Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #4
Compilation, Debugging, Project Management
Part I

0.1 Introduction

The experiment intends to present the famous `gcc` compiler, the `gdb` debugger and show students how to use `makefile`s to better manage projects. Students are assumed to be already familiar with the C-language. We've included an appendix at the end of the current experiment as a review for the C-language.

Students will be shown how to compile source code and create executables using the `gcc` compiler. Moreover, students will be shown how to create libraries (both static libraries and dynamic libraries) using the archive command `ar` and how to link to these libraries during the compilation phase. In the next experiment (part II), they will be shown how to use the `gdb` debugger in order to debug code and discover defects (or potential defects). Finally, students will be shown how to use a `makefile` that will help better manage a development project during compilation, setup and installation.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Give a quick overview of the GNU project.
- Show students on how to use the `gcc` compiler and the `ar` archive command.
- Show students on how to use the `gdb` debugger.
- Show students on how to use a `makefile` to manage a development project.

0.3 The GNU project

The GNU project was launched in 1984 to develop a complete Unix-like operating system which is free softwaresoftware which respects your freedom[1].

The GNU operating system is a complete free software system, upward-compatible with Unix. GNU stands for *GNU's Not Unix*. The name GNU was chosen because it met a few requirements among which it was a recursive acronym for "GNU's Not Unix"

The word *free* in "free software" pertains to freedom, not price. You may or may not pay a price to get GNU software. Either way, once you have the software you have three specific freedoms in using it. First, the freedom to copy the program and give it away to your friends and co-workers; second, the freedom to change the program as you wish, by having full access to source code; third, the freedom to distribute an improved version and thus help build the community. (If you redistribute GNU software, you may charge a fee for the physical act of transferring a copy, or you may give away copies).

The project to develop the GNU system is called the **GNU Project**. The GNU Project was

[1]The GNU Operating System - http://www.gnu.org - 2011.

conceived in 1983 as a way of bringing back the cooperative spirit that prevailed in the computing community in earlier days-to make cooperation possible once again by removing the obstacles to cooperation imposed by the owners of proprietary software. By the 1980s, almost all software was proprietary, which means that it had owners who forbid and prevent cooperation by users. This made the GNU Project necessary.

It was decided to make the operating system compatible with Unix because the overall design was already proven and portable, and because compatibility makes it easy for Unix users to switch from Unix to GNU.

A Unix-like operating system is much more than a kernel; it also includes compilers, editors, text formatters, mail software, and many other things. Thus, writing a whole operating system is a very large job.

The GNU Project is not limited to the core operating system. The aim is to provide a whole spectrum of software, whatever many users want to have. This includes application software.

In order to provide software for users who are not computer experts, a graphical desktop called (GNOME) was developed to help beginners use the GNU system.

The applications that GNU make available include editors such as `emacs`, the famous `gcc` compiler and `gdb` debugger. To see the list of all the applications, visit `http://directory.fsf.org`.

0.3.1 The `gcc` compiler

The GNU Compiler Collection is a full-featured ANSI C compiler with support for K&R C, as well as C++, Objective C, Java, Objective C++, Ada and Fortran. `GCC` provides many levels of source code error checking traditionally provided by other tools, produces debugging information, and can perform many different optimizations to the resulting object code.

Consider the following C-code:

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr, celsius;
  int lower, upper, step;

  lower = 0; /* lower limit of temperature scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */
  fahr = lower;

  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Copy the above code in a file called `celsius.c`. To compile the code and produce the object code `celsius.o`, we execute the following:

```
gcc -c celsius.c
```

If errors are encountered during the compilation phase, the line number at which the error(s) was (were) encountered will be displayed on the standard output. As an example, remove any of the semicolons in the above code and check on the error messages that get displayed.

You can list symbols from object files using the command `nm` as follows:

```
nm celsius.o
```

You'll get the following output:

```
00000000 T _main
         U _printf
```

The symbol `T` in the above output implies that the symbol is in the text (code) section while the symbol `U` implies that the symbol is undefined. Many more symbols are available and each has its own significance. Do a `man nm` to get to know the remaining symbols.

Note that the command `nm` applies on object files with extension `.o` as well as on executables.

If you intend to create an executable from file `celsius.c` (if the `main` function exists), you execute the following:

```
gcc celsius.c
```

An executable file named `a.out` will be created in the local directory. You can run the executable as follows:

```
./a.out
```

Instead of renaming the executable `a.out` to a different name, you can invoke the `gcc` command using the `-o` option as follows:

```
gcc celsius.c -o celsius
```

The executable named `celsius` will be created in the local directory and the executable can be invoked as follows:

```
./celsius
```

If you execute the command `nm` on the executable `celsius` (`nm celsius`), you might get the following output:

```
080495c0 A __bss_start
080482fc t call_gmon_start
080495c0 b completed.4583
080494c0 d __CTOR_END__
080494bc d __CTOR_LIST__
080495b4 D __data_start
080495b4 W data_start
08048464 t __do_global_ctors_aux
08048320 t __do_global_dtors_aux
080495b8 D __dso_handle
080494c8 d __DTOR_END__
080494c4 d __DTOR_LIST__
080494d0 D _DYNAMIC
080495c0 A _edata
080495c4 A _end
0804848c T _fini
080494bc A __fini_array_end
080494bc A __fini_array_start
080484a8 R _fp_hw
08048354 t frame_dummy
```

```
080484b8 r __FRAME_END__
0804959c D _GLOBAL_OFFSET_TABLE_
         w __gmon_start__
08048280 T _init
080494bc A __init_array_end
080494bc A __init_array_start
080484ac R _IO_stdin_used
080494cc d __JCR_END__
080494cc d __JCR_LIST__
         w _Jv_RegisterClasses
0804845c T __libc_csu_fini
0804840c T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
0804837c T main
080495bc d p.4582
080494bc A __preinit_array_end
080494bc A __preinit_array_start
         U printf@@GLIBC_2.0
080482d8 T _start
```

Notice the different symbols that you get in the second column above.

The compiler `gcc` can be invoked with plenty of options that can make it do more check-ups. A nice option is the `-Wunused` that points out all the defined but unused variables. It can be used as follows:

```
gcc -Wunused file.c
```

To try it, assume you have the following code:

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr, celsius;
  int lower, upper, step, noUseVar;

  lower = 0; /* lower limit of temperature scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */
  fahr = lower;

  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Save the above code in a file called `celsius_unused.c` and compile it as follows:

```
gcc -Wunused celsius_unused.c -o celsius_unused
```

Check on the warning message that you get regarding the variable `noUseVar`.

Another common source of problems is the use and manipulation of uninitialzed variables in your code. To detect such cases, you need to invoke the `gcc` compiler with the two options `-Wuninitialzed` and `-O`. Assume you have the following code:

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr;
  char celsius, noUseVar;
  int lower, upper, step;

  lower = 0; /* lower limit of temperature scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */
  fahr = lower;

  fahr = noUseVar;

  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%c\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Save the above code in a file called `celsius_uninitilazed.c` and compile it as follows:

```
gcc -Wuninitialized -O celsius_uninitilazed.c -o celsius_uninitilazed
```

Check on the warning message that you get regarding the variable `noUseVar`.

The `-O` option of the `gcc` invokes code optimization. The `gcc` compiler will thus remove from the generated code any piece of your program that it doesn't deem necessary (e.g. unused variables, loops that manipulate variables that will not be used when the loop ends, etc). You are thus advised to use it with caution.

As an example on optimization, consider the following code:

```
#include <stdio.h>

int main()
{
  int i, j;

  i = 1; j = 2;

  printf("i = %d\t j = %d\n", i, j);

  // for (i = 0; i < 1000; i++);
  // for (i = 0; i < 1000; i++);

  // while (j--);
  // while (j--);
```

```
  return(0);
}
```

Save the above code in a file called `optimize.c` and compile it as follows:

```
gcc optimize.c -o optimize
```

Note the size of the executable `optimize` by executing `ls -l optimize`.

Now remove the comments from the 4 commented lines in file `optimize.c` and compile it as follows:

```
gcc -Wuninitialized -O optimize.c -o optimize
```

Note the size of the newly generated executable by executing `ls -l optimize` and notice that its size did not increase despite the increase in code lines. The reason is that during optimization, the `gcc` compiler removed any code it did not think was necessary!

If you wish to get *all* types of warnings (other than those shown by the optimization option `-O`), you need to invoke the `gcc` compiler with the `-Wall` option as follows:

```
gcc -Wall celsius_uninitilazed.c -o celsius_uninitilazed
```

Check on the warning messages that you get.

Note 1:

Students always need to keep an open eye on warnings and not decide to ignore them. Sometimes warnings might hide potential malfunction of your application. Have a quick look on the warning messages and then decide if it is *safe* to ignore them or not.

Note 2:

Below are some specific warning options for the `gcc` compiler that can be selected individually. Do `man gcc` to get to know the remaining options.

gcc option	Meaning
`-Wcomment`	Warns about nested comments
`-Wformat`	Warns about the incorrect use of format strings in functions such as `printf` and `scanf`
`-Wimplicit`	Warns about any functions that are used without being declared
`-Wreturn-type`	Warns about functions that are defined without a return type but not declared `void`
`-Wconversion`	Warns about implicit type conversions that could cause unexpected results, such as conversions between floating-point and integer types
`-Wshadow`	Warns about the redeclaration of a variable name in a scope where it has already been declared

0.3.2 Creating executables from object files

An executable might be the result of compiling multiple C-files. In the above example, we have seen how to generate an executable from a single file. We'll see below how to create an executable that depends on multiple C-files.

Assume you have the following code:

```
#include <stdio.h>

/* print the sum of 2 variables */

main()
{
  int s1, s2, s3, s4, res1, res2;

  s1 = 1; s2 = 2;
  s3 = 3; s4 = 4;

  res1 = sumVal(s1, s2);
  res2 = sumVal(s3, s4);

  printf("res1 = %d\n", res1);
  printf("res2 = %d\n", res2);
}
```

Save the above code in a file called `object_1.c`.

Type the following code in a file named `object_2.c`:

```
#include <stdio.h>

int sumVal(int val1, int val2)
{
  return(val1 + val2);
}
```

If you compile the file `object_1.c` as follows:

```
gcc object_1.c -o object_1
```

you'll get an error message stating that the function `sumVal` is undefined! That was an expected error since we know that the function `sumVal` is not defined in the C-file `object_1.c`. It is defined in the C-file `object_2.c`.

To compile both files and generate an executable that depends on the 2 files, we do the following:

```
gcc -c object_1.c
gcc -c object_2.c
```

The object files `object_1.o` and `object_2.o` will be generated from the previous 2 compilation commands. To generate the executable named `object` now, we execute the command:

```
gcc object_1.o object_2.o -o object
```

We say that the 2 object files have been *linked* to generate the executable[2]. Indeed, the `gcc` compiler uses the linker `GNU ld` to link the 2 files. The linking phase is invisible to users. The executable `object` can now be invoked as:

`./object`.

0.3.3 Inclusion of header files during compilation

During compilation, the `gcc` compiler knows about the location of system header files such as `stdio.h` or `stdlib.h` and thus includes them if needed. By default, `gcc` searches the following directories for header files:

[2] http://www.network-theory.co.uk/docs/gccintro

```
/usr/local/include/
/usr/include/
```

The list of directories for header files is often referred to as the `include path`. The directories on these paths are searched in order, from first to last in the two lists above. For example, a header file found in `/usr/local/include` takes precedence over a file with the same name in `/usr/include`.

The `gcc` compiler does not know about the location of user-defined header files, unless they are located in the same directory that contains the C-files. If not, the `-I` directive can be used.

For example, assume you have the following code:

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

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main()
{
  int fahr;
  char celsius;
  int lower, upper, step;

  lower = 0; /* lower limit of temperature scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */
  fahr = lower;

  while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%c\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Save the above code in a file called `celsius_header.c`. Assume that the file `myHeader.h` is located in the folder `include` under the current directory. To compile successfully the file `celsius_header.c`, execute the following:

```
gcc -Iinclude celsius_header.c -o celsius_header
```

The `-I` directive instructs the `gcc` compiler to include the folder named `include` while searching for header files during the compilation phase.

Note:

You should never place the absolute paths of header files in `#include` statements in your source code, as this will prevent the program from compiling on other systems. The `-I` option or the `C_INCLUDE_PATH` variable should always be used to set the include path for header files. If we assume your shell to be `bash`, you can update the the `C_INCLUDE_PATH` environment variable as follows:

```
C_INCLUDE_PATH=$C_INCLUDE_PATH:/home/user/include
```

You need afterwards to export the environment variable `C_INCLUDE_PATH` as follows:

```
export C_INCLUDE_PATH
```

The shell command `export` is needed to make the environment variable available to programs outside the shell itself, such as the compiler. It is only needed once for each variable in each shell session, and can also be set in the appropriate login file.

0.3.4 Inclusion of libraries during compilation (Linking with libraries)

A library is a collection of precompiled object files which can be linked into programs. The most common use of libraries is to provide system functions, such as the square root function `sqrt` found in the C math library.

Libraries are typically stored in special archive files with the extension `.a`, referred to as *static libraries*. They are created from object files with a separate tool, the GNU archiver `ar`, and used by the linker to resolve references to functions at compile-time. We will see later how to create libraries using the ar command.

The standard system libraries are usually found in the directories `/usr/lib`, `/usr/local/lib` and `/lib`. For example, the C math library is typically stored in the file `/usr/lib/libm.a` on Unix-like systems. The corresponding prototype declarations for the functions in this library are given in the header file `/usr/include/math.h`. The C standard library itself is stored in `/usr/lib/libc.a` and contains functions specified in the ANSI/ISO C standard, such as `printf`. This library is linked by default for every C program.[3]

As an example, consider the following code:

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

int main (void)
{
  double x = sqrt (2.0);

  printf ("The square root of 2.0 is %f\n", x);
  return 0;
}
```

Save the above code in a file called `calc.c`. Trying to create an executable from this source file alone causes the compiler to give an error complaining about the unknown function `sqrt`[4].

The problem is that the reference to the `sqrt` function cannot be resolved without the external math library `libm.a`. The function `sqrt` is not defined in the program or the default library `libc.a`, and the compiler does not link to the file `libm.a` unless it is explicitly selected.

To enable the compiler to link the `sqrt` function to the main program `calc.c`, we need to supply the library `libm.a`. One obvious way to do this is to specify it explicitly on the command line:
`gcc -Wall calc.c /usr/lib/libm.a -o calc`.

The library `libm.a` contains object files for all the mathematical functions, such as `sin`, `cos`, `exp`, `log` and `sqrt`. The linker searches through these to find the object file containing the `sqrt` function.

The executable file `calc` includes the machine code for the main function and the machine code for the `sqrt` function, copied from the corresponding object file in the library `libm.a`

To avoid the need to specify long paths on the command line, the compiler provides a short-cut

[3] http://www.network-theory.co.uk/docs/gccintro

[4] Note that on some Linux distributions such as Ubuntu, the mathematical library is included by default. You might thus not get an error while compiling file `calc.c`

option `-l` for linking against libraries. For example, the following command:

```
gcc -Wall calc.c -lm -o calc
```

is equivalent to the original command above using the full library name `/usr/lib/libm.a`.

In general, the compiler option `-lNAME` will attempt to link object files with a library file `libNAME.a` in the standard library directories.

Note 1:

If a library is missing from the link path. It can be added to the link path using the option `-L` as follows:

```
-L/home/user/library_dir
```

If the user intends to link with a library called `gbdm` for example that is located in the above path, the compilation should be as follows:

```
gcc -Wall -L/home/user/library_dir calc.c -lgbdm
```

As for header files, libraries can be included in the link path using the environment variable `LIBRARY_PATH` as follows (assuming your shell is `bash`):

```
LIBRARY_PATH=$LIBRARY_PATH:/home/user/library_dir
```

Note 2:

To specify multiple search path directories on the command line, the options `-I` and `-L` can be repeated as follows:

```
gcc -I. -I/opt/gdbm-1.8.3/include -I/net/include -L. -L/net/lib calc.c -lgbdm
```

0.4 Creating a static library

We'll show below an example on how to create a static library using the GNU archiver `ar` without getting lost in the details[5]. We will create a small library `libhello.a` containing two functions `hello` and `bye`.

Assume you have the following code:

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

void hello (const char * name)
{
  printf ("Hello, %s!\n", name);
}
```

Save the above code in a file called `hello_fn.c`

Create the C-file `bye_fn.c` that contains the following code:

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

void bye (void)
{
  printf ("Goodbye!\n");
}
```

[5] http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Create the header file `hello.h` that contains the prototypes of the 2 functions seen above:

```
void hello (const char * name);
void bye (void);
```

Now compile the 2 C-files `hello_fn.c` and `bye_fn.c` as follows:

```
gcc -Wall -c hello_fn.c
gcc -Wall -c bye_fn.c
```

We can create the library by combining the 2 object files into a static library as follows:

```
ar cr libhello.a hello_fn.o bye_fn.o
```

The option `cr` stands for *create and replace*. If the library does not exist, it is first created. If the library already exists, any original files in it with the same names are replaced by the new files specified on the command line. Execute `man ar` to see a detailed description of options for the command `ar`. The first argument `libhello.a` is the name of the library. The remaining arguments are the names of the object files to be copied into the library.

The archiver `ar` also provides a *table of contents* option `t` to list the object files in an existing library:

```
ar t libhello.a
```

The output will be:

```
hello_fn.o
bye_fn.o
```

We can now create a third C-file called `main_hello.c` that uses the 2 functions `hello` and `bye` by linking with the library `libhello.a` as follows:

```
#include "hello.h"

int main (void)
{
  hello ("everyone");
  bye ();
  return 0;
}
```

We then link with library `libhello.a` as follows:

```
gcc -Wall main_hello.c libhello.a -o hello
```

or as follows:

```
gcc -Wall -L. main_hello.c -lhello -o hello
```

0.4.1 Creating a shared library

Shared libraries are handled with a more advanced form of linking, which makes the executable file smaller. They use the extension `.so`, which stands for *shared object*.

An executable file linked against a shared library contains only a small table of the functions it requires, instead of the complete machine code from the object files for the external functions. Before the executable file starts running, the machine code for the external functions is copied into memory from the shared library file on disk by the operating system–a process referred to as *dynamic linking*.

Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. Most operating systems also provide a virtual memory mechanism which allows one copy of a shared library in physical memory to be used by all running programs, saving memory as well as disk space.

Furthermore, shared libraries make it possible to update a library without recompiling the programs which use it (provided the interface to the library does not change).

Because of these advantages `gcc` compiles programs to use shared libraries by default on most systems, if they are available. Whenever a static library `libNAME.a` would be used for linking with the option `-lNAME`, the compiler first checks for an alternative shared library with the same name and a `.so` extension.

As with static libraries, an object file is created. The `-fPIC` of the `gcc` compiler tells `gcc` to create *position independant code* which is necessary for shared libraries:

```
gcc -Wall -c -fPIC hello_fn.c
gcc -Wall -c -fPIC bye_fn.c
```

Now you create the shared library as follows:

```
gcc -shared -Wl,-soname,libhello.so.1 -o libhello.so.1.0.1 hello_fn.o bye_fn.o
```

Note 1:
The library must start with the three letter `lib`.

Note 2:
When an executable that links with a shared library is run, its loader function must find the shared library in order to load it into memory. By default the loader searches for shared libraries only in a predefined set of system directories, such as `/usr/local/lib` and `/usr/lib`. If the library is not located in one of these directories it must be added to the load path.

The simplest way to set the load path is through the environment variable `LD_LIBRARY_PATH`. For example, the following command sets the load path to `/opt/gdbm-1.8.3/lib` (assuming your shell is `bash`):

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/gdbm-1.8.3/lib
```

0.4.2 Defining macros for the C preprocessor

The C preprocessor can define macros using the conditional `#ifdef` directive. Consider the below code:

```
#include <stdio.h>

int main (void)
{
#ifdef TEST
  printf ("Test mode\n");
#endif
  printf ("Running...\n");
  return 0;
}
```

Save the above code in a file called `test.c`.

When the macro is defined, the preprocessor includes the corresponding code up to the closing `#endif` command. In this example, the macro which is tested is called `TEST`, and the conditional

part of the source code is a `printf` statement which prints the message `Test mode`.

The `gcc` option `-DNAME` defines a preprocessor macro `NAME` from the command line. If the program above is compiled with the command-line option `-DTEST`, the macro `TEST` will be defined and the resulting executable will print both messages:

```
gcc -Wall -DTEST test.c -o test
```

If the same program is compiled without the `-D` option then the `Test mode` message is omitted from the source code after preprocessing, and the final executable does not include the code for it:

```
gcc -Wall test.c -o test
```

Note:

In addition to being defined, a macro can also be given a value. This value is inserted into the source code at each point where the macro occurs. Consider the following example:

```
#include <stdio.h>

int main (void)
{
  printf ("Value of NUM is %d\n", NUM);
  return 0;
}
```

Save the above code in a file called `testval.c`.

To define a macro with a value, the `-D` command-line option can be used in the form `-DNAME=VALUE`. For example, the following command line defines `NUM` to be `100` when compiling the program above:

```
gcc -Wall -DNUM=100 testval.c -o testval
```

If you run the generated executable `testval`, you get the following output:

```
Value of NUM is 100
```

As another example, you should know that whatever the value of the macro is, it is inserted directly into the source code at the point where the macro name occurs. For example, the following definition expands the occurrences of `NUM` to `2+2` during preprocessing:

```
gcc -Wall -DNUM="2+2" testval.c -o testval
```

If you run the generated executable `testval`, you get the following output:

```
Value of NUM is 4
```

If you intend to pass a string argument to the `gcc` compiler, you can execute the following command:

```
gcc -DMSG=\"Hello\" execFile.c -o execFile
```

Consider the following example that illustrates the passing of string arguments to the `gcc` compiler:

```
#include <stdio.h>

int main() {
  printf("MSG: %s\n", MSG);
}
```

Save the above code in a file called `define_text.c` and compile it as follows:

```
gcc -DMSG=\"Hello\" define_text.c -o define_text
```

Run the executable `define_text` and check the output that you get.

Appendix
Review of the C-language

The appendix below intends to give students a better insight in the C-language. Students are assumed to be already familiar with that programming language. We will not go in the details of control flow (e.g. `if-else`, `switch`, `while`, `for`, `do-while`, `break`, `continue`, `goto`). The focus of the experiment will be on tips and tricks that the students might even use in other programming languages[6].

0.5 Objectives

The objectives of the appendix is to teach the following:

- Give a brief introduction to the history of the C-language.
- Give students a solid background about the C-language without getting lost in details.
- Show students some tips and tricks about how to write efficient C-code.
- Prepare students to the compilation/debuging environment that will follow in the next experiment.

0.6 The C-language

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a **system programming language** because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (`if-else`), selecting one of a set of possible values (`switch`), looping with the termination test at the top (`while`, `for`) or at the bottom (`do`), and early loop exit (`break`).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively *low-level* language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

[6]Brian W. Kernighan and Dennis M. Ritchie, **The C programming Language** - Prentice-Hall, 1988.

0.6.1 Use of arrays

An array is a data structure that stores data of the same nature in successive locations. Students are familiar with definitions such as:

```
int nDigit[10]
```

In the above example, you are reserving space for exactly 10 integers that will occupy memory regardless of being used or not.

If you do not intend to use the above array during the lifetime of the process, it is a good technique to allocate memory when needed and free it once done with it. You can thus keep memory available for other usage and other processes as well. For *dynamic* memory allocation, you can define the array as follows:

```
int *nDigit = NULL
```

and use the `malloc` function as follows:

```
nDigit = (int *) malloc(sizeof(int) * 10);
```

To access a cell in the array `nDigit`, you can use either the conventional notation:

```
nDigit[5]
```

or the following notation:

```
*(nDigit + 5)
```

When the array `nDigit` is no more needed, you can free it as follows:

```
free(nDigit)
```

Note that you are not allowed to free the same array more than once. Note as well that it is good pratice to initialize dynamically allocated arrays to `NULL` and test if the array is allocated before using it. That might make you avoid lots of potential problems you might get into.

0.6.2 Use of symbolic constants

It's a bad practice to *bury* constants in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a *symbolic name* or *symbolic constant* to be a particular string of characters:

```
#define name replacement list
```

Examples can be as follows:

```
#define LOWER 0
#define UPPER 100
#define MID 50
```

It is good practice to have these symbolic constants in upper case always so as not to mix them with variables.

0.6.3 Functions

Functions are subroutines that you can call as many time as you need during the lifetime course of a process. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. Here is the function `power` and a main program to exercise it, so you can see the whole structure at once:

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

```
int power(int m, int n);

/* test power function */
main()
{
  int i;

  for (i = 0; i < 10; ++i)
    printf("%d %d %d\n", i, power(2,i), power(-3,i));
  return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
  int i, p = 1;

  for (i = 1; i <= n; ++i)
    p = p * base;

  return p;
}
```

Multiple things should be noted in the above function:

- The definition `int power(int m, int n);` is called the function prototype. It explains the return type of the function as well as the number and type of arguments that a user should pass when calling the function `power`. It is good practice to put function prototypes in header files (e.g. files that end with the extension `.h`) that are included by `#include` at the front of each source file.
- Writing the bulk of the code in functions has the advantage of keeping the developer focused on the problem he/she has at hand instead of getting lost in the details.
- If the function `power` end up being useful to multiple applications, it is good practice to create a library that contains the code for that function. In any application that needs the function `power`, you need to link with that library while compiling the application (covered in the next experiment).

0.6.4 External variables

Variables defined within functions are called *local variables* or *automatic variables* since their scope is limited to the functions to which they belong.

As an alternative to automatic variables, it is possible to define variables that are *external* to all functions, that is, variables that can be accessed by name by any function.

An external variable must be *defined*, exactly once, outside of any function; this sets aside storage for it. The variable must also be *declared* in each function that wants to access it; this states the type of the variable. The declaration may be an explicit `extern` statement or may be implicit from context.

```
#include <stdio.h>

int max; /* maximum length seen so far */
```

```
main()
{
  int len;
  extern int max;

  max = 0;
  while ((len = getline()) > 0)
    if (len > max) {
      max = len;
      printf("Longer line detected\n");
    }

  if (max > 0) /* there was a line */
    printf("%d", max);

return 0;
}
```

0.6.5 Enumeration constant

An enumeration is a list of constant integer values, as in:

```
enum boolean { NO, YES };
```

The first name in an `enum` has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
              /* FEB = 2, MAR = 3, etc. */
```

Enumerations provide a convenient way to associate constant values with names, an alternative to `#define` with the advantage that the values can be generated for you. Although variables of `enum` types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than `#define`s. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

0.6.6 Type conversion

It might be necessary to convert data from type to type. It is common to transform a string into an integer or float equivalent as follows:

```
char strValue[] = "12345";

int intValue = atoi(strValue); /* convert the string value to integer */

float floatValue = atof(strValue); /* convert the string value to float */

long longValue = atol(strValue); /* convert the string value to long */
```

0.6.7 Arithmetic Operators

The binary arithmetic operators are `+`, `-`, `*`, `/`, and the modulus operator `%`.

The `%` operator cannot be applied to a `float` or `double`.

The binary `+` and `-` operators have the same precedence, which is lower than the precedence of `*`, `/` and `%`, which is in turn lower than unary `+` and `-`. Arithmetic operators associate left to right.

0.6.8 Relational and Logical Operators

The relational operators are:

```
>   >=   <   <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
==   !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim - 1` is taken as `i < (lim - 1)`, as would be expected.

More interesting are the logical operators `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

The precedence of `&&` is higher than that of `||`, and both are lower than relational and equality operators.

0.6.9 Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

`&`	bitwise AND
`|`	bitwise inclusive OR
^	bitwise exclusive OR
`<<`	left shift
`>>`	right shift
~	one's complement (unary)

Note that the left shift operator `<<` is equivalent to multiply by 2 while the shift right operator `>>` is equivalent to divide by 2.

The bitwise AND operator `&` is often used to mask off some set of bits, for example:

```
n = n & 0177;
```

sets to zero all but the low-order 7 bits of `n`.

The bitwise OR operator `|` is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in `x` the bits that are set to one in `SET_ON`.

The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example:

```
x = x & ~077
```

sets the last six bits of `x` to zero.

0.6.10 Conditional Expressions

The conditional expression, written with the ternary operator `"?:"`, provides an alternate way to write conditional statements with the `if-else` clauses. As an example, the following expression:

```
z = (a > b) ?  a :  b; /* z = max(a, b) */
```

will set the value of `z` to either `a` or `b`, whichever is bigger.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of `?:` is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

0.6.11 Use of macros

A macro is a definition that has the form:

```
#define name replacement list
```

It calls for a macro substitution of the simplest kind - subsequent occurrences of the token `name` will be replaced by the replacement text. Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a \ at the end of each line to be continued.

Examples

```
#define MAX(A, B)   ((A) > (B) ? (A) : (B))

#define SWAP(A, B, type) { type tmp; \
                           tmp = (A); \
                           (A) = (B); \
                           (B) = tmp; \
                         }
```

Although it looks like a function call, a use of `MAX` expands into in-line code. Each occurrence of a formal parameter (here `A` or `B`) will be replaced by the corresponding actual argument. Thus the line:

```
x = MAX(p+q, r+s);
```

will be replaced by the line:

```
x = ((p+q) > (r+s) ?  (p+q) :  (r+s));
```

To ensure that a routine is really a function, not a macro, names may be undefined with the `#undef` statement as follows:

```
#undef getchar

int getchar(void) {...}
```

0.6.12 Conditional Inclusion

It is possible to control preprocessing with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The `#if` line evaluates a constant integer expression. If the expression is non-zero, subsequent

lines until an `#endif` or `#elif` or `#else` are included. (The preprocessor statement `#elif` is like `else-if`.) The expression `defined(name)` in a `#if` is 1 if the name has been defined, and 0 otherwise.

For example, to make sure that the contents of a file `hdr.h` are included only once, the contents of the file are surrounded with a conditional like this:

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif
```

The statement `#if` can be replaced by `#ifndef` as in the following example:

```
#ifndef HDR
#define HDR

/* contents of hdr.h go here */

#endif
```

Another useful example is shown below. It relates to deciding which header file to include based on the OS type:

```
#if SYSTEM == SYSV
  #define HDR "sysv.h"
#elif SYSTEM == BSD
  #define HDR "bsd.h"
#elif SYSTEM == MSDOS
  #define HDR "msdos.h"
#else
  #define HDR "default.h"
#endif
```

0.6.13 Header Files

Header files are files that end with the extension `.h`. They are mainly used to centralize the definitions and declarations shared among files, so that there is only one copy to get and keep right as the program evolves. Header files can contain other header files as well.

Usually, header files might contain the following elements:

- Other header files.
- Global variables shared among multiple C-files.
- Common structures or unions that are needed by some C-files.
- Function prototypes.
- Defined constants (by using the `#define` directive) and defined macros.

0.6.14 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is 1, there are no command-line arguments after the program name.

Example

```
./showAll 10 Khalil 2300
```

In the above example, `argc` is `4`, `argv[0]` is `./showAll`, `argv[1]` is `10`, `argv[2]` is `Khalil` and `argv[3]` is `2300`. Note that all arguments are passed as strings. As such, the arguments `10` and `2300` must be converted to integers if you intend to use them as such.

0.7 Structures

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called *records* in some languages). Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. The structure below describes an `employee` structure.

```
struct address {
  int  streetNumber;
  char streetname[20];
  char city[20];
  char state[20];
};

struct employee {
  char    surName[20];
  char    familyName[20];
  int     age;
  float   salary;
  address adr;
};
```

Structures can be passed as arguments to functions such as:

```
addEmployee(struct employee emp)
```

And functions can return variables of type structure as follows:

```
struct employee updateEmployee(struct employee emp, char *name)
{
  strcpy(emp.name, name);

  return(emp);
}
```

A structure can also self-reference such that it can contain an instance of itself. An example is the definition of a node in a binary tree as follows:

```
struct tnode { /* the tree node: */
  char *word; /* points to the text */
  int count;  /* number of occurrences */
  struct tnode *left;  /* left child */
  struct tnode *right; /* right child */
};
```

0.7.1 Typedef

C provides a facility called typedef for creating new data type names. For example, the declaration:

```
typedef int Length;
```

makes the name `Length` a synonym for `int`. The type `Length` can be used in declarations, casts, etc., in exactly the same ways that the `int` type can be:

```
Length len, maxlen;
Length *lengths[];
```

Similarly, the declaration:

```
typedef char *String;
```

makes `String` a synonym for `char *` or character pointer, which may then be used in declarations and casts:

```
String p,lineptr[MAXLINES];
int strcmp(String, String);
p = (String) malloc(100);
```

Another example would be:

```
typedef struct tnode { /* the tree node: */
      char *word; /* points to the text */
      int count;  /* number of occurrences */
      struct tnode *left;  /* left child */
      struct tnode *right; /* right child */
} Treenode;
```

0.7.2 Unions

A `union` is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

As an example suppose that a constant may be an `int`, a `float`, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union - a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```
union u_tag {
    int   ival;
    float fval;
    char *sval;
} u;
```

The variable `u` will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to `u` and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as:

```
union-name.member
```

or:

```
union-pointer->member
```

If the variable `utype` is used to keep track of the current type stored in `u`, then one might see code such as:

```
if (utype == INT)
  printf("%d\n", u.ival);
else if (utype == FLOAT)
  printf("%f\n", u.fval);
else if (utype == STRING)
  printf("%s\n", u.sval);
else
  printf("bad type %d in utype\n", utype);
```

0.7.3 Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word.

A *bit-field*, or field for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a *word*. Consider the following 3 fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

This defines a variable table called `flags` that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared `unsigned int` to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: `flags.is_keyword`, `flags.is_extern`, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus we can write:

```
flags.is_extern = flags.is_static = 1;
```

to turn the bits on:

```
flags.is_extern = flags.is_static = 0;
```

to turn them off; and:

```
if (flags.is_extern == 0 && flags.is_static == 0)
  ...
```

to test them.

0.7.4 File Access

Before a file can be read or written, it has to be *opened* by the library function `fopen`. The function `fopen` takes an external file name, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from `<stdio.h>` include a structure declaration called `FILE`. The only declaration needed for a file pointer is exemplified by:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

The `mode` can be `r` for reading, `w` for writing and `a` for appending.

You need always to check on the return value that `fopen` returns when opening a file. If the return value is `NULL`, then the file could not be opened. In read mode, that might mean that the file was not found or you have no permission to read it. If in write mode, that might mean that you have no free space on disk or have no permission to write on that file.

When you are done handling the file, it needs to be closed to free the memory structures used by the file. For that, we use the function `fclose` as follows:

```
fclose(fp)
```

The function `unlink(char *name)` removes the file name from the file system. It corresponds to the standard library function `remove`.