filename: debian_reference_ch12-programming_5pp_20241112.txt
https://www.debian.org/doc/manuals/debian-reference/ch12.en.html

Debian Reference

Chapter 12. Programming

I provide some pointers for people to learn programming on the Debian system enough to trace the packaged source code. Here are notable packages and corresponding documentation packages for programming.

Online references are available by typing "man name" after installing manpages and manpages-dev packages. Online references for the GNU tools are available by typing "info program_name" after installing the pertinent documentation packages. You may need to include the contrib and non-free archives in addition to the main archive since some GFDL documentations are not considered to be DFSG compliant.

Please consider to use version control system tools. See Section 10.5, "Git".

Do not use "test" as the name of an executable test file. "test" is a shell builtin.

```
*************************************************************************************************
* You should install software programs directly compiled from source into "/usr/local" or "/opt" to *
* avoid collision with system programs.                                                         *
*************************************************************************************************
```

12.1. The shell script

The shell script is a text file with the execution bit set and contains the commands in the following format.

```
#!/bin/sh
 ... command lines
```

The first line specifies the shell interpreter which read and execute this file contents.

Reading shell scripts is the best way to understand how a Unix-like system works. Here, I give some pointers and reminders for shell programming. See "Shell Mistakes" (https://www.greenend.org.uk/rjk/2001/04/shell.html) to learn from mistakes.

Unlike shell interactive mode (see Section 1.5, "The simple shell command" and Section 1.6, "Unix-like text processing"), shell scripts frequently use parameters, conditionals, and loops.

12.1.1. POSIX shell compatibility

Many system scripts may be interpreted by any one of POSIX shells (see Table 1.13, "List of shell programs").

- The default non-interactive POSIX shell "/usr/bin/sh" is a symlink pointing to /usr/bin/dash and used by many system programs.
- The default interactive POSIX shell is /usr/bin/bash.

Avoid writing a shell script with bashisms or zshisms to make it portable among all POSIX shells. You can check it using checkbashisms(1).

Table 12.1. List of typical bashisms

Good: POSIX	Avoid: bashism
`if [ "$foo" = "$bar" ] ; then ...`	`if [ "$foo" == "$bar" ] ; then ...`
`diff -u file.c.orig file.c`	`diff -u file.c{.orig,}`
`mkdir /foobar /foobaz`	`mkdir /foo{bar,baz}`
`funcname() { ... }`	`function funcname() { ... }`
octal format: "\377"	hexadecimal format: "\xff"

The "echo" command must be used with following cares since its implementation differs among shell builtin and external commands.

- Avoid using any command options except "-n".
- Avoid using escape sequences in the string since their handling varies.

Although "-n" option is not really POSIX syntax, it is generally accepted.

Use the "printf" command instead of the "echo" command if you need to embed escape sequences in the output string.

12.1.2. Shell parameters

Special shell parameters are frequently used in the shell script.

Table 12.2. List of shell parameters

shell parameter	value
$0	name of the shell or shell script
$1	first (1st) shell argument
$9	ninth (9th) shell argument
$#	number of positional parameters
"$*"	"$1 $2 $3 $4 ... "
"$@"	"$1" "$2" "$3" "$4" ...
$?	exit status of the most recent command

$$	PID of this shell script
$!	PID of most recently started background job

Basic parameter expansions to remember are as follows.

Table 12.3. List of shell parameter expansions

parameter expression form	value if var is set	value if var is not set
${var:-string}	"$var"	"string"
${var:+string}	"string"	"null"
${var:=string}	"$var"	"string" (and run "var=string")
${var:?string}	"$var"	echo "string" to stderr (and exit with error)

Here, the colon ":" in all of these operators is actually optional.
  * with ":" = operator test for exist and not null
  * without ":" = operator test for exist only

Table 12.4. List of key shell parameter substitutions

parameter substitution form	result
${var%suffix}	remove smallest suffix pattern
${var%%suffix}	remove largest suffix pattern
${var#prefix}	remove smallest prefix pattern
${var##prefix}	remove largest prefix pattern

12.1.3. Shell conditionals

Each command returns an exit status which can be used for conditional expressions.
  * Success: 0 ("True")
  * Error: non 0 ("False")

"0" in the shell conditional context means "True", while "0" in the C conditional context means "False".

"[" is the equivalent of the test command, which evaluates its arguments up to "]" as a conditional expression.

Basic conditional idioms to remember are the following.
  * "command && if_success_run_this_command_too || true"
  * "command || if_not_success_run_this_command_too || true"
  * A multi-line script snippet as the following

```
if [ conditional_expression ]; then
        if_success_run_this_command
else
        if_not_success_run_this_command
fi
```

Here trailing "|| true" was needed to ensure this shell script does not exit at this line accidentally when shell is invoked with "-e" flag.

Table 12.5. List of file comparison operators in the conditional expression

equation	condition to return logical true
-e file	file exists
-d file	file exists and is a directory
-f file	file exists and is a regular file
-w file	file exists and is writable
-x file	file exists and is executable
file1 -nt file2	file1 is newer than file2 (modification)
file1 -ot file2	file1 is older than file2 (modification)
file1 -ef file2	file1 and file2 are on the same device and the same inode number

Table 12.6. List of string comparison operators in the conditional expression

equation	condition to return logical true
-z str	the length of str is zero
-n str	the length of str is non-zero
str1 = str2	str1 and str2 are equal
str1 != str2	str1 and str2 are not equal
str1 < str2	str1 sorts before str2 (locale dependent)
str1 > str2	str1 sorts after str2 (locale dependent)

Arithmetic integer comparison operators in the conditional expression are "-eq", "-ne", "-lt", "-le", "-gt", and "-ge".

12.1.4. Shell loops

There are several loop idioms to use in POSIX shell.

* "for x in foo1 foo2 ... ; do command ; done" loops by assigning items from the list "foo1 foo2 ..." to variable "x" and executing "command".
* "while condition ; do command ; done" repeats "command" while "condition" is true.
* "until condition ; do command ; done" repeats "command" while "condition" is not true.
* "break" enables to exit from the loop.
* "continue" enables to resume the next iteration of the loop.

The C-language like numeric iteration can be realized by using seq(1) as the "foo1 foo2 ..." generator.

See Section 9.4.9, "Repeating a command looping over files".

12.1.5. Shell environment variables

Some popular environment variables for the normal shell command prompt may not be available under the execution environment of your script.

* For "$USER", use "$(id -un)"
* For "$UID", use "$(id -u)"
* For "$HOME", use "$(getent passwd "$(id -u)"|cut -d ":" -f 6)" (this works also on
Section 4.5.2, "The modern centralized system management")

12.1.6. The shell command-line processing sequence

The shell processes a script roughly as the following sequence.

* The shell reads a line.
* The shell groups a part of the line as one token if it is within "..." or '...'.
* The shell splits other part of a line into tokens by the following.
    + Whitespaces: space tab newline
    + Metacharacters: < > | ; & ( )
* The shell checks the reserved word for each token to adjust its behavior if not within "..." or '...'.
    + reserved word: if then elif else fi for in while unless do done case esac
* The shell expands alias if not within "..." or '...'.
* The shell expands tilde if not within "..." or '...'.
    + "~" --> current user's home directory
    + "~user" --> user's home directory
* The shell expands parameter to its value if not within '...'.
    + parameter: "$PARAMETER" or "${PARAMETER}"
* The shell expands command substitution if not within '...'.
    + "$(command)" --> the output of "command"
    + "` command `" --> the output of "command"
* The shell expands pathname glob to matching file names if not within "..." or '...'.
    + * --> any characters
    + ? --> one character
    + [...] --> any one of the characters in "..."
* The shell looks up command from the following and execute it.
    + function definition
    + builtin command
    + executable file in "$PATH"
* The shell goes to the next line and repeats this process again from the top of this sequence.

Single quotes within double quotes have no effect.
Executing "set -x" in the shell or invoking the shell with "-x" option make the shell to print all of commands executed. This is quite handy for debugging.

12.1.7. Utility programs for shell script

In order to make your shell program as portable as possible across Debian systems, it is a good idea to limit utility programs to ones provided by essential packages.

* "aptitude search ~E" lists essential packages.
* "dpkg -L package_name |grep '/man/man.*/'" lists manpages for commands offered by package_name package.

Table 12.7. List of packages containing small utility programs for shell scripts

package	popcon	size	description
dash	V:884, I:997	191	small and fast POSIX-compliant shell for sh
coreutils	V:880, I:999	18307	GNU core utilities
grep	V:782, I:999	1266	GNU grep, egrep and fgrep
sed	V:790, I:999	987	GNU sed
mawk	V:442, I:997	285	small and fast awk
debianutils	V:907, I:999	224	miscellaneous utilities specific to Debian
bsdutils	V:519, I:999	356	basic utilities from 4.4BSD-Lite
bsdextrautils	V:596, I:713	339	extra utilities from 4.4BSD-Lite
moreutils	V:15, I:38	231	additional Unix utilities

Although moreutils may not exist outside of Debian, it offers interesting small programs. Most notable one is sponge(8) which is quite useful when you wish to overwrite original file.

See Section 1.6, "Unix-like text processing" for examples.

12.2. Scripting in interpreted languages

Table 12.8. List of interpreter related packages

package	popcon	size	documentation
dash	V:884, I:997	191	sh: small and fast POSIX-compliant shell for sh

```
bash                      | V:838, I:999 | 7175 | sh: "info bash" provided by bash-doc
mawk                      | V:442, I:997 | 285  | AWK: small and fast awk
gawk                      | V:285, I:349 | 2906 | AWK: "info gawk" provided by gawk-doc
perl                      | V:707, I:989 | 673  | Perl: perl(1) and html pages provided by perl-doc and
                          |              |      | perl-doc-html
libterm-readline-gnu-perl | V:2,   I:29  | 380  | Perl extension for the GNU ReadLine/History Library: perlsh(1
)
libreply-perl             | V:0,   I:0   | 171  | REPL for Perl: reply(1)
libdevel-repl-perl        | V:0,   I:0   | 237  | REPL for Perl: re.pl(1)
python3                   | V:718, I:953 | 81   | Python: python3(1) and html pages provided by python3-doc
tcl                       | V:25,  I:218 | 21   | Tcl: tcl(3) and detail manual pages provided by tcl-doc
tk                        | V:20,  I:211 | 21   | Tk: tk(3) and detail manual pages provided by tk-doc
ruby                      | V:86,  I:208 | 29   | Ruby: ruby(1), erb(1), irb(1), rdoc(1), ri(1)
```

When you wish to automate a task on Debian, you should script it with an interpreted language first. The guide line for the choice of the interpreted language is:

- Use dash, if the task is a simple one which combines CLI programs with a shell program.
- Use python3, if the task isn't a simple one and you are writing it from scratch.
- Use perl, tcl, ruby, ... if there is an existing code using one of these languages on Debian which needs to be touched up to do the task.

If the resulting code is too slow, you can rewrite only the critical portion for the execution speed in a compiled language and call it from the interpreted language.

12.2.1. Debugging interpreted language codes

Most interpreters offer basic syntax check and code tracing functionalities.

- "dash -n script.sh" - Syntax check of a Shell script
- "dash -x script.sh" - Trace a Shell script
- "python -m py_compile script.py" - Syntax check of a Python script
- "python -mtrace --trace script.py" - Trace a Python script
- "perl -I ../libpath -c script.pl" - Syntax check of a Perl script
- "perl -d:Trace script.pl" - Trace a Perl script

For testing code for dash, try Section 9.1.4, "Readline wrapper" which accommodates bash-like interactive environment.

For testing code for perl, try REPL environment for Perl which accommodates Python-like REPL (=READ + EVAL + PRINT + LOOP) environment for Perl.

12.2.2. GUI program with the shell script

The shell script can be improved to create an attractive GUI program. The trick is to use one of so-called dialog programs instead of dull interaction using echo and read commands.

Table 12.9. List of dialog programs

```
package   | popcon       | size | description
x11-utils | V:192, I:566 | 651  | xmessage(1): display a message or query in a window (X)
whiptail  | V:284, I:996 | 56   | displays user-friendly dialog boxes from shell scripts (newt)
dialog    | V:11, I:99   | 1227 | displays user-friendly dialog boxes from shell scripts (ncurses)
zenity    | V:76, I:363  | 183  | display graphical dialog boxes from shell scripts (GTK)
ssft      | V:0, I:0     | 75   | Shell Scripts Frontend Tool (wrapper for zenity, kdialog, and dialog with
          |              |      | gettext)
gettext   | V:56, I:259  | 5818 | "/usr/bin/gettext.sh": translate message
```

Here is an example of GUI program to demonstrate how easy it is just with a shell script. This script uses zenity to select a file (default /etc/motd) and display it.

GUI launcher for this script can be created following Section 9.4.10, "Starting a program from GUI".

```
<code>
#!/bin/sh -e
# Copyright (C) 2021 Osamu Aoki <osamu@debian.org>, Public Domain

# vim:set sw=2 sts=2 et:
DATA_FILE=$(zenity --file-selection --filename="/etc/motd" --title="Select a file to check") || \
( echo "E: File selection error" >&2 ; exit 1 )

# Check size of archive
if ( file -ib "$DATA_FILE" | grep -qe '^text/' ); then
        zenity --info --title="Check file: $DATA_FILE" --width 640  --height 400 \
        --text="$(head -n 20 "$DATA_FILE")"
else
        zenity --info --title="Check file: $DATA_FILE" --width 640  --height 400 \
        --text="The data is MIME=$(file -ib "$DATA_FILE")"
fi
</code>
```

This kind of approach to GUI program with the shell script is useful only for simple choice cases. If you are to write any program with complexities, please consider writing it on more capable platform.

12.2.3. Custom actions for GUI filer

GUI filer programs can be extended to perform some popular actions on selected files using additional extension packages. They can also made to perform very specific custom actions by adding your

specific scripts.
  * For GNOME, see NautilusScriptsHowto.
  * For KDE, see Creating Dolphin Service Menus.
  * For Xfce, see Thunar - Custom Actions and https://help.ubuntu.com/community/ThunarCustomActions.
  * For LXDE, see Custom Actions.

12.2.4. Perl short script madness

In order to process data, sh needs to spawn sub-process running cut, grep, sed, etc., and is slow. On the other hand, perl has internal capabilities to process data, and is fast. So many system maintenance scripts on Debian use perl.

Let's think following one-liner AWK script snippet and its equivalents in Perl.

```
awk '($2=="1957") { print $3 }' |
```

This is equivalent to any one of the following lines.

```
perl -ne '@f=split; if ($f[1] eq "1957") { print "$f[2]\n"}' |
perl -ne 'if ((@f=split)[1] eq "1957") { print "$f[2]\n"}' |
perl -ne '@f=split; print $f[2] if ( $f[1]==1957 )' |
perl -lane 'print $F[2] if $F[1] eq "1957"' |
perl -lane 'print$F[2]if$F[1]eq+1957' |
```

The last one is a riddle. It took advantage of following Perl features.
  * The whitespace is optional.
  * The automatic conversion exists from number to the string.
  * Perl execution tricks via command line options: perlrun(1)
  * Perl special variables: perlvar(1)

This flexibility is the strength of Perl. At the same time, this allows us to create cryptic and tangled codes. So be careful.

12.3. Coding in compiled languages

Table 12.10. List of compiler related packages

package	popcon	size	description
gcc	V:167, I:550	36	GNU C compiler
libc6-dev	V:248, I:567	12053	GNU C Library: Development Libraries and Header Files
g++	V:56, I:501	13	GNU C++ compiler
libstdc++-10-dev	V:14, I:165	17537	GNU Standard C++ Library v3 (development files)
cpp	V:334, I:727	18	GNU C preprocessor
gettext	V:56, I:259	5818	GNU Internationalization utilities
glade	V:0, I:5	1204	GTK User Interface Builder
valac	V:0, I:4	725	C# like language for the GObject system
flex	V:7, I:73	1243	LEX-compatible fast lexical analyzer generator
bison	V:7, I:80	3116	YACC-compatible parser generator
susv2	I:0	16	fetch "The Single UNIX Specifications v2"
susv3	I:0	16	fetch "The Single UNIX Specifications v3"
susv4	I:0	16	fetch "The Single UNIX Specifications v4"
golang	I:20	11	Go programming language compiler
rustc	V:3, I:14	8860	Rust systems programming language
haskell-platform	I:1	12	Standard Haskell libraries and tools
gfortran	V:6, I:62	15	GNU Fortran 95 compiler
fpc	I:2	103	Free Pascal

Here, Section 12.3.3, "Flex - a better Lex" and Section 12.3.4, "Bison - a better Yacc" are included to indicate how compiler-like program can be written in C language by compiling higher level description into C language.

12.3.1. C

You can set up proper environment to compile programs written in the C programming language by the following.

```
$> apt-get install glibc-doc manpages-dev libc6-dev gcc build-essential
```

The libc6-dev package, i.e., GNU C Library, provides C standard library which is collection of header files and library routines used by the C programming language.

See references for C as the following.
  * "info libc" (C library function reference)
  * gcc(1) and "info gcc"
  * each_C_library_function_name(3)
  * Kernighan & Ritchie, "The C Programming Language", 2nd edition (Prentice Hall)

12.3.2. Simple C program (gcc)

A simple example "example.c" can compiled with a library "libm" into an executable "run_example" by the following.

```
$> cat > example.c << EOF
#include <stdio.h>
#include <math.h>
#include <string.h>

int main(int argc, char **argv, char **envp) {
        double x;
        char y[11];
        x=sqrt(argc+7.5);
```

```
        strncpy(y, argv[0], 10); /* prevent buffer overflow */
        y[10] = '\0';            /* fill to make sure string ends with '\0' */
        printf("%5i, %5.3f, %10s, %10s\n", argc, x, y, argv[1]);
        return 0;
}
EOF

$> gcc -Wall -g -o run_example example.c -lm
$> ./run_example
        1, 2.915, ./run_exam,     (null)
$> ./run_example 1234567890qwerty
        2, 3.082, ./run_exam, 1234567890qwerty
```

Here, "-lm" is needed to link library "/usr/lib/libm.so" from the libc6 package for sqrt(3). The actual library is in "/lib/" with filename "libm.so.6", which is a symlink to "libm-2.7.so".

Look at the last parameter in the output text. There are more than 10 characters even though "%10s" is specified.

The use of pointer memory operation functions without boundary checks, such as sprintf(3) and strcpy(3), is deprecated to prevent buffer overflow exploits that leverage the above overrun effects. Instead, use snprintf(3) and strncpy(3).

12.3.3. Flex - a better Lex

Flex is a Lex-compatible fast lexical analyzer generator.
Tutorial for flex(1) can be found in "info flex".
Many simple examples can be found under "/usr/share/doc/flex/examples/".

12.3.4. Bison - a better Yacc

Several packages provide a Yacc-compatible lookahead LR parser or LALR parser generator in Debian.

Table 12.11. List of Yacc-compatible LALR parser generators

package	popcon	size	description
bison	V:7, I:80	3116	GNU LALR parser generator
byacc	V:0, I:4	258	Berkeley LALR parser generator
btyacc	V:0, I:0	243	backtracking parser generator based on byacc

Tutorial for bison(1) can be found in "info bison".

You need to provide your own "main()" and "yyerror()". "main()" calls "yyparse()" which calls "yylex()", usually created with Flex.

Here is an example to create a simple terminal calculator program.

Let's create example.y:

```
/* calculator source for bison */
%{
#include <stdio.h>
extern int yylex(void);
extern int yyerror(char *);
%}

/* declare tokens */
%token NUMBER
%token OP_ADD OP_SUB OP_MUL OP_RGT OP_LFT OP_EQU

%%
calc:
 | calc exp OP_EQU    { printf("Y: RESULT = %d\n", $2); }
 ;

exp: factor
 | exp OP_ADD factor  { $$ = $1 + $3; }
 | exp OP_SUB factor  { $$ = $1 - $3; }
 ;

factor: term
 | factor OP_MUL term { $$ = $1 * $3; }
 ;

term: NUMBER
 | OP_LFT exp OP_RGT  { $$ = $2; }
  ;
%%

int main(int argc, char **argv)
{
  yyparse();
}

int yyerror(char *s)
{
```

```
  fprintf(stderr, "error: '%s'\n", s);
}
```

Let's create, example.l:

```
/* calculator source for flex */
%{
#include "example.tab.h"
%}

%%
[0-9]+ { printf("L: NUMBER = %s\n", yytext); yylval = atoi(yytext); return NUMBER; }
"+"    { printf("L: OP_ADD\n"); return OP_ADD; }
"-"    { printf("L: OP_SUB\n"); return OP_SUB; }
"*"    { printf("L: OP_MUL\n"); return OP_MUL; }
"("    { printf("L: OP_LFT\n"); return OP_LFT; }
")"    { printf("L: OP_RGT\n"); return OP_RGT; }
"="    { printf("L: OP_EQU\n"); return OP_EQU; }
"exit" { printf("L: exit\n");   return YYEOF; } /* YYEOF = 0 */
.      { /* ignore all other */ }
%%
```

Then execute as follows from the shell prompt to try this:

```
$ bison -d example.y
$ flex example.l
$ gcc -lfl example.tab.c lex.yy.c -o example
$ ./example
1 + 2 * ( 3 + 1 ) =
L: NUMBER = 1
L: OP_ADD
L: NUMBER = 2
L: OP_MUL
L: OP_LFT
L: NUMBER = 3
L: OP_ADD
L: NUMBER = 1
L: OP_RGT
L: OP_EQU
Y: RESULT = 9

exit
L: exit
```

12.4. Static code analysis tools

Lint like tools can help automatic static code analysis.

Indent like tools can help human code reviews by reformatting source codes consistently.

Ctags like tools can help human code reviews by generating an index (or tag) file of names found in source codes.

Configuring your favorite editor (emacs or vim) to use asynchronous lint engine plugins helps your code writing. These plugins are getting very powerful by taking advantage of Language Server Protocol. Since they are moving fast, using their upstream code instead of Debian package may be a good option.

Table 12.12. List of tools for static code analysis

package	popcon	size	description
vim-ale	I:0	2591	Asynchronous Lint Engine for Vim 8 and NeoVim
vim-syntastic	I:3	1379	Syntax checking hacks for vim
elpa-flycheck	V:0, I:1	808	modern on-the-fly syntax checking for Emacs
elpa-relint	V:0, I:0	147	Emacs Lisp regexp mistake finder
cppcheck-gui	V:0, I:1	7224	tool for static C/C++ code analysis (GUI)
shellcheck	V:2, I:13	18987	lint tool for shell scripts
pyflakes3	V:2, I:15	20	passive checker of Python 3 programs
pylint	V:4, I:20	2018	Python code static checker
perl	V:707, I:989	673	interpreter with internal static code checker: B::Lint(3perl)
rubocop	V:0, I:0	3247	Ruby static code analyzer
clang-tidy	V:2, I:11	21	clang-based C++ linter tool
splint	V:0, I:2	2320	tool for statically checking C programs for bugs
flawfinder	V:0, I:0	205	tool to examine C/C++ source code and looks for security weaknesses
black	V:3, I:13	660	uncompromising Python code formatter
perltidy	V:0, I:4	2493	Perl script indenter and reformatter
indent	V:0, I:7	431	C language source code formatting program
astyle	V:0, I:2	785	Source code indenter for C, C++, Objective-C, C#, and Java
bcpp	V:0, I:0	111	C(++) beautifier
xmlindent	V:0, I:1	53	XML stream reformatter
global	V:0, I:2	1908	Source code search and browse tools
exuberant-ctags	V:2, I:20	341	build tag file indexes of source code definitions
universal-ctags	V:1, I:11	3386	build tag file indexes of source code definitions

12.5. Debug

Debug is important part of programming activities. Knowing how to debug programs makes you a good Debian user who can produce meaningful bug reports.

Table 12.13. List of debug packages

package	popcon	size	documentation
gdb	V:14, I:96	11637	"info gdb" provided by gdb-doc
ddd	V:0, I:7	4105	"info ddd" provided by ddd-doc

12.5.1. Basic gdb execution

Primary debugger on Debian is gdb(1) which enables you to inspect a program while it executes.

Let's install gdb and related programs by the following.

```
$> apt-get install gdb gdb-doc build-essential devscripts
```

Good tutorial of gdb can be found:

- "info gdb"
- "Debugging with GDB" in /usr/share/doc/gdb-doc/html/gdb/index.html
- "tutorial on the web"

Here is a simple example of using gdb(1) on a "program" compiled with the "-g" option to produce debugging information.

```
$> gdb program
(gdb) b 1               # set break point at line 1
(gdb) run args          # run program with args
(gdb) next              # next line
...
( gdb) step             # step forward
...
( gdb) p parm           # print parm
...
( gdb) p parm=12        # set value to 12
...
( gdb) quit
```

Many gdb(1) commands can be abbreviated. Tab expansion works as in the shell.

12.5.2. Debugging the Debian package

Since all installed binaries should be stripped on the Debian system by default, most debugging symbols are removed in the normal package. In order to debug Debian packages with gdb(1), *-dbgsym packages need to be installed (e.g. coreutils-dbgsym in the case of coreutils). The source packages generate *-dbgsym packages automatically along with normal binary packages and those debug packages are placed separately in debian-debug archive. Please refer to articles on Debian Wiki for more information.

If a package to be debugged does not provide its *-dbgsym package, you need to install it after rebuilding it by the following.

```
$> mkdir /path/new ; cd /path/new
$> sudo apt-get update
$> sudo apt-get dist-upgrade
$> sudo apt-get install fakeroot devscripts build-essential
$> apt-get source package_name
$> cd package_name*
$> sudo apt-get build-dep ./
```

Fix bugs if needed.

Bump package version to one which does not collide with official Debian versions, e.g. one appended with "+debug1" when recompiling existing package version, or one appended with "~pre1" when compiling unreleased package version by the following.

```
$> dch -i
```

Compile and install packages with debug symbols by the following.

```
$> export DEB_BUILD_OPTIONS="nostrip noopt"
$> debuild
$> cd ..
$> sudo debi package_name*.changes
```

You need to check build scripts of the package and ensure to use "CFLAGS=-g -Wall" for compiling binaries.

12.5.3. Obtaining backtrace

When you encounter program crash, reporting bug report with cut-and-pasted backtrace information is a good idea.

The backtrace can be obtained by gdb(1) using one of the following approaches:

- Crash-in-GDB approach:
  - Run the program from GDB.
  - Crash the program.
  - Type "bt" at the GDB prompt.
- Crash-first approach:
  - Update the "/etc/security/limits.conf" file to include the following:

```
* soft  core unlimited
```

  - Type "ulimit -c unlimited" to the shell prompt.
  - Run the program from this shell prompt.
  - Crash the program to produce a core dump file.
  - Load the core dump file to GDB as "gdb gdb ./program_binary core" .

+ Type "bt" at the GDB prompt.

For infinite loop or frozen keyboard situation, you can force to crash the program by pressing Ctrl-\ or Ctrl-C or executing "kill -ABRT PID". (See Section 9.4.12, "Killing a process")

Often, you see a backtrace where one or more of the top lines are in "malloc()" or "g_malloc()". When this happens, chances are your backtrace isn't very useful. The easiest way to find some useful information is to set the environment variable "$MALLOC_CHECK_" to a value of 2 (malloc(3)). You can do this while running gdb by doing the following.

```
$> MALLOC_CHECK_=2 gdb hello
```

12.5.4. Advanced gdb commands

Table 12.14. List of advanced gdb commands

command	description for command objectives
(gdb) thread apply all bt	get a backtrace for all threads for multi-threaded program
(gdb) bt full	get parameters came on the stack of function calls
(gdb) thread apply all bt full	get a backtrace and parameters as the combination of the preceding options
(gdb) thread apply all bt full 10	get a backtrace and parameters for top 10 calls to cut off irrelevant outp ut
(gdb) set logging on	write log of gdb output to a file (the default is "gdb.txt")

12.5.5. Check dependency on libraries

Use ldd(1) to find out a program's dependency on libraries by the followings.

```
$> ldd /usr/bin/ls
        librt.so.1 => /lib/librt.so.1 (0x4001e000)
        libc.so.6 => /lib/libc.so.6 (0x40030000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x40153000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

For ls(1) to work in a `chroot`ed environment, the above libraries must be available in your `chroot`ed environment.

See Section 9.4.6, "Tracing program activities".

12.5.6. Dynamic call tracing tools

There are several dynamic call tracing tools available in Debian. See Section 9.4, "Monitoring, controlling, and starting program activities".

12.5.7. Debugging X Errors

If a GNOME program preview1 has received an X error, you should see a message as follows.

```
The program 'preview1' received an X Window System error.
```

If this is the case, you can try running the program with "--sync", and break on the "gdk_x_error" function in order to obtain a backtrace.

12.5.8. Memory leak detection tools

There are several memory leak detection tools available in Debian.

Table 12.15. List of memory leak detection tools

package	popcon	size	description
libc6-dev	V:248, I:567	12053	mtrace(1): malloc debugging functionality in glibc
valgrind	V:6, I:37	78191	memory debugger and profiler
electric-fence	V:0, I:3	73	malloc(3) debugger
libdmalloc5	V:0, I:2	390	debug memory allocation library
duma	V:0, I:0	296	library to detect buffer overruns and under-runs in C and C++ programs
leaktracer	V:0, I:1	56	memory-leak tracer for C++ programs

12.5.9. Disassemble binary

You can disassemble binary code with objdump(1) by the following.

```
$> objdump -m i386 -b binary -D /usr/lib/grub/x86_64-pc/stage1
```

gdb(1) may be used to disassemble code interactively.

12.6. Build tools

Table 12.16. List of build tool packages

package	popcon	size	documentation
make	V:151, I:555	1592	"info make" provided by make-doc
autoconf	V:31, I:230	2025	"info autoconf" provided by autoconf-doc
automake	V:30, I:228	1837	"info automake" provided by automake1.10-doc
libtool	V:25, I:212	1213	"info libtool" provided by libtool-doc
cmake	V:17, I:115	36607	cmake(1) cross-platform, open-source make system
ninja-build	V:6, I:41	428	ninja(1) small build system closest in spirit to Make
meson	V:3, I:22	3759	meson(1) high productivity build system on top of ninja
xutils-dev	V:0, I:9	1484	imake(1), xmkmf(1), etc.

12.6.1. Make

Make is a utility to maintain groups of programs. Upon execution of make(1), make read the rule

file, "Makefile", and updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist. The execution of these updates may occur concurrently.

The rule file syntax is the following.

```
target: [ prerequisites ... ]
 [TAB]  command1
 [TAB]  -command2 # ignore errors
 [TAB]  @command3 # suppress echoing
```

Here "[TAB]" is a TAB code. Each line is interpreted by the shell after make variable substitution. Use "\" at the end of a line to continue the script. Use "$$" to enter "$" for environment values for a shell script.

Implicit rules for the target and prerequisites can be written, for example, by the following.

```
%.o: %.c header.h
```

Here, the target contains the character "%" (exactly one of them). The "%" can match any nonempty substring in the actual target filenames. The prerequisites likewise use "%" to show how their names relate to the actual target name.

Table 12.17. List of make automatic variables

automatic variable	value
$@	target
$<	first prerequisite
$?	all newer prerequisites
$^	all prerequisites
$*	"%" matched stem in the target pattern

Table 12.18. List of make variable expansions

variable expansion	description
foo1 := bar	one-time expansion
foo2 = bar	recursive expansion
foo3 += bar	append

Run "make -p -f/dev/null" to see automatic internal rules.

12.6.2. Autotools

Autotools is a suite of programming tools designed to assist in making source code packages portable to many Unix-like systems.

- Autoconf is a tool to produce a shell script "configure" from "configure.ac".
  - "configure" is used later to produce "Makefile" from "Makefile.in" template.
- Automake is a tool to produce "Makefile.in" from "Makefile.am".
- Libtool is a shell script to address the software portability problem when compiling shared libraries from source code.

12.6.2.1. Compile and install a program

Do not overwrite system files with your compiled programs when installing them.

Debian does not touch files in "/usr/local/" or "/opt". So if you compile a program from source, install it into "/usr/local/" so it does not interfere with Debian.

```
$> cd src
$> ./configure --prefix=/usr/local
$> make # this compiles program
$> sudo make install # this installs the files in the system
```

12.6.2.2. Uninstall program

If you have the original source and if it uses autoconf(1)/automake(1) and if you can remember how you configured it, execute as follows to uninstall the program.

```
$> ./configure all-of-the-options-you-gave-it
$> sudo make uninstall
```

Alternatively, if you are absolutely sure that the install process puts files only under "/usr/local/" and there is nothing important there, you can erase all its contents by the following.

```
$> find /usr/local -type f -print0 | xargs -0 rm -f
```

If you are not sure where files are installed, you should consider using checkinstall(8) from the checkinstall package, which provides a clean path for the uninstall. It now supports to create a Debian package with "-D" option.

12.6.3. Meson

The software build system has been evolving:

- Autotools on the top of Make has been the de facto standard for the portable build infrastructure since 1990s. This is extremely slow.
- CMake initially released in 2000 improved speed significantly but was originally built on the top of inherently slow Make. (Now Ninja can be its backend.)
- Ninja initially released in 2012 is meant to replace Make for the further improved build speed and is designed to have its input files generated by a higher-level build system.
- Meson initially released in 2013 is the new popular and fast higher-level build system which

uses Ninja as its backend.

See documents found at "The Meson Build system" and "The Ninja build system".

12.7. Web

Basic interactive dynamic web pages can be made as follows.

* Queries are presented to the browser user using HTML forms.
* Filling and clicking on the form entries sends one of the following URL string with encoded parameters from the browser to the web server.
  + "https://www.foo.dom/cgi-bin/program.pl?VAR1=VAL1&VAR2=VAL2&VAR3=VAL3"
  + "https://www.foo.dom/cgi-bin/program.py?VAR1=VAL1&VAR2=VAL2&VAR3=VAL3"
  + "https://www.foo.dom/program.php?VAR1=VAL1&VAR2=VAL2&VAR3=VAL3"
* "%nn" in URL is replaced with a character with hexadecimal nn value.
* The environment variable is set as: "QUERY_STRING="VAR1=VAL1 VAR2=VAL2 VAR3=VAL3"".
* CGI program (any one of "program.*") on the web server executes itself with the environment variable "$QUERY_STRING".
* stdout of CGI program is sent to the web browser and is presented as an interactive dynamic web page.

For security reasons it is better not to hand craft new hacks for parsing CGI parameters. There are established modules for them in Perl and Python. PHP comes with these functionalities. When client data storage is needed, HTTP cookies are used. When client side data processing is needed, Javascript is frequently used.

For more, see the Common Gateway Interface, The Apache Software Foundation, and JavaScript.

Searching "CGI tutorial" on Google by typing encoded URL https://www.google.com/search?hl=en&ie=UTF-8&q=CGI+tutorial directly to the browser address is a good way to see the CGI script in action on the Google server.

12.8. The source code translation

There are programs to convert source codes.

Table 12.19. List of source code translation tools

package	popcon	size	keyword description
perl	V:707, I:989	673	AWK-->PERL convert source codes from AWK to PERL: a2p(1)
f2c	V:0, I:3	442	FORTRAN-->C convert source codes from FORTRAN 77 to C/C++: f2c(1)
intel2gas	V:0, I:0	178	intel-->gas converter from NASM (Intel format) to the GNU Assembler (GAS)

12.9. Making Debian package

If you want to make a Debian package, read followings.

* Chapter 2, Debian package management to understand the basic package system
* Section 2.7.13, "Porting a package to the stable system" to understand basic porting process
* Section 9.11.4, "Chroot system" to understand basic chroot techniques
* debuild(1), and sbuild(1)
* Section 12.5.2, "Debugging the Debian package" for recompiling for debugging
* Guide for Debian Maintainers (the debmake-doc package)
* Debian Developer's Reference (the developers-reference package)
* Debian Policy Manual (the debian-policy package)

There are packages such as debmake, dh-make, dh-make-perl, etc., which help packaging.

tweaks may be required to get them work under the current system.
