Build systems (Autotools)

ES 2016/2017

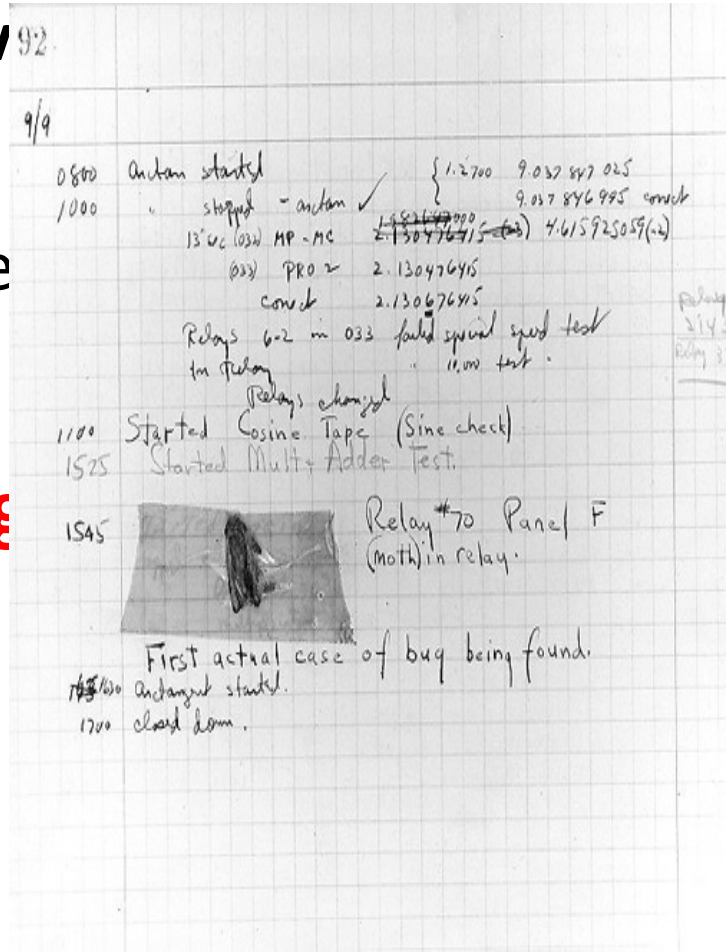
Adam Belloum

a.s.z.belloum@uva.nl

Material Prepared by Eelco Schatborn

Computers and programming

- first computers were **function specific**
  - work on a **specific problem** with **v**
- analog systems,
  - `programmed' by connecting tube
- programming took a **long time**
- finding problems **took even long**

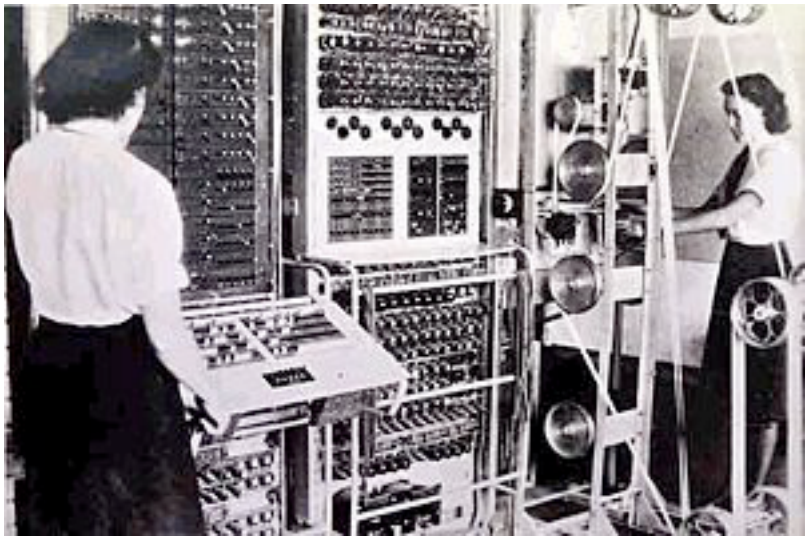


The `old' days

Colossus

- created during the 2nd world war, finished in 1943
- used to crack German codes

  (Bletchley Park in England)



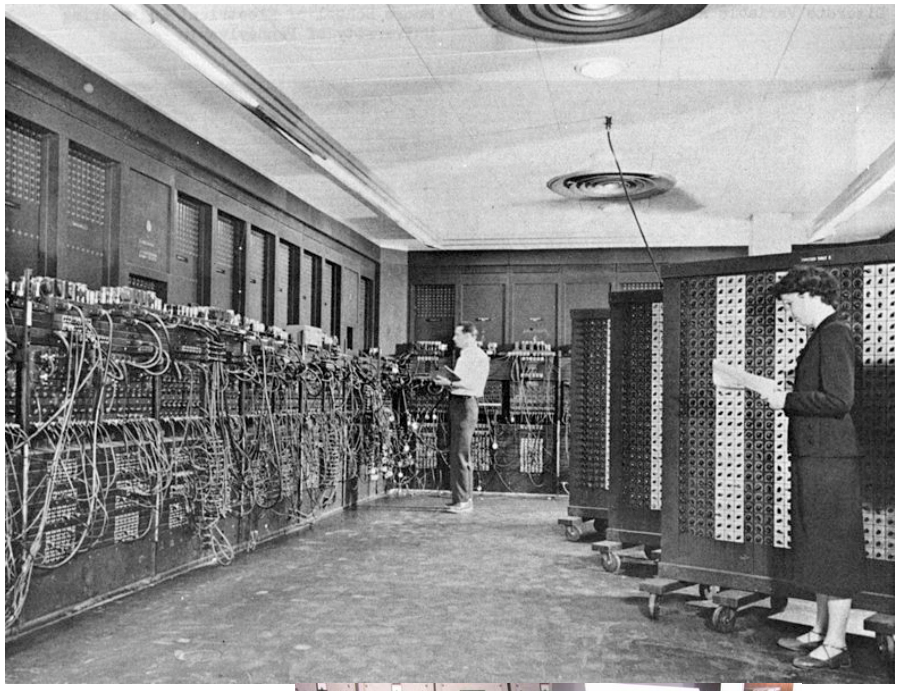
SNE visit to Bletchley park

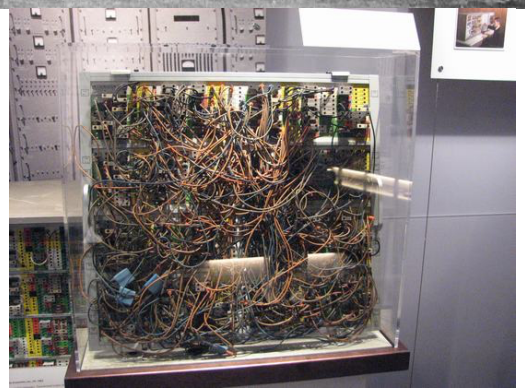
http://www.computer50.org/mark1/contemporary.html

The `old' days

ENIAC

- also created during the 2nd world war by the Americans, finished in 1946
- used to compute trajectories for grenades
- programming meant connecting the tubes with wires





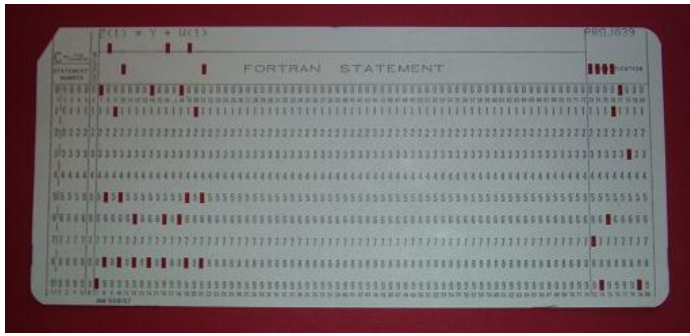
Evolution in programming

- **moved from hardware to software**, so no more messy wires
- programs now stored in memory
- BUT **each machine instruction** had to be entered by **hand**
- can take a **long time**, prone to errors
- **memory** would be **empty** at startup
- storing programs and data off-line would be nice

Mark Rendle - History of Programming, Norwegian Developer Conference 2014, http://vimeo.com/97541186

Storage: punch cards or tapes

- already invented for automated weaving looms
- paper with holes in it, each hole a bit
- can store any data, including programs
- writing programs meant punching the correct sequence of codes
- cumbersome, time consuming
- errors meant re-punching and re-evaluating



Storage: bigger and faster

- storage capabilities **grew**, no longer a **bottleneck**
- more and more **languages (high level)** start appearing
- **creating** and **changing** software became easier
- creation and testing **faster**, less time needed to program
- **larger** programs became possible

Mark Rendle - History of Programming, Norwegian Developer Conference 2014, http://vimeo.com/97541186

High level Languages and Compilers

- **Higher** level languages need to be translated to **machine code**
- compilers are needed

Bigger programs

- **larger** programs made programming more difficult
- **maintainability** an issue
- **compilation time** now also a factor

Problems:

- Long files are **harder** to **manage** (for both programmers and machines)
- Every change requires **long compilation**
- Many programmers **can not modify** the same file simultaneously

- Division to components is desired

Splitting source code

- sources can be **split** over more files
- problem: How to **merge** them into one?
- issues: dependency, multiple includes . . .

Solutions

1 concatenate the files and compile the whole

- can create very big sources, takes up a lot of memory
- compilation can take a long time

2 compile each far as possible and then glue them together afterwards

- adds extra step into the compilation process
- + less memory needed

Object files

each **source file** is compiled into an **object file**

- compiling the sources one by one **saves memory**
- object files are independent
  - checking is done at **linking** time
  - contains the machine code, data, program symbols, debugging information. . .
  - object files can **later be linked** together
- only have to **re-compile** the source file that were changed

What goes into an object file?

An object file contains five kinds of information.

- ***Header information:*** overall information about the file, such as the size of the code, name of the source file, and creation date.
- ***Object code:*** Binary instructions and data generated by a compiler or assembler.
- ***Relocation:*** A list of the places in the object code that have to be fixed up when the linker changes the addresses of the object code.
- ***Symbols:*** Global symbols defined in this module, symbols to be imported from other modules or defined by the linker.
- ***Debugging information:*** Other information about the object code not needed for linking but of use to a debugger. This includes source file and line number information, local symbols, descriptions of data structures used by the object code such as C structure definitions.

http://www.iecc.com/linker/linker03.html

Symbolic table in Object file

Consider the following program written in C

```c
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double  sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1;  i <= count;  i++)
        sum += bar((double) i);
    return sum;
}
```

A C compiler that parses this code will contain at least the following symbol table entries

Symbol name	Type	Scope
`bar`	function, double	extern
`x`	double	function parameter
`foo`	function, double	global
`count`	int	function parameter
`sum`	double	block local
`i`	int	for-loop statement

http://en.wikipedia.org/wiki/Symbol_table

Linkable, executable, loadable object files

- **A linkable** file contains extensive **symbol** and **relocation information** needed by the **linker** along with the object code.
  - The object code is often divided up into many small logical segments that will be treated differently by the linker.

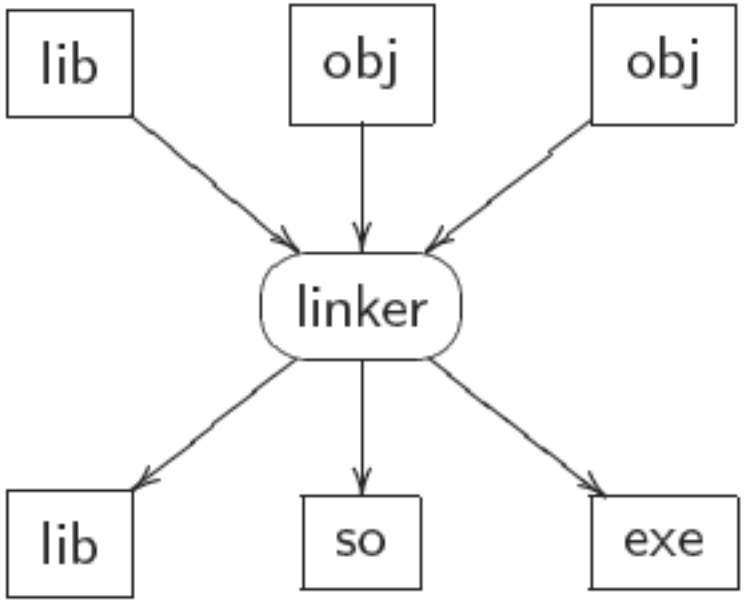
- **An executable** file contains object code, usually page aligned to permit the file to be mapped into the address space, but **doesn't need** any symbols (unless it will do runtime dynamic linking), and **needs little** or **no** relocation information.
  - The object code is a single large segment or a small set of segments that reflect the hardware execution environment.

- **A loadable (**depending on system's runtime environment) file may consist solely of object code, or may contain complete symbol and relocation information to permit **runtime symbolic linking**.

http://www.iecc.com/linker/linker03.html

Linking object files

- object files produced by a compiler can be linked together to form an executable
- function references are linked to the relevant code
- the linker **resolves** the **symbols** in **each object** file while linking



Example of Object file (1)

- **a.out** format served the Unix community for over a decade
  - but with the advent of Unix System V, AT&T decided that it needed something better to support **cross-compilation**, **dynamic** linking…
- **ELF** files come in three slightly different flavors:
  - **Relocatable** files are created by compilers and assemblers but need to be processed by the linker before running.
  - **Executable** files have all relocation done and all symbols resolved except perhaps shared library symbols to be resolved at runtime.
  - **Shared** objects are shared libraries, containing both symbol information for the linker and directly runnable code for runtime.

http://www.iecc.com/linker/linker03.html

Example of Object file (2)

- **MS-DOS .COM files (null object format):** COM file literally consists of nothing other than binary code.
- **IBM 360 object format :** designed in the early 1960s, but remains in use today. It was originally designed for 80 column punch cards, but has been adapted for disk files on modern systems
- **Microsoft Portable Executable format**: MS_NT has extremely mixed heritage including earlier versions of MS-DOS and Windows, Digital's VAX VMS (on which many of the programmers had worked), and Unix System V
- **Intel/Microsoft OMF files**: designed in the late 1970s for the 8086.
  - Over the years vendors, including Microsoft, IBM, and Phar Lap (wrote a widely used set of 32 bit extension tools for DOS), defined their own extensions.
  - The current Intel OMF is the union of the original spec and most of the extensions, minus a few extensions that either collided with other extensions or were never used.

http://www.iecc.com/linker/linker03.html

Shared Objects (Libraries)

Shared libraries are **loaded** by programs when they **start.**

- Shared libraries allow you:
  - **update** libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
  - **override** specific libraries or even specific functions in a library when executing a particular program.
  - do all this **while programs are running** using existing libraries.
- When a shared library is installed properly, all programs that start afterwards automatically use the new shared library.

http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Shared Objects (Libraries)

- object files can be linked to **libraries of symbols**
- **not executable**, but can be used for linking yet again
- Libraries are **reusable** for all programs
- can be **linked statically** or **dynamically**
  - **static** linking adds the library to an **Executable**
    - static linking makes for **bigger programs**
  - **dynamic** linking means a library is **loaded** into memory when the **executable** requires it
    - dynamic linking can be "**slower at starting up**"

Shared Objects (Libraries)

Shared Library Names

- The soname has the prefix ``lib'', the name of the library, the extension ``.so'', followed by a period and a version number that is incremented whenever the interface changes

Filesystem Placement

- The GNU standards recommend installing by default all libraries in /usr/local/lib when distributing source code.
- They also define the convention for overriding these defaults and for invoking the installation routines.
- According to the FHS (Filesystem Hierarchy Standard),
  - most libraries should be installed in **/usr/lib**,
  - libraries required for startup should be in **/lib**
  - libraries that are not part of the system should be in **/usr/local/lib**.

http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Shared Objects (Libraries)

- The list of directories to be searched is stored in the file
  /etc/ld.so.conf

- If you want to override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in /etc/ld.so.preload;
  - these ``preloading'' libraries will take precedence over the standard set. This preloading file is typically used for emergency patches;
  - a distribution usually won't include such a file when delivered.

http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Shared Objects (Libraries)

How Libraries are Used: in all Linux systems, starting up an ELF binary executable automatically causes:

- the program loader to be loaded and run. this loader is named **/lib/ld-linux.so.X** (where X is a version number).
- This loader, in turn, finds and loads all other shared libraries used by the program.

Note:

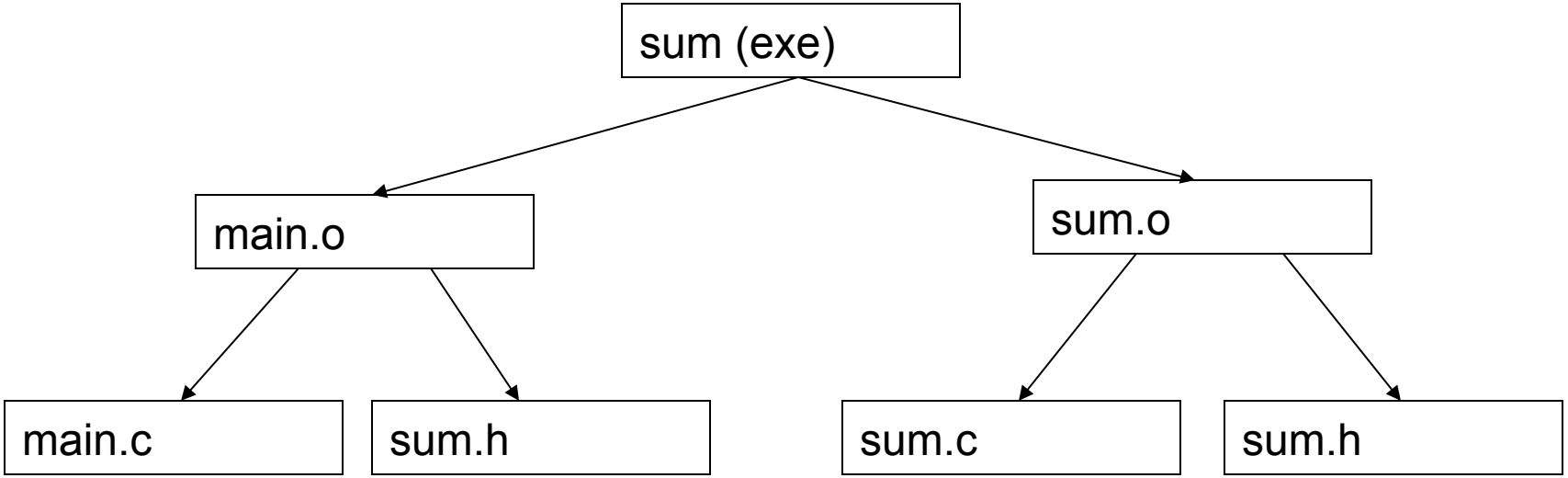
Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used.

- The program **ldconfig** by default reads in the file **/etc/ld.so.conf**, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to **/etc/ld.so.cache** that's then used by other programs.
- The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes;
- running ldconfig is often one of the steps performed by package managers when installing a library.
- On start-up, then, the dynamic loader actually uses the **file /etc/ld.so.cache** and then loads the libraries it needs.

http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Project structure

- Project **structure** and **dependencies** can be represented as a DAG (Directed Acyclic Graph)
- Example :
  - Program contains 3 files
  - main.c., sum.c, sum.h
  - sum.h included in both .c files
  - Executable should be the file sum


sum (exe)
main.o
sum.o
main.c
sum.h
sum.c
sum.h

Recap: building an executable

- say we have 2 source files for our program
- we compile each:

```
#> compile main.SNE
```

compiling... **main.obj** created

```
#> compile rest.SNE
```

compiling... **rest.obj** created

- we'll now link them into an executable, or binary

```
#> link main.obj rest.obj myprog.a
```

linking... **myprog.a** created

Compiling large Programs

- **compiling** and **linking** by **hand** becomes tedious when creating large programs
- automation was required
- created by hand per project, the programmer **must think** of **everything**
- introduced **inconsistencies**, only works on local system, dependency checking is boring. . .
- generic automation is required

Make (1)

- one of the first **build utilities**, **build automation**
- **very wide spread** due to inclusion into Unix
- created (originally) by Stuart Feldman in **1977** at Bell Labs
- Feldman received the ACM Software System Award
- make has undergone rewrites, many versions now
- BSD make, GNU make, Micrsoft nmake

- Makefile layout

```
target: dependencies
        command 1
        command 2
        ...
```

Make (2)

Provides

- **automated building** using a Makefile
- **only files** that have changed
- methods for maintaining dependencies
- **not language** specific

Problems

- **no dependency** checking, left to programmer
- **tailoring** to **multiple** platforms is problematic

Make (3)

Example Makefile

```
helloworld: helloworld.o
	cc -o $@ $<
helloworld.o: helloworld.c
	cc -c -o $@ $<
clean:
	rm -f helloworld helloworld.o
```

Build systems

- Tools for automating the creation of **platform specific building tools** (Makefiles and such)
- make-based systems
  - GNU automake, CMake, imake, qmake
- Others
  - Apache Ant - MSBuild
  - Debian Package Maker - …
  - GNU Build Tools
- See http://en.wikipedia.org/wiki/List_of_build_automation_software

GNU Build Tools (a.k.a autotools) (1)

- **suite of programming tools** produced by the GNU project
- designed to **assist** in **creating portable** (Unix) **source packages**
- widely used in many free software and open source packages
- intended to be used with other GNU tools like the GNU C Compiler
- the system requires (relatively) little input
  - `configure.ac (configure.in (old))`, `Makefile.am`, `acconfig.h`

http://sources.redhat.com/autobook/autobook/autobook.html#SEC_Top

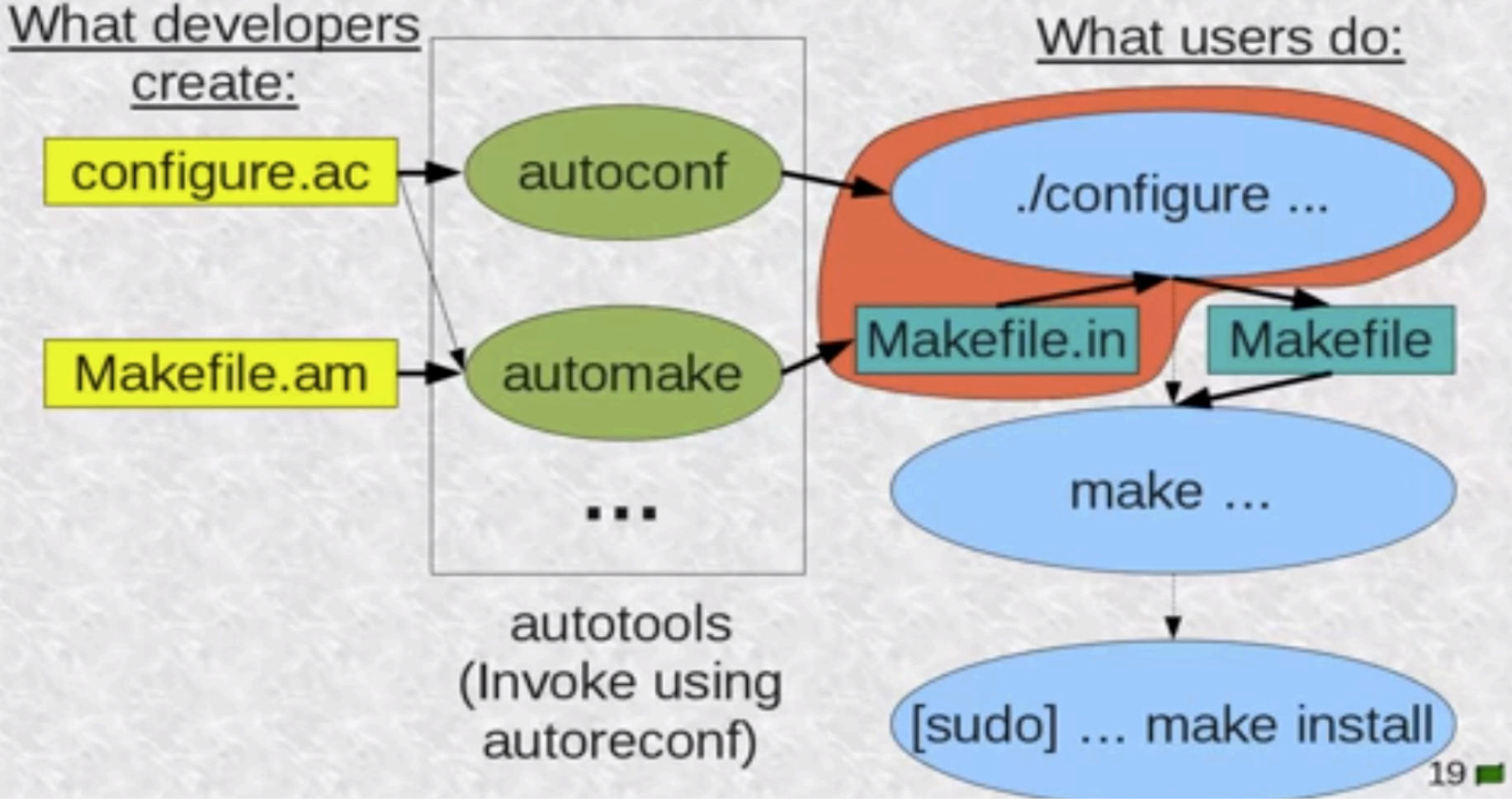
Some advantages when using GNU autotools

- The **installation** of a program is **straightforward**:

```
./configure; make; make install
```

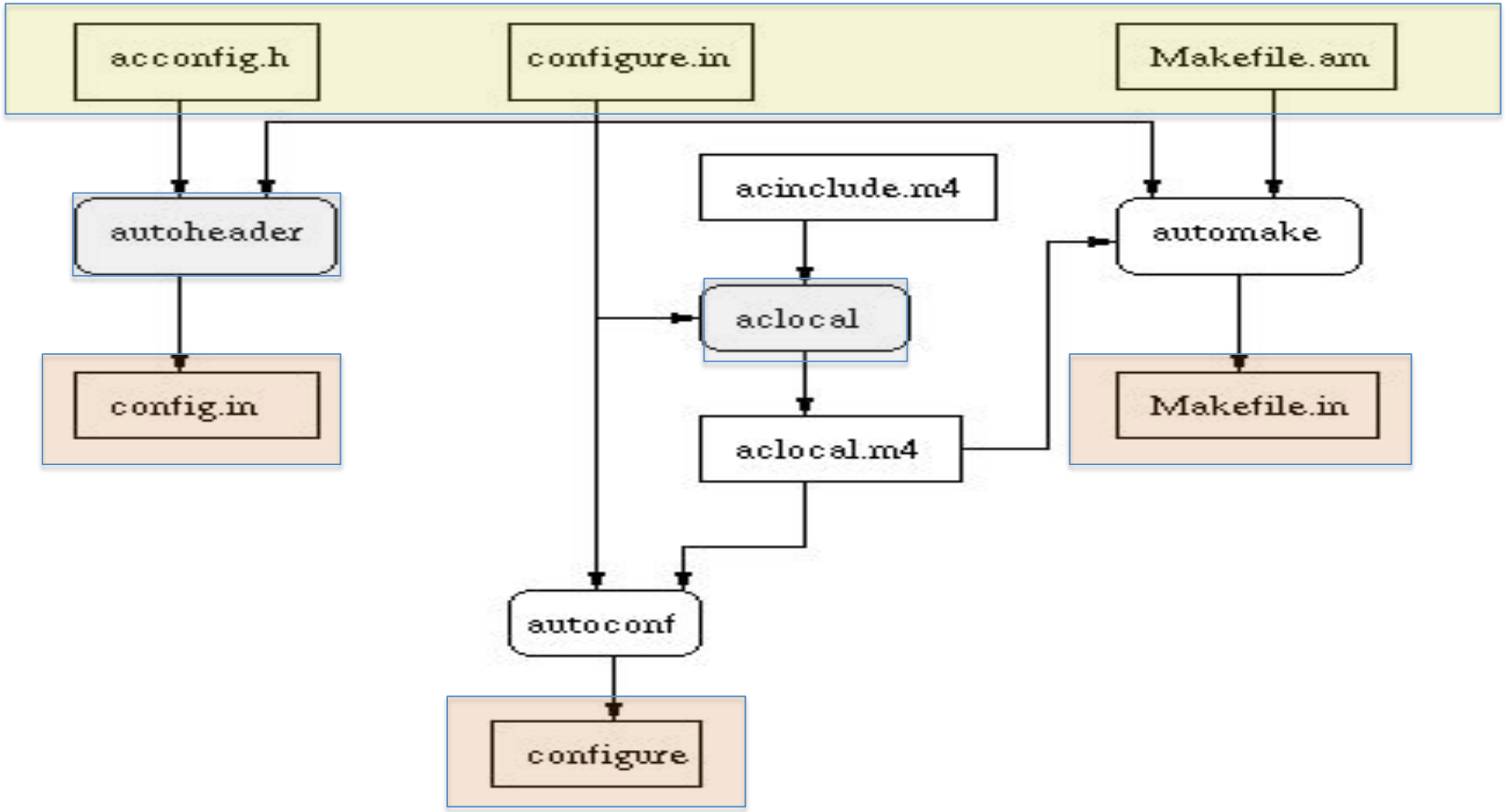
- When executing the configure script
  - **checks** for **system parameters**, **libraries**, **location of programs**, availability of functions …
  - and writes a **Makefile**
  - `./configure` supports many options to overwrite defaults settings (example --prefix=$HOME)

Recap: Autotools (simplified view)



David A. Wheeler

Developer: Created and generated files



GNU Build Tools (2)

- **addresses portability** by providing a way to check for conditions that may change from one system to another
  - header files, libraries, platform versions, . . .
- a special header file (config.h) is created for **inclusion into the sources**
  - it contains the gathered information on the system for use within your program

GNU Autoconf (1)

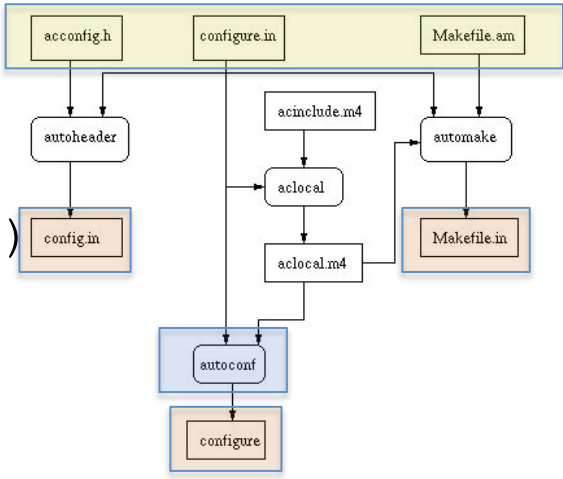
- used to **generate** a **configure** **script**
- **configure** script will
  - do the checking when used at **build time**
  - Process template files (**.in**) like **Makefile.in** to create a specific **Makefile**

Usage

1. create `configure.ac (configure.in (old))`

to specify what functionality is required

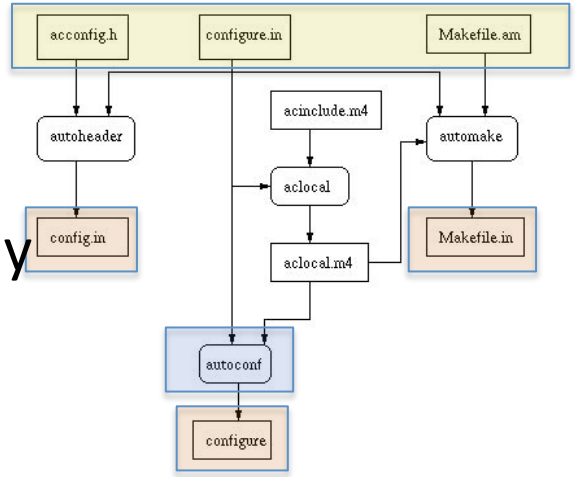
2. run `autoconf` to generate a configure script



GNU Autoconf (2)

Usage example

- not all systems support the `gettimeofday` function
- you want to use it when available, and some other function when not
- put `AC_CHECK_FUNCS(gettimeofday)` in configure.ac
  - if the function is available the (generated) configure script will arrange to define the preprocessor macro HAVE_GETTIMEOFDAY
  - otherwise it will not define the macro at all
- use `#ifdef`
  - to test whether it is safe to call gettimeofday from your code



GNU Autoconf (3)

Auxiliary programs part of Autoconf

- autoscan

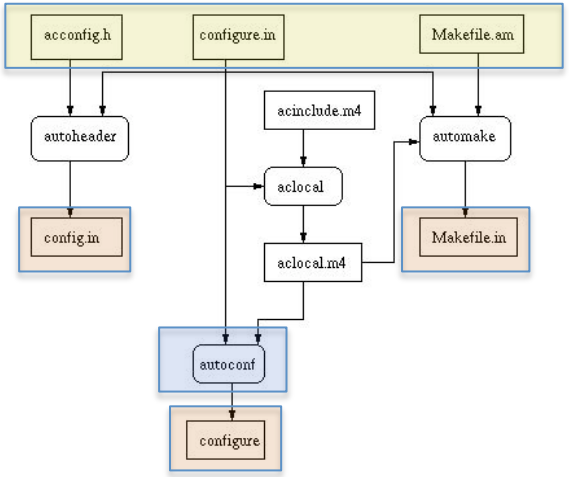
  create an initial input file for Autoconf (configue.scan), it will scan your sources to find possible portability problems

- autoheader

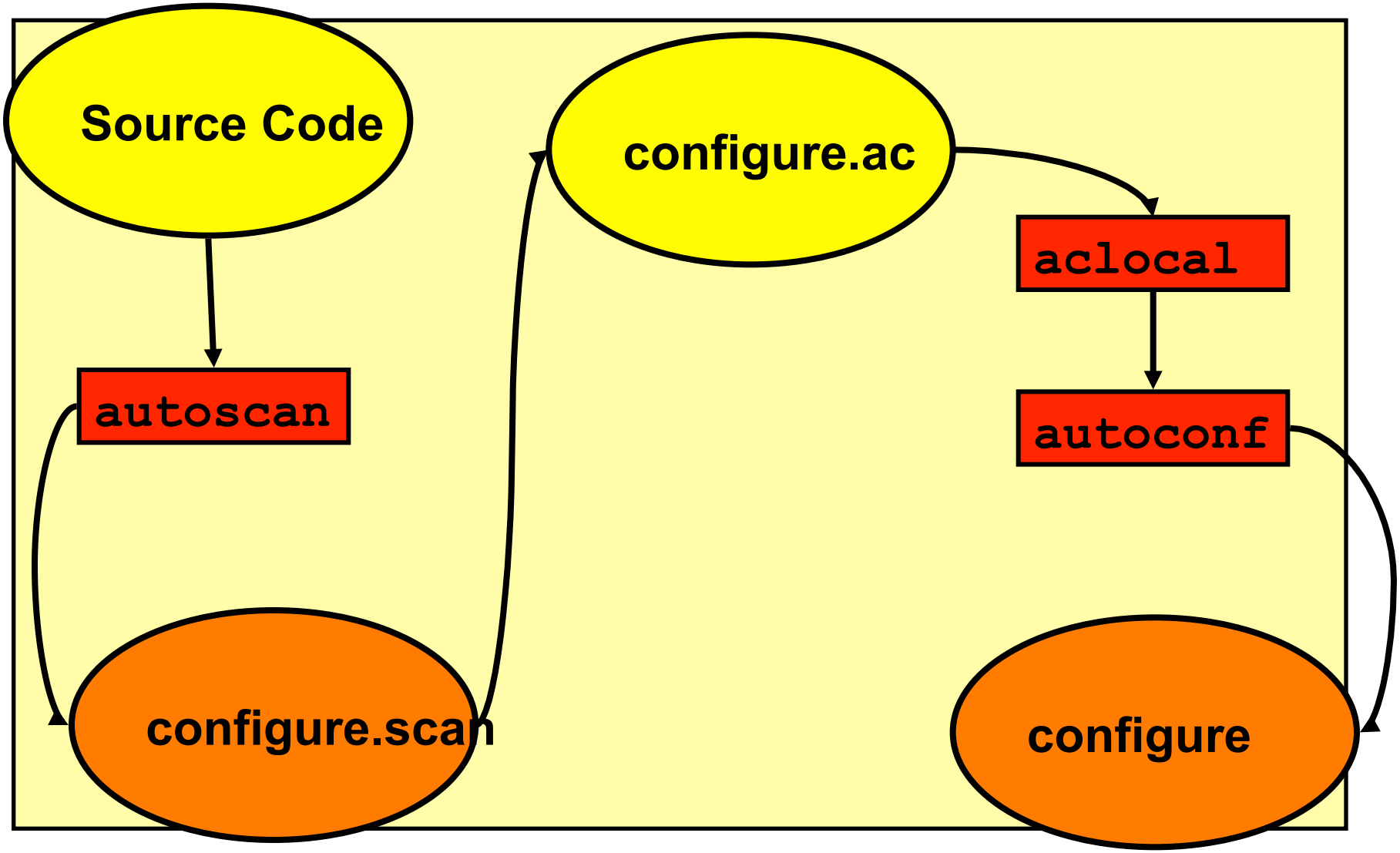
  manage C header files, will warn you if certain header requirements are not met (need to be added to acconfig.h)

- ifnames

  list C pre-processor identifiers already in use in your program



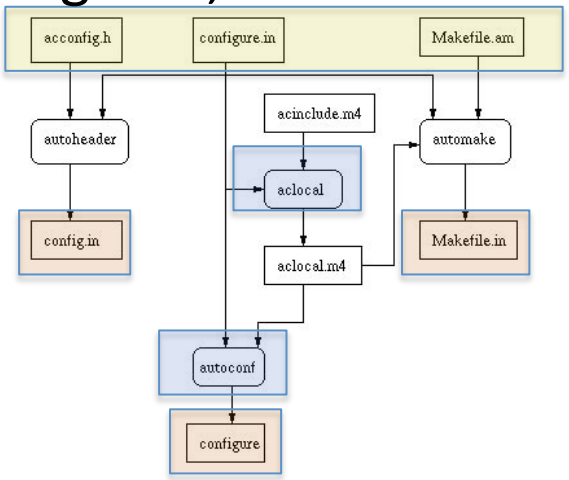
GNU Autoconf



Aclocal

- generates an **aclocal.m4** file getting information from **configure**.ac [needed] and **acinclude.m4** [optional].
  - **aclocal.m4:** contains the macro definitions

- **m4** macro processor : processes tokens (strings of letters and digits).
  - Reads token and determines if it is the name of a macro.
  - replaces the name of the macro with defining text,
  - pushes the resulting string onto the input to be rescanned



GNU Automake

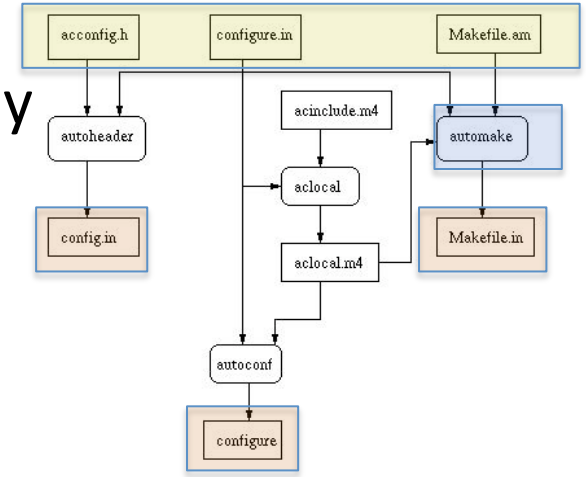
- helps to **create portable** Makefiles for use with the make utility
  →input **Makefile.am**, output **Makefile.in**

- **Makefile.in** is then used by the **configure** script to generate a Makefile

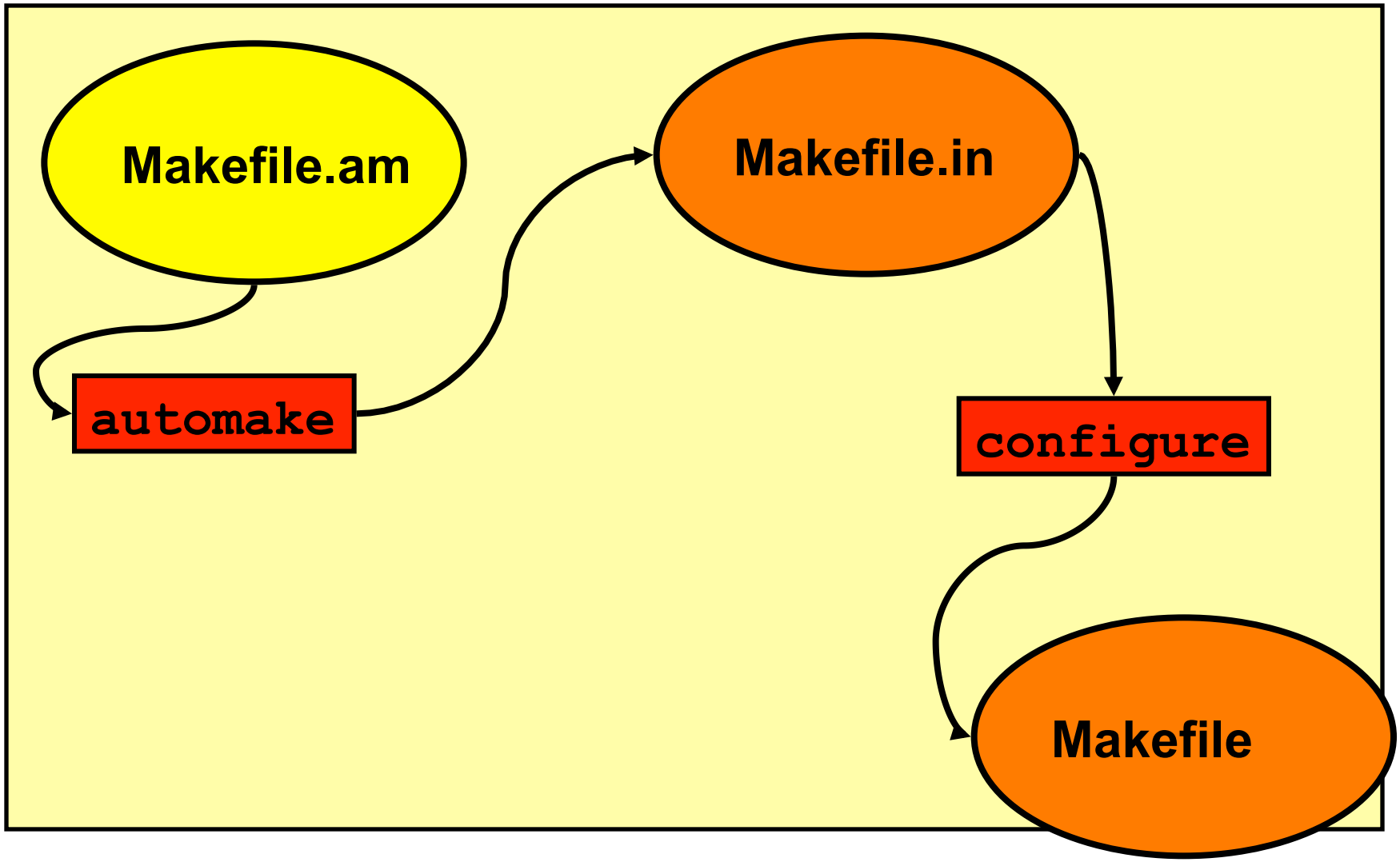
Usage example

- your package builds a program sne, simply add the following to the **Makefile.am** in the directory where the program is Built

```
bin_PROGRAMS = sne
sne_SOURCES = sne1 sne2 ...
```



GNU Automake



Example

foo.c :

```
#include <stdio.h>
 main()
 {
  printf("Cum grano salis\n");
 }
```

Makefile.am :

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

configure.ac :

```
AC_INIT(foo.c)
AM_INIT_AUTOMAKE(latin_words, 0.9)
AC_PROG_CC
AC_HEADER_STDC
AC_PROG_INSTALL
AC_OUTPUT([Makefile])
```

- **AC_INIT(sourcefile):**
  Initializes autoconf, should be the first macro called in configure.ac
- **AM_INIT_AUTOMAKE(latin_words, 0.9)**
  Runs many macros required for proper operation of the generated Makefiles
- **AC_PROG_CC**:
  Determines a C compiler to use, sets the CC variable, initilizes the CFLAGS
- **AC_HEADER_STDC** :
  Checks for stdlib.h, stdarg.h , string.h and float.h, defines STDC_HEADERS on success.
- **AC_PROG_INSTALL**
  Set variable INSTALL to the path of a BSD-compatible install program
- **AC_OUTPUT([Makefile])**
  Create output files in the case the Makefile.

autoheader

- Create a **template header** for **configure**

- Usage:
  - run **autoheader** on a directory with a **configure.ac** that contains a AC_CONFIG_HEADER macro call, and it will write the configue.in file

http://www.seul.org/docs/autotut/#autoheader

GNU Libtool

- helps **manage the creation** of **static** and **dynamic** libraries on various Unix-like operating systems
- hides the **complexity** of using shared libraries on **different Platforms**
- provides a **generic interface** for developers
  - clean Remove files from the build directory
  - compile Compile a source file into a libtool object.
  - Execute Automatically set the library path, then run a program.
  - finish Complete the installation of libtool libraries.
  - install Install libraries or executables.
  - link Create a library or an executable.
  - uninstall Remove libraries from an installed directory.

steps involved in **creating** a software

- create source
- create configure.ac
- create makefile.am
- aclocal
- autoconf
- automake -a //to add missing files

Directory Structure

- software name is software_01
- root of software is software_01
- software_01 will have

  lib/, src/, man/, doc/, makefile.am,
  configure.ac etc …

References

- http://en.wikipedia.org/wiki/Apollo_Guidance_Computer
- http://www.gnu.org/software/autoconf/
- http://www.gnu.org/software/libtool/libtool.html
- http://www.gnu.org/software/automake/automake.html
- http://www.gnu.org/software/m4/m4.html

Step 1: created source code

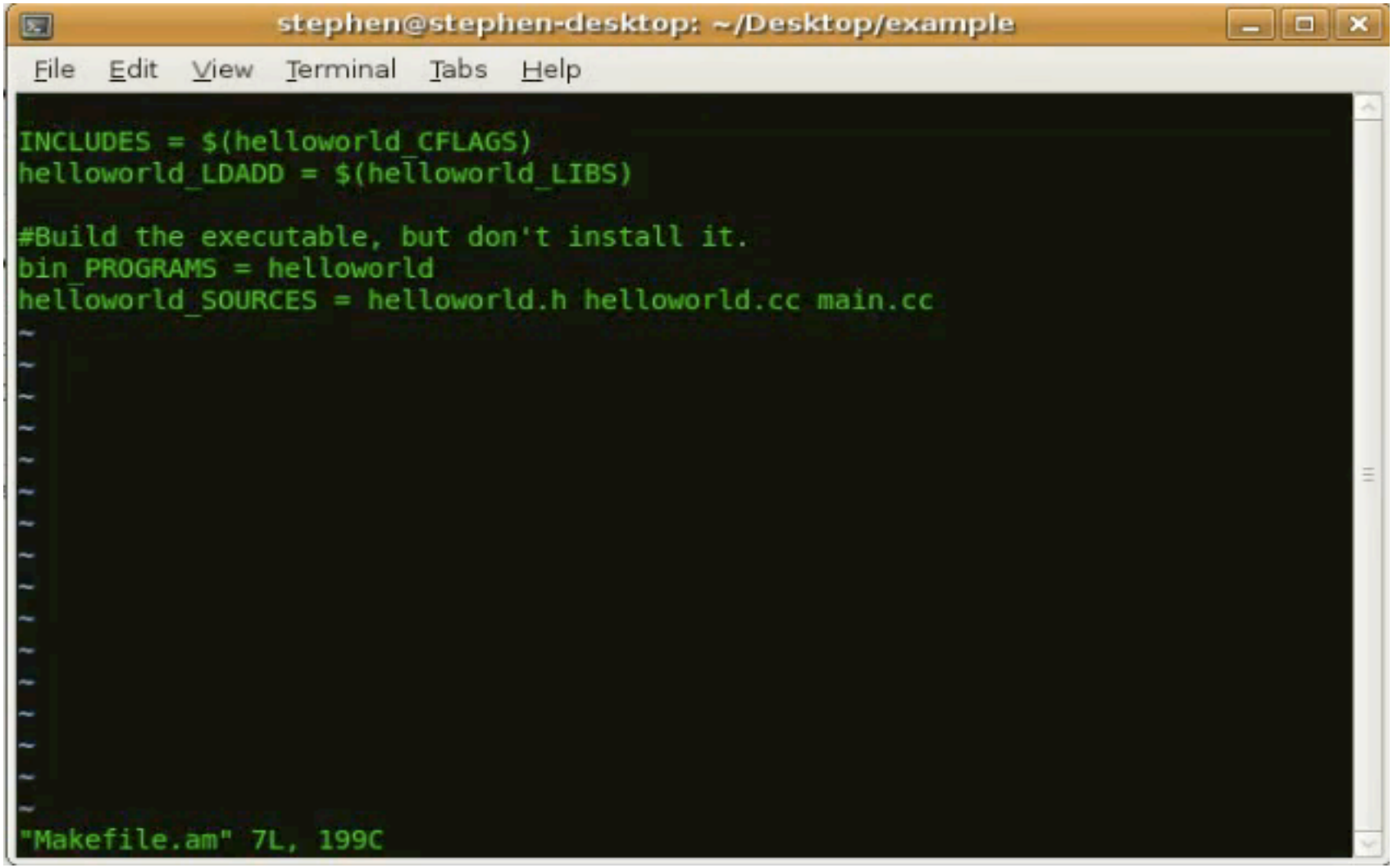
- Use the hello world example from the gtkmm tutorial
  - https://developer.gnome.org/gtkmm-tutorial/stable/
  - helloworld.h
  - main.cc
  - helloworld.cc

Step 1: created Makefile.am

```
INCLUDES = $(helloworld_CFLAGS)
helloworld_LDADD = $(helloworld_LIBS)

# Build the executable, but don't intall it
bin_PROGRAMS = helloworld
helloworld_SOURCES = helloworld.h helloworld.cc
   main.cc
```

Step 1: created Makefile.am



Step 2: run autoscan

Autoscan will create template file for the project
configure.scan

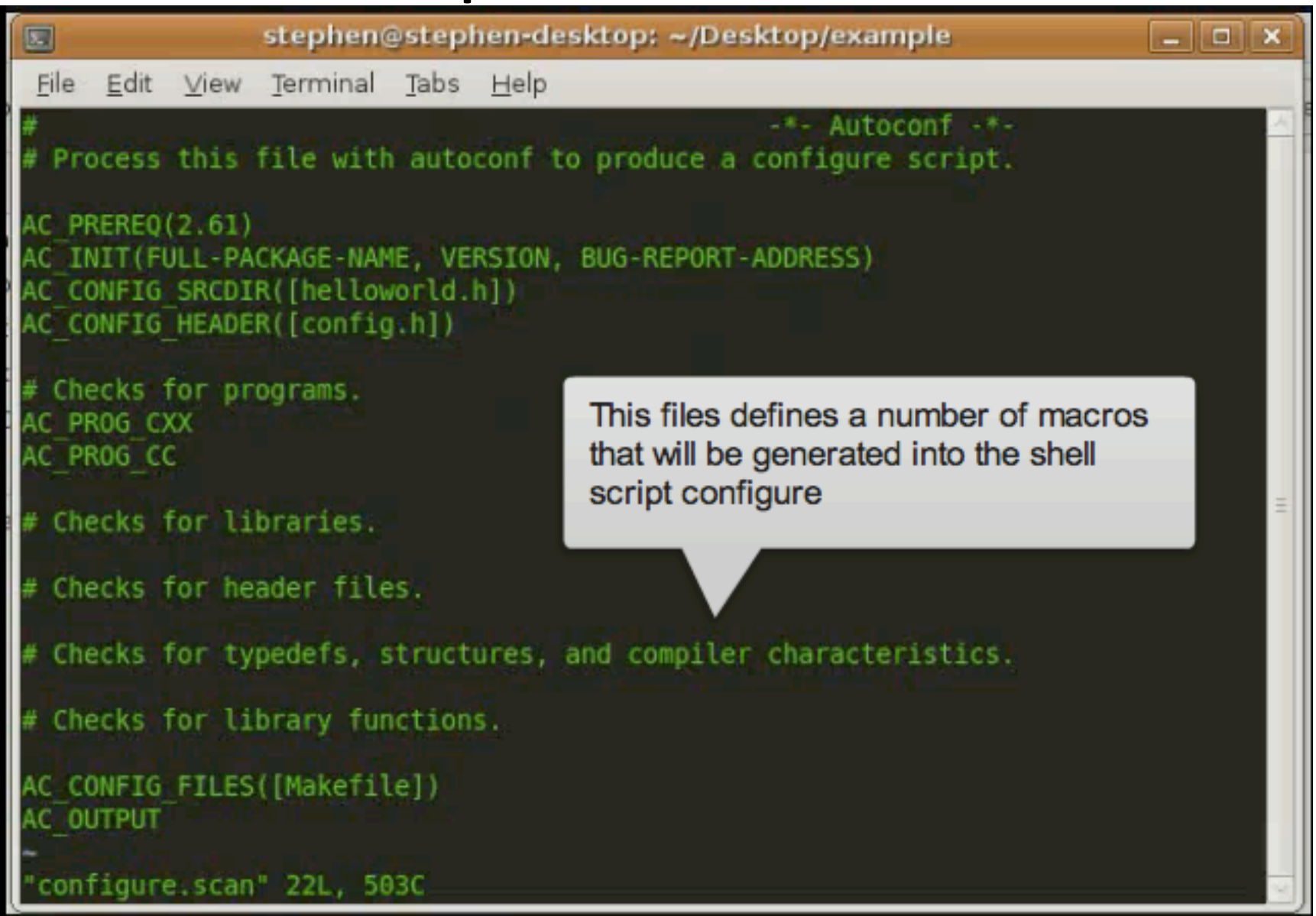
autoscan.log
configure.scan
Makefile.am
helloworld.cc
helloworld.h
main.cc

Step 2: run autoscan

Autoscan will create template file for the project
configure.scan

autoscan.log
configure.scan
Makefile.am
helloworld.cc
helloworld.h
main.cc

Step 2: run autoscan



Step 3: edit configure.scan

- to add Macros to initialize automake specifying the additional lib you are using and the position of the Makefile

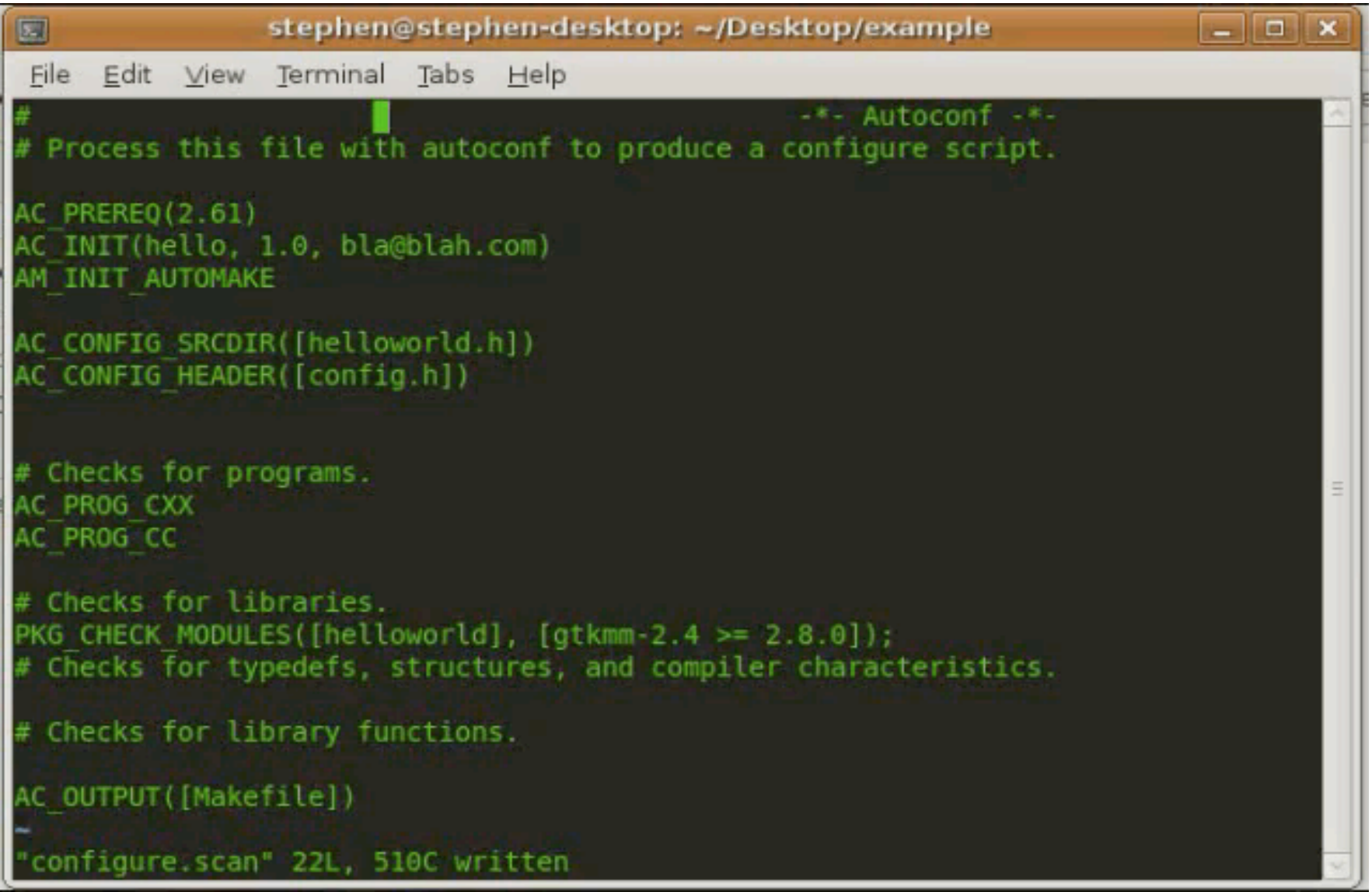
```
#                -*- Autoconfig -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([helloworld.cc])
AC_CONFIG_HEADER([config.h])
# check for Programs
AC_PROG_CXX
AC_PROG_CC
#Checks for libraries.
#Checks for typedefs, structure, and compiler characteristics
#Check for library functions.
AC_CONFIG_FILE([Makefile])
AC_OUTPUT
```

Step 3: edit configure.scan

- to add Macros to initialize automake specifying the additional lib you are using and the position of the Makefile

```
#                -*- Autoconfig -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2,61)
AC_INIT(hello, 1.0, bal@bah.com)
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([helloworld.h])
AC_CONFIG_HEADER([config.h])
# check for Programs
AC_PROG_CXX
AC_PROG_CC
#Checks for libraries.
PKG_CHECK_MODULES([helloworld], [gtkmm-2.4 >= 2.8.0]);
#Cheks for typedefs, structure, and compiler characteristics
#Check for library functions.
AC_OUTPUT([Makefile])
```

Step 3: edit configure.scan



Step 4: run autoheader

```
$ autoheader
$ ls -t
  automa4te.cache
  configure.h.in
  Makefile.am
  configure.ac
  autoscan.log
  configure.scan
  helloworld.cc
  helloworld.h
  main.cc
```

Step 5: run aclocal

```
$ aclocal
$ ll -t
  aclocal.m4
  automa4te.cache
  configure.h.in
   Makefile.am
   configure.ac
  autoscan.log
  configure.scan
  helloworld.cc
  helloworld.h
  main.cc
```

Step 6: run autoconf

```
$ autoconf
$ ls -t
  configure
  aclocal.m4
  automa4te.cache
  configure.h.in
  configure.ac
  autoscan.log
  configure.scan
  helloworld.cc
  helloworld.h
  main.cc
```

Step 7: automake

```
$ automake -ac
$ ls -t
  COPYING                    Makefile.am
  depcomp                    configure.ac
  INSTALL                    autoscan.log
  instakl-sh               configure.scan
  missing                    helloworld.cc
  Makefile.in                helloworld.h
  configure                  main.cc
  aclocal.m4
  automa4te.cache
  configure.h.in
```

Step 1: run ./configure (installing)

```
$ ./configure
$ ls -t
  Makefile
  mkinstalldirs
  confg.log
  config.status
  COPYING
  depcomp
  INSTALL
  instakl-sh
  missing
  configure
  aclocal.m4
  automa4te.cache
  configure.h.in
  Makefile.am
  configure.ac
  autoscan.log
  configure.scan
  helloworld.cc
  helloworld.h
  main.cc
```

Step 2: make (Installing)

```
$ make
$ ls -t
```

helloworld
helloworld.o
main.o
config.h
Makefile
Makefile.in
mkinstalldirs
confg.log
config.status
COPYING
depcomp

INSTALL
instakl-sh
missing
configure
aclocal.m4
automa4te.cache
configure.h.in
configure.ac
autoscan.log
configure.scan
Makefile.am
helloworld.cc
helloworld.h
main.cc

ANT:
Another Nice Tool

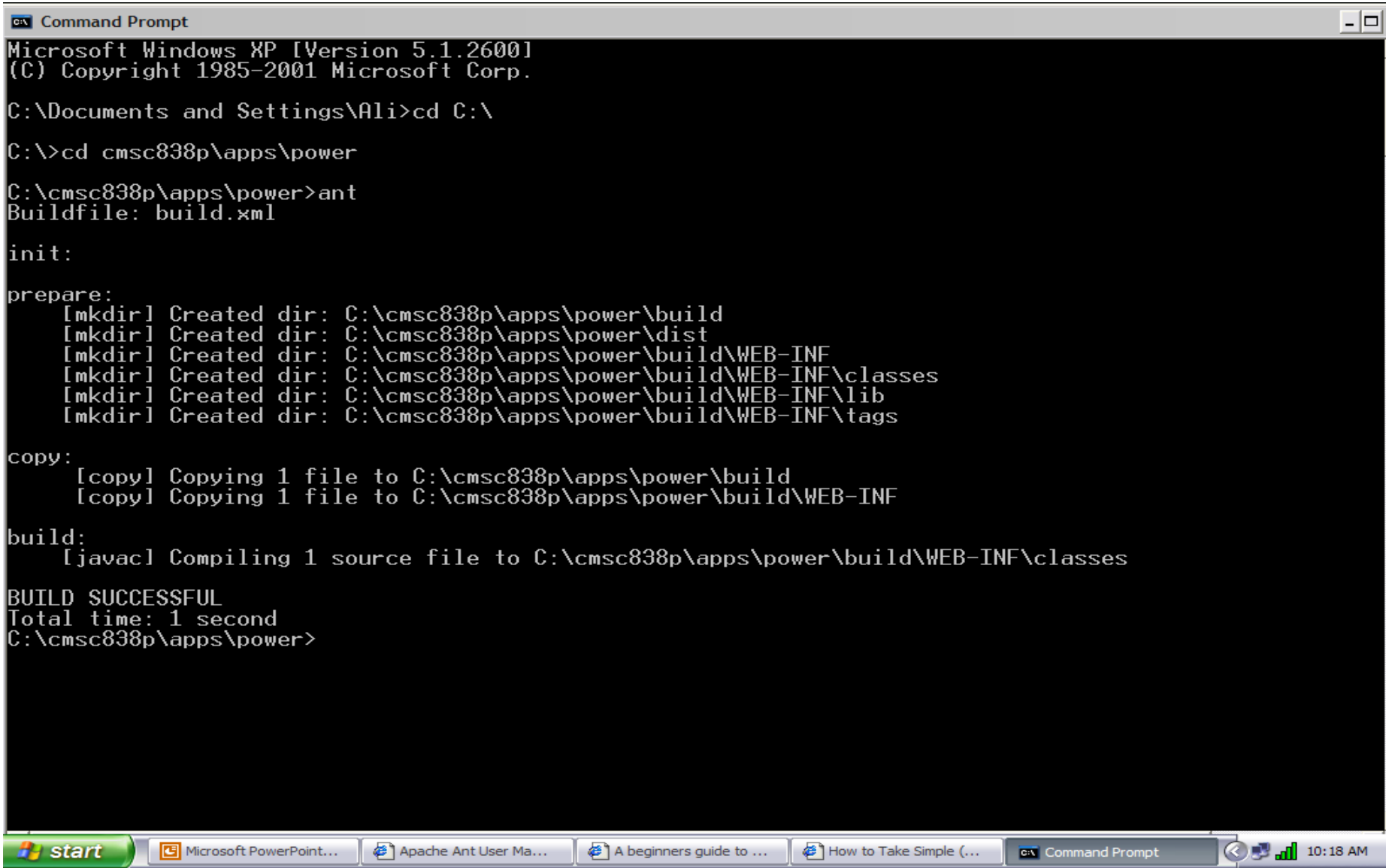
What is Ant?

- **Java**-based **Build** tool from **Apache**
- **De facto standard** for building, packaging, and installing Java applications
- Accomplishes same objectives that *make* does on Unix based systems
- Files are written in **XML**

Running Ant

- Type "**`ant`**" at the command line
- Automatically looks for **build.xml** file in current directory to run
- Type "`ant -buildfile buildfile.xml`" to specify another build file to run.

Ant Output


Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Ali>cd C:\
C:\>cd cmsc838p\apps\power
C:\cmsc838p\apps\power>ant
Buildfile: build.xml
init:
prepare:
[mkdir] Created dir: C:\cmsc838p\apps\power\build
[mkdir] Created dir: C:\cmsc838p\apps\power\dist
[mkdir] Created dir: C:\cmsc838p\apps\power\build\WEB-INF
[mkdir] Created dir: C:\cmsc838p\apps\power\build\WEB-INF\classes
[mkdir] Created dir: C:\cmsc838p\apps\power\build\WEB-INF\lib
[mkdir] Created dir: C:\cmsc838p\apps\power\build\WEB-INF\tags
copy:
[copy] Copying 1 file to C:\cmsc838p\apps\power\build
[copy] Copying 1 file to C:\cmsc838p\apps\power\build\WEB-INF
build:
[javac] Compiling 1 source file to C:\cmsc838p\apps\power\build\WEB-INF\classes
BUILD SUCCESSFUL
Total time: 1 second
C:\cmsc838p\apps\power>
start
Microsoft PowerPoint...
Apache Ant User Ma...
A beginners guide to ...
How to Take Simple (...
Command Prompt
10:18 AM

Sample build.xml

```
<project name="MyProject" default="dist" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init" >
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile" >
    <mkdir dir="${dist}/lib"/>
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean" >
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Ant Overview: Project

- Each **build file** contains exactly **one project** and at **least one target**
- Project tags specify the basic project attributes and have 3 properties:
  - name
  - default target
  - basedir
- Example: <project name="MyProject" default="build" basedir=".">

Ant Overview: Targets

- Target is a build module in Ant
- Each target contains **task(**s) for Ant to do
- One must be a project default
- Overall structure of targets:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>

<target name="D" depends="C,B,A"/>
```

Ant Overview: Tasks

- Each **target** comprises **one** or **more tasks**
- Task is a piece of executable Java code (e.g. javac, jar, etc)
- Tasks **do** the **actual** "**build**" **work** in **Ant**
- Ant has **core** (**built in**) **tasks** and the ability to create own tasks

Ant Overview: Tasks

Example:

```
<target name="prepare" depends="init" >
  <mkdir dir="${build}" />
</target>

<target name="build" depends="copy" >
  <javac srcdir="src" destdir="${build}">
     <include name="**/*.java" />
  </javac>
</target>
```

Ant Overview: Core Tasks

- javac – Runs the Java Compiler
- java – Runs the Java Virtual Machine
- jar (war) – Create JAR (GAR) files
- mkdir – Makes a directory
- copy – Copies files to specified location
- delete – Deletes specified files
- cvs – Invokes CVS commands from Ant

Ant Overview: Writing Own Task

- Create a Java class that extends

  `org.apache.tools.ant.Task`

- For each attribute, write a **setter** method public void and takes a single argument
- Write a public void execute() method, with **no arguments**, that throws a **BuildException**
  - this method implements the task itself

Ant Overview: Properties

- Special task for **setting up build file properties**:
- Example:

  <property name="src" value="/home/src"/>

  Can use ${src} in build file to denote /home/src

- Note:Ant provides access to **all system properties** as if defined by the <property> task

Ant Overview: Path Structures

- Ant provides means to set various **environment variables** like **PATH**, **CLASSPATH.**
- Example of setting CLASSPATH:

```
<classpath>
    <pathelement path="${classpath}"/>
    <pathelement location="lib/helper.jar"/>
</classpath>
```

Command Line Arguments

- -buildfile <*buildfile*>

  specify build file to use
- - *targetname*

  → specify target to run (instead of running default)
- -verbose, -quiet, -debug

  → Allows control over the logging information Ant outputs
- -logger <*classname*>

  →Allows user to specify own classes for logging Ant events

IDE Integration

- Eclipse, NetBeans, JBuilder, VisualAge, and almost any other Java IDE has **Ant integration built-in** to the system
- Refer to each IDE's documentation for how to use Ant with that IDE

Web Development with Ant

- Tomcat comes with special Ant tasks to ease Web application development and deployment
- Copy $TOMCAT_HOME/server/lib/catalina-ant.jar to $ANT_HOME/lib
- Ant tasks for Tomcat:
  - install
  - reload
  - deploy
  - remove