# An Introduction to Programming with GNU Tools

Imed Chihi (Synapse) <imed@suse.com.tn>

9 November 2003

**Abstract**

This document is a simple introduction to the development environment with GNU project programming tools. It only initiates the devolper to the interaction with the compiler and introduces some notions about the culture and the habits of software developement under Linux and Unix. This document assumes a basic knowledge of the C programming language and Linux/Unix shell. Please send any proposal for enhancement or omission/incorrectness report to the author.

## Change log

```
20030108 ichihi   Initial revision
20031109 ichihi   Added a section on setting up a CVS server
```

## 1 The gcc compiler

GCC is the GNU Compiler Collection, it's a whole compiling environment for C, C++, Fortran, Java and Objective-C. The following examples will be in C, there are uncountable good tutorials on the subject on the Net. The book by Brian Kernighan (and Dennis Ritchie) "The C Programming Language" is aslo an inevitable reference.

### 1.1 Ahlan (Hello World)

The classical example of programming books is the following piece of code that simply prints out the string "Ahlan" ("Hello World"). You may use your favorite text editor to create the sources.

```
$ cat ahlan-0.c
#include <stdio.h>
int main() {
printf("Ahlan\n");
return 0;
}
```

It's usually customary to add a header to every source file to specify critical information like its subject, the author and the date the development started.

```
$ cat ahlan.h
/* ahlan.h - Defines a constant
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * This program is free software, you may distribute/modify
 * it according to the terms of the GNU GPL.
 */

#define PI 3.14
#include <stdio.h>
#include <math.h>
```

We take the opportunity to create a second file (ahlan.h) to make a few declarations.

```
$ cat ahlan-1.c
/* ahlan-1.c - Prints sin(pi)
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * This program is free software, you may distribute/modify
 * it according to the terms of the GNU GPL.
 */

#include "ahlan.h"

int main() {
printf("Ahlan, sine of pi is %f\n", sin(PI));
return 0;
}
```

The Japanese date format (year-month-day) is the ISO format for date representation. It's, actually, ISO standard ISO-8601. An e-mail address at the beginning of every source file would be very welcome. In the case of the GNU GPL (*General Public License*), the notice about the license and the name of the author should never be altered.

## 1.2 indent

indent(1) is a tool for formatting C sources, it's usually documented is section 1 of the Linux/Unix manual. Therefore, you may use:

```
$ man 1 indent
```

to see the manual page for this tool. When the Unix litterature refers to commands (like indent(1)), functions or configuration files, it usually refers to the manual section where the command is documented too. You'd than see texts referring to gzip(1), read(2), lilo.conf(5), etc.

For example, to format our source code according to the GNU coding style, we may use:

```
$ cat ahlan-1.c | indent --gnu-style
```

## 1.3 Compiling

As we have already mentioned, GCC is a lot more than a compiler. Indeed, the conversion of a C source code into a binary executable by the machine is a multi-step process. In the case of GCC, it's a matter of:

1. preprocessing: the preprocessor language, which is not specific to C, is used to contol the compiling process itself. It's useful for such things as defining symbols and performing conditional compiling according to the architecture. Preprocessing is a way to prepare the source code for being compiled. In the Linux kernel, you'd find portions of code written in architecture-specific assembly language and which are included into the main source according to the settings. Try the command `cpp file.c` or `gcc -E file.c`,
2. compiling: this step consists into converting (compiling) a preprocessed C source (or other source) and to generate an assembly code specific to the given architecture. Try the command `gcc -S file.c` and have a look at the generated `file.s`,
3. assembly: it's the convertion (compiling) of an assembly source into binary code (.o suffix). Try the command `gcc -c file.c` or `gcc -c filr.s`,
4. linking: it's the last step to generate a machine executable program. The essence of this step consists into resolving references to symbols (variable and function names) spread out in multiple assembled modules (.o) and libraries.

Therefore, to compile our example described above, we may do:

```
$ gcc -E ahlan-1.c > ahlan-1.i
$ gcc -S ahlan-1.i
$ gcc -c ahlan-1.s
$ gcc -o ahlan-1 ahlan-1.o -lm
$ ./ahlan-1
```

Fortunately, GCC knows how to manage all these steps, and we can just issue:

```
$ gcc -o ahlan-1 ahlan-1.c -lm
$ ./ahlan-1
```

Notice the use of -lm to tell the compiler that it has to fetch certain symbols in a library called libm.o.

## 1.4 Libraries

Usually, all programs generated by GCC are, by default, dynamically linked to libc and ld-linux. libc is the C language standard library, it contains the functions defined by the standard along with entry points to the kernel system calls. ld-linux is the library containing the code responsible for the management of dynamic libraries like automatic loading. To check the list of libraries upon which a program depends, use:

```
$ ldd ./ahlan-1
      libc.so.6 => /lib/libc.so.6 (0x4002d000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

To compile a program with libraries statically linked, use the -static option:

```
$ gcc -static -o ahlan-1 ahlan-1.c -lm
$ ldd ./ahlan-1
      not a dynamic executable
```

As noted above, the compiler assumes the existance of a file named libm.so when it sees -lm and the existance of a file named libX11.so when it sees -lX11. The path to libraries are managed by ldconfig(8). For example, to list all the registered libraries, we may use:

```
$ ldconfig -v
```

The paths to directories containing dynamic libraries are listed in /etc/ld.so.conf. As ldconfig(8) updates a cache (/etc/ld.so.cache), it may require root privileges for certain operations. To add a directory of libraries to the context of a single user, we may set the LD_LIBRARY_PATH environment variable:

```
$ export LD_LIBRARY_PATH=/home/imed/src/libs/
```

Let's try to build a dynamic library offering the following functions:

```
$ cat libkaren.c
/* libkaren.c - Integer comparison library
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * This program is free software, you may distribute/modify
 * it according to the terms of the GNU GPL.
 */

int max(int a, int b) {
  return (a < b) ? b : a;
}

int min(int a, int b) {
  return (a < b) ? a : b;
}

$ cat libkaren.h
/* libkaren.h - Declarations for libkaren.so functions
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * This program is free software, you may distribute/modify
 * it according to the terms of the GNU GPL.
 */

int max(int a, int b);
int min(int a, int b);

$ gcc -fPIC -shared -o libkaren.so libkaren.c
```

The -fPIC option instructs the compiler to generate relative address references (*Position Independent Code*) which are indepent from the location where the program will be loaded. The -shared option tells the linker that it's dealing with shared code that will be linked to other code later on when it's loaded for execution and that it's has not to worry about unresolved symbols. We may now use our library as we have used libm.so earlier.

```
$ cat karen.c
/* karen.c - Comparisons using libkaren
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * This program is free software, you may distribute/modify
 * it according to the terms of the GNU GPL.*
 */

#include "libkaren.h"
#include <stdio.h>
int main() {
  printf("The max of 4 and 2 is %d, the min is %d\n", \
  max(4, 2), min(4, 2));
  return 0;
}

$ gcc -o karen karen.c -L. -lkaren
$ LD_LIBRARY_PATH=. ./karen
```

The -L. gives the linker a supplementary path to those managed by ldconfig(8) where it has to fetch dynamic libraries. We could have used LD_LIBRARY_PATH is a similar fashion.

# 2 make

make(1) is a tool that manages the process of compiling projects with multiple modules. It alleviates the burden of useless rebuilds when changes occur from the developer. Once configured, it can manage dependencies and performs the whole building process. The Makefile for the karen.c project should look like this:

```
$ cat Makefile
# Makefile - Main Makefile
# Imed Chihi (Synapse) <imed@suse.com.tn>
# 2003-01-08
#

SRCS= karen.c
CC= gcc
LIBSRCS= libkaren.c
CFLAGS= -O2 -g -Wall
LIBSPATH= -L.
LIBS= -lkaren
LDFLAGS= -fPIC -shared
RM= /bin/rm -rf

karen: $(SRCS) libs
        $(CC) $(CFLAGS) -o karen $(SRCS) $(LIBSPATH) $(LIBS)

libs: libkaren.so

libkaren.so: $(LIBSRCS) libkaren.h
        $(CC) $(LDFLAGS) -o libkaren.so $(LIBSRCS)

clean:
        $(RM) libkaren.so karen

$ make
gcc -fPIC -shared -o libkaren.so libkaren.c
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
$ make
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
$ vi libkaren.h
$ make
gcc -fPIC -shared -o libkaren.so libkaren.c
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
```

It's, actually, a dependence graph as illustrated in figure 1. make(1) is very picky when it comes to the Makefile formatting, you need to pay attention to spaces and tabs. Every node of the tree can be mapped in the Makefile by a "target" like: all, libs and karen in this case.

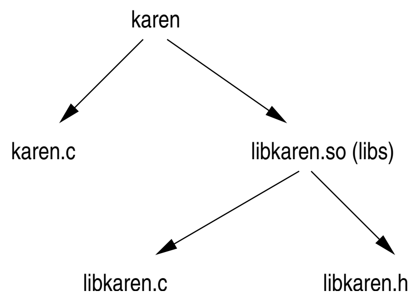


Figure 1: Dependencies graph for project "karen".

# 3 Configuration management

Configuration management is a term that's used to refer to all the systems and techniques used to manage the different versions of a text be it a document, a configuration file or a source code. Basically anything that's supposed to be modified through time possibly be many people. The most popular configuration management utility under Unix has been SCCS but it's deprecated by now, RCS is an evolution of SCCS and CVS is by far the most popular configuration management tool nowadays.

## 3.1 RCS and CVS

RCS(1) is the *Revision Control System*, it's a configuration management tool for projects comprised of multiple modules. It's used to track changes made to individual files and to restore the project to any past state in the development process. CVS(1) (*Concurrent Versioning System*) is the RCS successor, it adds extensions to support multiple concurrent developers performing changes to the fules of a project evetually at the same time. It's the preferred tool of configuration management in the Open Source world and knowledge of CVS is usually required in a multi-developer team.. This paragraph briefly introduces CVS, a more elaborate tutoriel is left as a future project, it will cover certain aspects like the details of setting up a CVS server.

To use CVS, we need to create a source repository first:

```
$ export CVSROOT=~/cvs
$ cvs init
```

Now, we have to create a project called karen in this repository:

```
$ mkdir ~/cvs/karen
```

At this stage, we need to create a working directory where we'll be altering a local copy of the files which we'll put back into the repository:

```
$ cd ~/src/
```

Let's get a copy of the source from the central repository, it should be empty for now:

```
$ cvs checkout karen
```

We create the projet files and we add them to our repository:

```
$ vi libkaren.c libkaren.h karen.c Makefile
$ cd ..
$ cvs add karen/Makefile; cvs add karen/karen.c; cvs add karen/libkaren.h;
cvs add karen/libkaren.c
$ cvs commit
```

We have to type in a comment like “Initial revision” when we’re prompted by the commit statement. Now we have our files in the repository and we can work on them along with the other developers. CVS can take care of certain situations of conflicts where many developers alter the same file at the same time. Everytime significant changes are done to a file, we have to run:

```
$ cvs update
$ cvs commit
```

If meanwhile another developer changes the repository, you’ll will be notified bye the ”cvs update” statement. If, by bad lack, he happens to modify the same file we’re working on, it will be clearly stated, if the altered parts of the file are distinguishably identifiable by CVS, it will merge them. Otherwise, it will modify the local copy to highlight the conflicts between the modifications and lets the developers solve the conflict manually. The history command gives the details of all the transactions performed on the repository:

```
$ cvs history -ae
O 2003-01-07 22:19 +0000 jamel karen =karen= ~/doc/dev/gnudev/cvs/*
A 2003-01-07 22:24 +0000 jamel 1.1 karen.c karen == ~/doc/dev/gnudev/cvs
A 2003-01-07 22:24 +0000 jamel 1.1 libkaren.c karen == ~/doc/dev/gnudev/cvs
A 2003-01-07 22:24 +0000 jamel 1.1 libkaren.h karen == ~/doc/dev/gnudev/cvs
O 2003-01-07 22:38 +0000 imed  karen =karen= ~/cvs/*
O 2003-01-07 22:38 +0000 imed  karen =karen= ~/cvs/*
M 2003-01-07 22:38 +0000 imed  1.2 karen.c karen == ~/cvs
C 2003-01-07 22:39 +0000 jamel 1.2 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:40 +0000 jamel 1.3 karen.c karen == ~/doc/dev/gnudev/cvs
U 2003-01-07 22:40 +0000 imed  1.3 karen.c karen == ~/cvs/karen
M 2003-01-07 22:41 +0000 imed  1.4 karen.c karen == ~/cvs
U 2003-01-07 22:41 +0000 jamel 1.4 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:42 +0000 imed  1.5 karen.c karen == ~/cvs
G 2003-01-07 22:42 +0000 jamel 1.5 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:42 +0000 jamel 1.6 karen.c karen == ~/doc/dev/gnudev/cvs
```

If you find this hard to read, you may want to look at various CVS clients like WinCVS and jCVS. webcvs has a very elegant read-only interface.

## 3.2 Setting up a CVS server

The CVS binary works as both a client and a server, it’s available on avery modern Linux distribution. The thing is that its configuration is a bit tricky and, unfortunately, there are very little resources on the web about this topic. Anyway, this is how I set it up under SuSE Linux:

1. Create the repository
```
# mkdir /usr/local/src
# cvs -d /usr/local/src init
```
2. Definition of modules
```
# mkdir /tmp/junk
# cd /tmp/junk
# cvs -d /usr/local/src checkout CVSROOT/modules
# cat >> CVSROOT/modules
ps synapse/ps
baytar suse/src/baytar
^d
# cvs -d /usr/local/src commit
```
Add a comment on the operation and leave vi
```
# export CVSROOT=/usr/local/src
# cvs release -d CVSROOT
```
Answer "y" for yes
```
# cd /usr/local/src/
# mkdir -p synapse/ps; etc.
```
3. Set permissions, I assume all CVS users belong to the "cvsauthor" group:
```
# chgrp -R cvsauthor synapse suse
# chmod -R g+w synapse suse
```

4. Start the service: edit /etc/xinetd.d/cvs so that the "disable" line says "no", mine looks like follows:
```
$ cat /etc/xinetd.c/cvs
# CVS pserver (remote acces to your CVS repositories)
# Please read the section on security and passwords in the CVS manual,
# before you enable this.
# default:  off
service cvspserver
{
disable = no
socket_type = stream
protocol = tcp
wait = no
user = root
server = /usr/bin/cvs
server_args = -f --allow-root=/usr/local/src pserver
}
$
```
..and run:
```
# /etc/init.d/xinetd restart
```

5. Create user accounts: use htpasswd(1) which comes with Apache to generate the encrypted passwords,
```
$ cd /tmp/
$ htpasswd -c -d passwd imed
$ htpasswd -d passwd jamel
# cp /tmp/passwd /usr/local/src/CVSROOT/
```

6. That's it, from a machine on the network you may try:
```
$ export CVSROOT=:pserver:imed@cvs.suse.com.tn:/usr/local/src
$ mkdir cvs ; cd cvs
$ cvs checkout ps
```
Create and/or modify some source files, then:
```
$ cvs commit
```

What's called anonymous CVS access is simply an account with a blank password.

### 3.3 diff and patch

diff(1) is used to compare files and to generate a list of the differences. It's very useful if a developer wants to send changes of a project to users or other developers without having to send the whole source tree. Let's duplicate the karen/ directory containing the source tree of the project into karen.orig/ and let's change the files under karen/ by, say, adding comments.

```
$ cd ~/src
$ ls
karen karen.orig
$ diff -u --recursive --new-file karen.orig karen > karen-comments.patch
```

This should generate a file containing all the changes done to the project in a universal (-u) format that makes the localisation of the changes easier.

A developer wishing to apply this patch to its (outdated) source tree to update it to the lates revision or to fix a bug may use the patch(1) command:

```
$ cd ~/src/karen/
$ patch -p1 < ../karen-comments.patch
```

The Linux kernel developers rely heavily on this diff and patch process.