer to an array is as a pointer to a character string because of notational convenience and efficiency. Using a variable of type pointer to char to reference a string gives you a lot of flexibility.

If you have an array of characters called text, you could similarly define a pointer to be used to point to elements in text:

char *textPtr;

If textPtr is set pointing to the beginning of an array of chars called text, **++textPtr;** sets textPtr pointing to the next character in text, text [1]. **–textPtr** sets textPtr pointing to the previous character in text, assuming that textPtr was not pointin to the beginning of text.

14.4. Challenge Counting string characters

Write a function that calculates the length of a string. It should take as parameter a const char pointer, it can only determine the length using pointer arithmetic (incrementation operator ++pointer). You are required to use a while loop using the value of the pointer to exit. The function should subtract two pointers, one pointing to the end of the string and one pointing to

the beginning of the string. The function should return an int that is the length of the string passed into the function.

The solution is:

```c
#include <stdio.h>

int StringLength(const char *pointer);
int main(int argc, char **argv)
{
    int length = 0;
    char str[] = "DanielCeron";
    const char *pStr = str;
    length = StringLength(pStr);
    printf("The length of the string is %i\n", length);
    return 0;
}

int StringLength(const char *pString)
{
    int length = 0;
    const char *pEnd = pString;
    for(; *pEnd!='\0'; pEnd++)
    {
    }
    length = pEnd - pString;

    return length;
}
```

Figura 14.1: Counting string characters challenge solution

15. Dynamic Memory Allocation

15.1. Overview

Whenever you define a variable in C, the compiler automatically allocates the correct amount of storage for you based on the data type. It is frequently desirable to be able to dynamically allocate storage while a programme is running. If you have a programme that is designed to read in a set of data from a file into an array in memory, you have three choices:

- Define the array to contain the maximum number of possible elements at compile time.
- Use a variable-length array to dimension the size of the array at runtime.
- Allocate the array dynamically using one of the C memory allocation routines.

The first approach is a fixed size. The data file cannot contain more than the elements that you select. No matter what value you select, you always have the possibility of running into the same problem again.

Using the dynamic memory allocation functions, you can get storage as you need it. This approach allows you to allocate memory while the programme is running.

Depends on the concept of a pointer and provides a strong incentive to use pointers in your code. The majority of production programmes will use dynamic memory allocation. It allows you to create pointers at runtime that are just large enough to hold the amount of data that you require for the task.

Dynamic memory allocation reserves space in a memory area called the heap, where you can arbitrarily change the occupied memory. The stack is another place where memory is allocated, function arguments and local variables are stored here. When the execution of a function ends, the space allocated in the stack for arguments and local variables is freed.

The memory in the heap is different because it is controlled by you. When you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required. You must free the space you have allocated to allow it to be reused.

15.2. malloc calloc and realloc

15.2.1. malloc()

The simplest standard library function that allocates memory at runtime is called **malloc()**. You need to include the stdlib.h header file. You specify the number of bytes of memory that you want allocated as the argument. Returns the address of the first byte of memory that is allocated. Because you get an address returned, a pointer is the only place to put it. Example:

int *pNumber = (int*)malloc(100);

In the above, you have requested 100 bytes of memory and assigned the address of this memory block to pNumber. pNumber will point to the first int location at the beginning of the 100 bytes

that were allocated. It assumes that type int requires 4 bytes and that it can hold 25 int values. Is better to remove any assumption of this type and write it:

int *pNumber = (int*)malloc(25*sizeof(int));

The cast (int*) converts the address returned by the function to the type pointer to int. **malloc()** returns a pointer of type pointer to void, so you have to cast it.

You can request any number of bytes. **malloc()** returns a pointer with the value NULL. It is always a good idea to check any dynamic memory request immediately using an if statement to make sure the memory is actually there before you try to use it:

if(!pNumber) // code to deal with memoty allocation failure

You can least display a message and terminate the programme, is much better than allowing the programme to continue and crash when using a NULL address to store something.

When you allocate memory dynamically, you should always release the memory when it is no longer required. Memory that you allocate on the heap will be automatically released when your programme ends, but it is better to explicitly release the memory when you are done with it, even if it is just before you exit from a programme.

A memory leak occurs when you allocate some memory dynamically and do not retain the reference to it, so you are unable to release the memory. It often occurs in loops. Because yo do not release the memory when is no longer required, the programme consumes more and more of the available memory in each loop iteration and eventually occupy it all.

To free the memory that you have allocated dynamically, you must still have access to the address that references the block of memory.

To release the memory for a block of dynamically allocated memory whose address you have stores in a pointer:

free(pNumber);

pNumber = NULL;

The **free()** function has a formal parameter of type void*, so you can pass a pointer of any type as argument. As long as pNUmber contains the address that was returned when the memory was allocated, the entire block of memory will be freed for further use. You should always set the pointer to NULL after the memory to which it points has been freed.

15.2.2. calloc()

The **calloc()** function offers a couple of advantages over **malloc()**: it allocates memory as a number of elements of a given size and initialises the memory that is allocated so that all bytes are zero. The function is declares in the stdlib.h header file.

calloc() function requires two argument values: number of data items for which space is required and size of data item:

int *pNumber = (int*)calloc(75, sizeof(int));

Creates a memory block reserved to pNumber of 75 elements.

The return value will be NULL if it was not possible to allocate the memory requested. Is very similar to **malloc()**, but with the big advantage that you know the memory area will be initialised to 0.

15.2.3. realloc()

realloc() function enables you to reuse or extend memory that you previosuly allocated using **malloc()** or **calloc()**. It expects two argument values: a pointer containing an address that was previously returned by a call to **malloc()** or **calloc()** and the size in bytes of the new memory that you want allocated.

realloc() allocates the amount of memory specified by the second argument, transfering the contents of the previously allocated memory referenced by the pointer that you supply as the first argument to the newly allocated memory. It returns a void* pointer to the new memory or NULL if the operation fails for some reason.

The most important feature of this operation is that this function preserves the contents of the original memory area.

15.2.4. Guidelines

Avoid allocations of lots of small amounts of memory: allocating memory on the heap carries some overhead with it. Allocating many small blocks of memory will carry much more overhead than allocating fewer larger blocks.

Only hang on to the memory as long as you need it, as soon as you are finished with a block of memory of the heap, release the memory.

Always ensure that you provide for releasing memoty that you have allocated. Decide where in your code you will release the memory when you write the code that allocates it.

Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it, this will cause a memory leak. Be especially careful when allocating memory within a loop.

15.3. Challenge Dynamic memory

Write a programme that allows a user to input a text string. The programme will print the text inputted. The programme will utilise dynamic memory allocation.

The user can enter the limit of the string they are entering. You can use this limit when invoking malloc.

The programme should create a char pointer only, no character arrays.

Be sure to release the memory that was allocated.

The solution is:

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

int main(int argc, char **argv)
{
    // Initialise variables
    int length = 0;
    char *pName = NULL;

    // Enter maximum length of the array
    printf("Enter the maximum number of the array: \n");
    scanf("%i", &length);

    // Allocate the memory dynamically
    pName = (char*)calloc(length, sizeof(char));

    // Ensure the memory has been correctly assigned
    if (pName != NULL)
    {
        // Enter the array to write
        printf("Enter the text to write: \n");
        scanf("%s", pName);

        // Print the array
        printf("The array is: %s", pName);

    }

    free(pName);

    return 0;
}
```

Figura 15.1: Dynamic memory challenge solution

16. Structures

16.1. Creating and Using structures

Structures in C provide another tool for grouping elements together.

Suppose you want to store a date inside the programme, we could create variables for month, day and year to store the date. If also needs to store the date of purchase of a particular item, you must keep track of other three variables for each date that you use. These variables are logically related and should be grouped together.

It would be much better to group these sets of three variables together. This is what structures in C allow you to do.

A structure declaration describes how a structure is put together, what elements are inside. The struct keyword enables to define a collection of variables of various types called a structure that you can treat as a single unit:

struct date

{

int day;

int month;

int year;

;

The above statement defines what a date structure looks like to the C compiler. There is no memory allocation for this declaration.

The variable names within the date structure: month, day, year, are called members or fields. The members of the structure appear between the braces that follow the struct tag name date.

The definition of a date defines a new type in the language. Variables can now be declared to be of type struct date.

struct date today;

The above statement declares a variable to be of type struct date. Memory is now allocated for the variable above for the three integer members per variable.

There is a difference between defining a structure and declaring variables of the structure type.

To access the members of a structure, you need a special syntax. A structure variable name is not a pointer. You refer to a member of a structure by writing the variable name followed by a period, followed by the member variable name. The period is called the member selection operator. There are no spaces between the variable name, the period, and the member name. To set the value to the day in the variable today to 25:

today.day = 25;

today.year = 2015;

When it comes to the evaluation of the expressions, structure members follow the same rules as ordinary variables do:

century = today.year/100 + 1;

An integer structure member divided by an integer is performed as an integer division.

You do have some flexibility in defining a structure; it is valid to declare a variable to be of a particular structure type at the same type that the structure is defined by including the variable name or names before the terminating semicolon of the structure definition:

struct date

{

int day;

int month;

int year;

today;

This creates a variable named today at the same time the structure is defined, being today a variable of type date. You can also assign initial values to the variables in a normal way.

You also do not have to give a structure a tag name. If all the variables of a particular structure type are defined when the structure is defined, the structure name can be omitted.

struct

{

int day;

int month;

int year;

today;

A disadvantage is that you can no longer define further instances of the structure in another statement. All variables of the structure type you want must be defined in one statement.

Initialising structures is similar to initialising arrays. The elements are listed inside a pair of braces, with each element separated by a comma. The initial values listed inside the curly braces must be constant expressions:

struct date today = 7, 2, 2015;

Just like an array initialisation, fewer values might be listed than are contained in the structure;

struct date date1 = 12,10;

sets date1.day to 12 and date1.month to 10, but gives no initial value to date.year.

You can also specify the member names in the initialisation list, enabling you to initialise the members in any order or only initialise specified members:

struct date date1 = .day = 12, .month = 10;

You can assign one or more values to a structure in a single statement using compound literals:

today = (struct date)25, 9, 2015;

This statement can appear anywhere on the programme. It is not a declaration statement, the type cast operator is used to tell the compile the type of the expression, the list of values follows the cast and are to be assigned to the members of the structure, in order. They are listed in the same way as if you were initialising a structure variable.

16.2. Declaring and initialising a structure

Write a programme that declares a structure and prints out its content. Create an employee structure with 3 members: name(character array), hireDate(int), salary(float). Declare and initialise an instance of an employee type. Read in a second employee from the console and store it in a structure of type employee. Print out the contents of each employee.

The solution is:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    // Create employee structure
    struct employee{
        char name[30];
        int hireDate;
        float salary;
    }employee2;

    // Define employee 1 structure
    struct employee employee1 = {"Jason", 500, 35.0e3};

    // Read employee 2 data
    printf("Introduce name of employee 2: ");
    scanf("%s", &employee2.name);
    printf("\nIntroduce hire date of employee 2: ");
    scanf("%i", &employee2.hireDate);
    printf("\nIntroduce name of employee 2: ");
    scanf("%f", &employee2.salary);

    // Print out the data of both employees
    printf("\n%s started to work %i days ago and earns %f euros",
           employee1.name, employee1.hireDate, employee1.salary);
    printf("\n%s started to work %i days ago and earns %f euros",
           employee2.name, employee2.hireDate, employee2.salary);

    return 0;
}
```

Figura 16.1: Declaring a structure solution

16.3. Structures and arrays

You can create multiple structures in the same array. A better method to handle multiple different structures is combining structures and arrays. Declaring an array of structures is like declaring any other kind of array:

struct date myDates[10];

defines an array called myDates which consists of 10 elements. Each element inside the array is defined to be of type struct date.

To identify members of an array of structures, you apply the same rule used for individual structures: follow the structure name with the dot operator and then with the member name. Referencing a particular structure element inside the array is quite natural. For example, to set the second date inside the myDates array to August 8, 1996:

myDates[1].day = 8;

myDates[1].month = 8;

myDates[1].year = 1996;

Initialisation of arrays containing structures is similar to initialisation of multidimensional arrays:

struct date myDates[5] = 12,10,1975, 30,12,1980, 15,11,2005;

sets the first three dates in the array myDates to 12/10/1975, 30/12/1980 and 15/11/2005.

The inner pair of braces are optional, but they help to the readability.

If you want to initialise just the third element of the array to the specified value:

struct date myDates[5] = [2] = 12,10,1975;

To set just the month and day of the second element of the myDates array to 12 and 30:

struct date myDates[5] = [1].month = 12, [1].day = 30;

It is also possible to define structures that contain arrays as members. The most common use is to set up an array of characters inside a structure.

Suppose you want to define a structure called month that contains as its members the number of days in the month as well as a three-character abbreviation for the month name.

struct month

{

int numberOfDays;

char name[3];

};

THis sets up a month structure that contains an integer member called numberOfDays and a character member called name. Member name is actually an array of three characters.

You can now define a variable to be of type struct month and set the proper field inside aMonth for January:

struct month aMonth;

aMonth.numberOfDays = 31;

aMonth.name[0] = 'J';

aMonth.name[1] = 'a';

aMonth.name[2] = 'n';

Another way is:

struct month aMonth = 31, 'J','a','n';

16.4. Nested structures

C allows to define a structure that itself contains other structures as one or more of its members. You have seen how it is posisble to logically group the month, day and year into a structure called date. How about grouping the hours, minutes and seconds into a structure called name:

struct time

{

int hours;

int minutes;

int seconds;

;

In some applications you may need to group both a date and a time together. You want a convenient way to associate both the date and the time together. You can define a new structure called, for example, dateAndTime, which contains as its members two elements: date and time.

struct dateAndTime

{

struct date sdate;

struct time stime;

;

The first member of this structure is of type struct date and is called sdate. The second member is of type struct time and is called stime. Variable can now be defined to be of type struct dateAndTime:

struct dateAndTime event;

To reference the date structure of the variable event, the syntax is the same as referencing any member:

event.sdate

To reference a particular member inside one of these structures, a period followed by the member name is tacked on the end. The below statement sets the month of the date structure contained within event to October and adds one to the seconds contained within the time structure:

event.sdate.month = 10;

++event.stime.seconds;

The event variable can be initialised like normal structures. To set the date in the variable event to February 1 2015 and the time to 3:30:00:

struct dateAndTime event = 2,1,2015,3,30,0;

You can use member names in the initialisation:

struct dateAndTime event = .month = 2, .day = 1, .year = 2015, .hour = 3, .minutes = 30, .seconds = 0;

It is also possible to set up and array of dateAndTime structures:

struct dateAndTime events[100];

The array events is declared to contain 100 elements of type struct dateAndTime. The fourth dateAndTime contained within the array is referenced in the usual way as events[3]. To set the first time in the array to noon:

events[0].stime.hour = 12;

events[0].stime.minutes = 0;

events[0].stime.seconds = 0;

You can define the Date structure within the time structure definition. The limitation is that the date structure can be used only inside the time structure, date would not exist outside the time structure.

16.5. Structures and pointers

C allows for pointers to structure. Pointers to structures are easier to manipulate than structures themselves. In older implementations, a structure cannot be passed as an argument to a function, but a pointer to a structure can. Even if you can pass a structure as an argument, passing a pointer is more efficient. Many data representations use structures containing pointers to other structures.

You can define a variable to be a pointer to struct as:

struct date *datePtr;

The variable datePtr can be assigned just like other pointers:

datePtr = &todaysDate;

You can then indirectly access any of the members of the date structure pointed to by datePtr:

(*datePtr).day = 21;

The above has the effect of setting the day of the date structure pointed to by datePtr to 21. Parentheses are required because the structure member operator . has higher precedence than the indirection operator *. It would try to access the data from a NULL pointer, essentially.

To test the value of month stored in the date structure pointed to by datePtr:

if ((*datePtr).month == 12)

Pointers to structures are so often used in C that a special operator exists. The structure pointer operator ->, which is the dash followed by the greater than sign permits

(*x).y;

to be more clearly expressed as:

x->y;

And the previous if statement can be conveniently written as:

if (datePtr->month == 12)

A pointer can also be a member of the structure:

struct intPtrs

{

int *p1;

int *p2;

;

A structure called intPtrs is defined to contain two integer pointers, the first one called p1 and the second one called p2. You can now define a variable of type struct intPtrs:

struct intPtrs pointers;

The variable pointers can now be used just like other structs. pointers itself is not a pointer, but a structure variable that has two pointers as its members.

You can do both structures of characters arrays or characters pointers. The pointers in a character struct should be used only to manage strings that were created and allocated elsewhere in the programme because the declaration of a structure of pointers to char do not allocate memory in the programme for the char arrays.

One instance in which it does make sense to use a pointer in a structure to handle a string is if you are dynamically allocating that memory, using a pointer to store the address. It has the advantage that you can ask **malloc()** to allocate just the amount of space that is necessary for a string.

The strings of pointers are not stored in a structure, they are stored in the chunk of memory managed by **malloc()**, The addresses of the two strings that are stores in the structure. Addresses are what string-handling functions typically work with.

16.6. Structures and functions

After declaring a structure, you can pass it as an argument to a function. The declaration of a function with an input structure would be: **bool siblings (struct Family member 1, struct**

Family member2). You would pass two structures of type Family as parameters: member 1 and member 2.

You should use a pointer to a structure as an argument. It can take quite a bit of time to cpy large structures as arguments, as well as requiring whatever amount of memory to store the copy of the structure. Pointers to structures avoid the memory consumption and the copying time (only a copy of the pointer argument is made). Example: **bool siblings (srutc Family *pmember1, struct Family *pmember2)**.

You can also use the const modifier to not allow any modification of the members of the struct (what the struct is pointing to): **bool siblings (Family const *pmember1, Family const *pmember2**.

You can use the const modifier to not allow any modification of the pointers address. Any attempt to change those structures will cause an error messahe during compilation, but it will be redundant since you are passing an address to the function: **bool siblings (Family *const pmember1, Family *const pmember2**. The indirection operator is now in front of the const keyword, not in front of the parameter name. You cannot modify the addresses stored in the pointer, they are the pointers who are protected here, not the structures to which they point to.

The function prototype must indicate the return value in the normal way: **struct Date my_fun(void);**. This is a prototype for a function that takes no arguments that returns a structure of type Date. It is often more convenient to return a pointer to a structure. When returning a pointer to a structure, it should be created on the heap.

You should always use pointers when passing structures of a function. In older versions on C, passing a structure may cause errors, passing a pointer will never be problematic. However, you have less protection for your data. SOme operations in the function could inadvertently affect data in the original structure. Use const qualifier solves the problem.

Advantages of passing structures as arguments: the function works with copies of the original data, which is safer than working with the original data; and the programming style tends to be clearer.

Disadvantages of passing structures as arguments: older implementations might not handle the code, wastes time ans space, specially wasteful to pass large structures to a function that uses only one or two members of the structure.

Programmers use structure pointers as function arguments for reasons of efficiency and use const when necessary. Passing structures by value is most often done for structures that are small.

16.7. Challenge Structures pointers and functions

Write a C programme that creates a structure pointer and passes it to a function. The structure name is item, with the following members: itemName (pointer), quantity (int), price (float), amount (float, stores quantity* price).

Create a function named readItem that takes a structure pointer of type item as a parameter. This function should read in from the user a product name, price and quantity. The contents read in should be stored in the passed in structure to the function.

Create a function named print that takes as a parameter a structure pointer of type item and

prints its content.

The main function should declare an item and a pointer to the item. You will need to allocate memory for the itenName pointer. THe item pointer should be passed into both the read and print item functions.

The solution is:

```c
#include <stdio.h>
#include <stdlib.h>

// Define the structure type
    struct item {
        char *itemName;
        int quantity;
        float price;
        float amount;
    };

void readItem(struct item *i);
void printItem(struct item *i);

int main(int argc, char **argv)
{
    // Create a structure of type item
    struct item itm;
    struct item *pItem;

    // Define a pointer to the structure
    pItem = &itm;

    // Allocate memory for the item name
    pItem->itemName = (char*)malloc(30*sizeof(char));

    if(pItem == NULL)
        exit(-1);

    // Call the read Item function for the user
    readItem(pItem);

    // Print the items introduced by the user
    printItem(pItem);

    return 0;
}
```

Figura 16.2: Challenge Structure Pointers 1

```c
// Function to read the item members
void readItem(struct item *Item)
{
    printf("Introduce the product name\n");
    scanf("%s", Item->itemName);
    printf("Introduce the product price\n");
    scanf("%f", &Item->price);
    printf("Introduce the product quantity\n");
    scanf("%i", &Item->quantity);
    Item->amount = (float)Item->quantity * Item->price;
}

// Function to print the item members
void printItem(struct item *Item)
{
    printf("The product name is %s, its price is %.2f and its quantity is %i. The total amount is %.2f",
    Item->itemName, Item->price, Item->quantity, Item->amount);
}
```

Figura 16.3: Challenge Structure Pointers 2

17. File Input and Output

17.1. Overview

All data that the previous programmes accessed was through memory. The scope and variety of applications you can create are limited.

All serious business require more data than would fit into main memory. Also depend on the ability to process data that is persistent and stored on an external device.

C provides many functions in the header file stdio.h for writing to and reading from external devices. The external device you would use for storing and retrieving data is typically a disk drive. However, the library will work with virtually any external storage device.

With all the examples up to now, any data the user enters is lost once the programme ends. If the user wants to run the programme with the same data, he must enter it again each time, wichi is very inconvenient and limits programming. This is referred to as volatile memory.

Programmes usually need to store data on permanent storage, non-volatile, that continues to be maintained after your computer is turned off. A file can store non-volatile data and is usually stored on a disk or a solid state device, a named section of storage. stdio.h is a file containing useful information.

C views a file as a continuous sequence of bytes, each bite can be read individually. It corresponds to the file structure in the Unix environment. A file has a beginning and an end, and a current position (defined as so many bytes from the beginning). You can move the current position to any point in the file (even the end).

There are two types of writing data to a stream that represents a file: text and binary.

Text data is written as a sequence of characters organised as lines (each line ends with a new line). Binary data is written as a series of bytes exactly as they appear in memory, representing image data or music encoding among others, not readable. You can write any data you like to a file, once written, the file just consists of a series of bytes.

You have to understand the format of the file in order to read it: a sequence of 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, etc. In binary mode, each and every byte of the file is accessible.

C programmes automatically open three files on your behalf: standard input (the normal input device for your system, usually your keyboard), standard output (usually your display screen) and standard error (usually your display screen).

Standard input is the file that is read by **getchar()** and **scanf()**. Standard output is used by **putchar()**, **puts()** and **printf()**. Redirection causes other files to be recognised as the standard input or standard output. The purpose of the standard error output file is to provide a logically distinct place to send error messages.

A stream is an abstract representation of any external source or destination for data. The keyboard, the command line on your display, and files on a disk are all examples of things you can work with the streams. The C library provides functions for reading and writing to or from data streams. You use the same input/output functions for reading and writing any external device that is mapped to a stream.

17.2. Accessing files

Files on disk have a name and the rules for naming files are determined by your operating system. You may have to adjust the names depending on your OS.

A programme references a file through a file pointer (or stream pointer, since it works in more than a file). You associate a file pointer with a file programmatically when the programme is running. Pointers can be reused to point to different file on different occasions.

A file pointer points to a struct of type FILE that represents a stream. It contains information about the file: whether you want to read or write or update the file, the address of the buffer in memory to be used for data and a pointer to the current position in the file for the next operation. This is all set through input/output file operations.

If you want to use several files simultaneously in a programme, you need a separate file pointer for each file. There is a limit to the number of files that you can open at one time, defined as FOPEN_MAX in stdio.h.

You associate a specific external file with an internal file pointer file variable through a process referred to as opening a file, via the **fopen()** function. It returns the file pointer for a specific external file. This function is defined in stdio.h. The structure is:

FILE *fopen(const char * restrict name, const char * restrict mode);

The first argument is a pointer to a string that is the name of the external file you want to process. You can specify the name explicitly or use a char pointer that contains the address of the character string that defines the file name. You can obtain the file name through the command line, as input from the user, or defined as a constant in your programme.

If you do not specify the path, the file must be in the workspace main folder.

The second argument is a character string that represents the file mode, it specifies what you want to do with the file. Is a character string between double quotes.

Assuming the call to **fopen()** is successful, the function returns a pointer of type FILE* that you can use to reference the file in further input/output operations using other functions in the library. If the file cannot be opened for some reason, it returns NULL.

The file modes are the following:

- **"w":** Open a text file for write operations. If the file exists, its current contents are discarded.
- **."a":** Open a text file for append operations. It writes at the end of the file.
- **r":** Open a text file for read operations.
- **"w+":** Open a text file for update (reading and writing) operations. If the file exists, it truncates the contents to zero. If not it creates the file.
- **."a":** Open a text file for update (reading and writing) operations. It writes at the end of the file if it exists. If not it creates the file.
- **r":** Open a text file for update (reading and writing) operations.

17.2.1. Write mode

If you want to write to an existing file with the name myfile.txt:

FILE *pfile = NULL;

char *filename = "myfile.text.";

pfile = fopen(filename, "w"); // Open myfile.txt to write it

This opens the file and associates the file with the name myfile.txt with your file pointer pfile. The mode "w"means that you can only write to the file, not read it. If myfile.txt does not exist, the function will create a new file with this name. If you only provide the file name without any path specification, the file is assumed to be in the current directory. You can also specify a string that is the full path and name for the file.

On opening a file for writing, the file length is truncated to zero, and the position will be at the beginning of any existing data for the first operation. Any data that was previously written to the file will be lost and overwritten by any write operations.

17.2.2. Append mode

If you want to add to an existing file rather than overwrite it, specify mode .ª", the append mode. This positions the file at the end of any previously written data or creates a new file if it does not exist. Example:

pFile = fopen("myfile.txt",.ª");

Do not forget that you should test the return value for null each time. When you open a file in append mode, all write operations will be at the end of the data in the file on each write operation; and all write operations append data to the file and you cannot update the existing contents in this mode.

17.2.3. Read mode

If you want to read a file, open it with mode argument r", you cannot write to this file. Example:

pFile = fopen("myfile.txt",r")

This positions the file to the beginning of the data. If you are going to read the file, it must already exist. If you try to open a file for reading that does not exist, it will return a NULL file pointer, so you always want to check the value returned from **fopen()**.

17.2.4. Renaming a file

Renaming a file is very easy using the **rename()** function:

int rename(const char *oldname, const char *newname);

The integer that is returned will be 0 if the name change was successful and nonzero otherwise. The file must not be opened when you call **rename()**, otherwise the operation will fail. If the file path is incorrect or the file does not exist, the renaming operation will fail.

17.2.5. Closing a file

When you have finished with a file, you need to tell the OS that it can free up the file using the **fclose()**.

fclose() accepts a file pointer as an argument. It returns EOF(int) if an error occurs. EOF is a special character called the End-Of-File character. It is defined in stdio.h as a negative integer, usually -1. If it is successful, returns 0. Example:

fclose(pfile); // Close the associated file with pfile.

The result is that the connection between the pointer, pfile and the physical file is broken, pfile is no longer able to access the file. If the file was being eritten, the current outputs of the output buffer are written to the file to ensure that data is not lost.

Is a good programming practice to close a file as sson as you have finished with it, it protects against output data loss. You must also close a file before attempting to rename it or remove it.

17.2.6. Deleting a file

You can delete a file by invoking the **remove()** function, declared in stdio.h:

remove("myfile.txt")

This will delete the file myfile.txt from the current directory. It cannot be opened when you try to delete it. You should always double check with operations that delete files, you could wreck your system if you do not.

17.3. Reading from a file

The **fgetc()** function reads a character from a text file thas has been previously opened for reading. Takes a file pointer as its only argument and returns the character read as type int:

int mchar = fgetc(pfile); // Reads a character into mchar with pfile a File pointer

The mchar is type int because EOF will be returned if the end of the file has been reached.

The function **getc()** is also available, requires an argument of type FILE* and returns the character read as type int. It is identical to **fgetc()** but it can be implemented as a macro, whereas **fgetc()** is a function.

You can read the contents of a file again when necessary with the **rewind()** function, that positions the file specified by the file pointer argument at the beginning:

rewind(pfile);

You can use the **fgets()** function to read from any file or stream:

char *fgets(char *str, int nchars, FILE *stream)

The function reads a string into the memory area pointed to by str, from the file specified by stream. Characters are read until either a null character is read or nchars-1 characters have been read from the stream, whichever occurs first.

If a newline is read, it is retained in the string, a \0 character will be appended to the end of the string. If there is no error **fgets()**returns the pointer, str. If there is an error NULL is returned. Reading EOF causes NULL to be returned.

You can get formatted input from a file by using the standard **fscanf()** function.

int fscanf(FILE *stream, const char *format, ...);

The first argument to this function is the pointer to a FILE object that identifies the stream. The second argument is the format: a C string that contains one or more of the following items: whitespace character, non-whitespace character, format specifiers. The usage is similar to scanf, but from a file.

The function returns the number of input items successfully matcjed and assigned.

17.4. Challenge Find number of lines

Write a programme to find the total numbers of lines in a text file. Create a file that contains some lines of text. Open your test file, use the fgetc function to parse characters in a file until you get to the EOF. If EOF, increment counter. Display as output the total numbers of lines in the file.

The solution is:

```c
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "ReadFiles.txt"

int main(int argc, char **argv)
{
    FILE *fp = NULL;
    char ch;
    int linesCount = 0;

    fp = fopen(FILENAME, "r");

    if (fp== NULL)
    {
        printf("File does not exist");
        return -1;
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        if (ch == '\n'
        {
            linesCount++;
        }
    }

    fclose(fp);
    fp = NULL;

    printf("Total number of lines are: %d\n",linesCount);
    return 0;
}
```

Figura 17.1: Find Number of lines solution

17.5. Writing to a file

The simplest write operation is provided by the function **fputc()**, that writes a single character to a text file:

int fputc(int ch, FILE *pfile);

The function writes the character specified by the first argument to the file identified by the second argument (file pointer). It returns the character that was written if successful or EOF if failure.

In practice, characters are not usually written to a physical file one by one, it is extremely inefficient. The **putc()** function is equivalent to this function, with the same arguments and return type, but can be implemented as a macro, whereas **fputc()** is a function.

You can use the **fputs()** function to write to any file or stream:

int fputs(const char *str, FILE *pfile)

The first argument is a pointer to the character string that is to be written to the file and the

second argument the file pointer. This function will write characters from a string until it reaches the \0 character. However, it does not write this character to the file, which can complicate reading back variable-length strings from a file that has been written by **fputs()**. It expects to write a line of text that has a new line character at the end.

The standard function for formatted output to a stram is **fprintf()**.

int fprintf(FILE *stream, const char *format,...)

The first argument for this function is a pointer to a FILE object that identifies the stream. The second argument is the format: a C string that contains one or more of the following items: whitespace character, non-whitespace character, format specifiers. The usage is similar to scanf, but from a file. If successful, the total number of characters written is returned; otherwise, a negative number is returned.

17.6. Convert characters in a file to uppercase

Write a programme that converts all characters of a file to uppercase and write the results to a temporary file. Then rename the temporary file to the original filename and remove the temporary filename. Use the fgetc and fputc functions. Use the rename and remove functions. Use the islower function, you can convert a character to uppercase by subtracting 32 from it. Display the contents in uppercase to standard output.

The solution is:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>


#define FILENAME "ConvertUppercase.txt"

int main(int argc, char **argv)
{
    // Define pointers and variables
    FILE *fp = NULL;
    FILE *fp_temp = NULL;
    char ch = ' ';

    // Read the file
    fp = fopen(FILENAME, "r");

    // Return -1 if there is a problem reading the file
    if (fp == NULL)
    {
        return -1;
    }

    // Create a temporary file
    fp_temp = fopen("temp.txt", "w");

    // Return -1 if there is a problem writing the file
    if (fp_temp == NULL)
    {
        return -1;
    }

```

Figura 17.2: Challenge Convert characters to Uppercase Solution 1

```
33      // Read the file until reaching the end
34      while ((ch = fgetc(fp)) != EOF)
35      {
36          if (islower(ch))
37          {
38              // If it is lowercase, turn it to uppercase
39              ch = ch -32;
40          }
41          // Save it in temporary file
42          fputc(ch, fp_temp);
43      }
44      // Close the files
45      fclose(fp);
46      fclose(fp_temp);
47
48      // Rename temp file to ConvertUppercase
49      rename("temp.txt", FILENAME);
50
51      // Remove temporary file
52      remove("temp.txt");
53
54      // Open new ConvertUppercase
55      fp = fopen(FILENAME, "r");
56
57      // Return -1 if there is a problem reading the file
58      if (fp == NULL)
59      {
60          return -1;
61      }
62
```

Figura 17.3: Challenge Convert characters to Uppercase Solution 2

```
68
69      // Close file
70      fclose(fp);
71
72      // Set the pointers to NULL
73      fp = NULL;
74      fp_temp = NULL;
75      return 0;
76  }
77
```

Figura 17.4: Challenge Convert characters to Uppercase Solution 3

17.7. Finding your position in a File

For many applications, you will need to access data in a file other than sequential order. There are various functions that you can use to access data in random sequence.

There are two aspects to file positionning: finding out where you are in a file and moving to a given point in file.

You can access a file at a random position regardless of whether you opened the file.

You have two functions to tell you where you are in a file: **ftell()** and **fgetpos()**. The first one has the structure:

long ftell(FILE *pfile);

This function accepts a file pointer as an argument and returns a long integer value that specifies the current position in the file. Example:

long fpos = ftell(pfile);

The fpos variable now holds the current position in the file and you can use this to return to this position at any subsequent time. The value is the offset in bytes from the beginning of the file.

fgetpos() is a little more complicated. Its structure is:

int fgetpos(FILE *pfile, fpos_t *position);

The first parameter is a file pointer. The second parameter is a pointer to a type that is defined in stdio.h, a type that is able to record every position within a file.

fgetpos() is designed to be used with the positioning function **fsetpos()**. **fgetpos()** stores the cuttent position and file state information for the file in position and returns 0 if the operation is successful or a nonzero integer value for failure. Example:

fpos_t here;

fgetpos(pfile, &here);

In here we have to pass the ampersnad, the address of it, because it is not a pointer. The above records the current file position in the variable here. You must declare a variable of type fpos_t, but not a pointer of type fpos_t* because there will not be any memory allocated to store the position data.

As complement to **ftell()**, you have the **fseek()** function. Structure:

int fseek(FILE *pfile, long offset, int origin);

The first parameter is a pointer to the file that you are repositioning. The second and third parameters define where you want to go in the file. The second parameter is an offset from a reference point specified by the third parameter. Reference point can be one of three values specified by the predetermined names:

- **SEEK_SET:** defines the beginning of the file.
- **SEEK_CUR:** defines the current position in the file.
- **SEEK_END:** defines the end of the file.

For a text mode file, the second argument must be a value returned by **ftell()**. The third argument for text mode files operations must be SEEK_SET. For text files, all operations with **fseek()** are performed with reference to the beginning of the file. For binary files, the offset argument is simply a relative byte count. Therefore, it can supply positive or negative values for the offset when the reference point is specified as SEEK_CUR.

fsetpos() go with **fgetpos()**. Its structure is:

int fsetpos(FILE *pfile, const fpos_t *position);

The first parameter is a pointer to the open file. The second is a pointer of the fpos_t type, the position that is stored at the address was obtained by calling **fgetpos()**. **fsetpos()** returns a nonzero value on error and 0 when it succeeds.

This function is designed to work with a value that is returned by **fgetpos()**. You can only use it to get to a place in a file that you have been before. **fseek()** allows you to go to any position just by specifying the appropiate offset.

17.8. Challenge print contents of file in reverse order

Write a programme that will print the contents of a file in reverse order. Use the fseek function to seek to the end of the file. Use the ftell function to get the position of the file pointer. Display as output the file in reverse order.

The solution to the challenge is:

```c
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "File.txt"

int main(int argc, char **argv)
{
    // Variable declaration
    FILE *fp = NULL;
    int cnt = 0, i = 0;

    // Read the file with the file pointer
    fp = fopen(FILENAME, "r");

    // End if fail in the assignment
    if (fp == NULL)
    {
        return -1;
    }

    // Move the file pointer to the end of the file
    fseek(fp, 0, SEEK_END);

    // Get the number of bytes of the file
    cnt = ftell(fp);

```

Figura 17.5: Challenge print reverse file 1

```c
    // While loop to print character in reverse order
    while (i < cnt)
    {
        i++;
        // i bytes before the end each time
        fseek(fp, -i, SEEK_END);

        printf("%c", fgetc(fp));
    }
    printf("\n");

    // Close file and pointer to null
    fclose(fp);
    fp = NULL;

    return 0;
```

Figura 17.6: Challenge print reverse file 2

18. The Standard C Library

18.1. Standard header files

The C standard library offers functions that you can use in your programmes without having to code anything. In this Subsection, there will be an explanation of the main standard header files to use in C

18.1.1. stddef.h

This header contains some standard definitions:

- **NULL:** A null pointer constant.
- **offset of(struc, memb):** The offset in bytes of the member memb from the start of the structure struc, the type of the result is size_t.
- **ptrdiff_t:** The type of integer produced by subtracting two pointers.
- **size_t:** The type of integer produced by the sizeof operator.
- **wchar_t:** The type of integer required to hold a wide character.

18.1.2. limits.h

Contains various implementation-defined limits of character and integer data types.

18.1.3. stdbool.h

Contains definitions for working with Boolean variables: type _Bool:

- **bool:** Substitute name for the basic _Bool data type.
- **true:** Defined as 1.
- **false:** Defined as 0.

18.2. Various functions

A reminder of the C standard library that have already been addressed.

18.2.1. String functions

To use any of these functions, you need to include the header file string.h.

char *strcat(s1,s2): Concatenates the character string s2 to the end of s1, placing a null character at the end of the final string. The function returns s1.

char *strchar(s,c): Searches the string s for the first occurrence of the character c. If it is found, a pointer to the character is returned, otherwise, a null pointer is returned.

int strcmp(s1,s2): Compares strings s1 and s2 and returns a value less than zero if s1 is less than s2, equal to zero if s1 is equal to s2 and greater than zero if s1 is greater than s2.

char *strcpy(s1,s2): Copies the string s2 to s1, returning s1.

size_t strlen(s): Returns the number of characters in s, excluding the null character.

char *strncat(s1,s2,n): Copies s2 to the end of s1 until either the null character is reached or n characters have been copied, whichever occurs first. Returns s1.

int strncomp(s1,s2,n): Compares strings s1 and s2 and returns a value less than zero if s1 is less than s2, equal to zero if s1 is equal to s2 and greater than zero if s1 is greater than s2. Only compares the first n characters of the strings.

char strncpy(s1,s2,n): Copies s2 to s1 until either the null character is reached or n characters have been copied, whichever occurs first. Returns s1.

char *strrchr(s,c): Searches the string s for the last occurrence of the character c. If founf, a pointer to the character in s is returned, otherwise, the null pointer is returned.

char *strstr(s1,s2): Searches the string s1 for the first occurrence of the string s2. If found, a pointer to the start of where s2 is located inside the s1 is returned, otherwise, if s2 is not located inside s1, the null pointer is returned.

char *strtok(s1,s2): Braks the string s1 into tokens based on delimiter characters in s2.

18.2.2. Character functions

To use these character functions, you must include the file ctype.h.

Examples: isalnum, isalpha, isblank, islower, isspace...

Another two functions are **int tolower(c)** (returns the lowercase equivalent of c. If c is not an uppercase letter, c itself is returned) and **int toupper(c)** (returns the uppercase equivalent of c. If c is not an lowercase letter, c itself is returned).

18.2.3. Input/output functions

The most common I/O functions from the C library are included in the header file stdio.h.

Also, definitions for the names EOF, NULL, stdin, stdout, stderr and FILE.

int fclose(Ptr): closes the file identified by filePtr and returns zero if the close is successful or EOF if an error occurs.

int feof(filePtr): returns nonzero if the identified file has reached the end of the file and zero otherwise.

int fflush(filePtr): flushes (writes) any data from internal buffers to the indicated file, returning zero on success and EOF if an error occurs.

int fgetc(filePtr): Returns the next character from the file identified by filePtr of EOF if an end of file condition occurs. The function returns an int.

int fgetpos(filePtr, fpos): gets the current file position for the file associated with filePtr, storing it into the fpos_t variable (defined in stdio.h) pointed to by fpos. fgetpos returns zero on success and nonzero on failure.

char *fgets(buffer, i, filePtr): reads characters from the indicated file, until either i-1 characters are read or a newline character is read, whichever occurs first.

FILE *fopen(fileName, accessMode): opens the specified file with the indicated access mode.

int fprintf(filePtr, format, arg1, arg2,..., argn writes the specified arguments to the file identified by filePtr, according to the format specified by the character string format.

int fputc(c,filePtr): writes the value of c to the file identified by filePtr, returning c if the write is successful and EOF otherwise.

int fputs(buffer, filePtr): writes the characters in the array pointed to by buffer to the indicated file until the terminating null character in buffer is reached.

int fscanf(filePtr, format, arg1, arg2,...,argn): reads data items from the file identified by filePtr, according to the format specified by the character string format.

int fseek(filePtr, offset, mode): positions the indicated file to a point that is offsed (a long int) bytes from the beginning of the file, from the current position in the file, or from the end of the file, depending upon the value of mode (an integer).

long ftell(filePtr): Returns the relatve offset in bytes of the current position in the file identified by filePtr or -1L on error.

printf(format, arg1, arg2,...,argn): writes the specified arguments to stdout, according to the format specified by the character string. Returns the number of characters written.

int remove(fileName): removes the specified file. A nonzero value is returned on failure.

int rename(fileName1, fileName2): renames the file fileName1 to fileName2. A nonzero value is returned on failure.

int scanf(format, arg1, arg2,...argn): reads items from stdin according to the format specified by the string format.

18.2.4. Conversion functions

To use these functions that convert character strings to numbers, you must include the header file stdlib.h.

double atof(s): Converts the string pointed to by s into a floating point number, returning the result.

double atoi(s): Converts the string pointed to by s into a int number, returning the result.

double atol(s): Converts the string pointed to by s into a long int number, returning the result.

double atoll(s): Converts the string pointed to by s into a long long int number, returning the result.

18.2.5. Dynamic Memory functions

To use these functions that allocate and free memory dynamically, you must include the stdlib.h header file.

void *calloc(n,size): allocates contiguous space for n items of data, where each item is size bytes in length. The allocated space is initially set to all zeroes. On success, a pointer to the allocated space is returned, on failure, the null pointer is retruned.

void free(pointer): Returns a block of memory pointed to by pointer that was previously allocated by a **calloc**, **malloc** or **realloc** call.

void *malloc(size): allocates contiguous space of size bytes, returning a pointer to the beginning of the allocated block if successful and the null pointer otherwise.

void *realloc(pointer,size): changes the size of a previously allocated block to size bytes, returning a pointer to the new block (which might have moved), or a null pointer if an error occurs.

18.3. Math functions

To use common math functions, you must include the math.h header file and link it to the math library.

double acosh(x): returns the hyperbolic arccosine of x, with x greater or equal to 1.

double asin(x): returns the arcsine of x, with x in the range -1 to 1. The angle is expressed in radians in the range $-\pi/2$ to $\pi/2$.

double atan(x): returns the arctangent of x. The angle is expressed in radians in the range $-\pi/2$ to $\pi/2$.

double ceil(x): returns the smallest integer value greater than or equal to x. The value is returned as double.

double cos(r): returns the cosine of r.

double floor(x): returns the smallest integer value lower than or equal to x. The value is returned as double.

double log(x): returns the natural logarith of x, with x greater than or equal to zero.

double nan(s): returns a NaN, if possible, according to the content specified by the string pointed to by s.

double pow(x,y): returns x^y. If x is less than zero, y must be an integer. If x is zero, y must be greater than zero.

double remainder(x,y): returns the remainder of x divided by y.

double round(x): returns the value of x rounded to the nearest integer in floating point format. Halfway values are always rouned away from zero (0.5 is rounded to 1.0).

double sin(r): returns the sine of r.

double sqrt(x): returns the square root of x, x greater than or equal to zero.

double tan(r): returns the tangent of r.

And so many more, there is a complex arithmetic library.

18.4. Utility functions

These functions are usually in the header file stdlib.h

int abs(n): returns the absoulte value of its int argument n.

void exit(n): terminates programme execution, closing any open files and returning the exit status specified by its argument n. EXIT_SUCCESS and EXIT_FAILURE are defined in stdlib.h. Other related routines in the library are abort and atexit,

char getenv(s): returns a pointer to the value of the environment variable pointed to by s or a null pointer if the variable does not exist. It is used to get environment variables.

void qsort(arr,n,size, comp_fn): sorts the data array pointed to by the void pointer arr. There are n elements in the array, each size bytes in length. n and size are of type size_t. The fourth argument is of type pointer to function that returns int and that takes two void pointers as arguments. qsort calls this function whenever it need to compare two elements in the array, passing ir pointers to the elements to compare.

int rand(void): returns a random number in the range 0 to RAND_MAX, where RAND_MAX is defined in stdlib.h and has a minimum value of 32767.

void(srand(seed): seeds the random number generator to the unsigned int value seed.

int system(s): gives the command contained in the character array pointed to by s to the system for execution, returning a system-defined value. system("mkdir/usr/tmp/data").

18.4.1. Assert library

The assert library, supported by the assert.h hader file is a small one desgined to help with debugging programmes.

It consists of a macro called **assert**, that takes as its argument an integer expression. If the expression evaluates as false (nonzero), the macro writes an error message to the standard error stream (stderr) and calls the **abort()** function, which terminates the programme. Example:

z = x*x - y*y;

assert(z>=0); // asserts that z is greater than or equal to zero.

18.4.2. Other useful header files

time.h: defines macros and functions supporting operations with dates and times.

errno.h: defines macros for the reporting of errors.

locale.h: defines functions and macros to assit with formatting data such as monetary units for different countries.

signal.h: defines facilities for dealing with conditions that arise during programme execution, including error conditions.

stdarg.h: defines facilities that enable a variable number of arguments to be passed to a function.

19. Conclusions

19.1. Further topics of study

There are still a few advanced concepts of C not studied in this course: more on data types (defining your own types with typedef), more on the preprocessor (string concatenation) more on void*, static libraries and shared objects, macros, unions, function pointers (pointers that point to a function), advanced pointers (pointers that point to a pointer), variable arguments of functions (variadic functions), dynamic linking (dlm_open), signals, forking and inter-process communication, threading and concurrency, sockets.

There are functions for restoring previous states: setjmp and longjmp. Other topics are memnory management, fragmentation, portability of the programme, interfacing with kernel modules, compiler and linker flags, advanced use of the debugger (gdb), profiling and tracing tools (gprof, dtrace, strace), memory debugging tools such as valgrind...

19.2. Course Summary

The main topics of the course are:

- CodeBlocks as environment
- **Basic Operators:** logical, arithmethic, assignment.
- **Conditional Statements:** Making decisions: if, switch.

- **Repeating Code:** looping: for, while, do-while.
- **Arrays:** Defining and initialising, multi-dimensional.
- **Functions:** Declaration and use, arguments and parameters, call by value vs call by reference.
- **Debugging:** Call stack, common mistakes, understanding compiler messages.
- **Structs:** Initialising, nested structures.
- **Character strings:** basics, arrays of chars, character operations.
- **Pointers:** Most important topic: definition and use, using with functions and arrays, malloc, pointer arithmethic.
- **Preprocessor:** #define, #include.
- **Input and Output:** command line arguments, scanf, printf.
- **File Input and Output:** Reading and writing to a file: fgetc, fgets, fputc, fseek, etc.
- **Standard C Library:** string functions, math functions, utility functions, standard header files.

The main course outcomes are the following:

- You are now able to write beginner C programmes.
- You are now able to write efficient, high quality C code: modular, low coupling (small number of dependencies), naming conventions, indentation.
- You are now able to find and fix your errors: you understand compiler messages and know how to use a debugger.
- You now know understand fundamental aspects of the C programming language.