;login:
THE MAGAZINE OF USENIX & SAGE
June 2003 • volume 28 • number 3

inside:

BEST PRACTICES
.profile
by Travis Howard

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

.profile

by Travis Howard

Travis Howard is the founder of Asymptotic Systems, a security consulting organization. He has 17 years of experience in computer security and specializes in applying research advances to harden systems.

auto92089@hushmail.com

Of all the dot files, *.profile* presents the greatest opportunity for customization. This file is read by *sh* derivatives as part of the login process, and usually sets environment variables that influence the behavior of many programs invoked as part of that login session. This article describes how to customize *.profile*, but first we should examine how *.profile* is used and what is appropriate to place in it.

Specifically, *sh* derivatives consider a shell a login shell when argv[0] begins with a dash ("-"; ASCII value 45). Note that this is a violation of the convention that the first element of argv[] contain the last component of the executed program's path (for more information see *execve(2)*). This is the only way to flag *sh* as a login shell. However, *ksh* accepts -l and *bash* accepts -login as alternate ways of flagging a shell as a login shell.

All *sh* derivative login shells first process the systemwide /etc/profile if it exists. The next file processed depends on the shell; *sh* and *ksh* process $HOME/.profile, while *bash* processes the first it finds of $HOME/.bash_profile, $HOME/.bash_login, or $HOME/.profile. Note that *bash* has a flag -noprofile, which inhibits processing any of these files.

You may wonder where *sh* derivative login shells get their notion of $HOME. The HOME environment variable is set by *login(1)*, as are SHELL, PATH, TERM, LOGNAME, USER (if BSD), MAIL (if not BSD), and TZ (if Solaris). For portability's sake, we should rely only on the common subset of environment variables set by *login(1)* (if we rely on any of them at all).

Note that *login(1)* will print a variety of messages, unless $HOME/.nologin exists. By the same rationale, we should feel free to output information from *.profile* under the same conditions. The nologin support is primarily for UUCP and other automated logins, so we need not heed this rule too carefully, unless you intend to use your *.profile* for your automated users as well.

Since we have the full power of the shell at our disposal when processing *.profile*, we may easily use its branching constructs to process different parts of this file under different conditions. This allows us to potentially use the same *.profile* in many locations – unlike some other dot-files – simplifying the customization of multiple environments into a single file distribution problem.

Furthermore, we will wish to do some similar and repetitive tasks in our *.profile*, so it will be advisable to use its native function definitions as a macro facility. If we were generating different *.profile* files using some kind of macro language such as m4 or cpp, we could avoid depending on the availability of functions in our shell. However, the environments I am interested in all support functions within /bin/sh, so that is an implicit assumption in my *.profile*. This also allows runtime recursion, which preprocessors could not provide (not that we will need it).

It's extremely difficult to come up with guiding principles and structure for a *.profile*. For one thing, several dependencies must be taken into account when creating a linear order for your statements. There are site-specific, OS-specific, release-specific, and architecture-specific components, and components that are specific to two or more of these categories.

Often it is not clear which category certain statements fall into. For example, BSD UNIX uses BLOCKSIZE to affect the reporting units in df, du, and some other programs. It is not clear whether this is a variable which should be set globally, or perhaps in a BSD-specific portion of the *.profile*.

There are design tradeoffs, such as the desire for an uncluttered environment and the desire to keep the *.profile* simple. Another tradeoff occurs when two OSes have similarities. For example, SunOS and BSD both have /sbin directories for the superuser. Do you add /sbin to the path in both cases, causing code duplication, or do you make a case statement that puts them together, and add /sbin to the path there? Thus, this *.profile* is most definitely a compromise among several competing desires.

Since *.profile* is only processed once per login session, it's tempting to do a little more work in order to make the *.profile* simpler. For example, you might want to invoke Perl and have it tell you where its man pages are located, rather than guessing and testing several possibilities. Similarly, rather than putting /sbin handling into all BSD-like OS-specific sections, it might be cleaner to test for the presence of /sbin and handle, adding it to the PATH no matter which OS you are running. Also note that I usually don't bother to preserve environment variables that are already set via */etc/profile*, although you may wish to do so.

```
# $Id: .profile,v 1.108 2002/11/12 08:12:47 username Exp $
# Hey Emacs this is -*- sh -*-

# This is sourced at login-time by sh, ash, ksh, and bash.
```

```
# Assumptions:   login(1) sets HOME SHELL PATH TERM LOGNAME
#                XDM(1) sets DISPLAY PATH SHELL XAUTHORITY

# Login script order:
# sh, ash, ksh     /etc/profile ~/.profile
# bash             /etc/profile (~/.bash_profile ~/.bash_login ~/.profile)
```

I keep my *.profile* under CVS, so I have an $Id:$ keyword in the first line. I also tell Emacs that this is a shell script in the following line (I believe that more recent versions of Emacs allow mode:sh). This is followed by documentation of assumptions and a little reminder of when this file was evaluated.

Next I give the user some indication that the *.profile* is being run (and send it to the console, not to an output file):

```
## Show login stuff:
# Echo to fd 2 (stderr).
e2 () { echo "$@" >&2; }
e2 'Running .profile'
```

The first thing the *.profile* should do is give an indication that it has been invoked. This will definitely help debug login problems as well as give a visual indicator of how heavy the system load is (if it takes a long time to show this, the load is extremely heavy). This code snippet defines a function (e2), which echoes all of its arguments to the standard error file descriptor (read that operator as "standard output gets tied to fd 2"). There are occasions where standard output is buffered, but standard error is usually line-buffered at most, so that is why I use it. Note the use of double-quoted $@; this incantation preserves whitespace and argument boundaries, even if the arguments include whitespace. It is a good idea to get into the habit of using this instead of the boundary-destroying $*. The code snippet then invokes e2 to display a simple message. Should you wish, you could easily test for the presence of a ~/.nologin file to suppress printing any output.

Next I inform the OS that I wish to make all my file ownerships private:

```
## Set a paranoid umask.
umask 077
```

I want to make sure that this *.profile* can find the programs I want to invoke, so I then must attend to setting the PATH environment variable, which controls the search path for said programs. Clearly, I will want some functions to help me manipulate the colon-separated list of directories:

```
## Set colon-separated search path elements:

# Test a directory (sanity check).
# Returns true (0) only if it is a directory and searchable.
test_directory () {
    test "$#" -eq 0 && e2 "Usage: test_directory dirname" && return 2
    test -d "$1" && test -x "$1"
}

# Check to see if a directory is already in a search path.
in_search_path () {
    test "$#" -lt 2 && e2 "Usage: in_search_path path dirname" && return 2
    local n="$1"
    local d="$2"
    # Save input field separators.
    local OLDIFS="$IFS"
    # Break up on colon boundaries
    IFS=":$IFS"
    # Now set the positional args to elements of the named variable.
    eval set -- $n
    # Restore input field separator.
    IFS="$OLDIFS"
```

```
    # Loop through each colon-separated element.
    while test "$#" -gt 0; do
        # Compare to dirname.
        # TODO: how portable is relying on -ef?
        test "$1" -ef "$d" && return 0
        shift
    done
    return 1
}

# Sanity-check then append a directory to a search path.
dirapp () {
    test "$#" -lt 2 && e2 "Usage: dirapp varname dirname" && return 2
    local n="$1"
    local d="$2"
    test_directory "$d" || return 1
    eval in_search_path \"\$$n\" $d && return 1
    if eval test -n \"\$$n\"; then
        eval $n=\"\$$n:$d\"
    else
        eval $n=\"$d\"
    fi
}

# Sanity-check then prepend a directory to a search path.
# TODO: Allow caller to "move" directory to front with this funcall.
dirpre () {
    test "$#" -lt 2 && e2 "Usage: dirpre varname dirname" && return 2
    local n="$1"
    local d="$2"
    test_directory "$d" || return 2
    # eval in_search_path \"\$$n\" $d && return 1
    if eval test -n \"\$$n\"; then
        eval $n=\"$d:\$$n\"
    else
        eval $n=\"$d\"
    fi
}

# Call dirapp for a list of directories.
dirapplist () {
    test "$#" -lt 2 && e2 "Usage: dirapplist varname d1 d2 ..." && return 2
    local n="$1"
    shift
    while test "$#" -gt 0; do
        dirapp "$n" "$1"
        shift
    done
}

# Call dirpre for a list of directories.
# NOTE: Directories will appear in reverse order in varname.
dirprelist () {
```

```
    test "$#" -lt 2 && e2 "Usage: dirapplist varname d1 d2 ..." && return 2
    local n="$1"
    shift
    while test "$#" -gt 0; do
        dirpre "$n" "$1"
        shift
    done
}
```

Note that I will try to use the term *path* to refer to a list of directories, starting with the root and ending with a file or directory, whereas I use the term *search path* to refer to a colon-separated list of paths.

I return 2 to distinguish from the exit code of the last command, which could be 0 or 1.

Next I set up PATH elements which should always be present. I think all reasonable operating systems start PATH with these elements, but it doesn't hurt to be sure:

```
# These should be present on any target system.
# In fact, they should already be in the search path.
dirapplist PATH /bin /usr/bin
# I like to be able to run e.g., ifconfig, sendmail.
dirapplist PATH /sbin /usr/sbin /usr/games /usr/libexec /usr/ccs/bin
dirpre PATH /usr/ucb
export PATH

# Set the search path for manual pages.
dirapplist MANPATH /usr/share/man /usr/share/man/old /usr/contrib/man
export MANPATH
```

Note that I mark these variables as exportable. In general, my policy is to do so the first time I manipulate the variable.

```
## Do shell-specific handling:

# This code distinguishes between various shell versions.
if test "$(echo ~)" != "$HOME"
then
    # This is the standard Bourne shell.
    :
else
    # This is not the Bourne shell, so it supports aliases.
    test -r "$HOME/.kshrc" && test -z "$ENV" && export ENV="$HOME/.kshrc"
    # TODO: Figure out a deterministic way to distinguish shells.
    # Should I just check $SHELL instead?
    if test "${RANDOM:-0}" -eq "${RANDOM:-0}"
    then
        # This is ash, which does not have a type built-in.
        # TODO: Recent versions of ash do indeed have type, so I should test for its presence somehow.
        test -r "$HOME/.ashtype" && . "$HOME/.ashtype"
        # Tell ash where to find our shell functions.
        test_directory "$HOME/functions" && dirapp PATH "$HOME/functions%func"
    fi
fi
```

This code tries to figure out which *sh* variant we are using. It then takes care of the shell-dependent initialization commands. Note that since it uses dirapp, it has to occur after dirapp has been defined, else I would have placed it earlier in the file.

```
## Run this stuff on logout.
if test -r "$HOME/.shlogout"
```

```
then
    trap '. $HOME/.shlogout' 0
else
    # Make a reasonable attempt to clear the screen.
    trap 'clear' 0
fi
```

This code takes care of cleaning up when we log out. If $HOME/.shlogout exists and is readable, we evaluate that and exit with the last command's exit code; otherwise we just clear the screen. That is, this code runs when a login shell exits. If this varied from site to site, it might be more appropriate to put it in the site-specific section.

Clearly, the universal *.profile* will need facilities for examining its environment to determine which parts should be processed in a particular situation. Now that we've set a reasonable PATH (enough to reach the system binaries, anyway), we can try executing some commands to ascertain facts about the platform:

```
## Set the environment variables:

# Try to set the envar called by name in arg1 to the output of the commands that follow, one argument per command.
setvarcmd () {
    test "$#" -lt 2 \
        && e2 "Usage: setvarcmd varname \"cmd1 args\" \"cmd2 args\" ..." \
        && return 2
    local n="$1"
    shift
    while test "$#" -ge 1 && eval test -z \"\$$n\"; do
        eval "$n=\"$($1 2>/dev/null)\""
        shift
    done
    # TODO: what if no commands generate output?
    export $n
}
```

We realize that we will be calling a lot of commands in sequence until we find one that gives us the information we need, which we will then export. Therefore, we enshrine this process in a shell function, setvarcmd. Note that I use the newer $() syntax rather than ''; this allows levels of command expansion to be nested (although we do not do that here). Errors, such as invalid options or nonexistent commands, are directed to /dev/null when executing the commands, because we will be trying several commands in order and don't care if some of them fail.

Next we use setvarcmd to try yo set four critical variables:

```
setvarcmd OS_NAME "uname -s" "uname"
e2 "Operating system: $OS_NAME"

setvarcmd OS_RELEASE "uname -r"
e2 "Release: $OS_RELEASE"

setvarcmd HW_NAME "arch" "uname -m"
e2 "Hardware/Architecture name: $HW_NAME"

# TODO: This frequently does not include the domain name.
setvarcmd HOST_NAME "hostname -f" "uname -n" "hostname"
e2 "Host Name: $HOST_NAME"
```

These variables will be used to decide which portions of the *.profile* to evaluate. There is no error checking here, because if these commands fail to ascertain the desired information, it is not clear what else we could do. However, the user logging in would see that some of the variables were not set, so we can hand-wave here and say the user should investigate and remedy. It may well be that a system which does not readily yield its details may be an unsuitable platform for a universal *.profile*.

Now that OS_NAME is set, I can run some OS-dependent commands to manipulate the environment:

```
# Find some other binary directories, but only for the right architecture.
# Set MAIL to point to mailbox so shell can tell us when we have mail.
# TODO: fix for mailbox in $HOME/mbox.
case "$OS_NAME" in
    AIX)
        dirapp PATH /public/ibm/bin
        ;;
    SunOS*)
        dirapp PATH /public/sun4/bin
        # Find my mailbox and have this shell check it periodically.
        # NOTE: Do not export or subshells will check mail.
        test_directory /usr/spool/mail && MAIL=/usr/spool/mail/$LOGNAME
        ;;
    *BSD)
        test_directory /var/mail && MAIL=/var/mail/$LOGNAME
        ;;
esac
```

The additions to PATH are from one of the sites that I used to work at and are nonstandard. Therefore, should they exist, it's fairly certain it's because this *.profile* is being invoked at that site. Should these directories exist at some other site, just by chance, then maybe I'll want them in my path there, too. If not, I'll have to change my *.profile* accordingly.

The MAIL environment variable tells the shell to alert me between commands if new mail has arrived. If I exported it, subshells, such as the ones created by system(3), or those created by xterms under X, would also alert me, and I don't want to be told more than once.

Next I want to try to set an environment variable to point to the first directory in a list of potential directories:

```
# Set a specified variable to equal the first valid directory in a list.
# TODO: Should I check to see if it is set already?
setvardir() {
    test "$#" -lt 2 && e2 "Usage: setvardir varname dir1 dir2 ..." && return 2
    local n="$1"
    shift
    while test "$#" -gt 0; do
        test_directory "$1" && eval "$n=\"$1\"" && export $n && return 0
        shift
    done
    return 1
}
```

First I use setvardir to find Openwin:

```
# I had to use Openwin on some SunOS machines.
if setvardir OPENWINHOME /usr/openwin; then
    dirapp PATH "$OPENWINHOME/bin"
    dirapp MANPATH "$OPENWINHOME/share/man"
    dirapp LD_LIBRARY_PATH "$OPENWINHOME/lib"
fi
```

Next I try to locate the X Windowing System binaries:

```
# XWINHOME is used by some startx(1), XF86Setup(1), and apparently xman(1), XF86_S3(1), etc.
# Technically, I should only accept /usr/X386 if we are on an x86,
# but what would it be doing there on another architecture anyway?
if setvardir XWINHOME /usr/X11R6 /usr/X386; then
```

```
    # put X executables in search path
    dirapp PATH "$XWINHOME/bin"
    # put X man pages in search path
    dirapp MANPATH "$XWINHOME/man"
fi
```

I then define a short function that I use to find the Perl man pages, given a "root" like /usr/local:

```
# TODO: There has got to be a good way to find the Perl manual pages by querying Perl.
findperlmanpages() {
    local r="$1"
    dirapplist MANPATH   "$r/lib/perl5/man" \
                         "$r/lib/perl/man" \
                         "$r/share/perl5/man" \
                         "$r/share/perl/man"
}
```

I then want to set a variable LOCALIZED to point to where the locally installed programs reside:

```
# Find locally installed programs.
if setvardir LOCALIZED /usr/local /local /lusr /opt; then
    # Prepend locally installed program dir so it can override system binaries.
    dirpre PATH "$LOCALIZED/bin"
    # Search here for manual pages.
    dirpre MANPATH "$LOCALIZED/share/man"
    # BSD systems might have this, others probably will not.
    dirapp PATH "$LOCALIZED/sbin"
    # Some sites insist on per-package bin directories, sigh.
    # NOTE: This could go later in this file, as a site-dependent section,
    # but these directories probably will not exist on most systems.
    for i in tex gnu tk tcl elm expect ghostscript lotus netmake newsprint tk mh; do
        dirapp PATH "$LOCALIZED/$i/bin"
        dirapp MANPATH "$LOCALIZED/$i/man"
    done
    findperlmanpages $LOCALIZED
    dirapp MANPATH "$LOCALIZED/teTeX/man"
    # This is the info path for GNU info hypertext command, and Emacs
    dirapplist INFOPATH  "$LOCALIZED/info" \
                         "$LOCALIZED/share/info" \
                         "$LOCALIZED/teTeX/info"
    # This was required to run Lotus Notes at one site.
    dirapp LD_LIBRARY_PATH "$LOCALIZED/lotus/common/lel/r100/sunspa53"
    # Set the cool Concurrent Version System repository directory.
    setvardir CVSROOT "$LOCALIZED/share/cvsroot"
fi
```

Next I want to set up any user-specific binaries so that I can compile my own copy of a program and have that program be called in place of the system-installed binary of the same name. In other words, this gives the user of this *.profile* the ability to mask system binaries with his/her own copies. We're implicitly creating search paths that start with the user's binaries, followed by the system's (local) binaries and, finally, the OS binaries. One disadvantage of this is that sometimes upgrading the OS will cause the OS binaries to be more up-to-date than the ones closer to the front of the search path. Note that this is done only if the HOME directory is not the root, since /bin is already in the search path. Note also that we allow a user to have binaries that are OS-specific or even release-specific. This is useful in a heterogeneous environment with shared user home directories.

```
# Find my installed programs.
# NOTE: If we boot up in single-user mode, home directory is root.
```

```
if test "$HOME" != "/"; then
    # I have even more control over these so they get prepended.
    dirprelist PATH   "$HOME/bin" "$HOME/bin/$OS_NAME" "$HOME/bin/$OS_NAME/$OS_RELEASE"
    dirpre LD_LIBRARY_PATH "$HOME/lib"
    dirapp MANPATH "$HOME/share/man"
    findperlmanpages $HOME
    dirapp INFOPATH "$HOME/share/info"
    # Set the Pretty Good Privacy filepath (where it finds its files).
    setvardir PGPPATH "$HOME/pgp"
    # Set the cool Concurrent Version System repository directory.
    setvardir CVSROOT "$HOME/share/cvsroot"
fi
```

Note that CVSROOT is being set here again and can thus override the earlier value based on the LOCALIZED dir.

Next I do some variable exporting which wouldn't fit in easily to the above clauses:

```
# There is no convenient place to do this above so do it here.
export INFOPATH
export LD_LIBRARY_PATH
```

Now that we have a complete PATH (among other things), I would like to set certain environment variables that want complete path information. To this end, I create a shell function to find executables:

```
# Echo the full file name of the executable in the path to stdout.
# NOTE: All args are call-by-value.
findinpath () {
    test "$#" -lt 2 && e2 "Usage: findinpath exe_basename path" && return 2
    local f="$1"
    local IFS=":$IFS"
    set -- $2
    while test "$#" -gt 0
    do
        test -x "$1/$f" && echo "$1/$f" && return 0
        shift
    done
    return 1
}
```

This function searches a given path in much the same way that the shell and execlp(3) and execvp(3) do. It echoes the full pathname to standard out. Now that we've got that ability, we set some variables with the pathnames of desired binaries (and a group of standard variables, too).

```
# This is my preferred editor.
EDITOR=$(findinpath vi $PATH)
export EDITOR

# This is the visual, or full-screen editor of choice.
VISUAL="$EDITOR"
export VISUAL

# This is the editor for the fc built-in (for ksh).
FCEDIT="$EDITOR"
export FCEDIT

# EX init file or commands (used in vi(1))
EXINIT="set tabstop=4 showmode"
export EXINIT
```

```
# TODO: is this test sufficient and correct?
test "$OS_NAME" = "NetBSD" && EXINIT="$EXINIT verbose"

# This is my personal CVS working area.
setvardir CVSHOME "$HOME/dev/cvs"

# TEMP is a temporary directory for many programs: cc gcc mailq merge newaliases sendmail rcs (and friends)
# ghostscript i386-mach3-gcc perlbug perldoc
setvardir TEMP /var/tmp

# TMPDIR is a temporary directory for these programs:
# sort
# NOTE: gcc tries TMPDIR, then TMP, then TEMP
setvardir TMPDIR /tmp

# Use the large tmp dir for metamail.
setvardir METAMAIL_TMPDIR /var/tmp

# BLOCKSIZE is the size of the block units used by several commands: df, du, ls
# For more information see NetBSD environ(7).
BLOCKSIZE="1k"
export BLOCKSIZE

# CVS_RSH lets us use ssh instead of rsh for client/server
CVS_RSH=$(findinpath ssh $PATH)
export CVS_RSH
```

The next step is to set the environment variable PAGER to either less or, failing that, more:

```
# Set the pagination program for man, mailers, etc.
if PAGER=$(findinpath less $PATH); then
    # We found less, so set less options:
    #  -M = more verbose than "more"
    #  -f = force special files to be opened
    LESS="-Mf"
    export LESS

    # latin1 selects the ISO 8859/1 character set. latin-1 is the same as ASCII, except characters between 161
    #  and 255 are treated as normal characters.
    LESSCHARSET="latin1"
    export LESSCHARSET
else
    # Every system should have this.
    PAGER=$(findinpath more $PATH)
fi
export PAGER
```

Note that the conditional part of the if statement is true if and only if findinpath returns a true value.

Next comes the site-dependent clauses:

```
# This is the site-dependent section.
case "$HOST_NAME" in
    hostname*|*company.com)
        NNTPSERVER="news.company.com"
        http_proxy=http://proxy.company.com:8000/
        export http_proxy
        ;;
    otherhostname*|*othercompany.com)
        NNTPSERVER="nntp-server.othercompany.com"
```

```
        ;;
esac
export NNTPSERVER
```

I then print a fortune:

```
## Show fortune for fun:
type fortune > /dev/null 2>&1 && fortune -a
```

Then I do some last-minute fixes for certain OSes:

```
## OS-dependent fixes:

case "$OS_NAME" in
    NetBSD*)
        case "$OS_RELEASE" in
            0*|1.0*|1.1)
                # MANPATH does not work in early releases  of NetBSD
                unset MANPATH
                ;;
        esac
        ;;
esac
```

Next I have the terminal-handling routines:

```
## Set up terminal and start X if appropriate.

# Determine if we started under XDM.
if test -z "$DISPLAY"; then

    # If tset exists, use it to set up the terminal.
    if type tset > /dev/null 2>&1
    then
        # Set TERM and TERMCAP variables
        if test "$TERM" = "pcvt25h" && test "$OS_NAME" = "NetBSD" && test "$OS_RELEASE" = "1.1A"
        then
            e2 "Skipping tset due to $OS_NAME termcap buffer overflow bug"
        else
            eval $(tset -s -m 'network>9600:?xter' -m 'unknown:?vt100'  -m 'dialup:?vt100')
        fi
    fi

    export LINES
    export COLUMNS

    # This is OS-dependent terminal setup.
    case "$OS_NAME" in
    *BSD*)
        if ispcvt 2> /dev/null
        then
        # 28 lines on screen, HP function keys, 80 columns
        # NOTE: due to bug in scon, it only sets the row/col
        # of ttys if it is done in two commands like so:
        scon -s 28 && scon -H && scon -8
        LINES=25
        COLUMNS=80
        else
        # TODO: Set up wscons.
```

```
        :
        fi

            # Start X if we are on PCVT or WSCONS
            case $(tty) in
               /dev/ttyv*|/dev/ttyE*)
                  sx
                  ;;
            esac
            ;;
        *)
            # Be conservative about screen if not known.
            LINES=24
            COLUMNS=80
            ;;
    esac
fi
```

This rather large block has several functions. First, note that it is only processed if we are *not* starting under XDM (which would set DISPLAY to something). Next, it tests for the presence of tset using the type built-in. If tset exists, it first makes sure it is okay to use tset on this OS. Assuming it is, it tries to figure out what kind of terminal we're on. Note that tset -s actually produces shell commands that must be evaluated. For more information see the tset man page.

After we have set up the terminal, it marks LINES and COLUMNS as exported to child processes. Then we do some OS-specific checks to see what kind of terminal we're on, and if we are on the console, we start X using a shell function sx. Note that we put all the commands related to starting X in a shell function, so we can invoke it from the command line as easily as from *.profile*.

Finally, I end with a true value:

```
# Exit with true value for "make test".
:
```