An Introduction to C Programming

F. Safai Tehrani
(francesco.safaitehrani@roma1.infn.it)

International School of Trigger and Data Acquisition 2011

Document version 0.2 - 02 Feb 2011

Course Syllabus

The main aim of these notes is to provide the students participating to the International School of Trigger and Data Acquisition (ISOTDAQ) 2011 with enough understanding of C programming on Linux to fully profit from the school and its associated labs.

This is by no means intended to be a complete C programming course (or a programming course at all), since it mainly addresses the elements of C that are of direct interest for TDAQ systems. Hence we will be highlighting specific features of the language while barely mentioning other ones.

Some advanced subjects (multiprocessing and multithreading in particular) will be the subject of a school lesson, so they will be ignored in this note.

Course structure

This document provides the theoretical framework and a number of pointer to further treatment of the material. The student should review this material and attempt to solve the exercises before the beginning of the actual School. One of the initial lessons of the School will be dedicated to discussing the exercises and reviewing them. The student will be expected to be reasonably familiar with the C and Python languages (a note on Python will also be provided).

The content of this note is largely preliminary, since it is in its first draft. I do apologize in advance for the mistakes that you will find throughout this text. Feel free to contact me for comments, complaints and clarifications on the material or simply to point out the mistakes. If I get enough comments, I might release a updated release before the school begins. If that happens I will also provide pointers as to what has been changed between the releases so you can decide whether the new version would be useful for you or not.

You should read this text while on a computer, with a Linux environment ready to test the code provided and with an active Internet connection. A fundamental part of learning a language is writing programs to solve specific problems in it. To this end a number of exercises have been provided throughout the text.
The level of the exercises is tailored to your expected level of understanding of the C language, and the exercises should be attempted before moving further ahead with the material. Feel free to contact me if you need help.

A large number of links has also been provided, often to Wikipedia pages. You should take the time to at least skim these pages and maybe go back to them at a later time as you need deeper understanding of specific issues. Some links point to large amounts of material, and are provided for further reference only.

Tools and prerequisites

You need a running instance of Linux with the GCC compiler in order to test the provided code and implement the exercises.

If you are unable to or don't feel like performing a full installation of a Linux distribution, we suggest the use of one of the virtual machines (VM) provided by the CernVM project. This VM runs a recent version of the SL5 Linux distribution and can be freely downloaded from http://cernvm.cern.ch/cernvm/.

Two versions of the VM are available: Basic which is command-line only and lacks a GUI and Desktop which runs a minimal Gnome environment. The Desktop version is more resource-hungry,

but probably simpler to use for the absolute beginner. Both versions are perfectly suitable for this course.

At the time of writing the latest version of the CernVM is 2.1.0 and is distributed in various formats which are meant to be run using different virtualization platforms. All the most common platforms are supported: VMWare, VirtualBox and Parallels.

We suggest the use of VirtualBox which is freely available from http://www.virtualbox.org/ both for Mac and for Windows. At the time of writing the latest release of VirtualBox is 3.2.12.

Unless specified otherwise on the CernVM site you should always use the latest version of VirtualBox and of the CernVM. The CernVM site also provides an How-To procedure needed to configure the VM and to make it work. You will have to carefully review and follow this procedure in order for the VM to work correctly and be usable.

The current links to the HowTo-s are:

Virtual Box Quick Start for the Windows and Linux versions of VirtualBox and
Virtual Box On Mac Quick Start for the Mac version.

Please check on the CernVM website that these links are still current.

What you should know already and where to learn it: some pointers

You should already be familiar with the common programming concepts like variables, bounded and unbounded loops, boolean operators and so on. The specific syntax for these entities in C will be introduced, but the rationale behind them won't be discussed.

You should also be able to think in an algorithmic way: starting from a problem being able to reformulate it in terms of an algorithm to be implemented in a computer program.

You need to be able to perform common tasks with Linux (login, file and directory manipulation, process control and so on), and be able to create and edit a text file, using an editor of your choice. We will provide additional instructions for specific tasks (e.g. compiling a source file).

We offer the following links for self study, but your favourite search engine can find many more.

For an introduction to Linux check:

http://tldp.org/LDP/intro-linux/html/

For C programming:

http://www.cprogramming.com/tutorial.html

and look for the C Tutorials on this page.

The C Language

The C language was created between 1969 and 1972 by Dennis Ritchie while working at Bell Labs as a member of the ALGOL family of languages (via BCPL and B). It was meant to be a system implementation language, specifically thought for the implementation of the Unix operating system but it has been since used extensively as an application language. Well known examples of applications written in C are Mathematica and MATLAB.

C is a procedural/imperative language (as opposed to C++ which is object oriented), statically but weakly typed (variables have a type, but this type can change). It is usually compiled from source code to an executable (more on this later). It is a relatively small language (it has a small number of reserved keyword) whose more complex features are hosted in external libraries.

For a lot more about the C language and many additional pointers see:

http://en.wikipedia.org/wiki/C_language

Unix and Linux

The Unix operating system was created by a group of programmers working at AT&T's Bell Labs in 1969. The name Unix (originally Unics, as in 'single user') is a tongue-in-cheek reference to Multics, an earlier mainframe operating system which ran on much larger machines and supported multiple 'time-sharing' users. Unix was developed to run on a minicomputer (originally a Dec PDP-7).

An important part of the Unix technology is that, after an initial implementation phase, Unix was rewritten in C (architecture independent language) with minimal 'core' elements implemented in assembly language (which are architecture dependent). Previous operating systems had been developed in machine/assembly language and were completely architecture-dependent. A new architecture would then require a full rewrite of the operating system.

The Unix approach made it relatively simple to 'port' Unix to run on different architectures which, in addition to the Bell Labs policy of freely distributing the Unix source code to whoever asked for it, made Unix very popular in the academic and business world. This also created a large number of alternative implementations of Unix running on very different architectures (AIX, HP/UX, Ultrix, OSF/1, ConvexOS, IRIX and so on).

Fast forward to the early nineties, when the first version of the Linux kernel was released in the wild and used as the kernel of a fully open version of Unix, with most of the additional software needed to create an actual distribution coming from the Free Software Foundation's GNU project.

Fast forward again to 2010 and you'll find that Linux is extensively used in the scientific field as the main operating system for data processing and often also for data acquisition.

Some links about Unix (if you are a beginner, you should really check these):

http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/index.html

For more material about Unix, Linux and so on, see:

http://en.wikipedia.org/wiki/Unix
http://en.wikipedia.org/wiki/Linux
http://en.wikipedia.org/wiki/GNU_Project

Before we start: man pages

As mentioned above the student is supposed to be familiar with Unix/Linux, so this technical section might feel redundant. We feel it's important to briefly describe the Unix `man` facility, which is particularly relevant when developing C/Unix code.

The `man` (short for manual) command allows the user to access the documentation ("man pages") for most, if not all, Unix commands and library functions. The man pages are sometimes substituted with larger documentation accessible via the `info` command (`man info` if you want to know more about this). The man page usually points to the info documentation where available.

In addition to offering support for most commands (although the man page is barely sufficient for the more advanced tools that Unix provides), all the standard library functions are described. As an example let's get the man page for `printf`. In a C program, the function `printf` is used to produce formatted output. To check its syntax, we can run the command `man printf` which yields:

```
PRINTF(1)                       User Commands                       PRINTF(1)

NAME
       printf - format and print data

SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION

DESCRIPTION
       Print ARGUMENT(s) according to FORMAT.
....
```

which does not look right, especially since at the top it says 'User Commands' and `PRINTF(1)`. This is because , unless we specify otherwise, man first looks at the 'User Command' section (also known as Section 1), and as it happens there is a system command called `printf`. With unique names you would get the correct page, without having to specify the section.

To access function documentation you need to know that man pages are logically divided in sections (identified by numbers from 1 to 9 or letters) with the correct section number for functions being 3. Let's try again: `man 3 printf` yields:

```
PRINTF(3)                 Linux Programmer's Manual                 PRINTF(3)

NAME
       printf,   fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
....
```

which instead describes the C function `printf` (and many more similar ones). The heading "Linux Programmer's Manual" also indicates that we are looking at the correct documentation.

For a lot more about man and man pages, including the section numbers on various versions of Unix, take a look at http://en.wikipedia.org/wiki/Man_page.

`man man` also provides an useful reading. An important feature is the `-k` switch that implements a simple man page search facility. Say you are looking for the C compiler and don't remember the command name: you could try running `man -k compiler` which yields:

```
....
g77                  (1)  - GNU project Fortran 77 compiler
gcc                  (1)  - GNU project C and C++ compiler
gcc                 (rpm) - Various compilers (C, C++, Objective-C, Java, ...)
gcc [g++]            (1)  - GNU project C and C++ compiler
....
```

Each line contains the program name, the man page section and a brief one-line description of the command/function.

Before we start: printf

We will be using `printf` to perform screen output and see the result of most of our program. The function `printf` is a member of the standard C library together with a large number of other output functions. To use `printf` you need to `#include stdio.h`. All the members of the `printf` family share the same logical approach: they format their output via a very simply string description language. This is the syntax of `printf`:

```
printf("format string", arg1, arg2, ...);
```

Syntax of printf

`Printf` is a slightly uncommon function since it can have a variable number of arguments. Functions that can take a variable number of arguments are called variadic[1].

An example of `printf` use is:

```
printf("This is the output string: a number: %d, a character: %c\n", 42, 'a');
```

This prints:

```
This is the output string: a number: 42, a character: a
```

The output string contains special metacharacters, introduced by the `%` symbol, that describe what goes into that specific position. `%d` indicates an integer number, `%c` a character and so on.

Further format specifications can be added, like number field length, justification and so on. The replacement is purely positional: the first metacharacter is replaced with the first argument following the format string, the second with the second and so on. C will try to convert the argument into the correct type, but it might fail and create unpredictable results. Be sure to use the correct metacharacter for what you want displayed. Also note that the number of arguments must be equal to the number of metacharacters, or you will get an error.

Other metacharacters are used for output string formatting, like `\n` which indicates a newline.

[1] http://en.wikipedia.org/wiki/Variadic_function

For a full description of `printf`, the metacharacters and the formatting options see `man 3 printf` or http://en.wikipedia.org/wiki/Printf.

Lifecycle of a program: from source to executable

We start our overview of C programming by describing the full lifecycle of a program. First we write a very simple program. Tradition calls for this program to simply print the message “Hello, World!” upon execution. The source code is:

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

hello_world.c

There’s no need to worry right now about how the program works. Just type the code in your favourite editor (or simply cut-and-paste) and save it in a file called `hello_world.c`.

The process of turning a source code into an executable (that is: a program that can be run) is called *compilation*. Compilation, for C, is a multi-step process containing the following steps: preprocessing, compilation to an object file and linking. Different compilers can alter these steps (e.g. there might be two steps in the compilation phase), but this is usually transparent to the user.

Note that even though this time we will look at each step of the compilation process in detail, running them one-by-one, this is not necessary as the standard compilation command runs all the step implicitly producing an executable directly from a source file. Also, in general when dealing with more complex programs (which often contain tens of thousands or more of source files) the compilation steps are performed by other programs (like make or cmt) which are used to automate complex compilation processes.

Preprocessing

The preprocessing step takes care of converting preprocessor directives (the instructions which start with a # character) into C code. Preprocessor directives can be used to include header files (#include), define symbols (#define) and a large number of other tasks. We will address the most common usage of these directives later on, but advanced features like macros will be completely left out.

The preprocessing step is performed by the `cpp` command (short for C pre-processor). Running cpp hello_world.c produces a lot of output:

```
# 1 "hello_world.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello_world.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
....
```

This code contains the original code from our file plus a lot of additional code generated by processing the `#include <stdio.h>` directive. Very rarely it might be necessary to examine the result of the preprocessing by hand to solve some peculiarly complex error, but most of the time this step is completely transparent. The output of this step is still source code.

The preprocessor can also be invoked from `gcc` using the `-E` command switch.

Compilation

This step transforms the C code into assembler code, which is a direct representation of CPU instructions in some form of mnemonic code. The output of this step is a .s file containing the assembler source code.

The compiler step is be invoked using the `-S` switch: `gcc -S hello_world.c` generates `hello_world.s`. An excerpt of the content of `hello_world.s` is:

```
        .file   "hello_world.c"
        .section        .rodata
.LC0:
        .string "Hello, World!"
        .text
.globl main
        .type   main, @function
main:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
....
```

Assembler

This step compiles the assembler source code into an object file. An object file contains code that has been transformed (“compiled”) into an executable representation, but lacks the hooks to connect to system libraries. Usually object files have a `.o` extension and are binaries (not text).

The assembler step of the C compiler is invoked via the `gcc` command, and using the `-c` switch it does not perform the link step giving us just the object file. Running `gcc -c hello_world.c` produces `hello_world.o`.

Linux also provides a standalone assembler `as` which can be used to compile assembler code directly: `man as` if you’re interested.

Link

The link step ‘connects’ the object file to the operating system libraries, thus transforming it into an executable. This step is performed by a linker, which on Linux is called `ld`. Invoking `ld` by hand is extremely delicate and complex, requiring detailed knowledge about the version of the compiler, the operating system and other details. Luckily `gcc` can do this work for us, and link our object file into an executable. This step is performed using the `-o` switch followed by the name we want to give to the executable: `gcc -o hello_world hello_world.o` links `hello_world.o` producing the executable `hello_world`.

Let's look at the steps we've performed so far and the result in the printout of a session:

```
~> ls -l
-rw-r--r--  1 safai zp     107 Dec 11 15:10 hello_world.c
~> gcc -S hello_world.c
~> ls -l
-rw-r--r--  1 safai zp     107 Dec 11 15:10 hello_world.c
-rw-r--r--  1 safai zp     949 Dec 11 15:10 hello_world.s
~> gcc -c hello_world.s
~> ls -l
-rw-r--r--  1 safai zp     107 Dec 11 15:10 hello_world.c
-rw-r--r--  1 safai zp     949 Dec 11 15:10 hello_world.s
-rw-r--r--  1 safai zp    1512 Dec 11 15:10 hello_world.o
~> gcc -o hello_world hello_world.o
~> ls -l
-rwxr-xr-x  1 safai zp    6725 Dec 11 15:10 hello_world
-rw-r--r--  1 safai zp     107 Dec 11 15:10 hello_world.c
-rw-r--r--  1 safai zp     949 Dec 11 15:10 hello_world.s
-rw-r--r--  1 safai zp    1512 Dec 11 15:10 hello_world.o
~> ./hello_world
Hello, World!
```

How to compile for real

As mentioned above, there is a way to invoke gcc which performs all of the above steps transparently, going from source code to executable using a single command.

We start from `hello_world.c` and issue the command `gcc -o hello_world hello_world.c`. This will produce the `hello_world` executable with a single command execution.

An important caveat: `gcc -o hello_world.c hello_world.c` will compile the source program `hello_world.c` into an executable program `hello_world.c` **overwriting** your original source code. No warnings, no ifs, no buts. You have been warned.

For a more detailed look at the entire compilation process, with (a lot) more information about the details see:

http://www.lisha.ufsc.br/teaching/os/exercise/hello.html

The structure of a C program

A C program is simply a set of instructions that are executed in sequence. Program execution starts with the 'first' instruction and end with the last.

C program execution always starts at a special entry point known as the `main` function. The instructions in the `main` function are executed first to last, transferring control of the execution flow as needed (typically during a function invocation). If a program finishes correctly (that is: it does not crash due to an error or unforeseen problem) the last instruction to be executed is the `return` instruction at the end of the `main`.

Let's look again at `hello_world.c` and discuss the various parts of the code:

```c
#include <stdio.h>

int main(int argc, const char * argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

The first line is a preprocessor directive, instructing gcc to load the content of the file `stdio.h`. Somewhere in your system, there is a text file named `stdio.h` (a common place for this is `/usr/include`). This kind of file is known as a *header file* and contains a list of function and symbols that are used to perform screen oriented I/O. Directives can be interspersed throughout the code as needed, although `#include`s and `#define`s are usually located at the beginning of source files.

Next there is the program entry point: the `main` function. We will worry about how to define a function and the meaning of all the elements on this line at a later time. For now, it's simply the way we define the `main` function. Note that at the end of the line there is a `{` character, which is matched by a `}` at the end of the code. The curly brackets `{` and `}` indicate respectively the beginning and the end of the body of a function. The instructions contained between the `{` and the `}` describe what the function does.

The group of instructions contained between curly brackets is also known as a code block and is logically equivalent to a single instructions, that is to say that wherever I can put a single instruction, I can alternatively put a group of instructions grouped between curly brackets.

Now we are inside the body of the main function, that in this specific case contains two instructions: a call to the `printf` function and a `return 0` instruction. All instructions in C are closed with a semicolon (`;`), so the line `printf("Hello, World!\n");` is an instruction and the line `return 0;` is another.

The end of the `hello_world.c` file informs the compiler that there's nothing more to do for this program.

Large C programs are usually organized in a number of source (.c) files which are referenced via their associated header files (.h). The code is logically organized in smaller, more manageable units (modular programming) which in C take the shape of functions. Each function can, in turn, transfer control to other functions (like the main function invoking the printf function) once it finishes its work or simply return control to the calling function.

A first approximation to the philosophy of software development (and software engineering in general) is: the developer has to decide how to break down the problem in logical 'units' which can then be mapped onto language entities (functions, objects, ...). The design needs to address the problem's complexity while at the same time producing code that is correct and easily manageable.

The elements of C

We are now going to look at the syntax of C.

Comments

Comments in C code are obtained by surrounding test between `/*` and `*/` like this:

```
/* This is a comment */
```

Comment syntax

All the text between `/*` and `*/` is ignored by the compiler.

Some preprocessor directives

We have already met one of the most common preprocessor directive `#include` which is used to import a header file into the current file. Importing a header file allows us to use library functions and in general modularize our code.

The syntax is `#include <library.h>` or `#include "library.h"`. There is an important difference between the two forms:

```
#include <lib.h>  /* look for lib.h in the default system include directory */
#include "lib.h"  /* look for lib.h in the current directory                 */
```

#include syntax

It is also possible to specify an explicit path (partial or full) in the `#include` directive.

The `#define` directive associates a symbol to a value. During the preprocessing phase the symbol is replaced by the value in the code. The syntax is:

```
#define SYMBOL value
```

#define syntax

Another useful directive is the one used for conditional compilation:

```
#ifdef SYMBOL
<lines to compile if SYMBOL is defined>
#else
<lines to compile if SYMBOL is undefined>
#endif
```

#ifdef syntax

The lines between `#ifdef` and `#else` are used if `SYMBOL` is defined, while the lines between `#else` and `#endif` are used otherwise. This process happens before the compilation, in the preprocessing step hence the compiler simply ignores the unselected code. `SYMBOL` is usually provided on the command line that was used to invoke the compiler via the `-D` `gcc` switch.

Variables and variable types

In programs it is usually necessary to create 'spaces' that we can store the logical entities that we need to implement our algorithm. These 'spaces', called variables since their value can change

throughout the execution of our program, have three (four really, but we'll get back to that later) important properties:

- a name: a sequence of number and characters used to identify it
- a type: a language specifier used to indicate what kind of entity can be stored in the space
- a value: the entity stored in the space

Let's look at an example:

```
int a = 21;
int b = a + 21;
```

The first 'space' is named `a` and we tell the compiler that we want to store `int`eger numbers in it. Using the `=` operator, we fill the `a` space with the number `21`. Next we create a new 'space', call it `b` and then use it to store `a + 21`. This tells us that the 'space' named `a` is simply a placeholder for its value. A more common way to do this in an actual C function would be:

```
int a, b;
a = 21;
b = a + 21;
```

This time we have split the variable *declaration*, that is the place where we associate a name with a type, from its *initialization*, that is the place where we associate the name with a value. The first line declares `a` and `b` as variables of type `int`. The next line initializes `a` with `21`, and the third line initializes `b` to the value of `a + 21`. As usual the semicolon closes all the instructions.

Note that C requires that all variable declaration are placed at the very beginning of a function body.

The variable type tells the compiler what kind of entity we plan on storing in the variable, so that the compiler can complain when we try to store the wrong kind of entity. C is a relatively easy going language (more formally: C is weakly typed, it has types but it can quietly convert one type into another to complete a computation without complaining), so it actually allows us to freely mix types in our operations as long as that makes sense.

It is also possible to explicitly specify that we want the content of a variable to be seen as one of a different type via an operation called a *cast* (as in: we cast an integer variable to a char). This is often necessary for complex memory manipulations and will be addressed in detail at a later time when we'll talk about memory management and pointers. A simple example is still needed here to talk about a peculiar (as in: different from expected) behavior of C: the integer division.

Let's write a bit of code:

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int a = 1;
    printf("1/2 = %f\n", a/2);
    return 0;
}
```

integer_division.c

Compiling and running this program produces this output:

```
1/2 = 0.000000
```

which does not look correct. What's happening? C has found a division between two integers and it treats it as an integer division, resulting in 0. Intuitively we would expect to get 0.5, but to get that we need to explicitly tell C to consider this a real division. This can be done by changing the division to:

```
    printf("1/2 = %f\n", a/2.);
```

The `.` after the 2 indicates that the constant 2. is float rather than integer (this is an implicit cast), hence triggering a real division. Another alternative is via an explicit *cast*, that is telling C that even though a has been defined of type `int`, we really want it to be used as a `float`:

```
    printf("1/2 = %f\n", (float)a/2);
```

The cast approach is more general, especially when you don't have any numeric constant in your calculation to perform an implicit cast.

Now let's briefly look at the most common default data type that C offers:

- `char`, `short`, `int`, `long`, `long long`: these all represent an integer number.

The main difference between these is the range, that is the minimum and maximum number that can be stored in a variable of this type. A `char` is 8 bit long, hence it can store numbers between -127 and 128 (or 0 to 255 if unsigned), a `short` is 16 bit long so it can store a number between -32767 and 32768 (or 0 to 65535 if unsigned) and so on.

- `float`, `double`, `long double`: these all represent a floating point number

As for integer types, the main difference is in the bit length, hence in the range and precision of the number which can be stored in a variable of such type. All these types can also be specified as unsigned to indicate that the values stored in them are always positive.

A char variable can also represent a character constant, via the ASCII code of the character, like this:

```
char f = 'F';
```

The `f` variable now contains 70 which is the ASCII code of the letter `F`.

C also supports other variable types and more complex structures, but we won't deal with those. The Wikipedia page for the C syntax contains a lot more details about the supported data types:

http://en.wikipedia.org/wiki/C_syntax

and with further details:

http://en.wikipedia.org/wiki/C_variable_types_and_declarations

including the ranges for the various data types.

Anybody who deals with numerical computing should also take a look at 'What Every Scientist Should Know About Floating-Point Arithmetic':

http://docs.sun.com/app/docs/doc/800-7895/.

Arrays

It is often necessary to work with collections of variables of the same type (like all the values from a specific measurement). To do that, C provides the array data structure. An array is a collection of 'spaces' all of the same type, indexed via an integer number and of predefined length. An array is defined like an ordinary variable, but with the added specification of the array size appended after the variable name, like this:

```
int distance[256];
```

The variable `distance` is now an array of `int`egers, that is a collection of 256 `int`s. Accessing an element of an array is done via the `[..]` operator, like this:

```
distance[128] = 345;
distance[129] = distance[128] + 1;
```

These command set the 129th element of the distance array to 345, and the 130th to 346.

Array indexes are 0-based, so the first element of `distance` is `distance[0]` and the last one is `distance[255]`. Trying to access `distance[256]` (or larger indexes) will cause your program to crash with a 'Segmentation Fault' error.

The array size is defined at declaration time and cannot be changed if a larger array is needed, hence it's common when doing defensive programming to specify the array size via a `#define`:

```
#define SAMPLES 44100
...
float sample[SAMPLES];
...
```

So if at a later time we need to raise the number of our samples to 96000, we won't need to modify the code, but only change the `#define`-d value. This works even better if we are consistent in our use of `#define`-d symbols. More on this later. Also a lot more about arrays will be said when we talk about memory management and pointers.

Structs

It is often useful to logically organize simple types into more complex ones to simplify code. C offers the possibility of creating data structures (`struct`s) to group entities. In addition to `struct`s, C provides also `union`s and `bitfield`s, that we will not address. You can find some information about them in the Wikipedia C syntax page, and a lot more can be found with your favourite search engine.

A `struct` is a collection of simple data type. Let's define a `sample` entity containing an integer value (the sample index), and two float values: the sample value and the sample error:

```
struct sample_s {
    int index;
    float value;
    float error;
} aSample;
```

This code defines a structure named `sample_s` containing three fields: `index` (int), `value` and `error` (float). It also defines a variable of type `sample_s` called `aSample`. To access the fields we can use the `name.field_name` syntax:

```
aSample.index = 255;
aSample.value = 3.14;
aSample.error = 0.03;
float maxValue = aSample.value + aSample.error;
```

To create new variables of type `sample_s` we need to inform the compiler that `sample_s` is a struct by writing:

```
struct sample_s anotherSample;
```

We can also use the `typedef` facility of C, which creates new datatypes, and change our original definition to:

```
typedef struct sample_s {
    int index;
    float value;
    float error;
} sampleType;
```

Now we will be able create new variables of type `sampleType` directly.

Flow control

Now we know how to create spaces to store values, and how to name and organize them, so it's time we move to control the program flow. The flow of a program is usually controlled via tests:

- `if condition_is_true do_this otherwise do_that`

bounded iteration

- `do this this many times`

or unbounded iteration

- `do this until this condition is true/false/...`

Tests and logical operators

Tests in C are introduced by the keyword `if` followed by a *predicate* (a logical condition that can be true or false), a code block to be executed if the predicate is true (the *then clause*) and, optionally, the keyword `else` followed by a code block to be executed otherwise (the *else clause*):

```
if (predicate) {
    do this if predicate is true;
} else {
    do this if predicate is false;
}
```

if keyword syntax

Let's look at some real code:

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int a = 42;
    int b = 21;
    if (a>b) printf("a is greater than b\n");
    else {
           printf("b was greater than a ... but not anymore!\n");
           a = b+1;
    }
    return 0;
}
```

if_test.c

The predicate `a>b` is true, so the instruction `printf("a is greater than b\n")` is executed. Note how in the body of the `else` we have specified two instructions, so we had to surround them with curly brackets to turn them into a code block, while the then clause contained only one instruction so it did not need curly brackets.

Operators which take a single argument are called unary operators and operators that take two arguments (like `+`, `-`, ...) are called binary operators.

All the common math comparison operators are supported as in the following list:

- `>` greater than
- `<` lesser than
- `>=` greater or equal
- `<=` lesser or equal
- `==` equal (beware of the double =, a single = works but does something completely different!)
- `!=` not equal

In addition to these there are the logical operators which are used to create more complex boolean conditions:

- `c1 && c2` (and): true is both `c1` and `c2` are both true[2]
- `c1 || c2` (or): true if at least one of `c1` or `c2` is true
- `!c1`: negates its argument: true if `c1` is false, and vice versa

C has a special form of `if` for assignments called the *ternary operator*:

```
aVariable = condition?value_if_true:value_if_false;
```

the value stored in `aVariable` depends on the condition. If condition is true then `aVariable` will contain `value_it_true`, while it will get `value_if_false` otherwise.

For a lot more about C operators (including the bitwise ones and notes on the operator precedence) see:

[2] Trueness in C: C does not have a dedicated entity/datatype for the logical True/False, hence anything that is not zero/`NULL` is true, zero/`NULL` is always false.

http://en.wikipedia.org/wiki/Operators_in_C_and_C++.

It is also possible to use the `switch` statement to select between different possibilities. See the C syntax page for a description of `switch`.

Iteration

Bounded iteration in C is implemented via the `for` keyword. The syntax for `for` is:

```
for(initialization; continue_iteration_predicate; increment) {
     instructions_to_repeat;
}
```

for keyword syntax

The initialization clause sets the loop index to its initial value, the increment clause specifies how the loop index should be increased while the predicate is a logical condition that causes the loop to continue until it becomes false.

In practice:

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
     int index;
     for (index=0; index<10; index=index+1) {
             printf("Index is at %d\n", index);
     }
     return 0;
}
```

count_to_ten.c

The integer `index` variable is initialized to 0 and the beginning of the loop, at each step of the loop the `index` variable is increased by one and the loop is repeated as long as `index` is less than ten. The value of the increment, and more in general, the increment technique can vary widely.

A few additional notes:

- the operation `index=index+1` means: take the value of index, add one to it and replace it as the new value of `index`. Mathematically incorrect, but it makes perfect sense in the context of programming.
- since operations like `index=index+increment` are very common, C supports the augmented assignment operators `+=`, `-=`, `*=` and so on. So instead of writing `index=index+increment`, we can write `index+=increment`. See:

http://en.wikipedia.org/wiki/Augmented_assignment

- the self-increment-by-one (and decrement) operation is so common that it has its own special dedicated operators: `++` and `--`. So `index=index+1` can be written as `index++` or `++index`.
- the self-increment and self-decrement operators can be used prefix or postfix with a slightly different meaning. The underlying idea is when the increment actually happens: with the prefix operator first you use the value *then* you increment, while with the postfix operator it's vice versa. Look for the `prefix_postfix_increment.c` example a little later in the text.

Unbounded iteration is implemented via two different constructs: the `while` statement and the `do ... while` statements. Both take a logical condition as argument and keep repeating their associated code block while the associated logical condition remains true. The syntax is:

```
while (condition) {
    instructions;
}
```

while keyword syntax

```
do {
    instructions;
} while (condition);
```

do ... while keyword syntax

The main difference between the two keywords is that the `while` body might never be executed if the condition is found to be false at the beginning of the `while`, while the `do ... while` body gets executed at least once.

Unless we are trying to produce an infinite loop, something within the `while` / `do ... while` body should modify the variables involved in the logical condition. In code:

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int index=0;
    while (index<10) {
            printf("index: %d\n", index);
            index++;
    }
    return 0;
}
```

upTo10.c

The value of `index` is modified within the loop, so the while loop finishes when its value hits ten.

Functions

The logical unit to modularize the C code is the function. We have already met two functions: `main` and `printf`. From a logical standpoint we can see a function as a black box: an entity that accepts some arguments and returns a value. We don't really need to know that happens within the function body to use it. We know that `printf` accepts arguments and its effects is to print its arguments on the terminal console. There is no need for us to discover how that actually happens, or even if `printf` does the actual job or simply delegates it to some other function.

Just like with `printf`, we would like to be able to write our code organizing it in functions that we will simply invoke when we need them to do their work.

A function is identified by four elements:
- its name
- its return type
- its argument list (also known as the function signature or interface)
- its body

The syntax for a function definition is:

```
return_type function_name(arg1, arg2, ...) {
     function_body;
     return value_to_return;
}
```

function definition syntax

Let's define a simple function that calculates the sum of three numbers:

```
#include <stdio.h>

float sum3(float v1, float v2, float v3) {
     float tmp = v1+v2+v3;
     return tmp;
}

int main(int argc, const char * argv[]) {
     float rv = sum3(1.41, 2.73, 3.14);
     printf("sum3: %f\n", rv);
    return 0;
}
```

sum3.c

The function `sum3` takes three float arguments (locally called `v1`, `v2` and `v3`) and sums them, returning their sum. The correspondence between the formal arguments and the value in the invocation is positional: `v1` takes the value `1.41`, `v2` takes `2.73` and `v3` takes `3.14`.

The `return` keyword "returns" the value, which is then taken to be 'the value of the function'. From this standpoint C functions behave like their mathematical counterparts: `y = f(x)`, `y` takes the value of `f` calculated in `x`.

Is is legal to invoke a function and ignore its return value (like with `printf`). It is also possible to define a function that does not return a value by indicating its return type as `void`.

In general, before using a function, the C compiler needs to be able to see its definition. This might create a problem when using library functions, whose code is usually not immediately available. This is solved by including in the source code the function prototypes which are forward declarations that inform the compiler about the return value and the signature of a function before you use it.

It is not actually necessary for the compiler to see the function code until your program runs. This is the idea underlying libraries. Libraries are collections of precompiled functions that your program can use (`printf` is an example of this). To use `printf` in your program you just need to declare its prototype somewhere and then, at link time (static library) or at run time (dynamic library) the actual code will be pulled in. Libraries are usually provided with special files containing the prototypes for the library functions in the form of *header*/*include* files. So what we are really doing with the line `#include <stdio.h>` at the beginning of our code is pulling in the function prototypes for the I/O oriented functions of the standard (`std`) library. We can also create our own libraries and header files for our own private libraries as we'll demonstrate later on.

Now we can go back to the issue of the prefix/postfix self-increment/decrement operator. Let's repeat what was said when we introduced them:

- The underlying idea is when the increment actually happens: with the prefix operator first you use the value then you increment, while with the postfix operator it's vice versa

To display this behavior clearly we implement a simple function (`pp`) that prints out its numeric argument and then invoke it once with prefix self-increment and once with postfix self-increment:

```
#include <stdio.h>

void pp(int datum) {
     printf("%d", datum);
}

int main(int argc, const char * argv[]) {
    int index,i2;
    for (index=0; index<10; index++) {
            i2 = index;
            printf("index: %d -- pp(index++): ", index);
            pp(i2++);
            i2 = index;
            printf(" -- pp(++index): ");
            pp(++i2);
            printf("\n");
    }
    return 0;
}
```

prefix_postfix.c

Compile and run this program and you can see the different effect of the prefix and postfix self-increment operators. This works because the increment happens before function invocation (with the prefix notation) or after function invocation (with the postfix notation). The consequence is that the function receives the value of `i2` already incremented (prefix) or not yet incremented (postfix).

Recursive functions

It is sometimes useful to define functions in terms of themselves, that is define a function that performs its work by doing some operations and then calling itself again. This is called recursion. The default example of this is the factorial function:

```
x! = 1 * 2 * ... * x
```

The only detail we need to be careful about is that the computation needs to finish, otherwise we will create an endless loop that will cause our program to crash (the precise reason why it crashes will be explained later on). Let's implement the factorial function in an iterative and in a recursive fashion:

```
#include <stdio.h>

int recursive_factorial(int num) {
     if(num==1) return 1;
     return num * recursive_factorial(num-1);
}
```

```c
int iterative_factorial(int num) {
     int i, result=1;
     for (i=num; i>1; i--) result *= i;
     return result;
}

int main(int argc, const char * argv[]) {
     printf("Iterative 6! :%d\n", iterative_factorial(6));
     printf("Recursive 6! :%d\n", recursive_factorial(6));
    return 0;
}
```

factorial.c

In both algorithm we have used backward iteration, going from `num` to 1. This makes the recursive approach somewhat simpler. The following exercise will help you analyze the advantages and disadvantages of the two approaches:

Exercise 1. Factorial

Rewrite both the iterative and recursive factorial functions using forward counting (from 1 to num).

For the next exercise, you will need to know about Fibonacci numbers. If you're not familiar with them, be sure to read this:

http://en.wikipedia.org/wiki/Fibonacci_number

Exercise 2. Fibonacci numbers

The formula to calculate the n-th Fibonacci number F_n is:

$F_n = F_{n-1} + F_{n-2}$
$F_0 = F_1 = 1$

Write a program containing two functions (one iterative and one recursive) which take `n` as arguments and return F_n.

Function prototypes and headers

We have already mentioned and used header/include files, mainly in the context of system libraries. A large program will use many system libraries, but it will also create some of its own so you should be able to correctly create and use header files.

Let's create and compile a program, splitting some of its functionalities into a library/header file. To do that we recover the `factorial.c` program and factor out its functionalities. First we factor out the factorial functions to a file:

```c
#include "fact_lib.h"

int recursive_factorial(int num) {
    if(num==1) return 1;
    return num * recursive_factorial(num-1);
}

int iterative_factorial(int num) {
```

```
    int i, result=1;
    for (i=num; i>1; i--) result *= i;
    return result;
}
```

fact_lib.c

Next we need to create the header file for this code. The header file contains simply the prototype of the functions in `fact_lib.c`:

```
int recursive_factorial(int);
int iterative_factorial(int);
```

fact_lib.h

Note how the function prototype simply needs to specify the types but not the names of the formal parameters. It is also legal to specify the name, but it is useless.

Now let's create another function to print out the result of the evaluation of 6!:

```
#include "print_lib.h"

void calc_fact_6() {
    printf("Iterative 6! :%d\n", iterative_factorial(6));
    printf("Recursive 6! :%d\n", recursive_factorial(6));
 }
```

print_lib.c

and its associated header file:

```
#include <stdio.h>
#include "fact_lib.h"

void calc_fact_6();
```

print_lib.h

Finally, let's create the main program:

```
#include <stdio.h>
#include "fact_lib.h"
#include "print_lib.h"

int main (int argc, const char * argv[]) {
    calc_fact_6();
    return 0;
}
```

fact6.c

Before we analyze the code, let's see how we have to compile this program. The 'lib' files cannot be compiled to an executable on their own, since they lack a `main` function. What we do is compile them to object files:

```
~> gcc -c fact_lib.c print_lib.c
~> ls -la *lib.o
```

```
-rw-r--r-- 1 safai zp 1528 Dec 18 18:16 fact_lib.o
-rw-r--r-- 1 safai zp 1744 Dec 18 18:16 print_lib.o
```

Now we just need to link them into our main executable. To do that, we inform `gcc` that we want to use them:

```
~> gcc -o fact6 fact6.c print_lib.o fact_lib.o
```

So we have produced the `fact6` executable, which contains and uses the 'library' functions `recursive_factorial`, `iterative_factorial` and `calc_fact_6`.

The library infrastructure is completely described by the interplay between include files and include directives. Note how each 'library' `#includes` its own header: this is to ensure that header and implementation are in sync with regards to function signatures. The compiler would in fact signal an error if we were to try to create a function with the wrong signature. Next we should note that to use a function from a library, we need its prototype. So, to use `calc_fact_6`, we need to include `print_lib.h`, which in turn includes the headers for the functions that are used within `print_lib.c`.

Also note how to include headers that are in the current directory we have surrounded the header name with double quotes rather than brackets:

```
#include "fact_lib.h"
```

instead of

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

As a side note, headers tend to contain a lot more stuff when writing C++ code. Sometimes this is described as: headers contain *interface* code and source files contain *implementation* code. This will be a lot more interesting when you will learn some C++.

Let's suppose that we now want to distribute our library, without distributing the source code. We can simply hand over to other developers the object files (organized in some way) and the relevant header files. That is all other developers need to use our code. Collections of function objects can also be organized in special files known as static or dynamic libraries, which then can be linked against your executables. The main difference between a static library and a dynamic library is when the library is 'connected' to the program.

With a static library the connection between the executable and the library happens at link time, creating what is called a monolithic executable. A monolithic executable contains its entire environment within the executable file and does not need any additional file to run. While this might seem desirable the shortcomings are the big (sometimes very big) file sizes of the executable which cause large load times and in general make the program more difficult to transfer between machines.

The alternative is a dynamic (or, in Unix speak, shared objects) library: is this case external functions are loaded at run time 'as needed'. The executable is much smaller since it does not need to contain all of the code that is used, but it has a strong dependency on the operating system environment. This also diminishes code replication, since programs share a single copy of the library code, without needing to include one within their own executable.

Another widely used name for dynamic libraries is DLL, dynamic link libraries. Programs depending on dynamic libraries are, by far, the most common on a modern Unix/Linux system.

The creation of static and dynamic libraries is vastly outside the scope of the present document, but the interested reader can check:

http://en.wikipedia.org/wiki/Library_(computing)
http://en.wikipedia.org/wiki/Static_library
http://en.wikipedia.org/wiki/Dynamic_loading
http://tldp.org/HOWTO/Program-Library-HOWTO/index.html
http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Exercise 3. Unit conversion library

For this exercise you will implement a set of unit conversion library. You can find the conversion factors and algorithms online.

Start with a library to convert centimeters to inches, meters to feet and vice versa, then add miles to kilometers. Add as many as you want.

Now create another library to convert weights, and implement the conversion between kilograms and pounds. Add as many as you want.

Now create a library to convert between different temperature scales, Celsius to Fahrenheit and vice versa, Celsius to Kelvin, Kelvin to Fahrenheit and so on.

Create a test program to use these libraries and print various conversions. Check that the result are correct.
Now, unless you've done some design in advance, you will find yourself with a lot of functions which do exactly the same thing (more or less).

Would it be possible to rewrite your conversion libraries to minimize code repetition, maybe by implementing some utility functions in a special dedicated library? (Utility functions are functions which solve a specific problem in a more general way).

Rewrite your libraries to maximize code reuse. Is it simpler now to add new conversions? Discuss your solution.

Unix programming

In this section we will provide some basic information regarding programming in the Unix environment. Due to the limited amount of space, most of the material is meant to be informative rather than technical. Pointers will be provided to more technical documentation.

Before reading on, you should read the Wikipedia page about the Unix architecture:

http://en.wikipedia.org/wiki/Unix_architecture

Tools, pipes and default I/O streams

When developing software, and in general working with Unix, you will often be working with the command line (in a shell). Unix provides a large number of utility applications, most of which follow the so called 'tool philosophy': rather than implementing large applications for generic tasks, it is preferable to implement small, very specialized applications ('tools') for specific tasks (check the `yes` application for an extreme example of this). These applications can then be 'connected' in a 'tool chain' at the command line to perform more complex tasks via pipes and redirection.

An example will help clarify this approach. To understand what follows you need to know that `ls` lists the content of a directory, `grep` selects text lines in its arguments (which can be files or streams) conforming to a specific pattern, and `wc` counts characters, letters or words. Reading the man pages for `ls`, `grep` and `wc` provides more details, especially for `grep` which is a very powerful command.

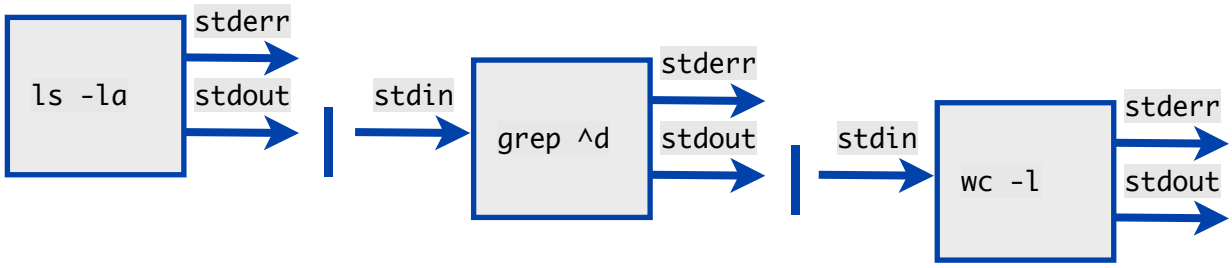
Let's count the number of subdirectories contained in the current directory:

```
ls -la | grep ^d | wc -l
```

How does this work? Each tool has three standard channels (streams) that it offers: `stdin`, `stdout` and `stderr`. A tool should read its input from `stdin`, print its regular output to `stdout` and its error messages to `stderr`. Each stream is directly accessible via redirection (check the documentation for the shell you are using if you don't know about redirection) and via pipes. The `|` character is called pipe and provides the service of connecting the `stdout` stream of the application on its left to the `stdin` stream of the application on its right.

So, rereading the command above: `ls -la` prints out the complete list of files in a directory to its `stdout`. If the `stdout` is not piped into another application, the output gets printed to your terminal window. This time we have plugged it into `grep`'s `stdin`. So `grep` 'reads' the data stream from `ls`, filters is by selecting all the strings that start with 'd' and prints them to its `stdout`[3]. `grep`'s `stdout` is piped into `wc`'s `stdin` which is instructed by the -l switch to just count lines. `wc` then prints to its `stdout` the number of lines it counted and this number ends up on your screen.

[3] The ^d in grep is a regular expression (regex), where ^ indicates the beginning of a string and d is simply the letter d, so this means "all the strings that begin with d". Regular expressions are very powerful and useful so you should probably spend some time learning how to build and use them. The main shortcoming is that different languages format their regexes differently so that will require additional work. A good starting point can be found on Wikipedia: http://en.wikipedia.org/wiki/Regular_expression



You might wonder why there is not `stdin` stream in `ls`. While it is common for tools to implement and consistently use all three streams it is by no means mandatory. Some applications simply cannot take 'input' from other applications. More unpleasant are applications which use non-standard output techniques, hence their output cannot be filtered, redirected and in general manipulated. Luckily they are very uncommon. When developing software, you should always wonder about the potential use of your program and if they might be needed in a tool chain and be a good Unix citizen and implement your standard streams correctly and consistently.

Programs that do their work mainly by taking input from the `stdin` and writing their output to the `stdout` are also known as *filters*.

The Unix standard library provides all the facilities that are needed to implement, manage and use pipes, which can be especially useful when used programmatically to interface multiple processes and threads within your programs. The IPC (Inter-Process Communication) page on Wikipedia offers a glimpse of the available facilities:

http://en.wikipedia.org/wiki/Inter-process_communication

which is by no means exhaustive.

Concurrency and concurrent programming, of which IPC is a special case, is an active research field, often spawning new languages and technologies that take a while to find their way into the mainstream. Again some Wikipedia pages offer a very basic entry point to the field:

http://en.wikipedia.org/wiki/Concurrency_(computer_science)
http://en.wikipedia.org/wiki/Dining_philosophers_problem
http://en.wikipedia.org/wiki/Concurrent_computing

For a more general view (not Unix specific) about pipes and pipelines:

http://en.wikipedia.org/wiki/Pipes_and_filters

For Unix pipes and pipelines:

http://en.wikipedia.org/wiki/Pipeline_(Unix)

For something useful you can actually do with pipes and Unix:

http://en.wikibooks.org/w/index.php?title=Ad_Hoc_Data_Analysis_From_The_Unix_Command_Line

Everything is a file

Another core tenet of the Unix philosophy is the 'everything is a file' statement. This indicates the the file metaphor is used an an abstraction layer to implement all 'parts' of Unix. The advantage of modeling all entities as files is that it provides a unified interface to everything. So we

can ‘open’ and ‘read’ a file that can really be a file (a collection of bytes on a filesystem), a device (e.g. a USB port) or a socket (a network connection).

From the hardware standpoint this has interesting consequences. A hardware device gets interfaced with the operating system (OS) via an element of the kernel (called module). The kernel module acts like a device driver exposing standard operations that allow the OS and the users to access and use the device. Devices are logically mapped to element of the filesystem mimicking files which are usually stored in the `/dev` filesystem. Using such a device can be as easy as ‘opening’ the file for I/O, and then performing standard ‘read’ and ‘write’ operations.

The file abstraction breaks when it comes to operation which would be file-like (e.g. like searching in a file) that do not really make sense with some device, like a printer. Still the abstraction is consistently used within the system and makes working with external devices a lot easier.

Another interesting part of Linux is the /proc filesystem which contains a mirror image of the entire running system. There is one subdirectory per process (identified by the process id) and a number of other subdirectories that contain informations about buses, devices, disks and so on.

For a lot more information about advanced Linux programming, including many subjects we haven’t even mentioned in passing, you should take a look at Advanced Linux Programming. This book can be freely downloaded from:

http://www.advancedlinuxprogramming.com/

and contains a large amount of information for the Linux system programmers. Chapter 7 of the current edition at the time of writing is dedicated to the `/proc` filesystem. For even more advanced material about writing Linux kernel module, another full book is available for download:

http://lwn.net/Kernel/LDD3/

Both these link contain material ranging from advanced to very advanced. Caveat emptor. Less daunting, but still very advanced:

http://tldp.org/LDP/lkmpg/2.6/html/index.html

The Standard C Library

While C is a relatively small language (few keywords and datatypes) it comes with a relatively large standard library. The content of the library is part of the C standard as approved by the ISO, so that the interfaces are constant between implementations. The actual implementation can change between different compilers and operating systems, but these details are largely transparent to the developer.

The C standard library contains 24 headers, largely system independent, which are described here:

http://en.wikipedia.org/wiki/C_standard_library

In the context of Unix and Linux the standard C library also contains the POSIX standard library which aims to standardize the infrastructure between different implementations of Unix:

http://en.wikipedia.org/wiki/POSIX
http://en.wikipedia.org/wiki/C_POSIX_library

These Wikipedia pages contain links to the Wikipedia pages of the specific headers which also describe the functions contained in these libraries.

You can get similar information via the `man` command. For the `stdio.h` library:

```
man stdio.h
```

and so on. The most common implementation of the standard C library available on Linux systems is the `glibc` provided by the GNU Project of the Free Software Foundation. You can find its documentation (and a lot more) at:

http://www.gnu.org/software/libc/libc.html

Memory

Memory management is one of the most complex task that you will have to handle while developing C code. Using memory management correctly requires a deep understanding of the operating system infrastructure and lots of care. Mistakes in memory management can create a large variety of consequences ranging from slowing down the program over time (e.g. memory leaks) to complete crashes (e.g. when accessing unallocated memory or releasing memory blocks that have already been released).

Memory Structure

Computer memory is often organized in the form of a byte addressable memory. The smallest memory unit is the byte, which in most modern computer is made of 8 bit (short for binary digit, an entity which can contain a 0 or a 1). Each byte can be identified and accessed via its address, a number that univocally identifies a memory location.

The Operating System is in charge of memory management and grants/regains resources as they are needed for running programs. A program can expect the OS to allocate a certain amount of memory, in which the program has to perform all of its operation. A program can usually also ask for more memory as needed, and the OS can grant such request or deny it depending on its state.

The memory of a C program comes organized in two main 'structures': the *stack* and the *heap*. The *stack* is the memory space where functions and their resident data live. Stack is a generic name for a common type of data structure also known as a *LIFO* (Last In, First Out), which suits well the function invocation format of the C language.

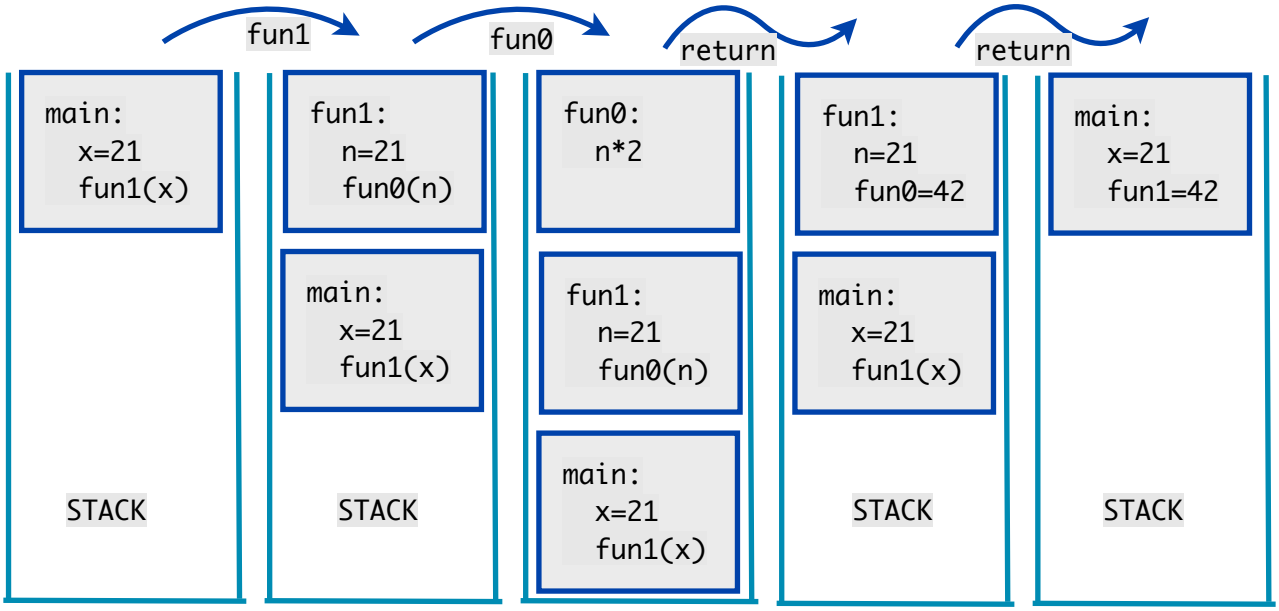
At any given time during the execution of a C program there is one single function that is being executed. This function lives at the top of the stack (TOS). When the current function invokes a new function, it becomes the new current function, a chunk of memory is allocated for it and placed upon the stack, at the top. The new TOS function gets executed and when it finishes, its memory chunk is removed from the TOS and control is returned to the previous function, which is again at the TOS. A simple drawing should help clear what's going on. Let's write a very simple program and then look at its stack:

```
int fun0(int n) {
    return n*2;
}

int fun1(int n) {
    return fun0(n);
}

int main(int argc, const char * argv[]) {
    int x = 21;
    fun1(x);
}
```

This drawing shows you what happens during the program's life:



The size of the stack is relatively limited, hence it is possible that under specific conditions the stack space will be exhausted and the program will crash. Note that structures that are defined on the stack (variables within a function) stop existing ('are released') as soon as the relevant chunk of memory is released. Caution must be exercised to avoid back-propagating memory addresses that are on the stack as this can create problems to the memory allocation mechanism.

Exercise 4. Crash the stack

Write a program to crash the stack.
As a bonus point, add a counter to check the stack depth.

During the execution of a program, a specific amount of memory could be required. Allocating memory on the stack can be undesirable due to its relatively limited size (and also if you need to propagate a memory block or similar situations). In these situations it is preferable to use dynamic memory allocation, that is request blocks of memory from the OS. These blocks are allocated on the heap, which is a memory space that can be organized by the OS as needed according to requests from various programs. The standard lifecycle of an allocated memory block is:

```
request memory block
use memory block
dispose of/release memory block
```

Carefully following these steps will ensure that the heap is handled correctly. Once a memory block on the heap has been assigned, it is not available anymore to other processes until release. That is why it's fundamental to correctly release unneeded resources.

More about functions

An important issue is how parameters are passed to functions. Unless we specify otherwise, parameters are passed 'by value', that is the value of the argument is copied onto the formal argument, and that value is used in the context of the function. Modifying that value within the function does not affect the value of the parameter in the calling function. Still sometimes we need to be able to modify the value of the actual argument that we passed to a function and have this modification propagated to the calling function. Let's try to define the swap function:

```
a = 42; b = 21;
swap(a,b);
```

```
a -> 21; b -> 42;
```

Let's try by writing:

```
#include <stdio.h>

void swap(int as, int bs) {
    int tmp = as;
    as = bs;
    bs = tmp;
    printf("swap: a -> %d - b -> %d\n", as, bs);
}

int main(int argc, const char * argv[]) {
    int a = 42;
    int b = 21;
    printf("a -> %d - b -> %d\n", a, b);
    swap(a, b);
    printf("a -> %d - b -> %d\n", a, b);
    return 0;
}
```

swap_by_value.c

Testing this code will immediately show that it is not doing what we would like it to. The function `swap` is swapping the content of `a` and `b`, but the new values are not being propagated to the main function. What's happening is that the function call is creating a copy of the values of a and b and passing them to the `swap` function, where the formal parameters `as` and `bs` now contain the copied values. Swapping the content of `as` and `bs` has the expected effect, but the values of the original parameters are unaffected.

Before describing the solution to our problem, we need to go back to the variables. You might remember that we mentioned that variables have four important properties. Three we have already described:

- a name, a type and a value

and now we can add:

- an address: the memory address where the variable is stored

The address of a variable can be extracted using the `&` operator and it can be stored in a special variable called a pointer. A pointer is simply an address to a variable. The pointer has to be of the same type of the variable it points to and is indicated with a `*` appended to the variable type. So for example:

```
int a = 42;
int* pa = &a;
```

In addition to this a pointer can be used to access the variable content, via the *dereference* operation. So to get the content of `a`, via `pa`:

```
*pa
```

and to modify the content of `a`, always via `pa`:

```
*pa = 21;
```

Needless to say, it is perfectly legal to create a pointer to a pointer and so on. Pointers are simply variables.

Now we can write the real swap function, passing the arguments by *reference* instead of by *value*:

```
#include <stdio.h>

void swap(int* as, int* bs) {
    int tmp = *as;
    *as = *bs;
    *bs = tmp;
    printf("swap: a -> %d - b -> %d\n", *as, *bs);
}

int main(int argc, const char * argv[]) {
    int a = 42;
    int b = 21;
    printf("a -> %d - b -> %d\n", a, b);
    swap(&a, &b);
    printf("a -> %d - b -> %d\n", a, b);
    return 0;
}
```

swap_by_reference.c

Compiling and running this program will show that this is a correct solution to the problem, but why does this work? The issue here is what we are passing. The arguments are still being passed by value, that is the address returned by `&a` is simply copied into `as`, but now instead of working on the value of `as` we work on the value contained in the memory location pointed to by the content of `as` (which is a pointer, hence it contains a memory address). Since the address contained in `as` is the address of `a`, modifying the content of memory location pointed to by `as`, does in fact modify the content of `a`.

This technique is also often used to return multiple values from a function. Pass some variables to the function by reference and store the return values in them. Often the function value (the one returned via the return keyword) is used to indicate anomalous situations and errors, hence the calling function would check the return value and used the other returned values only if no error condition is signaled.

Exercise 5. Return multiple values

Write a function that accepts two positive numbers, and returns their sum, their difference and their mean value. Also make it so that the function returns something indicating an error if one of the arguments is negative.
Write a program to use this function and print its results.

So far we have mentioned arrays, but never really used them. To pass an array to a function, we need to specify that the specific argument is an array. Similarly to what we do with pointers, where we add the * specification to the variable type, for arrays we add the [] specification (without

specifying any array dimension). Let's write a function that takes an array of ints and returns its sum:

```
#include <stdio.h>
#define LEN 10

int sum(int ints[], int len) {
     int index, result=0;
     for(index=0; index<len; index++) {
            result += ints[index];
     }
     return result;
}

int main(int argc, const char * argv[]) {
     int index;
     int ints[LEN];

     for(index=0; index<LEN; index++) {
            ints[index] = index;
     }

     printf("The sum of the first %d integers is %d\n", LEN, sum(ints, LEN));
     return 0;
}
```

sum_array.c

A few highlights about the program: the `sum` function accepts an array as indicated by its signature. Since we have no other way of doing this, we also need to explicitly pass the array size as an argument to the function.

Exercise 6. Numeric integration

Write a program to calculate a numeric integral using the the composite trapezoidal rule (http://en.wikipedia.org/wiki/Trapezoidal_rule).
The program should define a function that accepts an array containing the values of the function to integrate and any other relevant parameter: `float integrate(float values[], ...)`

The main part of the program should fill the values array, with values calculated from the function to be integrated. Ideally this function should also be stored into a function (okay, the mathematical function to be integrated should be stored in C function).

This logical separation allows you to write the integration code and reuse it as needed, while making it also possible to easily implement other integration algorithms and reuse the same mathematical functions.

You should be careful when defining the integration interval, the integration steps and all the relevant parameters. You might also want to define some utility function to map the integer indexes of the values array onto the integration step.
The `math.h` header contains a number of mathematical functions which might be useful.

Arrays and pointer algebra

Arrays are always passed by reference, this is because the array name is really a pointer to the first element of the array, hence passing as array simply passes a pointer to its first element:

```
int values[256];
values[0] = 42;
*values -> 42;
```

What's even more interesting is the way this pointer behaves. Let's write a small program and then discuss it:

```
#include<stdio.h>
#define LEN 10

int main() {
     int index;
     int ints[LEN];

     for(index=0; index<LEN; index++) {
            ints[index] = index;
     }
     for(index=0; index<LEN; index++) {
            printf("Array element %d = %d\n", index, *(ints+index));
     }
     return 0;
}
```

array_pointer.c

So while `*ints` points to the first element of the array, `*(ints+1)` points to the second, `*(ints+2)` to the third and so on. Incrementing a pointer does 'the right thing', that is it produces a pointer to the next element of an array. In fact we could write that:

```
array[n] == *(array+n)
```

Notice that this works even for modification:

```
*(array+n) = 42; == array[n] = 42
```

This works because C is doing some work behind the scenes, so that "incrementing the memory address by 1" does not in fact increment the actual address by one. This behavior is usually described by saying that the memory access is *aligned*. This is extremely important when we want to use dynamic memory.

Data types and memory

Let's go back to variables and in particular to variable types. The type of a variable indicates what kind of entities can be stored in it. This is related to the amount of space allocated to the variable. Since the base unit of allocation is the byte, the 'size' of a variable is simply the number of bytes that it uses to store its value.

An `unsigned short` is 2 bytes long (16 bit) so it can store values from 0 to 2^{16}-1 (65535), an `unsigned int` is 4 bytes long (32 bit) so it can store values from 0 to 2^{32}-1 (4,294,967,295) and so on. C provides an operator to discover the size of a variable type: `sizeof`. This is necessary when

requesting dynamic memory blocks. The main facility for dynamic memory allocation is the `malloc` function contained in `stdlib.h`. The signature of `malloc` is:

```
void *malloc(size_t size);
```

This call returns a pointer to a memory block which is long `size` bytes. The special data type `void` guarantees that access to this memory block is unaligned, so it is accessible byte by byte.

So let's create a dynamic array of 10 ints:

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

int main() {
     int index;
     int* data_block = (int*)malloc(sizeof(int)*10);

     for(index=0; index<10; index++) {
            *(data_block+index) = index;
     }

     for(index=0; index<10; index++) {
            printf("%d -> %d... ", index, data_block[index]);
     }

     free(data_block);
     return 0;
}
```

array_on_heap.c

The core part of the program is the memory allocation requested via `malloc`. The memory block we request on the heap is long 10 times the size of an `int`, thus allowing us to use it as an int array of 10 elements. The only important issue is that the pointer returned by `malloc` (which returns a null pointer if the allocation fails) is unaligned, so before using it to store `int`s we need to align it. To do that we cast it into an `int` pointer (`int*`). After this `data_block` behaves exactly like an array, and we use both common array access syntaxes to store and retrieve values from it.

To clean up and be good citizens we take care of explicitly releasing the allocated memory block using the `free` function. Be very careful as `free`-ing the same memory block twice will cause your program to crash.

So, to go back to the access syntax *(array+index) what is happening from the memory standpoint is that C knows the type of the array pointer, so the actual address is incremented by index*sizeof(array_pointer) thus realizing an aligned memory access.

Notice that just like we cast the `malloc` return value from `void*` to `int*`, it is always possible to cast a memory block to `void*`, thus allowing unaligned access. This is sometimes useful for more complex operations that require reorganizing the internal representation of a variable. One such case can be due to the endianness, that is the way a processor internally represents numbers. Two main approaches are known: little endian and big endian.

Let's look at the way an int can be represented. Let's use the number 1,937,425,042.

In little endian representation the values of the four bytes are 146 194 122 115
In big endian representation they are 115 122 194 146

In little endian representation the first byte is the least significant, while in big endian it is the most significant.

To obtain the original number: $115 * 2^{24} + 122 * 2^{16} + 194 * 2^{8} + 146$

Exercise 7. Endianness

Figure out the endianness of the computer you are using using a C program.

For more about endianness:

http://en.wikipedia.org/wiki/Endianness

Structs and pointers

It is also possible to define pointers to structs and to dynamically allocate them via the `malloc` function. The `sizeof` operator applied to a struct will return the correct value to use with `malloc`.

One important difference is in the access operation to structure members. Let's retrieve the definition of the `sample_s` struct:

```
typedef struct sample_s {
    int index;
    float value;
    float error;
} sampleType;
```

Now we can define a pointer to a `sampleType` struct like this:

```
sampleType* aSample = (sampleType*)malloc(sizeof(sampleType));
```

Now to access the fields of the `sampleType` structure pointed to by `aSample` we can simply dereference (the parentheses are needed since the dereference operator has lower priority than the `.` operator):

```
(*aSample).index = 255;
(*aSample).value = 3.14;
(*aSample).error = 0.03;
float maxValue = (*aSample).value + (*aSample).error;
```

or we can use the arrow operator (`->`) to directly access the fields without the need to explicitly dereference:

```
aSample->index = 255;
aSample->value = 3.14;
aSample->error = 0.03;
float maxValue = aSample->value + aSample->error;
```

More exercises

Now that you have been exposed to some parts of C, it is important to take the time and actually write some code with it. In this section you will find a few more exercises to play with. Note that these exercises are somewhat complex and will require a bit of work to be implemented.

Exercise 8. Integration with Monte Carlo method

Write a program to implement the 1D Monte Carlo integration.

The Monte Carlo integration is a numerical integration algorithm that uses random numbers.

The algorithm works as follows:
1. inscribe your function in a rectangle whose left and right sides are the same as the integration limits, and whose lower side lays on the x axis
2. generate a random point within this rectangular area
3. if the point is under the curve, increment a counter

Repeat 2. and 3. N times. With N large enough, the integral of the curve is
~= (counter/N)*rectangle_area

See: http://en.wikipedia.org/wiki/Monte_Carlo_integration and `man rand` for information about random number generation in C.

Exercise 9. 1D Cellular Automata

Implement a program to generate 1D Cellular Automata. A Cellular Automaton is a discrete model of a system which evolves over time.

A 1D Cellular Automaton consists of 'cells' living on a line. Each cell can be dead (0) or alive (1).

Each cell has two neighbors and its evolution is determined by the state of the two neighbors, its own state and a set of evolution rules, which determine the new state at the next iteration.

Write a program that evolves a cellular automaton given an initial configuration and a set of evolution rules.

See: http://en.wikipedia.org/wiki/Cellular_automaton and http://www.stilldreamer.com/mathematics/1d_cellular_automaton/

Exercise 10. The Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple iterative algorithm to generate a table of prime numbers.

Take the list of the first 100 numbers, and start by removing the multiples of 2. Then proceed to remove the multiples of 3. Then the multiples of 5 (4 has been removed when we removed the multiples of 2) and so on. At the end of this process, what's left are only the prime numbers between 1 and 100.

Write a program to implement this algorithm and use it to calculate the prime factors of an integer number.

See:
http://en.wikipedia.org/wiki/Prime_factor
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes