cs3157 lecture #12 notes.

mon 14 apr 2003

http://www.cs.columbia.edu/~cs3157

- news
  - homework #4 due next week (april 21)
  - no labs this week!!!
- today
  - web programming
  - programming tools
  - autoconf

web programming (1).

- web services vs. ”classical” web programming
- client vs. server programming
  - client: JavaScript, Java
  - HTML-centric vs. program-centric
  - HTML-centric: PHP, ASP
  - cgi, fast-cgi
  - (Java) servlet
- data model: Java servlet, database

web programming (2).

- web services vs. web programming
- web services = remote procedure call
  - methods and responses
  - generally, for machine consumption
- web programming $\rightarrow$ generate HTML pages
  - for humans
  - often, database-driven

web programming (3).

- client vs. server programming
- execute code on client:
  - download Java applet $\rightarrow$ self-contained programming environment
  - JavaScript:
    - modify and get values from HTML — "document object model" (DOM)
- execute code on server:
  - generate document
  - state maintenance (HTTP stateless)
    - login, shopping cart, preferences

web programming (4).

- taxonomy

	embedded in HTML	separate
server	SSI (server-side include) ASP (active server pages) PHP (hypertext preprocessor) JSP (java server page) CFM (cold fusion markup language)	server API (NSAPI — netscape server application programmer interface) CGI (common gateway interface) servlets (applet that runs on a server)
client	JavaScript	Java applet plug-ins

web programming (5).

- web state
- state choices:
  - stateless
  - state completely stored on client
  - state referenced by client, stored on server (most common)
- mechanisms:
  - hidden form fields
  - URL parameters
  - cookies (HTTP headers)

web programming (6).

- web as remote procedure call (RPC)
- request = HTTP GET, PUT
- response (result): headers + body
- object identifier ~URL
- typed data (XML) vs. HTML
- ranges from constant ← mostly constant ← completely on-demand

web programming (7).

- server-side include (SSI)
- *.shtml* documents (or configured by default for all .html documents)
- a type of HTML comment that directs the web server to dynamically generate data for the web page whenever it is requested
- include in HMTL comments:

```
<!-- #element attribute=value attribute=value ... -->
```

- limited scripting: if/else, include, exec, variables
- primarily for conditional inclusion, boilerplate
- security issues: `#exec`

web programming (8).

- SSI example = test.shtml

```
<html>
<body>
<p>
last modified
<!--#flastmod file="$document_name"-->
<p>
file size =
<!--#fsize file="$document_name" -->
<p>
<!--#exec cgi="test.cgi" -->
<p>
<!--#printenv -->
</body>
</html>
```

web programming (9).

- common gateway interface (CGI)
- earliest attempt at dynamic web content
- language-independent
- passes HTTP request information via:
  - command line (ISINDEX) — rarely used
  - environment variables: system info + query string (GET)
  - request body (POST) $\rightarrow$ standard input
- return HTML or XML via standard output
- non-parsed headers (NPH) return complete response

web programming (10).

- cgi arguments
- `application/x-www-form-urlencoded format`
  - space characters $\rightarrow$ "+"
  - escape (%xx) reserved characters
  - name=value pairs separated by &
- GET:

```
foo.cgi?name=John+Doe&gender=male&family=5&city=New+York&other=abc%0D%0Adef&nickname=J%26D
```

- POST:
  - include in body of message

web programming (11).

- CGI mechanics
- either called .cgi in HTML directory or stored in cgi-bin
- in CS, both `/home/alice/html/foo.cgi` or `/home/alice/secure_html/foo.cgi` work
- executable (script file)
- usually runs as user `nobody`
- store secret data off the document tree!

programming tools (1).

- tools: what are they for?
  - Creating code modules (compiler)
  - Creating program from modules (linker)
  - Compiling groups of programs (dependencies)
  - Debugging and tracing code
  - Profiling and optimization
  - Documentation: derive from code
  - Coordination and “memory”
  - Testing
  - User installation
  - User feedback

programming tools (2).

- compiler
  - convert source code to object modules
  - in `.o` file, external references not yet resolved
  - `gcc -c` compile only, to produce `.o` files — it doesn't link!

programming tools (3).

- example

```
/* h.c */

#include <stdio.h>

int main() {
  char s[10] = "simon";
  printf( "hello %s\n",s );
}
```

programming tools (4).

- compile:

```
bash# gcc -g -c h.c -o h.o
```

- look at compiler output (`nm` lists symbols from object files):

```
bash# nm h.o
00000000 T main
         U memset
         U printf
bash# gcc -g h.o -o h.x
bash# nm h.x
00000000 a *ABS*
00000000 a *ABS*
00010698 t .nope
00020918 ? _DYNAMIC
000209c4 b _END_
0002089c ? _GLOBAL_OFFSET_TABLE_
000208a8 ? _PROCEDURE_LINKAGE_TABLE_
00010000 ? _START_
```

programming tools (5).

- linker
- combine `.o` and `.so` and/or `.a` into single executable module
- `.so` and `.dll`: dynamically loaded at run-time
- example:

```
bash# ldd a.out
libc.so.1 =>     /usr/lib/libc.so.1
libdl.so.1 =>    /usr/lib/libdl.so.1
```

programming tools (6).

- creating a static library
- static library for linking: `libsomething.a`
- first create `.o` files:

```
bash# gcc -c helper.c
```

- then create library:

```
bash# ar rv libsomething.a *.o
bash# ranlib libsomething.a
```

- then use library:

```
bash# gcc -L/your/dir -lsomething
```

programming tools (7).

- creating a dynamic library
- details differ for each platform
- creating shared library on unix:

```
bash# gcc -shared -fPIC -o libhelper.so *.o
```

- usage is same as for static library
- you can also define `LD_LIBRARY_PATH`

programming tools (8).

- testing
- every module and functionality needs to have an (automated) test
- regression testing
  - change
  - test new functionality
  - then test old functionality to make sure it is backwards compatible
- easy for simple functions

programming tools (9).

- program tracing
- simple debugging: find out what system calls a program is using
- `truss` on Solaris, `strace` on Linux
- does not require access to source code
- does not show stdio calls, but can use -u libc
- -f: follow children
- -p: attach to existing process (e.g., truss -p 27878 to see what process is doing when doing certain action)

programming tools (10).

- truss example

```
bash# truss h.x
execve(".//h.x", 0xEFFFFB38, 0xEFFFFB40)  argc = 1
open("/usr/lib/libc.so", O_RDONLY)              = 3
fstat(3, 0xEFFFF744)                            = 0
mmap(0x00000000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xEF7B0000
mmap(0x00000000, 655360, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xEF700000
munmap(0xEF786000, 57344)                       = 0
mmap(0xEF794000, 33408, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 540672) = 0xEF794000
open("/dev/zero", O_RDONLY)                     = 4
mmap(0xEF79E000, 3524, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 4, 0) = 0xEF79E000
close(3)                                        = 0
open("/usr/lib/libdl.so.1", O_RDONLY)           = 3
fstat(3, 0xEFFFF744)                            = 0
mmap(0xEF7B0000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 0) = 0xEF7B0000
close(3)                                        = 0
close(4)                                        = 0
ioctl(1, TCGETA, 0xEFFFECE4)                    = 0
hello simon
write(1, " h e l l o   s i m o n\n", 12)        = 12
lseek(0, 0, SEEK_CUR)                           = 15398
_exit(1)
```

programming tools (11).

- memory utilization — top
- show top consumers of CPU and memory
- updates periodically (type “q” to exit)

```
bash# top
11:28am  up 13 days, 14:18, 12 users,  load average: 0.00, 0.00, 0.00
117 processes: 116 sleeping, 1 running, 0 zombie, 0 stopped
CPU0 states:  0.3% user,  0.0% system,  0.0% nice, 99.2% idle
CPU1 states:  0.0% user,  0.1% system,  0.0% nice, 99.4% idle
Mem:   515344K av,  510136K used,    5208K free,       0K shrd,  119612K buff
Swap:  265064K av,   42172K used,  222892K free                 147696K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 6560 sklar     17   0  1204 1204   928 R     0.5  0.2   0:00 top
 1091 root      10   0  1140  956   852 S     0.1  0.1   2:13 nmbd
    1 root       8   0   520  480   460 S     0.0  0.0   0:13 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:01 keventd
    3 root      19  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU0
    4 root      19  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU1
    5 root       9   0     0    0     0 SW    0.0  0.0   0:18 kswapd
    6 root       9   0     0    0     0 SW    0.0  0.0   0:00 bdflush
    7 root       9   0     0    0     0 SW    0.0  0.0   0:08 kupdated
    8 root      -1 -20     0    0     0 SW<   0.0  0.0   0:00 mdrecoveryd
   64 root       9   0     0    0     0 SW    0.0  0.0   0:00 khubd
```

programming tools (12).

- debugging
- interact with program while running
  - step-by-step execution
    - instruction
    - source line
    - procedure
  - inspect current state
    - call stack
    - global variables
    - local variables

programming tools (13).

- more debugging
- requires compiler support:
  - generate mapping from PC to source line
  - symbol table for variable names
  - compile with `-g` option (`gcc -g`)
- unix debugger: `gdb`
- useful for debugging core dumps:

```
bash# gdb h.x core
...
(gdb) where
```

programming tools (14).

- debugging example

```
bash# gdb h.x
GNU gdb 5.2.1
Copyright 2002 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 "sparc-sun-solaris2.5.1"...
(gdb) break main
Breakpoint 1 at 0x10750: file h.c, line 4.
(gdb) run
Starting program: /hmt/skol/u/3/e/eis20}03/cs3157/h.x

Breakpoint 1, main () at h.c:4
4         char s[10] = "simon";
(gdb)
```

programming tools (15).

- gdb hints
- make sure your source file is around and doesnt get modified
- gdb does not work (well) across threads

programming tools (16).

- gdb commands

run arg	run program
call f(a,b)	call function in program
step N	step N times into functions
next N	step N times over functions
up N	select stack frame that called current one
down N	select stack frame called by current one

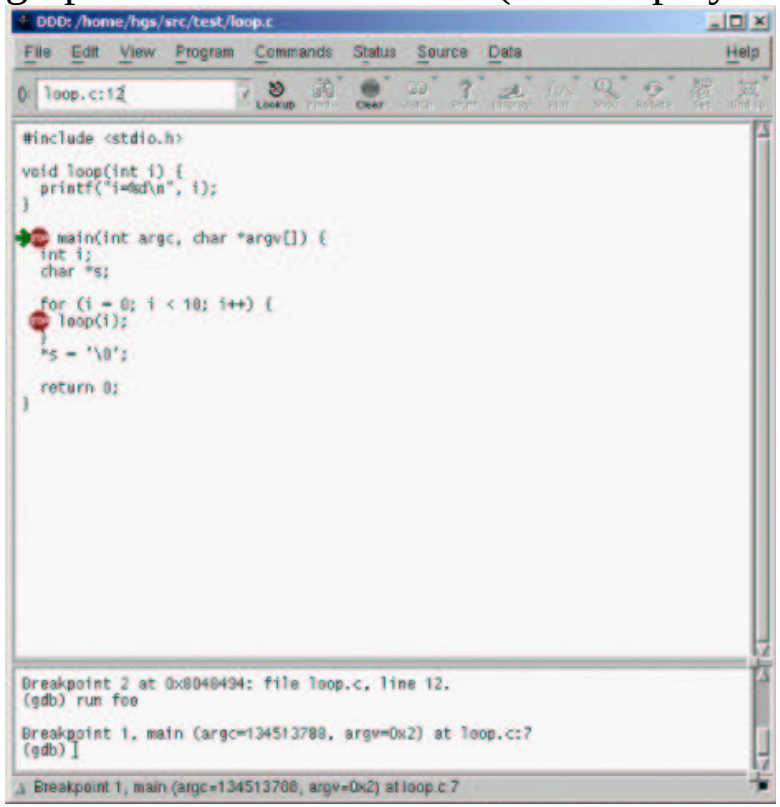
programming tools (17).

- gdb break points

break main.c:12	set break point
break foo	set break at function
clear main.c:12	delete breakpoint
info break	show breakpoints
delete 1	delete break point 1
display x	display variable at each step

programming tools (18).

- graphical interface — ddd (data display debugger)



programming tools (19).

- building programs
- programs consist of many modules
- dependencies:
  - if one file changes, one or more others need to change
  - `.c` depends on `.h` $\rightarrow$ re-compile
  - `.o` depends on `.c` $\rightarrow$ re-compile
  - `.x` (executable) depends on `.o`s $\rightarrow$ link
  - `.a` or `.so` (library) depends on `.o` $\rightarrow$ archive
  - recursive!

programming tools (20).

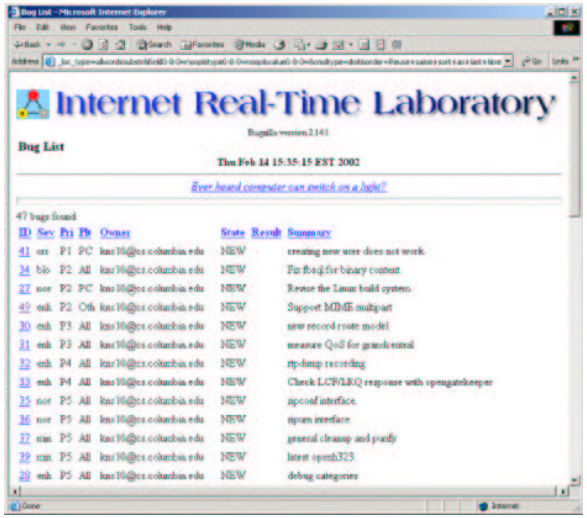
- make
- make maintains dependency graphs
- based on modification times
- `Makefile` or `makefile` as default name
- we covered this last time...

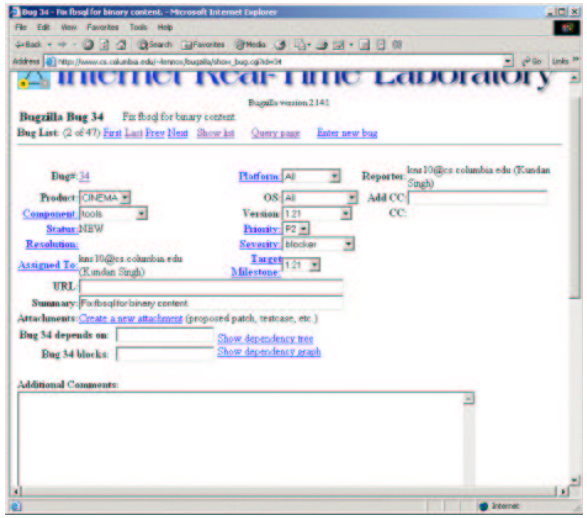
programming tools (21).

- user feedback — bug tracking
- automatically capture system crash information
  - non-technical users
  - privacy?
- user and developer bug tracking
  - make sure bugs get fixed
  - estimate how close to done

programming tools (22).

- bug tracking





programming tools (23).

- development models
- Integrated Development Environment (IDE)
  - integrate code editor, compiler, build environment, debugger
  - graphical tool
  - single or multiple languages
  - e.g., VisualStudio, JCreator, Forte, CodeWarrior, ...
- Unix model
  - individual tools, command-line

programming tools (24).

- source code management
- problem: lots of people working on the same project
  - source code (C, Perl, ...)
  - documentation
  - specification (protocol specs)
- mostly on different areas
- sometimes on different versions
- sometimes on different hardware and OS versions
- releases may need to run on different hardware and OS versions
  - test many places!
  - different hardware: memory, processor speed, network capabilities

programming tools (25).

- installation
- traditional:
  - `tar` (archive) file and `gzip` (compress)
    - `*.tgz`
    - `*.tar.gz`
  - `unzip`, verb+untar+ to uncompress
  - then compile (make) or distribute binaries
  - often (should!) contains documentation, etc.
  - look for README, INSTALL, etc
- Linux rpm
- Solaris pkg
- InstallShield (windows)

programming tools (25).

- creating a `*.tar.gz` (tarred, gzipped) file:

```
bash# tar cvf myfile.tar *
```

  - c = create
  - v = verify
  - f = file name

```
bash# gzip myfile.tar
```

  - creates `myfile.tar.gz`
  - compresses argument

- or

```
bash# tar czvf myfile.tgz *
```

  - z = zip

programming tools (26).

- using a `*.tar.gz` (tarred, gzipped) file:

```
bash# gunzip myfile.tar.gz
```

  - creates `myfile.tar`
  - decompresses argument

```
bash# tar xvf myfile.tar *
```

  - x = extract
  - v = verify
  - f = file name

- or

```
bash# tar xzvf myfile.tgz *
```

  - z = (un)zip

programming tools (27).

- rpm
- RedHat Linux package manager
  - `http://www.redhat.com/docs/books/max-rpm/`
- activities for an application:
  - Installation – on different architectures
  - Updates
  - Inventory: whats installed
  - Un-install
- Each Unix architecture seems to have one: Solaris pkg, RPM (www.rpm.org), ...

programming tools (28).

- more rpm
- package label, e.g., perl-5.001m-4:
  - software name
  - software version
  - package release
- Package-wide information
  - date and time built
  - description of contents
  - total size of all files
  - grouping information
  - digital signature

programming tools (29).

- more more rpm
- per-file information:
  - name of file and where to install it
  - file permissions
  - owner and group specification
  - MD5 checksum
  - file content

programming tools (30).

- using rpm:
  - `rpm -i` : install package, check for dependencies
  - `rpm -e` : erase package
  - `rpm -U` : upgrade package
  - `rpm -q` : query packages (e.g., -a = all)

programming tools (31).

- rpm example:

```
bash# rpm -q -i telnet
Name        : telnet                    Relocations: (not relocateable)
Version     : 0.17                      Vendor: Red Hat, Inc.
Release     : 18.1                      Build Date: Wed Aug 15 15:08:03 2001
Install date: Fri Feb  8 16:50:03 2002  Build Host: stripples.devel.redhat.com
Group       : Applications/Internet     Source RPM: telnet-0.17-18.1.src.rpm
Size        : 88104                     License: BSD
Packager    : Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>
Summary     : The client program for the telnet remote login protocol.
Description :
Telnet is a popular protocol for logging into remote systems over the
Internet.  The telnet package provides a command line telnet client.

Install the telnet package if you want to telnet to remote machines.

This version has support for IPv6.
```

programming tools (32).

- doc++
- documentation system for C/C++ and Java
  - generate LaTeX for printing and HTML for viewing
  - hierarchically structured documentation
  - automatic class graph generation (Java applets for HTML)
  - cross references
  - formatting (e.g., equations)
- similar to `javadoc`

programming tools (33).

- configuration tools
  - autoconf: configuration files
  - automake: make files
- code generation:
  - indent (e.g., indent -kr -i2 hello.c): automated indentation for C programs
  - lex, flex: lexical analyzers
  - yacc, bison: compiler generator

autoconf (1).

- autoconf: software portability
- many software products need to run on lots of platforms
  - Unix, Windows, (old) Macintosh, VMS, ...
  - Varieties of Unix: Linux, Solaris, SunOS 4.x, Free/Net/OpenBSD, MacOS X (Darwin), Tru64, AIX, HP/UX, SVR4, SCO, Minix, ...
- Open-source software especially needs to be portable
  - Create a developer community

autoconf (2).

- historical practice
- ignore the problem:
  - 1982: “All the worlds a VAX” (running BSD 4.2)
  - 1992: “All the worlds a Sun” (running SunOS 4.x)
  - 2002: “All the worlds Linux” (on an x86)
- this is great, for as long as its true...

autoconf (3).

- first solution:
- code contains a sea of platform-specific `#ifdef` statements:

```
#ifdef __linux__
   linux_specific_code()
#elif defined(__sun__) && defined(__svr4__) /* Solaris */
   solaris_specific_code()
#else
  #error "What system is this?"
#endif
```

- this only works for platforms youve already ported your code to
- it can quickly become unmanageable

autoconf (4).

- second solution:
- Makefile documents -D flags, -L flags, etc., to pass to compiler for specific systems or compilation options
- user modifies the programs Makefile by hand, in a text editor
- this works okay for very small projects; runs into problems very quickly
- it is error-prone; users often forget to specify flags, mis-type them, or give the wrong ones
- porting to a new platform can be very difficult

autoconf (5).

- third solution:
- variant of second solution: interactive scripts to set all the options
- run a shell script. It asks you lots of questions like “does this system support the argle(3) function with extended frobnitz? (y/n):”
- shell script automatically creates your makefile
- very bad for inexperienced software builders
- not (easily) possible to build software non-interactively
- with good per-system defaults, this can work, however. (Perls build system works like this.)

autoconf (6).

- today's solution: autoconf
- *configure* script
  - automatically checks the characteristics of the build system
    - programs, libraries, header files, system calls
  - generates a Makefile from a programmer-supplied template
  - Generates a config.h file defining compiler and system characteristics

autoconf (7).

- autoconf philosophy
- check *features*, not *systems*
  - Does this system support `select()` or `poll()`?
  - Not “Is this BSD or SysV”?
- where possible, check features *directly*
  - for example, try to compile a program that invokes a function — see if it compiles/links/runs (depending on the feature)
  - this isnt always possible, e.g., “what kind of audio drivers does this OS use?”

autoconf (8).

- on the part of the developer: *configure.in*

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT([littleserver], [1.0])
AC_CONFIG_SRCDIR(littleserver.c)
AC_CONFIG_FILES(Makefile)
AC_CONFIG_HEADERS(config.h)

dnl Checks for programs.
AC_PROG_CC

dnl Checks for libraries.
AC_CHECK_LIB(nsl, gethostbyaddr)
AC_CHECK_LIB(socket, bind)

dnl Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS(limits.h sys/time.h unistd.h crypt.h string.h stdlib.h)
AC_HEADER_TIME

dnl Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
AC_TYPE_SIGNAL

dnl Checks for library functions.
AC_CHECK_FUNCS(select socket strftime strtol)
AC_REPLACE_FUNCS(strerror strdup)

AC_OUTPUT
```

autoconf (9).

- on the part of the installer:

```
bash# ./configure
bash# ./make
```

- example:

```
bash# ./configure
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for gethostbyaddr in -lnsl... yes
checking for bind in -lsocket... yes
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for limits.h... yes
checking for sys/time.h... yes
checking for unistd.h... yes
checking for crypt.h... yes
checking for string.h... yes
checking for stdlib.h... yes
checking whether time.h and sys/time.h may both be included... yes
checking for working const... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
creating config.h
```

autoconf (10).

- autoconf: configure.in structure
- `AC_INIT (package, version, [bug-report-address])`
  - start configure.in
- `AC_CONFIG_SRCDIR (unique-file-in-source-dir)`
  - uniquely identify source directory
- `AC_CONFIG_FILES (file..., [cmds], [init-cmds])`
  - create files from templates (e.g. Makefile)
- `AC_CONFIG_HEADERS (header ..., [cmds], [init-cmds])`
  - create header files (e.g. config.h)
- `AC_OUTPUT`
  - output all the generated files

autoconf (11).

- configure.in program checks
- autoconf can check if the build system has certain programs
  - `AC_PROG_AWK`
    - Sets output variable AWK to mawk, gawk, nawk, or awk
  - `AC_PROG_LEX / AC_PROG_YACC`
    - Find lex (flex) or yacc (byacc, bison)

autoconf (12).

- configure.in: compiler checks
- autoconf can find the compiler and check its characteristics
  - `AC_PROG_CC, AC_PROG_CXX`
    - find the C or C++ compiler
  - `AC_PROG_C_STDC`
    - check if the C compiler is ANSI C, after trying various compiler options to make it do so
  - `AC_C_CONST`
    - check if the C compiler supports `const`
    - If not, `#define` const to the empty string, so you can use it anyway
  - `AC_C_BIGENDIAN`
    - check if the system is *big-endian*; i.e., stores integers most-significant-byte first. (Sparc is big-endian; x86 is little-endian.)

autoconf (13).

- configure.in library checks
- autoconf can determine whether certain libraries are available
  - `AC_CHECK_LIB(library, function)`
    - Check whether the library can be linked (as llibrary), and then whether the function can be found inside it.
    - Once a library is found, its linked in by default for future calls to `AC_CHECK_LIB`, so you can have one library depend on another.

autoconf (14).

- configure.in: header checks
- autoconf can check whether certain header files are available
  - `AC_CHECK_HEADER`
    - Check whether a header file is available
  - `AC_HEADER_STDC`
    - Check whether the system header files conform with ANSI C (not the same as `AC_PROG_C_STDC`, or `__STDC__`)
  - `AC_HEADER_TIME`
    - Check whether `<time.h>` and `<sys/time.h>` can both be included

autoconf (15).

- configure.in: type checks
- autoconf can check characteristics of structures and types in the compilation environment
- type checking code `#includes` all detected header files checked so far
  - `AC_CHECK_MEMBER(aggregate.member)`
    - check whether the given aggregate (struct or union) is defined, and if so, whether it contains the given member
  - `AC_CHECK_TYPE(type)`
    - check whether the compiler knows about a specific type
  - `AC_TYPE_SIZE_T`
    - check whether the compiler knows about the type `size_t`; if not, typedef it to `unsigned`

autoconf (16).

- configure.in: function checks
- autoconf can check whether system and library functions are available
  - `AC_CHECK_FUNCS(functions...)`
    - check whether the given functions are available
  - `AC_REPLACE_FUNCS(functions...)`
    - check whether the given functions are available, and if not, link in replacement code re-implementing them

autoconf (17).

- output = Makefile
- some autoconf output is needed by the Makefile, so is defined as template substitutions
  - libraries, programs, search paths
- developer must write Makefile.in: template Makefile
  - other than template variables, looks exactly like a normal Makefile
- patterns in Makefile.in are substituted with results of autoconf tests
  - `@CC@` $\rightarrow$ C compiler
  - `@AWK@` $\rightarrow$ Awk executable
  - `@CFLAGS@` $\rightarrow$ compiler flags
  - `@LIBS@` $\rightarrow$ matched libraries

autoconf (18).

- output = config.h
- other autoconf output is needed by the source code, so symbols are defined in `config.h`
- source code `#includes <config.h>`, then makes decisions based on the symbols defined.
- for example

`HAVE_SYS_TIME_H`	`<sys/time.h>` exists
`WORDS_BIGENDIAN`	integers are big-endian
`HAVE_SELECT`	select() was found
`HAVE_STRUCT_PASSWD_PW_GECOS`	struct passwd has the pw_gecos field.

autoconf (19).

- system-dependent tests
- some things cant be checked automatically
  - things that only work as root
  - details of system object formats
- for these, autoconf provides system-dependent checks
- check the system type of either the build or the host system
  - standard GNU naming system
    - i686-unknown-linux-gnu
    - sparc-sun-solaris
- Use shell pattern matching on these names

autoconf (20).

- custom tests
- sometimes you need to check for things that autoconf doesnt already have tests for
- you can write custom tests:
  - `AC_TRY_CPP, AC_TRY_COMPILE, AC_TRY_LINK, AC_TRY_RUN`
    - try to preprocess / compile / link / run a specific fragment of C code.
    - specify actions to take if test succeeds or fails.

autoconf (21).

- results of custom tests
- custom tests need to be able to output their results
  - `AC_DEFINE`
    - define a C preprocessor symbol in config.h
  - `AC_SUBST`
    - substitute a variable pattern into Makefile.in
  - `AC_CACHE_CHECK`
    - cache a variable in config.cache for future configure runs
  - `AC_MSG_CHECKING / AC_MSG_RESULT`
    - output messages telling the user that somethings being checked

autoconf (22).

- subtleties of custom tests
- autoconf actually works by using the m4 macro processor to create a shell script
- so you can embed your own shell (/bin/sh) code in your custom tests
- HOWEVER:
  - you cant just write bash code and expect everything to work!
  - since the point of the ./configure script is to run anywhere, you need to write shell code that can run on any Unix systems shell
  - Lowest-common-denominator scripting!

autoconf (23).

- custom libraries of tests
- if you need to execute your own tests, you can write autoconf functions
  - `AC_DEFUN` defines new functions
- custom functions can be embedded into a custom file
  - aclocal.m4: project-specific custom functions
  - acsite.m4: system-wide custom functions

autoconf (24).

- other parts of the GNU build environment
- automake
  - automates creation of Makefile.in.
  - automatically supports make clean, make install, building outside the source directory, creating .tar.gz distributions, etc.
  - good for simple projects; not very flexible for complex projects
- libtool
  - creating shared libraries, and dynamically-loadable-libraries, is wildly different on all the platforms that support them
  - libtool is a shell script that encapsulates the knowledge of how to do this, how to set load paths automatically, and so forth.