A Tutorial for
C/C++ Programming on Linux

Shridhar Daithankar

September 5th, 2004

Table of Contents

Introduction

I meet a lot of computer students on PLUG(Pune Linux Users Group). Invariably I find them using `TC` i.e. Turbo C for their C/C++ programming assignments. At PLUG, we attempt to help them getting started with C/C++ programming on Linux and answer any queries they have. This tutorial is to help them getting started on their own.

A lot of these students also tell me that they use `TC` because their teachers insist on using it. We encourage teachers as well as students to get away from `TC` as soon as and as far as possible. This tutorial is aimed at teachers and students alike.

There are several disadvantages of Turbo C. They lead to programming practices which are not exactly beneficial when a student completes his/her graduation and aspires to be a software professional.

- Most Turbo C installations uses a version which is very old and/or incomplete. It lacks essential features like namespaces and templates in C++. Hence students practice their skills on a platform which is limited.
- Usage Turbo C promotes DOS programming paradigm which is obsolete on all practical counts. Student keep thinking that directly writing to VGA memory is a big thing while it is a complete waste of efforts.
- Most Turbo C installations lack a complete C library. Hence students are completely unaware of some very important features such as threading or localization. Turbo C has a minimal C library installed with it, which is just enough for basic programming.
- TC being an IDE, students never grasp the concept of project management. Furthermore the assignments in the course do not promote good project management practices. Many students end up doing all their assignments in a single file and consequently never learn to reuse the code. Some of the essential practices students miss are,
    - Use of Makefiles
    - Usage of Version Control systems
    - Practice of modularizing the code in different files/libraries etc.
- Students are never exposed to diversity. They think TC is C/C++. What they do not understand is TC is but one implementation of C/C++. There are several others, each with it's own strengths, weaknesses and gotchas. They learn to think in terms of tools rather than language, theory and problem at hand. The lack of exposure to diverse tools keeps them from maturing rapidly.

This tutorial aims to help students and teachers enriching the learning experience in a undergraduate course. Though this tutorial explains C/C++ on Linux, I would encourage students to explore usage other compilers such a Microsoft Visual C/C++ and Intel C/C++.

I want to make it clear here that I am not going to illustrate usage of any IDE on Linux in this tutorial. Following are the reasons that I can think of, top of my head,

- I think they are easy enough to figure out if you know any other IDE or basic C/C++ programming.
- I intend to show how to use basic C/C++ usage. If I start with an IDE, it will be no different than `TC`. Students will think of '`Kdevelop` as C/C++' on which is no better than thinking '`TC` is C/C++'
- Explaining an IDE in totality is a big job. It could take a book rather than a tutorial to do it.

There could be more reasons but I think these are good enough to skip an IDE right now.

Starting C programming with Linux

Basic Checks

To start programming on Linux, you need a Linux installation, which has development packages installed. If you are not familiar with Linux installation, I would recommend getting help from friends/teachers or PLUG members.

Following example illustrates how to check existence of development tools. You should get a similar output. All these commands were executed on my home PC which runs Slackware 10.

```
shridhar@darkstar:~$ which gcc
/usr/bin/gcc
shridhar@darkstar:~$ which g++
/usr/bin/g++
shridhar@darkstar:~$ which make
/usr/bin/make
shridhar@darkstar:~$ which vi
/usr/bin/vi
shridhar@darkstar:~$ which pico
/usr/bin/pico
```

If you have afore-mentioned tools available, you are set to start. The last two, `vi` and `pico` are text editors. You can do with any text editor of your choice. However if you don't know where to look for, above are two very basic choices. Personally, I use `kate` which is bundled with KDE.

I have put an entire section at the end of the document, illustrating these editors, especially `vi`. From here, I will assume that you have one editor available for writing code.

The first encounter

Let us write a small program and compile it. Run command `pico` from command line and type in following code.

```
#include <stdio.h>
int main(void)
{
 printf("Hello World\n);
 return(0);
}
```

Press `Ctrl-X` to quit. Pico will ask to save the file. Press `Y`. It will ask for file name. Supply a name of your choice. I chose `hello.c` here.

Let us compile the code.

```
shridhar@darkstar:~$ gcc -o hello hello.c
shridhar@darkstar:~$ ls -la hello*
-rwxr-xr-x  1 shridhar users 10584 2004-07-29 08:01 hello*
-rw-r--r--  1 shridhar users    78 2004-07-29 08:01 hello.c
```

The first command compiled the file `hello.c` to a executable `hello`. There was no error or warning given. It means that everything went fine. This is Unix way of doing things.

However for people from DOS/Windows background, this could be confusing. So I issued an `ls` command, which is similar to `dir` command on DOS. This confirms that the executable called as `hello` was indeed created.

Let us run the program.

```
shridhar@darkstar:~$ ./hello
Hello World
```

```
shridhar@darkstar:~$
```

Here we run the program. There are several things to note in above example. First of all, the current directory '.' is included explicitly in file name. On Unix, the current directory is not necessarily included in the path. So running the executable with full path is recommended as a good portable practice, even if you don't have to do it on your Linux installation.

Secondly the directory separator is '/' on Linux rather than '\' on DOS. If you are using Linux for some time, I am sure you have noticed it by now. The reasons behind the difference are topic of a continuous debate between Windows/DOS and Unix users. Let us skip them for now.

Summary

Here we have learnt several things.

- On Linux, editing a file and compiling are two different things.
- The compiler is a command line tool which has nothing to do with the way you create/edit source code.
- The C compiler on Linux is called as `gcc`.
- The directory separator on Linux is '/' instead of '\'.

C/C++ Compiler

The C compiler on Linux is a part of compiler suite, known as GCC(GNU Compiler Collection). This suite offers compilers for several languages. Following is a list of typical ones offered.

- C
- C++
- Objective C
- Fortran

The name of C compiler program on Linux is `gcc` and C++ compiler is called as `g++`. You can find out the version of the compiler using `-version` option. This is the output produced on my home machine.

```
shridhar@darkstar:~$ gcc --version
gcc (GCC) 3.3.4
Copyright (C) 2003 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Command-line Options

The command line options to the C/C++ compiler control their behavior. There are large number of options available. The compiler documentation has the detailed explanation but I will cover the commonly used ones.

A compiler option is denoted by '`-`' such as `-s`. This is different than Dos/Windows where the options are denoted by '/'. In our first program, you can see that we have already made use of `-o` option.

Note that these options are case-sensitive.

`-c`

This option instructs the compiler to just compile the file and produce an object file, instead of creating a program. Creating a program is the default behavior.

This is typically used when a program or a library consist of more than one source files and/or the sources spread across multiple modules. The object files produced has a `.o` extension rather than `.obj` as with Dos/Windows.

-o <name>

This option instructs the compiler to produce the target with a specific name, overriding the default name.

When a source is compiled into an object file, the extension changes to `.o` e.g. A source file `hello.c` will produce an object file named `hello.o`. However you can change the name of object file using this option. Following commands demonstrate this usage.

```
shridhar@darkstar:~$ gcc -c -o hello1.o hello.c
shridhar@darkstar:~$ ls -al hello1.o
-rw-r--r--  1 shridhar users 844 2004-09-05 20:36 hello1.o
```

If this option is not specified, the name of program produced is always `a.out`. This is different than Dos/Windows where a source file `hello.c` results into a program with name `hello.exe`. Hence on Linux, this option is almost always used while producing the program.

-O<n>

This option instructs compiler to produce optimized programs. Here `n` denotes the level of optimization. This is an optional argument. The level of optimization range from 1 to 3. The most commonly used optimization level is 2.

There are several options that control specific optimizations. This option is a convenient way of specifying a group of most commonly used. More details on these options are available in compiler documentation.

-g

This options instructs compilers to produce a program with debug information included. Unless a program is compiled with this option, it can not be debugged.

If a program is created from more than one object files, all of the object files must be compiled with this flag, so that the entire program can be debugged.

-s

This option instructs the compiler to remove any symbol and object relocation information from the program. This is used to reduce the size of program and runtime overhead.

Coupled with `-O2`, these options produced programs that are used in production. Note that this option should not be used in conjunction with `-g` as it removes the debugging information as well.

-I <directory name>

This option instructs compiler to add the specified directory to include search path. Compiler will search in directory when it is looking for header files included by the programs.

By default, the compiler search in directory `/usr/include` and hence it need not be specified. The `#include` directive in source can take relative form. Let us say a program has a line such as follows.

```
#include <sys/socket.h>
```

Then the compiler will match the file `/usr/include/sys/socket.h`. Compiler will attempt to find a file among all the include directory path specified before throwing an error.

For C++, `/usr/include/c++/<version>` is the additional default include directory, where standard library headers for C++ are located. Here version is the compiler version. So on my machine, it translates to the directory `/usr/include/c++/3.3.4/`.

`-L <directory name>`

This option instructs the compiler to add the specified directory to library search path. This option is actually used by linker. However since linker is invoked via compiler in most cases, this option is passed to the compiler. The compiler passes it to the linker.

By default compiler searches the directory `/usr/lib`. I will describe libraries in more details later.

`-l <library name>`

This option instructs the compiler to link against the specified library. This option follows a specific naming convention. The library name specified does not include library name suffix or prefix. E.g. To link against a library `libnurses.so`, one has to specify `-lncurses` since `lib` and `.so` are standard library suffix/prefix.

Libraries

A library is like a executable program in that it contains the compiled code in machine specific assembly language. It differs from a program in that libraries are collection of reusable code. They are no meant to be run like a normal programs.

Using libraries in programs

To use libraries with a program, one needs corresponding header files and libraries. The header files are included in code. The compiler option `-I`, explained above, tells compiler where to find these specific headers. The compiler can obtain function declarations from these header files. After compilation, the compiler produces object files which has empty slots for functions/symbols[1] declared in library header files. Later linker fills in these slots.

To produce a program, compiler invokes linker with appropriate linker directories and libraries. The linker puts all the object files together in the final program. It creates a list of empty slots from all the object files. Then it searches for these symbols in libraries specified. For each symbol found in libraries , it marks the library as a dependency.

If it can not find a symbol in any of libraries specified, it throws a `Undefined symbols` error. It means that the linker could not find any library which contains the symbol definition. Necessary libraries need to be specified so as to get the program linked successfully.

Types of libraries

There are two types of libraries, shared and static. They differ in how the compiled code is reused by the programs.

---

1 Linker calls function and variable declarations as symbols. It does not deal with functions alone and it is not specific to a particular language such as C/C++. It can link object files produced by any language as long as they are in specific format..

- Static Libraries

When a program is linked against a static library, the linker copies the symbol definition i.e. the code for function implementation into the resulting program. Hence the program does not need the library installed in order to run. This results in a bigger program size at the cost of ease of installation. Furthermore to take advantage of newer version of a library, the program must be recompiled and reinstalled.

By convention, static libraries has a `lib` prefix and `.a` extension. Thus a `libmyprog.a` is file name for library `myprog`. While compiling/linking, only the library name is passed and not the complete file name as linker can locate the file from the library name.

- Shared Libraries

On the other hand, while linking against a shared library, the linker marks the symbols in shared library as external. While running the program, the runtime linker searches through installed libraries for necessary library and the required symbol definition in the library. If either the library or the necessary symbol definition is not found, a runtime error is thrown and program execution is aborted.

Using shared libraries, the program size can be kept to minimum. If more than one programs are using same library, only one copy of library is loaded saving memory at the runtime. This is not possible with static libraries. If the installed library is upgraded, all the programs depending upon it get the benefit of newer version.

The standard prefix for shared libraries is `lib` and file extension is `.so`. Thus a library `myprog` will have the file name `libmyprog.so`.

Creating a library

Creating a shared library is very similar to creating a program. One has to include appropriate headers and specify additional libraries for linker. A shared library can depend upon other shared libraries. In order to run the program, all the dependent libraries are required to be installed.

However some additional flags need to be passed to compiler and linker to create a shared library. On Linux, following are the necessary flag.

`-fPIC`

This option instructs the compiler to generate position independent code. This is a necessary option to create an object file that can be put into shared library. It prevents the compiler from using the optimizations that are dependent upon code/symbol location since these locations are not guaranteed in a shared library.

`-shared`

This option is passed to linker. Instead of creating a program the linker creates a shared library.

To create a static library, one must produce a set of object files and pass them to a utility called as `ar`.

Libraries and running a program

When a program is invoked, a system program called as runtime linker is run before running the user code. This program is a library and the compiler links every program against it. The compiler creates necessary code so that this program is run before running any of the user code.

The runtime linker loads the program like any other normal file and reads all the dependencies marked by the

linker.

Then it searches the required libraries in the system. A system-wide configuration file `/etc/ld.so.conf`, lists the directories searched by the runtime linker. Once it locates the necessary libraries, it loads them and searches for the symbol definitions that are marked as external in the program. In order to run the program, the runtime linker has to locate all the external symbols in the program.

The runtime linker loads the necessary libraries in the programs address space and recalculates the address of each symbol in the shared library. After this step is completed, the control is passed to the user code.

To find out the shared library dependencies of a program, a utility called as `ldd` can be used. For our first example, the dependencies are listed as follows.

```
shridhar@darkstar:~$ ldd hello
        libc.so.6 => /lib/libc.so.6 (0x4002d000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Another utility called as `nm` can be used to list all the symbols in a program. Again, for our first example program, the listing is as follows. Note that the function printf is marked as 'U', which means that another shared library has to supply the implementation of this function. The particular shared library is the standard C library `libc.so` in this case.

```
shridhar@darkstar:~$ nm hello
08049494 D _DYNAMIC
08049570 D _GLOBAL_OFFSET_TABLE_
08048470 R _IO_stdin_used
         w _Jv_RegisterClasses
08049560 d __CTOR_END__
0804955c d __CTOR_LIST__
08049568 d __DTOR_END__
08049564 d __DTOR_LIST__
08049490 r __FRAME_END__
0804956c d __JCR_END__
0804956c d __JCR_LIST__
08049588 A __bss_start
08049484 D __data_start
08048420 t __do_global_ctors_aux
08048310 t __do_global_dtors_aux
08049488 D __dso_handle
08049484 A __fini_array_end
08049484 A __fini_array_start
         w __gmon_start__
08049484 A __init_array_end
08049484 A __init_array_start
080483e0 T __libc_csu_fini
080483b0 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
08049588 A _edata
0804958c A _end
08048450 T _fini
0804846c R _fp_hw
08048278 T _init
080482c0 T _start
080482e4 t call_gmon_start
08049588 b completed.1
```

```
08049484 W data_start
08048350 t frame_dummy
08048384 T main
0804948c d p.0
         U printf@@GLIBC_2.0
```

The C/C++ programming environment

Man pages for C calls

POSIX is a standard which specifies the programming calls that a standard Unix operating system must provide. These calls are implemented either in kernel or C standard library. Linux is a POSIX compliant OS and hence provides all of these calls.

In order to find out what these calls are, I would generally recommend a book on Unix programming such as one from Kernnighan and Ritchie. However there is a simpler reference to all these calls that is installed on each Linux system .These are man(manual) pages accessible using `man` command.

All the manual pages for programming calls are located in directory `/usr/man/man2`. These man pages follow a specific naming standard. A typical man page `/usr/man/man2/unlink.2.gz` means that it documents a command called as `unlink` in section 2 of the manual. This section number is required if two sections contain man page with same name. In this case, the required `man` command to be issued on shell prompt would be '`man 2 unlink`'.

Even though you can use the `man` command to view man pages, I would recommend a GUI viewer such a `konqueror` in KDE since it can hyperlink other man pages and source file.

C/C++ standard library documentation

The C/C++ libraries on Linux are provided by the GNU project. These libraries are developed along with the compiler. Detailed documentation for these libraries are available from the respective websites. The necessary URLs are as follows.

- GNU Libc Documentation at http://www.gnu.org/software/libc/manual/
- C++ standard library documentation at http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html
- libstdc++ FAQ at http://gcc.gnu.org/onlinedocs/libstdc++/faq/

STL documentation

STL is Standard Template Library which provides many useful containers and algorithms that can be used in C++. The standard C++ library on Linux includes an STL implementation.

To use STL effectively, I recommend the STL documentation from SGI.

- STL documentation from SGI at http://www.sgi.com/tech/stl/

Debugging

When a program does not work as expected, one has to debug it. Debugging is a integral part of software development. So to learn to develop programs on Linux, we need to learn to debug them as well.

The most primitive technique of debugging is to introduce plethora of print statements in the programs to execute state of program at each stage. It certainly work for some class of problems. But it is no replacement for a proper debugger.

I will cover gdb in this chapter which is a command line debugger on Linux. While there are GUI front-ends such as DDD available for gdb, I will cover gdb only since it is the minimum required for debugging. Again no matter what Linux system you encounter, you can count on gdb to be available.

Preparing and debugging the program

A program can not be debugged unless it contains debug information. Let us start with a fresh program to demonstrate debugging.

Type following program in a file called as print.c.

```
#include <stdio.h>

int main(void)
{
 int i=0;
 printf("The value of i is %d\n",i);

 return(0);
}
```

Let us compile it for debugging and check it's size.

```
shridhar@darkstar:~$ gcc -g -o print print.c
shridhar@darkstar:~$ ls -al print
-rwxr-xr-x  1 shridhar users 15880 2004-09-05 23:51 print*
shridhar@darkstar:~$
```

Notice the '-g' flag passed to compiler. It instructs to add debugging information in the program. Let us run the program.

```
shridhar@darkstar:~$ ./print
The value of i is 0
```

It just runs like any other program. The next step is debugging. Here is a gdb session which runs the program.

```
shridhar@darkstar:~$ gdb print
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i486-slackware-linux"...Using host
libthread_db library "/lib/libthread_db.so.1".

(gdb) r
Starting program: /home/shridhar/print
The value of i is 0

Program exited normally.
```

```
(gdb) q
shridhar@darkstar:~$
```

The debugger accepts the name of program as the argument. You are dropped to a prompt where you can enter the debugger commands. Here the command '`r`' runs the program and '`q`' quits the debugger.

Breakpoints and variable values

Breakpoints allow to halt program execution and gives control to the debugger. The programmers can inspect and/or change variable values at break points.

Following session demonstrates these features of gdb.

```
(gdb) b main
Breakpoint 1 at 0x8048394: file print.c, line 5.
(gdb) r
Starting program: /home/shridhar/print
Breakpoint 1, main () at print.c:5
5        int i=0;
(gdb) print i
$3 = 1073826048
(gdb) n
6        printf("The value of i is %d\n",i);
(gdb) print i
$4 = 0
(gdb) n
The value of i is 0
8        return(0);
(gdb) n
9       }
(gdb) n
0x40042936 in __libc_start_main () from /lib/libc.so.6
(gdb) n
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
```

Note that the user gets control at break point. The source line shown is the one which will be executed**next**. Hence the value of i is random when printed first and it is set to zero after the statement is executed. The command '`n`' is used to run one statement at a time. If the line contains a user defined function call, then command '`s`' steps into the function rather than treating the entire line as one step.

Break points can be set in two ways. One way is to specify a function as we have done here. Other way is t specify in form of `filename:line number`. This is how it is done.

```
(gdb) b print.c:5
Breakpoint 2 at 0x8048394: file print.c, line 5.
```

By default, gdb searches the source files in current directory. You can tell it to search specific directories using `dir` command.

```
(gdb) dir /home/shridhar
Source directories searched: /home/shridhar:$cdir:$cwd
(gdb) dir /mnt4/shridhar/Downloads
Source directories searched: /
```

```
mnt4/shridhar/Downloads:/home/shridhar:$cdir:$cwd
```

Note that each invocation of `dir` command adds the directory to the list of directories to be searched, rather than replacing it. Unfortunately `gdb` does not understand directory tree. So if you have a multilevel source tree, you can not just tell `gdb` the top level directory and expect it to search entire tree. You need to specify each directory separately.

Changing variable values

You can change a variable value while you are debugging. It allows a developer to test a potential bug fix without recompiling the program. This is how it is done.

```
(gdb) r
Starting program: /home/shridhar/print

Breakpoint 2, main () at print.c:5
5        int i=0;
(gdb) n
6        printf("The value of i is %d\n",i);
(gdb) print i
$5 = 0
(gdb) set var i=10
(gdb) print i
$6 = 10
(gdb) n
The value of i is 10
8        return(0);
(gdb) n
9       }
(gdb) n
0x40042936 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

GDB command reference

What do I miss from Turbo C?

Linux and Dos are two different operating systems. Turbo C and GCC are two different compilers. Consequently not everything from Dos/Turbo C can be used on Linux/GCC. This section lists some of the things missed most frequently.

Headers

dos.h

This is a header in Turbo C that provides interrupt related functions and entry to Dos kernel. This header is not available on Linux.

Linux does not allow direct use of interrupts from user application programs. Unlike Dos, Linux is a multi-user multi-tasking OS which provides standard IPC (interprocess communication) and other APIs.

Linux can do all the things provided by this header and much more. However the syntax and APIs are drastically different. If you have any programs that rely on this header, you need to rewrite them in Linux.

conio.h

This header file in Turbo C provides functions that can manipulate screen and cursor. On Linux, one needs to use a library called as `ncurses` to achieve similar effects. The discussion of porting to `ncurses` is out of scope of this tutorial.

Graphics

Turbo C provides a simple graphics programming API that can handle VGA graphics and provide low level routines for graphics manipulation.

On Linux, a library called as SVGALib is available for similar effects. However this library is discontinued in favor of other options. Direct frame buffer rendering is another option available.

However it is strongly recommended that instead of using any of these options, one directly use X and higher level toolkits such as GTK/Qt/FLTK. The discussion of X and any of these toolkits is beyond the scope of this document. There is plenty of material available on internet for these topics.

Whats next?

Further reading

This document touches on many issues but can not cover them in detail. I recommend that the reader pursue following subjects as well. These are important tools for developing professional and large scale software applications. I will provide a brief description of each of these.

Make

`Make` is a program to maintain a project build in a consistent state. It reads the project description in form of dependencies of sources and targets. In context of programming, sources files are source and programs are targets. However the definition of sources and targets is provided by application developer. Hence `Make` can be used in variety of creative ways.

The source target dependency description is provided in a file called as Makefile. The `Make` documentation covers the format of Makefiles.

Based on timestamps of each sources and target, it determines the targets that needs to be updated. It updates only those targets that are older than corresponding sources.

A source can be a source file or another target e.g. An object file can be source for a program though it is not a source file. Make can nest the dependencies so all the dependent targets are rebuilt. In this example, if the source file is updated, it causes the object file to be rebuilt and hence the program that depends upon the object file.

`GNU Make` is installed on most Linux systems. It provides some extensions over traditional Unix `Make`. There are some other programs that provide similar functionality. `Jam` from Perforce Inc. and `scons/cons` are two such programs.

Autoconf

Autoconf is a system of tests that tests availability of certain features in operating system. It is used for building software from same sources on multiple platforms. If a test succeeds, a preprocessor directive is set in a configuration header file. Application developer can add code for the particular system if the directive is defined.

Autoconf is written in `m4` which is a macro processing language available on most Unix systems.

It is responsibility of application developer to write tests. However since large number of Free software projects use Autoconf, there is a rich set of tests already made and which can be reused.

Automake

Automake is a tool that makes writing makefiles easy. It has a simpler syntax compared to `Make`. Automake processes a file called as `Makefile.am` and produce Makefile. The produced makefile has many useful built-in targets such as `clean`, `install`/`uninstall` etc.

It supports constructs like directory recursion, program and library creation in a much simpler manner compared to make.

Version Control Systems

A version control system keeps tracks of changes in a set of file. In a software project, it can keep track of every change made in sources and hence one can retrieve any older version for comparison.

Typical version control operations include
- Checking in a file with changes
- Comparing two revisions of a file based on revision number or dates
- Maintaining a change log which describes why each change was made. [2]
- Retrieving any particular version of any file
- Branching of source code allowing parallel development
- Creating and applying patches to source tree

A version control system brings order and maintainability to a software project. Without a version control system, it is not possible to maintain a software project of considerable size.

CVS and SubVersion are two version control systems that are widely used in Free software projects. One should be familiar with at least one of them.

Alternate Operating Systems

Linux is just one of the many Free Operating Systems available. There are many other OS's which are Unix and Free software. BSD family of Unix which includes FreeBSD/NetBSD/OpenBSD are the most popular among such operating systems.

As explained earlier, diversity is very important from a software developer's point of view. It brings maturity and experience that is invaluable.

Once you are familiar and comfortable with Linux, I recommend that you fiddle with at least one other OS. FreeBSD is the closest. Developing on FreeBSD with expose you to cultural differences between these two which is a good learning experience in my opinion.

Contact information

Editors

As we have learnt in last section, any text editor can be used to write C/C++ programs. There are plethora of them available on Linux. Here is brief introduction for two of them with mention of others. The idea is to

2Of course, entering a correct change log is application developers responsibility.

make sure that if a particular editor is unavailable, the other one can be used.

If you don't know any editor in Linux, read one of following sections to start with.

Vi

Vi is very age-old and powerful editor. Typically, Linux has a flavor of vi, called as `Vim`(Vi Improved) installed.

Vi has a reputation of being tough for newbies. However it's user base swears by it. It should be easy enough to use if the command reference provided here is handy. Of course it would be little unfamiliar to start with but that phase will last for first few days only. Learning vi is recommended because you will find it available no matter what Unix OS you use.

Vi operates in two modes, command mode and editing mode. By default, it starts in command mode. So anything you type is treated as a command. Any one of the editing commands starts the editing mode. To switch back to command mode, `Esc`(escape) is used.

If you start vi with a file, you can move around the file using arrow keys but you can not edit it because you are in command mode. If you start vi with no file, there is nothing to move around.

Remember that vi commands are case sensitive. Also vim allows normal navigation controls such as arrows keys, page up/page down to be used. But traditional vi key sequences are still mentioned for compatibility with other Unix OSs.

A brief vi command reference can be found in appendix. Vim also has a GUI version gvim which can be used with menus and all other GUI goodies. GVim is also available for Windows.

Pico

Pico is a very simple editor. It is the default editor in text mail client pine. The help can easily be obtained using `Control-g`. The editor has a on-screen help menu which covers most basic operations. It does not have modes i.e. unlike Vi, you just start typing. All the commands are paired with Control key.

Joe

This is another very simple editor. It has a wordstar like key bindings and can be easy to use for people familiar with wordstar.

One thing to remember is that pico and joe are not as easily available as vi on other Unix OS.

Others

Following are other editors that are typically available on a Linux system. Note that Emacs is not an editor but virtually an OS in itself. There are/used-to-be religious wars between vi and emacs fans. However I am not covering it here because I have never used it personally.

- Emacs
- GNOME editors such as GEdit
- KDE editors such as Kate/KWrite
- NEdit

You can use any of these. The choice is yours. Use an editor that suits your needs.

Appendix

Brief Vi command reference

File Handling

Command	***Explanation***	***Command***	***Explanation***
:q	Quit the editor	:q!	Quit the editor without saving
:w	Save the file being edited	:e file	Open file for editing.
:n#	Switch to next file for editing	:!command	Execute shell command such as `ls`

Editing

Command	***Explanation***	***Command***	***Explanation***
i	Start Editing where currnt cursor is	o	Start editing on a new line after current line
O	Start editing on a new line before current line	x	Delete current character
a	Append at the current position. Starts next to cursor	.	Repeat last command.(This is not really a editing command. It can be used with any other command)

Navigation

Command	***Explanation***	***Command***	***Explanation***
l	Move cursor to right	h	Move cursor to left
j	Move cursor down one line	k	Move cursor up one line
Control-F	Page down	Control-B	Page up
Shift-g	Go to the end of the file	Control-g	Tell position in file
^	Start of current line	$	Endof current line

Buffer Handling

Command	***Explanation***	***Command***	***Explanation***
dd	Cut the current line	*n*dd	Cut n line starting from current line e.g. 3Dd cuts 3 lines
p	Paste buffer after current line. There is no *n*p command. All the lines cut/copied are pasted.	P	Paste buffer before current line

Setting Options

Command	*Explanation*	*Command*	*Explanation*
:set nu	Turn line numbering on	:set ai	Set auto indent on
:set all	Show all typical options and their values	:set everything	Show all options. This listing is huge

Search and replace

Command	*Explanation*	*Command*	*Explanation*
/<pattern>	Search the pattern in forward direction	?<pattern>	Search the pattern in backward direction
n	Repeat last search		