Portability

The extent to which a program written on one platform can be run on a different platform.

Basic strategies for portability

Portability by **virtualization**: create a virtual platform that is the same, regardless of where you try to run the program.

Portability by **code branching**: create source code branches that deal with specifics of platforms.

Portability by **abstraction**: utilize libraries that abstract away architectural details, e.g., Windows versus linux.
Java and portability: virtualization
Every java program executes on the same Java virtual machine.
The JVM is re-implemented on every hardware platform.
Good news and bad news:
Good news: If the program only accesses the JVM, then portability is guaranteed.
Bad news: it is very difficult to limit dependencies to the JVM: the platform dependencies must be considered also.

Thus java is often described as "write once, debug everywhere".

Portability by virtualization is not enough.
Example: JVM and browsers

In a browser, the JVM runs in an environment constructed by the browser. That environment
  Differs ever so little between browsers. Is seen by the JVM differently in every browser.
Differences include:
  Exposure of devices.
  Exposure of HTML.
  Exposure of scripting.

Must write your JVM app for the lowest common denominator of all targeted browsers.
C and portability

The way that C is translated into machine code differs by architecture and choice of compiler.
Thus one must debug everywhere.

Some common C portability traps

Word length: IA32 versus IA64

Integers are defined in C as the most efficient word length.
So the length of an int changes with architecture!
In practice, never declare anything as an int! Use long and short instead.

Machine byte order:
Some machines are little-endian, some are big-endian,

Little-endian:
first byte in memory is most significant.
last byte in memory is least significant.

Big-endian:
first byte in memory is least significant.
last byte in memory is most significant.
i.e., is the sequence of bytes 0x0 0x0 0x0 0x1 interpreted as 1 (big-endian) or 2^25 (little-endian)
(This comes into play when one does one's own manipulation of integers via bitwise operations.)

Versions of libraries and their capabilities.
Version skew between dynamic and static libraries.
Locations of files in different distributions of linux.
Why C software portability is difficult:

• **Heterogeneity**: systems upon which you wish to run the compiler differ in configuration.

• **Conditional compilation**: dealing with heterogeneity by modifying the program that gets compiled for different circumstances.

• **Environmental probing**: how one figures out whether the program needs modification.

• **Environmental probe design**: creating software that always, on every platform, figures out what customizations are needed.
Portability is even challenging between environments that are roughly the same
  E.g., Linux, AT&T Unix, SCO Unix, Solaris...
It is even difficult to assure portability between linux distributions
  E.g., RedHat, Ubuntu, ...

How is this done?
• Here's how we deal with adapting programs to run in multiple environments (autoconf/automake model):
  Makefile.am: description of all goals
    | |
    | +----- configure.in: map of all constraints
    | | |
    | |automake |autoconf
    | | |
    v v |
Makefile.in |
    | |
    | |configure<---+
    | |
    v Makefile
    |
    |make |
    |
    v executable program!

• In this diagram:
  o nodes are files.
  o arrows are processes.
  o arrow labels are programs that accomplish the processes.

• In English:
  o automake reads Makefile.am and configure.in and creates Makefile.in
  o autoconf reads configure.in and creates configure.
  o configure reads Makefile.in and creates Makefile.
  o make reads Makefile and creates desired compiled binaries.
• gcc: compiler
  ○ doesn't know what to compile or link
  ○ must figure out how to compile things by hand.
• make: dependency analyzer
  ○ determines what needs to be done to build an executable.
  ○ does dependency analysis on intermediate files.
  ○ creates an execution order for making the files.
• imake: site dependency envelope
  ○ remembers things that don't change for your site/machine.
  ○ generates a Makefile containing those things.
  ○ unique to X11 and older packages
  ○ not useful for modern programs or for working around bugs in operating systems.
• configure: environment analyzer and portable code generator
  ○ tells C programs what your environment is like.
  ○ allows 'portable' c code to be written.
  ○ de-facto standard for portability analysis.
• autoconf and automake: how to create configure scripts!
  ○ you describe what needs to be located or determined.
  ○ it builds a configure script that locates it!
From the bottom up:

1. make
2. configure
3. autoconf and automake
Make:

- analyzes dependencies between actions
- orders actions into a linear sequence.
- calls compilers as appropriate.
- rules specified in a file Makefile (or makefile)
General form of a make rule:

- line starting with word is a rule:
  target files : source files to use
- lines thereafter, starting with the `<TAB>` character, are commands to execute:
  `<TAB>` command to use to make the target files from the source files.
  `<TAB>` another command
  `<TAB>` another command
- commands executed inside `/bin/sh` (not `/bin/tcsh`, so syntax differs slightly)
- first line not starting with `<TAB>` ends command stream.
Sample makefile: from the old comp15

a1: a1.cc
   g++ -g a1.cc -o a1
a2: a2.o /g/15/class/a2/t2.o
   g++ -g a2.o /g/15/class/a2/t2.o -lcurses -ltermcap -o a2
a2.o: a2.cc
   g++ -g -c a2.cc

• If you say:
   make a1

then the line
  g++ -g a1.cc -o a1

gets executed.

• If on the other hand you say:
  make a2

then a rather complex process happens:
1. Make figures out the *sequence of things to do* by converting
   the *partial order* of things to make
   a2.o precedes a2
   t2.o precedes a2
   a2.cc precedes a2.o

   into some *total order* of things to make:
   a2.cc -> a2.o -> t2.o -> a2

   This is called a *topological sort*. 
2. then make proceeds to make everything it knows how to make
that's *older than its precedent*, using that sequence and the
commands given.
Make's functions

Tuesday, October 30, 2012
7:16 PM

Make automagically:

- manages dependencies between files.
- reads a control file Makefile that designates dependencies between files and commands that can be used to make one file from others.
- uses the timestamps on files to remake any files older than their sources.
- converts the dependency partial order into a total order and executes appropriate commands in order.
If rules specify how to make partial results, then the commands are performed in the correct order to get an up-to-date end-result. E.g.

```
  a1.o: a1.c a1.h
       g++ -g -c a1.c
  t1.o: t1.c a1.h
       g++ -g -c t1.c
  t1 : t1.o a1.o
       g++ -g -o t1 t1.o a1.o -lcurses -ltermcap
```

- the command `make t1' will observe that in making t1, one has to make t1.o and a1.o, and make them first.
- But if one or both of them are up to date, in the sense that they're newer than their sources, they're not remade.
Default rules

• Default rules and make variables simplify complex makefiles.

• The default transformation from a .c file to a .o file is
  $(CC) $(CFLAGS) -c file.c
  where CC and CFLAGS are specified by the user.
  These are make variables, similar to shell variables.

• This means that the Makefile:
  
  CC= g++
  CFLAGS= -g
  LIBS= -lcurses -ltermcap

  a1.o: a1.c a1.h
  t1.o: t1.c a1.h
  t1 : t1.o a1.o

  $(CC) $(CFLAGS) -o t1 t1.o a1.o $(LIBS)

calls two implicit rules.
Changing a default rule:

- The code:

  .c.o:

  $(CC) $(CFLAGS) $(MYFLAGS) -g -c $< -o $@

  does the same thing as the one above, but has all site-specific information at the top.
Stupid make trick: uncompressed and unpacking files.

- The following Makefile 'remembers' how to unpack files!
  hello-1.3.tar: hello-1.3.tar.gz
    gunzip $<
  hello-1.3: hello-1.3.tar
    tar xf $<
- $<: special make variable contains name of dependencies
Make target conventions:

- Certain conventions for target names help us remember what they do across all makefiles.
- **all**: make everything
  
  ```
  make all
  ```

  makes all targets defined in the makefile, e.g.
  ```
  all: foo bar faw faux
  ```

- **install**: make available to users
  
  ```
  make install
  ```

  installs all targets in a reasonable place where normal users can get to them to use them, e.g.,
  ```
  install: foo bar faw faux
  cp $< /usr/local/bin
  ```

- **clean**: clean up after a make
  
  ```
  make clean
  ```

  removes all intermediate files that aren't needed in the final result, e.g.
  ```
  clean:
  rm *.o
  ```

- **realclean**: make clean enough to redistribute
  
  ```
  make realclean
  ```

  removes all created files regardless of function, e.g.:
  ```
  realclean:
  rm *.o foo bar faw faux
  ```
Automatic dependency generation

• Any .c file depends on ALL .h FILES IT INCLUDES.
• If any header changes, the .c file has to be RECOMPILED.
• Fortunately, gcc and g++ can automatically generate dependency parts of a makefile by tracing #include's.
  To do this,
  
g++ -MM file.c file2.c file3.c ... >Make.deps

  which generates a file of dependencies, and put the line
  
  include Make.deps

  in your Makefile as the first line.
• Method: G++ runs the C preprocessor, picks up all the #includes, collates a list of every file you include!
Things to watch out for in using make:

- make doesn't handle *cross-package* dependencies. E.g., to compile g++, you have to first compile bison and flex and put them into your path. Only the README can help here.

- make doesn't handle anything you don't tell it to. E.g., if you don't say things depend upon one another it'll never remake them!

- you can't write a Makefile by terminal cut and paste.
  - cut and paste *don't preserve tab characters*.
  - make requires these to be preserved!
Next level: managing make

- write software using conditional compilation rules.
- never change program again.
- edit Makefile to change conditions.
**Simple example: calling ps in a portable way.**

- Suppose your C program needs to call 'ps' to list processes
- Solaris: `ps -efl`
- Linux: `ps aux`
- Let's write a program `pssr.c` that does that:
  ```
  #include <stdio.h>
  main()
  {
    #ifdef LINUX
      system("ps aux");
    #else LINUX
      system("ps -efl");
    #endif LINUX
  }
  ```
  - If LINUX is defined, then it uses the first form.
  - If LINUX isn't defined, then it uses the second form.

Let's write a Makefile that handles both cases:
```
# uncomment this for LINUX:
# DEFINES= -DLINUX
pssr: pssr.c
  $(CC) $(DEFINES) pssr.c -o pssr
```

- Let's analyze what happens:
  1. If the DEFINES line is uncommented, then the compile command is:
     ```
     gcc -DLINUX pssr.c -o pssr
     ```
  2. If the DEFINES line is commented, then the compile command is:
gcc pssr.c -o pssr

- net effect is to switch which line gets compiled into the program!
The next level: writing the Makefile differently

- portability trick: `uname` command.
- prints system description.
- `uname -a`: all information known.
  SunOS conbrio 5.7 Generic_106541-11 sun4u sparc
  SUNW,Ultra-250
- `uname -n`: name of operating system
  (Linux or SunOS).
  SunOS
REWITING THE MAKEFILE

Let’s do this in a simple way:

```bash
#!/bin/csh
if (`uname -n` == "Linux") then
    echo "DEFINES= -DLINUX" >Makefile
else
    echo "# DEFINES= -DLINUX" >Makefile
endif
```

```bash
pssr: pssr.c
    $(CC) $(DEFINES) pssr.c -o pssr
```

This changes the makefile depending upon whether `uname -n` is |Linux| or |SunOS|!

So when `make` is called, the program compiles correctly!

If we call this program `configure` we know what we're really doing.

This is of course much simpler than a real configure.
But is this a realistic technique?

- This is well and good for simple programs, but realistic programs can have *hundreds* of environmental dependencies.
- Old approaches: imake, xmkmf.
- Current approach: autoconf.
Up another rung: autoconf and automake

- `imake` is a *passive* strategy for determining configuration.
- need to instead be able to make an *active probe* of configuration information.
- main problem is to make the resulting mess manageable.
  - An early example: Larry Wall's `Configure`.
  - Current practice: autoconf, automake.
Configure

- General map of Configure operation

```
Configure (discovery script)
    |    \    \ execute!
    |     \    
    v      \  
config.sh  config.h  (to be included in program)
    |    |
    |    v
Makefile
    |    |
    |    make
    v
your program!
```

- `config.sh`: describes `configuration' (reusable)
- `config.h`: describes `configuration' to C programs.
Configure details

• (BA)SH Script!
• Figures out system dependencies.
• Writes header files describing system.
• Writes Makefile
• So you don't have to figure out dependencies.
• So you don't have to port code.
Highlights of (an old) SSH Configure

- In the following, *italics are comments that are not part of the code.*
- Figure out if our compiler works!

  `contents of meta/config1.txt...`
  `put a test of confdefs.h into conftest.c and compile`

  ```
  cat > conftest.$ac_ext <<EOF
  #line 1014 "configure"
  #include "confdefs.h"
  main(){return(0);}
  EOF
  
  use a mysterious incantation to compile the test program
  if { (eval echo configure:1018: "$ac_link") 1>&5; (eval $ac_link) 2>&5; } && test -s conftest; then
    it compiled! try to run it!
    ac_cv_prog_cc_works=yes
    # If we can't run a trivial program, we are probably using a cross compiler.
    if (./conftest; exit) 2>/dev/null; then
      ac_cv_prog_cc_cross=no
    else
      ac_cv_prog_cc_cross=yes
    fi
  else
    oops! it didn't compile. Give up!
    echo "configure: failed program was:" >&5
    cat conftest.$ac_ext >&5
    ac_cv_prog_cc_works=no
  fi
  rm -fr conftest*
  ```

  if something is really wrong, inform user
  ```
  echo "$ac_t""$ac_cv_prog_cc_works" 1>&6
  if test $ac_cv_prog_cc_works = no; then
    { echo "configure: error: installation or configuration problem: C compiler cannot create executables." 1>&2; exit 1; }
  fi
  ```
echo $ac_n "checking whether the C compiler ($CC $CFLAGS $LDFLAGS) is a cross-compiler"...
$ac_c" 1>&6
echo "configure:1038: checking whether the C compiler ($CC $CFLAGS $LDFLAGS) is a cross-compiler" >&5
echo "$ac_t"$ac_cv_prog_cc_cross" 1>&6
cross_compiling=$ac_cv_prog_cc_cross

preprocess a simple program that contains 'yes' if this is gcc
echo $ac_n "checking whether we are using GNU C"...
$ac_c" 1>&6
echo "configure:1043: checking whether we are using GNU C" >&5
if eval "test \"`echo '$''{ac_cv_prog_gcc}'+set}'`\" = set"; then
  if we've already done this, then don't repeat it
  echo $ac_n "(cached) $ac_c" 1>&6
else
  if not, make up a program that does this
  (not a real C program)
  cat > conftest.c <<EOF
  #ifdef __GNUC__
  yes;
  #endif
  EOF
  preprocessed this program and if the output contains a yes, we're golden
  if { ac_try='$\{CC-cc\} -E conftest.c'; { (eval echo configure:1052: \"$ac_try\") 1>&5; (eval $ac_try) 2>&5; }; } | egrep yes >/dev/null 2>&1; then
    ac_cv_prog_gcc=yes
  else
    ac_cv_prog_gcc=no
  fi
fi

...end of meta/config1.txt

• Figure out which install program to use!

contents of meta/config2.txt...
  # Find a good install program. We prefer a C program (faster),
  # so one script is as good as another. But avoid the broken or
  # incompatible versions:
# SysV /etc/install, /usr/sbin/install
# SunOS /usr/etc/install
# IRIX /sbin/install
# AIX /bin/install
# AFS /usr/afs/bin/install, which mishandles nonexistent args
# SVR4 /usr/ucb/install, which tries to use the nonexistent group "staff"
# ./install, which can be erroneously created by make from ./install.sh.

echo $ac_n "checking for a BSD compatible install"... $ac_c" 1>&6
echo "configure:1325: checking for a BSD compatible install" >&5
if test -z "$INSTALL"; then
  first check whether we've already done this before
  (if so the answer is 'cached' in config.cache)
  if eval "test "$ac_cv_path_install" = set"; then
    echo $ac_n "(cached) $ac_c" 1>&6
  else
    IFS="$IFS="; ac_save_IFS="$IFS; IFS="$IFS:"
    for ac_dir in $PATH; do
      # Account for people who put trailing slashes in PATH elements.
      case "$ac_dir/" in
        /|/|/|/|/etc/*|/usr/sbin/*|/usr/etc/*|/sbin/*|/usr/afs/bin/*|/usr/ucb/*)
        for ac_prog in ginstall installbsd scoinst install;
        check for the first one that exists and record its name
        if test -f $ac_dir/$ac_prog; then
          if test $ac_prog = install &&
            grep dspmsg $ac_dir/$ac_prog >/dev/null 2>&1;
          then
            # AIX install. It has an incompatible calling convention.
      esac
    esac
  fi
fi
# OSF/1 installbsd also uses dspmsg, but is usable.

```bash
: else
   ac_cv_path_install="$ac_dir/$ac_prog -c"
   break 2
  fi
fi
done
;;
esac
done
IFS="$ac_save_IFS"
fi

if you didn't find one, use the default
if test "${ac_cv_path_install+set}" = set; then
   INSTALL="$ac_cv_path_install"
else
   # As a last resort, use the slow shell script. We
don't cache a
   # path for INSTALL within a source directory, because
   # break other packages using the cache if that
directory is
   # removed, or if the path is relative.
   INSTALL="$ac_install_sh"
  fi
fi
echo "$ac_t""$INSTALL" 1>&6

...end of meta/config2.txt

• Put this into Makefile!
  INSTALL = ./install-sh -c
  INSTALL_PROGRAM = ${INSTALL}
  INSTALL_DATA = ${INSTALL} -m 644
  INSTALL_SCRIPT = ${INSTALL_PROGRAM}

• Figure out whether we can include standard #includes
contents of meta/config3.txt...
checks for headers by compiling trivial programs that include them
for ac_hdr in utime.h ulimit.h sys/resource.h
do
  ac_safe=`echo ""$ac_hdr" | sed 'y%/+-%p_%'``
echo $ac_n "checking for $ac_hdr"... $ac_c" 1>&6
echo "configure:2965: checking for $ac_hdr" >&5
check whether we already found this out and stored it in cache=
if eval "test "$ac_cv_header_$ac_safe"+set"; then
  echo $ac_n "(cached) $ac_c" 1>&6
else
  this is the C program that tests whether headers exist
  cat > conftest.$ac_ext <<EOF
  #line 2970 "configure"
  #include "confdefs.h"
  #include <$ac_hdr>
  EOF
  compile this program
  ac_try="$ac_cpp conftest.$ac_ext >/dev/null 2>
  conftest.out"
  { (eval echo configure:2975: "$ac_try") 1>&5; (eval
  $ac_try) 2>&5; }
  ac_err=`grep -v '^ *+' conftest.out`
  if test -z "$ac_err"; then
    if there are no errors it worked and we have the header
    rm -rf conftest*
    eval "ac_cv_header_$ac_safe=yes"
  else
    oops, there was an error, no header available
    echo "$ac_err" >&5
    echo "configure: failed program was:" >&5
    cat conftest.$ac_ext >&5
    rm -rf conftest*
    eval "ac_cv_header_$ac_safe=no"
  fi
  rm -f conftest*
fi
if eval "test "$ac_cv_header_"$ac_safe"\" = yes"; then
  echo "$ac_t""yes" 1>&6
  ac_tr_hdr=HAVE `echo $ac_hdr | sed 'y%abcdefghijklmnopqrstuvwxyz./%ABCDEFGHIJKLMNOPQRSTUVWXYZ___%'`
record existence of header as #define

cat >> confdefs.h <<-EOF
#define $ac_tr_hdr 1
EOF

else
  echo "$ac_t""no" 1>&6
fi
done

...end of meta/config3.txt

• Put a line into sshconf.h (to be included by all programs):

  #define HAVE_ULIMIT_H 1

• Use this in source code:

  #ifdef HAVE_ULIMIT_H
  #include <ulimit.h>
  #endif /* ULIMIT_H */

  ...#ifdef HAVE_ULIMIT_H
    /* Set up the file size ulimit if ULIMIT is set. */
    def = ssh_child_get_env(defenv, "ULIMIT");
    if (def != NULL && atoi(def) > 0)
      ulimit(UL_SETFSIZE, atoi(def));
  #endif /* HAVE_ULIMIT_H */
The next level: autoconf

- Helps programmers *create configure scripts*!
- Has a battery of standardized tests you can apply.
- Allows you to include customized tests
"Simple" Autoconf Example

- configure.in:
  
  dnl Process this file with autoconf to produce a configure script.

  AC_INIT(hello.c) these are the source files
  AC_PROG_CC find a c compiler, please
  AC_PROG_CPP find a c preprocessor, please
  AC_PROG_INSTALL find an install program
  AC_STDC_HEADERS check for whether stdlib.h exists
  AC_HAVE_HEADERS(string.h fcntl.h sys/file.h) and whether these exist
  AC_ALLOCA is there an alloca() function in the c library?
  AC_OUTPUT(Makefile) write a Makefile as a result

produces configure.
Makefile.am produces Makefile

Makefile.am: (approximately)

```makefile
bin_PROGRAMS = hello
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h

what to link to get hello
hello_LDADD = @INTLLIBS@ @hello@

where to find locale information
localedir = $(datadir)/locale
INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"

produces Makefile.in. which configure uses to produce Makefile!
```
autoconf and automake are written in m4.

m4: the macro processor.
  ○ More powerful than C preprocessor.
  ○ Very different syntax.

autoconf and automake scripts are simply macro calls that get expanded into sh script and make control file, respectively.
The bad news

- It takes a *term* to fully understand how to assure C program portability. And two books:
  - One on the real powers of make.
  - One on autoconf, automake, and m4.
- Over 2000 separate tests of system state.
- If that's not enough you can write your own.

And this only covers portability among *nixes!
At the top level, portability is a simple concept. The key to making it possible is to raise the level of abstraction. Makefile.am and configure.in are very high-level depictions of what needs to be known. autoconf translates these into concrete descriptions of what is known.
Another form of portability: abstraction layers/middleware

Craft a middleware layer on all platforms. Assure portability of the middleware layer. Write your application on top of the middleware layer.

Do not worry about portability; the middleware layer does it.
Problem: Browsers don't handle javascript in exactly the same ways.
Solution: JQuery/Bootstrap middleware libraries that

Accomplish many things portably across a broad variety of browsers.
Enable server/browser communication portably.
Problem: write a portable multi-platform application
Solution: a multi-platform portability framework:
   Windows
   Linux
   Mac
that provides the same interfaces on all platforms.
(Qt is pronounced "Cute")
"Lowest common denominator": the capabilities of the framework are those **shared** among the target platforms. First thing to go is performance: cannot presume that rendering, etc are as fast as native methods.