Tema 6 - Configuración y adaptación automática de proyectos (DCA)

HISTORIAL DE REVISIONES			
NÚMERO	FECHA	MODIFICACIONES	NOMBRE

Índice

1. Introducción (I) 1
2. Introducción (II) 1
3. Herramientas de configuración 1
4. Autotools en una imagen... 2
5. Historia de Autoconf. 2
6. Uso de Autoconf (I) 2
7. Uso de Autoconf (II) 3
8. Uso de Autoconf (III) 3
9. Autoconf. Macros habituales (I) 3
10. Autoconf. Macros habituales (II) 3
11. Autoconf. Macros habituales (III) 4
12. Autoconf. Macros habituales (IV) 4
13. Autoconf. Macros habituales (V) 4
14. Autoconf. Macros habituales (VI) 5
15. Autoconf. Macros habituales (VII) 5
16. Autoconf. Macros habituales (VIII) 5
17. Autoconf. Creación de Makefiles 5
18. Automake. Creación de Makefiles (I) 6
19. Automake. Creación de Makefiles (II) 6
20. Automake. Creación de Makefiles (III) 6
21. Autoconf. Imprescindibles... 7
22. Autoconf + Automake (I) 7
23. Autoconf + Automake (II) 7

24. Autoconf + Automake (III) 8
25. Autoconf + Automake (IV) 8
26. Autoconf. Hola Mundo... 8
27. Autoconf Archive 9
28. Cmake (I) 9
29. Cmake (II) 9
30. Cmake (III) 10
31. Cmake (IV) 10
32. CMake. Generadores. 11
33. Estructura de CMakeLists.txt (I) 11
34. Estructura de CMakeLists.txt (II) 11
35. Estructura de CMakeLists.txt (III) 12
36. Estructura de CMakeLists.txt (IV) 12
37. Estructura de CMakeLists.txt (V) 12
38. Estructura de CMakeLists.txt (VI) 13
39. Estructura de CMakeLists.txt (VII) 13
40. Estructura de CMakeLists.txt (VIII) 14
41. Cmake en proyectos reales: 14
42. Cmake: Gráficos de dependencias 15
43. Cmake: No olvides... 15
44. Meson (I) 16
45. Meson (II) 16
46. Prácticas en grupo: 16
47. Prácticas individuales: 16
48. Aclaraciones 17

Logo DLSI

Tema 6 - Configuración y adaptación automática de proyectos (DCA)
Curso 2018-2019

1. Introducción (I)

- Cualquier proyecto software debe crearse teniendo en mente:
  - Su portabilidad
  - La facilidad de reconstruirlo cada vez que sea necesario...
  - Incluso la facilidad de construirlo en un sistema diferente al que se ha usado en su desarrollo.
- No tiene sentido crear un proyecto software hoy en día pensando en que se ejecutará en una única plataforma.
- El hecho de que sea más fácil de reconstruir evitará posibles situaciones de error al ser recompilado.
- Las herramientas que permiten configurar y adaptar la construcción de un proyecto software lo hacen realizando unos tests y chequeos antes de proceder a la compilación.
- Según los resultados de estos tests podemos adaptar el proceso de compilación.

2. Introducción (II)

UN SISTEMA DE CONFIGURACIÓN AUTOMÁTICA DEBERÍA PODER:

- Buscar automáticamente los programas necesitados en el momento de la configuración.
- Poder llevar a cabo la construcción del proyecto *fuera* de los directorios de código fuente.
- Generar durante la fase de configuración nuevos archivos que se emplean durante la fase de construcción.
- Permitir seleccionar componentes opcionales (p.e. *bibliotecas*) en tiempo de configuración.
- Generar proyectos/soluciones/makefiles para que las herramientas del s.o. destino puedan construir el software una vez configurado. Si esto lo pueden hacer a partir de un sencillo archivo de texto... mejor.
- Permitir cambiar entre un modelo de generación de archivos binarios estáticos y dinámicos.
- Tener en cuenta las dependencias entre ficheros y soportar construcciones en paralelo.

3. Herramientas de configuración

En el mundo del *Software Libre* disponemos de varias herramientas de configuración y adaptación del proceso de compilación de un proyecto software.

En la asignatura nos vamos a centrar en dos (AutoTools y CMake) y comentaremos algo de pasada sobre Meson.

1. AutoTools, las cuales constan de tres componentes: Autoconf, Automake, Libtool.

Funcionan las tres en s.o. tipo `unix`, y en Windows bajo el paraguas que proporciona `cygwin`.

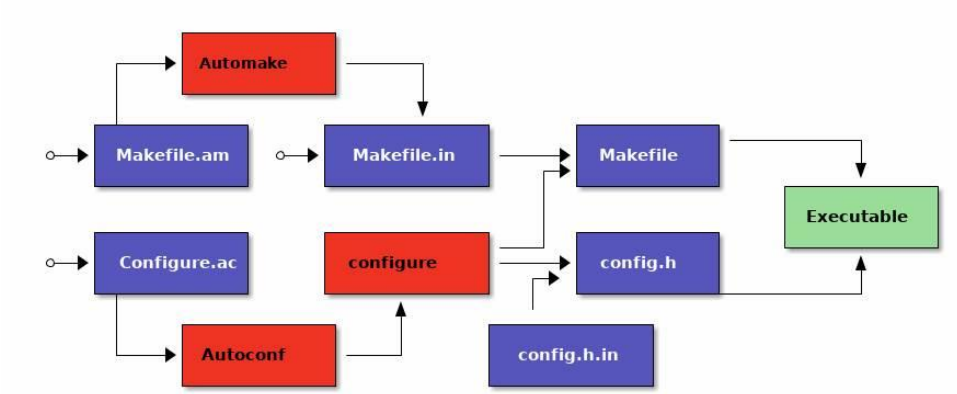
1. CMake. Además dispone de una serie de herramientas adicionales, como son CPack, CTest y CDash.

PERO PUEDES CONSULTAR OTRAS:

1. Por ejemplo en esta comparativa/resumen con SCons.
2. O más recientes como por ejemplo Meson build system.

4. Autotools en una imagen...

- Es muy fácil perderse entre todas las herramientas que componen las AutoTools...
- Por eso ten presente siempre esta imagen... te ayudará a entender el proceso.



5. Historia de Autoconf.

- Aparece en el año 1991, creado por David MacKenzie.
- Produce guiones de shell para sistemas compatibles Posix.
- Los guiones que produce no requieren tener instalado `autoconf` cuando son ejecutados. Son independientes de `autoconf`.

6. Uso de Autoconf (I)

- El guión **`configure`** se crea a partir de un fichero llamado **`configure.ac`**.
- Tendremos un fichero **`configure.ac`** en el directorio principal de nuestro proyecto.
- La estructura básica de un **`configure.ac`** sería esta:

```
# Process this file with autoconf to produce a configure script.
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/hola.cc])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CXX
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

7. Uso de Autoconf (II)

CONFIGURE.AC CONSTA DE:

- **Comentarios**: Todo lo que sigue a un **#** o también a **dnl**.
- **Macros**: Escritas en mayúsculas. Pueden ser de `autoconf` o nuestras. Las de `autoconf` comienzan por **AC_** , por ejemplo **AC_CONFIG_HEADER**. Las de *autoconf archive* suelen comenzar por **AX_** .
- **Código de shell**: Escrito diréctamente en el fichero, se queda en el fichero `configure` tal y como lo hemos escrito.

8. Uso de Autoconf (III)

- Una vez tenemos `configure.ac` ejecutamos **autoconf** para generar el fichero `configure`.
- `configure` realiza las comprobaciones pedidas en `configure.ac` y finalmente...
- Genera los ficheros pedidos en las macros **AC_CONFIG_FILES([])** seguida de **AC_OUTPUT**

9. Autoconf. Macros habituales (I)

- **AC_INIT(full-package-name, version, bug-report-address)**: Inicia autoconf seguido de **AC_CONFIG_SRCDIR([unique-file-in-source-dir])**.
- **AC_CONFIG_AUX_DIR(dir)**: Crea un directorio para instalar ficheros auxiliares necesitados por `autoconf`.
- **AC_CONFIG_FILES([_idem_])**: seguido de **AC_OUTPUT** genera los ficheros de salida.
- **AC_PROG_MAKE_SET**: Comprueba si `make` predefine la variable `MAKE`. Si lo hace `SET_MAKE` se define a vacío, si no `SET_MAKE` contiene `MAKE=make`.

10. Autoconf. Macros habituales (II)

- **AC_CONFIG_HEADER (header-to-create ...)**: Crea el o los ficheros de cabecera que recibe como parámetros y define el símbolo **HAVE_CONFIG_H**. Este fichero contiene sentencias "`#define`" resultado de los tests, además de sustituciones de símbolos de la forma "`@DEFS@`".
- **AC_DEFINE(variable, [value], [description])**: Define un símbolo del preprocesador. Este símbolo se deposita en el fichero "`config.h`" si lo usamos, o se pasa directamente al compilador con la opción "`-D`*símbolo*".

- **AC_SUBST(variable, [value])**: Define una variable de salida a partir de una variable de shell, la cual será sustituída en los ficheros de salida, normalmente "`Makefiles`". La sustitución se realizará al transformar un fichero "`Makefile.in`" en "`Makefile`". Las variables en "`Makefile.in`" tienen la forma **@variable@**.

11. Autoconf. Macros habituales (III)

- **AC_PROG_CC**: Determina el compilador de `C` a usar y lo guarda en la variable "`CC`".
- **AC_PROG_CXX**: Idem pero con el compilador de `C++`, y lo guarda en la variable "`CXX`".
- **AC_CHECK_PROG(variable, prog-needed, val-if-found [,val-if-not-found [,path]])**: Busca un programa en los directorios especificados por la variable "`PATH`" y deposita en **variable** el camino completo hasta él si lo encuentra.

12. Autoconf. Macros habituales (IV)

- **AC_LANG(language)**: Los tests de compilación realizados se harán en el lenguaje elegido: `C`, `C++`, etc...
- **AC_TRY_COMPILE(includes,func-body,[act-compile],[act-not-compile])**: Intenta compilar (y no enlazar) en el lenguaje actual elegido la función "*func-body*", si lo consigue ejecuta las acciones "*act-compile*", y si no...
- **AC_TRY_LINK(includes,func-body,[act-link],[act-not-link])**: Idem al anterior, pero ahora mira si "*func-body*" se *enlaza* bien y en ese caso ejecuta "*act-link*" y si no ejecuta...

13. Autoconf. Macros habituales (V)

```
AC_TRY_COMPILE(
   [#include <iostream>],[int i = static_cast<int>(2.5);],
   [AC_MSG_RESULT([si])],
   [
    AC_MSG_ERROR([Lo siento, necesitas un compilador de C++ mejor.])
   ]
 )
AC_TRY_LINK(
   [#include <vector>
   template <class T>
   class Matriz
   {
   private:
     std::vector< std::vector<T> > la_matriz;
   public:
     Matriz(int);
   };

   template <class T>
   Matriz<T>::Matriz(int h) : la_matriz(h) {}
   ],

   [Matriz<int> me = Matriz<int>(10);],
   [AC_MSG_RESULT([si])], [AC_MSG_ERROR([Lo siento, necesitas un compilador de C++ mejor
       .])]
 )
```

14. Autoconf. Macros habituales (VI)

- **`AC_DEFUN(macro-name, macro-body)`**: Permite que definamos una macro nueva. Esta macro la podemos poner en `configure.ac`. De este modo podemos agrupar bajo un único nombre un conjunto de comprobaciones y así emplearlo de forma sencilla.
- Disponemos de un `repositorio` oficial de macros de terceros.

```
# Comprobamos las nuevas conversiones de tipo de C++
AC_DEFUN(DCA_CXX_NEW_CASTS,
[
AC_MSG_CHECKING(Si el compilador de C++ tiene las nuevas conversiones de tipo)
AC_TRY_COMPILE([],[int i = static_cast<int>(2.5);],
                [AC_MSG_RESULT([si])],
                [AC_MSG_ERROR([Lo siento, necesitas un compilador de \
                               C++ mejor.])])
])
```

15. Autoconf. Macros habituales (VII)

- **`AC_ARG_ENABLE(feat,help-string,[act-if-given],[act-not-given])`**: Si el usuario llama a `configure` con las opciones "`--enable-feat`"" o "`--disable-feat`" se ejecuta "`[act-if-given]`" o...

```
AC_ARG_ENABLE(g47, [--enable-g47     Usa la version 4.7 de g++.])
if test $enable_g47; then
  CXX=g++-4.7
else
  CXX=g++
fi
```

16. Autoconf. Macros habituales (VIII)

- **`AC_CHECK_FILE(file,[act-found],[act-not-found])`**: Comprueba si existe `file` y si es así ejecuta `act-found` y en otro caso ejecuta...
- **`AC_CHECK_LIB(lib,func,[act-found], [act-not-found], [other-libs])`**: Comprueba si la biblioteca "`lib`" contiene la función "`func`". "`lib`" es el nombre base de la biblioteca, por ejemplo, la matemática "`libm`" se especificaría como sólo como "`m`".

17. Autoconf. Creación de Makefiles

- Crearemos un fichero "`Makefile.in`" en el directorio raíz del proyecto...
- Y un fichero "`Makefile.in`" por cada subdirectorio del proyecto.
- Estos ficheros son leídos por "`autoconf`" como una plantilla -al ser especificados en la macro `AC_OUTPUT`- y se reescriben con el nombre "`Makefile`" *casi sin tocar*.
- Sólo se realizan en ellos sustituciones de símbolos de la forma "`@DEFS@`" que hubiéramos especificado en `configure.ac` mediante la macro "`AC_SUBST`".

18. Automake. Creación de Makefiles (I)

- Autoconf *sólo* configura... seguimos teniendo que crear los *Makefile* a mano.
- Además, los *Makefile* de distintos proyectos se parecen mucho... podríamos tener un "*esqueleto*" fijo y añadir lo específico de cada caso... la verdad es que no somos los primeros en pensar en esto, de hecho...
- ...Podemos emplear otra aplicación de la familia de las *autotools* para generar los "`Makefile.in`", se llama "`automake`".
- El uso de "`automake`" junto con "`autoconf`" requiere hacer algún cambio en "`configure.ac`" (`AM_INIT_AUTOMAKE`).
- Los archivos de entrada de "`automake`" son los llamados "**`Makefile.am`**". Cuando usamos `automake` debemos escribir ficheros "`Makefile.am`" en lugar de los "`Makefile.in`".

19. Automake. Creación de Makefiles (II)

- `Automake` busca o lee archivos `Makefile.am`. Son muy parecidos a un `Makefile`.
- Por defecto `Automake` crea proyectos al estilo GNU es decir, son obligatorios los archivos: `NEWS README AUTHORS ChangeLog`.
- Se pueden crear así: **`touch NEWS README AUTHORS ChangeLog`**.
- Se puede hacer que los ficheros `Makefile` que genera `automake` no emitan *demasiado ruido*, para ello empleamos la macro `AM_SILENT_RULES([yes])` en `configure.ac`. Los efectos de esta macro se pueden revertir temporalmente al llamar a `make` así:

```
make V=1
```

20. Automake. Creación de Makefiles (III)

- ¿Cómo le decimos a `automake` cuáles son nuestros ficheros de código fuente? Aquí interviene un concepto, el de *PRIMARIOS* (**primaries**):

```
# Makefile.am
bin_PROGRAMS = hw
hw_SOURCES = hw.cc main.cc
hw_LDADD =
hw_CFLAGS =
hw_CXXFLAGS =
```

- Si tuvieramos varios subdirectorios con `Makefile.am` en ellos:

```
SUBDIRS = src app gui
```

21. Autoconf. Imprescindibles...

autoscan

- Examina el directorio actual y crea un archivo "`configure.scan`" que nos sirve como punto de partida.

autoreconf

- Ejecuta `autoconf`, `autoheader`, `aclocal`, `automake`, `libtoolize`, y `gettextize` cuando corresponde.

autoupdate

- Ejecútalo cada cierto tiempo, comprobará si tu `configure.ac` necesita algún cambio con respecto a la versión de `autoconf` que tengas instalada actualmente.

22. Autoconf + Automake (I)

- Necesitamos poner en "`configure.ac`" la macro **AM_INIT_AUTOMAKE**.
- En resumen, una vez creados los "`Makefile.am`" necesarios, el proceso es:

```
autoscan                        - Edit configure.scan and save as configure.ac
touch NEWS README
touch AUTHORS ChangeLog

autoreconf --force --install    - runs aclocal, autoconf, autoheader and ↩
   automake
                                in the right order to create config.h.in,
                                Makefile.in, configure and a number
                                of auxiliary files

./configure                     - creates Makefile from Makefile.in and config. ↩
   h
                                from config.h.in

make
```

Un tutorial mucho más detallado Consúltalo en la web de Alexandre Duret-Lutz.

23. Autoconf + Automake (II)

AUTOMAKE NO RECURSIVO

- Trabajar con un proyecto cuyo código se divide en varios directorios... hace que en cada directorio acabemos teniendo un "`Makefile.am`"...
- Esto es lo que se conoce como un uso de *Automake* y de *make* recursivo...
- Algunos usuarios de *Automake* no recomiendan el uso recursivo del mismo. Es por eso que desde hace unas cuantas versiones *Automake* soporta construir un proyecto de manera no recursiva.
- A grandes rasgos basta con iniciar `automake` en `configure.ac` así:


```
AM_INIT_AUTOMAKE([subdir-objects])
```

Y tener sólo un `Makefile.am` en el directorio principal, en el cual, al indicar los nombres de los ficheros con código fuente especificamos su ruta, p.e.:

```
hw_SOURCES = src/hw.c
```

24. Autoconf + Automake (III)

automake puede iniciarse de otras formas desde configure.ac, por ejemplo:

```
AM_INIT_AUTOMAKE([no-dist-gzip dist-xz foreign])
```

no-dist-gzip
: No comprime con `gzip` el fichero `tar` de la distribución.

dist-xz
: Comprime con `xz` el fichero `tar` de la distribución.

foreign
: No requiere la existencia de los ficheros `NEWS`, `AUTHORS`, etc...

Y no son las únicas opciones, hay más. Consulta la documentación para conocerlas.

25. Autoconf + Automake (IV)

- `Automake` genera unos ficheros `Makefile` muy completos, tanto que tienen en cuenta incluso dependencias de los ficheros de configuración, es decir, al teclear `make` comprueban si se debe ejecutar `configure` previamente...y en ese caso, lo hacen.
- Estos `Makefiles` incluyen *por defecto* una serie de objetivos muy útiles, por ejemplo (y no son las únicas):

  dist
  : Genera el archivo `tar` de la distribución de nuestro proyecto.

  clean
  : Borrar la mayoría de los archivos intermedios generados. Para indicarle a `automake` qué otros archivos debe borrar, consulta en la documentación para qué sirven las variables de `automake` *CLEANFILES* y *DISTCLEANFILES*.

  distcheck
  : Genera el archivo `tar` de la distribución de nuestro proyecto, lo expande en un directorio temporal y trata de configurarlo. Si la configuración funciona, entonces ejecuta `make` para ver si puede compilarlo y no se nos ha olvidado incluir ningún fichero necesario.

26. Autoconf. Hola Mundo...

SHOW TIME...

- Preparamos una demo desde cero...

1. Primero todo el código en un solo directorio.
2. La modificamos para crear un subdirectorio para el código.

- Uso de la variable **EXTRA_DIST** y objetivo "`dist`" (*make dist*).
- Ejecución de "`make`" en modo "*verboso*": `make V=1`
- Observad la definición: "`-DHAVE_CONFIG_H`"

```
#if HAVE_CONFIG_H
    #include <config.h>
#endif
```

27. Autoconf Archive

¿Se puede contribuir a Autoconf? ¡Desde luego!, echa un vistazo a Autoconf Archive:

importante

"The GNU Autoconf Archive is a collection of more than 450 macros for GNU Autoconf that have been contributed as free software by friendly supporters of the cause from all over the Internet."

28. Cmake (I)

- Comienza a desarrollarse en 1999, como parte del *Insight Toolkit*.
- Posteriormente ha sido adoptado por otros proyectos software (KDE, Awesome Window Manager, etc...).
- Soporta el paso de `tests` mediante el componente CTest.
- Soporta la distribución sencilla del software mediante el componente CPack.
- Soporta la generación de los ficheros necesarios para construir el software configurado en plataformas "`*nix`" (**Makefile**s), `Windows` (Visual Studio) y `OS X` (XCode).
- Soporta *compilación cruzada*, bien para otros S.O., bien para dispositivos embebidos.
- Se comporta como *Autoconf* + *Automake* juntos.
- Su documentación.

29. Cmake (II)

- La descripción de un proyecto para Cmake se hace en ficheros de texto.
- Estos ficheros se llaman "`CMakeLists.txt`".
- Solemos tener uno en el directorio principal de nuestro proyecto y uno por cada subdirectorio del mismo donde haya que construir algo.

- Por ejemplo...nuestro proyecto `estrella` (ya visto con *Autotools*) "Hola Mundo", en su versión en la que el código está en el directorio principal, quedaría así:

```
# CMakeLists.txt
project (hw)
add_executable ( hw hw.cc )
```

SENCILLO...¿NO?

- ¿ Ves porqué hemos dicho que se parece a *Autoconf* + *Automake* ? No hay un *"`configure.ac`"* por un lado y un *"`Makefile.am`"* por otro...**sólo "`CMakeLists.txt`"**.

30. Cmake (III)

Un CMakeLists.txt un poco más elaborado...

```
# CMakeLists.txt
project (hw)

set (HW_SRCS hw.cc main.cc another.cc)

if (WIN32)
    set (HW_SRCS ${HW_SRCS} WinStuff.cc)
else (WIN32)
    set (HW_SRCS ${HW_SRCS} UnixStuff.cc)
endif (WIN32)

add_executable ( hw ${HW_SRCS} )
```

31. Cmake (IV)

¿CÓMO GENERAMOS EL MAKEFILE EN EL CASO DE S.O.'S UNIX?

- En el caso más sencillo, estando en el directorio principal del proyecto ejecutamos:

```
cmake .
```

- Este tipo de generación del proyecto es lo que `cmake` denomina "*in-source build* ".
- `cmake` soporta "*out-source builds* ", estando en el directorio principal del proyecto ejecutamos:

```
mkdir -p build && cd build
cmake ..
```

32. CMake. Generadores.

- CMake no solo puede generar Makefiles...
- CMake entiende el concepto de *generador*
- Un *generador* es el tipo de archivo de construcción del software que va a producir.
- Los generadores soportados son estos:

Visual Studio	**Borland Makefiles**	**NMake Makefiles**
Watcom WMake	**MSYS Makefiles**	**Unix Makefiles**
Ninja	**CodeBlocks**	**Eclipse CDT4**
KDevelop3	**Sublime Text 2**	

33. Estructura de CMakeLists.txt (I)

- Comentarios: **#**
- **CMakeLists.txt** del directorio principal debe tener obligatoriamente:

```
project ( projectname [C][CXX][Java][NONE])
```

- Opcionalmente alguna o varias de estas acciones:

```
1       include (file)
2       add_executable (exename sources)
3       add_library (libname [SHARED] sources)
4       target_link_libraries (exename libname)
5       add_subdirectory (dirname)
6       set ( VARIABLE value )
7       add_custom_target ( Name [ALL] [command1 [args1...]]
8                                [COMMAND command2 [args2...] ...]
9                                [DEPENDS depend depend depend ... ]
10                               [SOURCES src1 [src2...]] )
```

34. Estructura de CMakeLists.txt (II)

Flujo de control:

```
1       if (variable)
2     # cierta: si la variable es no: vacia, 0, FALSE, OFF, NOTFOUND
3   else (variable)
4     # falsa
5   endif (variable)
```

Iteración:

```
1       foreach (f v1 v2 v3 ... vn)
2          # actions for each value of 'f'
3       endforeach (f)
```


35. Estructura de CMakeLists.txt (III)

Condiciones:

```
1     NOT variable, var1 AND var2, var1 OR var2
2
3     DEFINED var
4
5     EXISTS file-name, EXISTS dir-name
6
7     IS_DIRECTORY name
8
9     variable EQUAL/LESS/GREATER number
10
11    string EQUAL/LESS/GREATER number
12
13    variable STREQUAL/STRLESS/STRGREATER number
14
15    string STREQUAL/STRLESS/STRGREATER number
```

Entorno: Mediante el uso de `ENV`: `$ENV{USER}`, `$ENV{HOME}`, etc...

36. Estructura de CMakeLists.txt (IV)

SOPORTE DE MÓDULOS:

- Mediante el uso de "`find_package`":

```
find_package(<name>)
```

- Busca en los directorios estándar de módulos uno llamado "*name* " y define las siguientes variables:

```
1       <name>_FOUND       # true iff the package was found
2
3       <name>_INCLUDE_DIR # a list of directories containing the
4                               # package's include files
5
6       <name>_LIBRARIES         # a list of directories containing the
7                               # package's libraries
8
9       <name>_LINK_DIRECTORIES  # the list of the package's libraries
```

37. Estructura de CMakeLists.txt (V)

Opciones y mensajes:

```
option (DEBUG "Generate output suitable for debugging" ON)
message ("DEBUG=[${DEBUG}]")
if (DEBUG)
        set (CMAKE_BUILD_TYPE "Debug")
        message ("Compiling for debugging.")
        # Podriamos anyadir definiciones para el compilador, p.e.:
        add_definitions ( -DDEBUG_BUILD )
else (DEBUG)
        set (CMAKE_BUILD_TYPE "Release")
        message ("Compiling for releasing.")
endif (DEBUG)
```

- En este ejemplo vemos también el soporte para la generación del proyecto en modo *DEBUG* y en modo *RELEASE*.

Acceso al sistema de ficheros:

```
file (GLOB backups *~ *.i *.s *.html)
```

La variable *backups* contendría una lista de los archivos existentes cuyo nombre coincida con los patrones indicados.

38. Estructura de CMakeLists.txt (VI)

Instalación: Soporta la instalación de varios tipos de *elementos*.

- De *objetivos* creados al compilar:

```
install (TARGETS my-app DESTINATION bin)
```

- De ficheros en general:

```
install (FILES version.h DESTINATION include RENAME pkg-version.h)
```

- De directorios completos:

```
install (DIRECTORY data/icons DESTINATION share/my-project)
```

39. Estructura de CMakeLists.txt (VII)

SELECCIÓN DE UN GENERADOR

- Se hace por nombre y con la opción **-G**, por ejemplo:

```
cmake -G "CodeBlocks - Ninja"
cmake -G "Unix Makefiles"
cmake -G "Eclipse CDT4 - Unix Makefiles"
```


- Los generadores disponibles están en la página de manual:

```
man cmake  # Aqui buscamos la seccion: GENERATORS
```

En versiones recientes los buscaremos en esta otra página.

```
man cmake-generators
```

- O también ejecutando:

```
cmake --help
```

- Al cambiar de generador es posible que nos avise de borrar la *cache*.

40. Estructura de CMakeLists.txt (VIII)

Archivo "config.h"

```
# CMakeLists.txt:     config.h
configure_file (${CMAKE_SOURCE_DIR}/src/config.h.in
                ${CMAKE_BINARY_DIR}/src/config.h)
```

Archivo "config.h.in"

```
    #ifndef CONFIG_H
    #define CONFIG_H
        #cmakedefine DATADIR "@DATADIR@"
        #cmakedefine PKGDATADIR "@PKGDATADIR@"
        #cmakedefine GETTEXT_PACKAGE "@GETTEXT_PACKAGE@"
        #cmakedefine RELEASE_NAME "@RELEASE_NAME@"
        #cmakedefine VERSION "@VERSION@"
        #cmakedefine VERSION_INFO "@VERSION_INFO@"
    #endif // CONFIG_H
```

41. Cmake en proyectos reales:

1. Awesome Window Manager (*lenguaje C*)
   Descargamos el código de *awesome* mediante:

   ```
   git clone https://github.com/awesomeWM/awesome.git
   ```

   Y vemos el contenido de `CMakeLists.txt`.
2. Geary (*lenguaje Vala*)

```
"Geary is an email application built around conversations, for the
GNOME 3 desktop. It allows you to read, find and send email with a
straightforward, modern interface."
```

Descargamos el código de *geary* mediante:

```
git clone git://git.gnome.org/geary
```

Veamos el contenido de los distintos `CMakeLists.txt` que proporciona.

3. Algo. . . ¿más grande?: VTK (*lenguaje C++*)

Ojo!. . . es GRANDE. . .

```
git clone http://vtk.org/VTK.git
```

42. Cmake: Gráficos de dependencias

- Prueba la opción --graphviz de cmake

```
cmake --graphviz=deps.dot .
```

- Dibuja los gráficos de las dependencias de diversos modos

```
dot -Tpng -o deps.png deps.dot
circo -Tpng -o deps.png deps.dot
```

43. Cmake: No olvides. . .

1. La página de manual: `man cmake`
2. Los interfaces curses (*ccmake*) y GUI (*cmake-gui*)
3. La opción "-E" (command mode): `cmake -E ...`
4. La opción de ayuda general es "--help": `cmake --help`
5. Algunas de las opciones de ayuda más particulares son:

```
--help-command-list
--help-command cmd
--help-module-list
--help-module module
--help-variable-list
--help-variable var
...
```


44. Meson (I)

1. Desarrollo muy reciente. Web.
2. Multiplataforma (GNU/Linux, OS X y Windows).
3. Depende de Python3 y de Ninja.
4. La idea básica: *sencillez de uso* y construcción rápida del software que configura.
5. La configuración se hace en archivos de texto llamados `meson.build`:

```
1  #Configuracion de hola-mundo
2  project('tutorial', 'c')
3  executable('demo', 'main.c')
```

6. A continuación tecleamos:

```
meson build; cd build; ninja
# O simplemente
meson build; ninja -C build
```

45. Meson (II)

1. Emplea Ninja como *backend* por defecto. Dependiendo de la versión de meson que tengas pudes elegir como *backend* proyectos de Visual Studio o de Xcode. Compara con CMake donde también puedes elegir *backend*.
2. Sólo permite crear *out-of-source builds*.
3. Los cambios en los archivos `meson.build` son detectados automáticamente al usar ninja.
4. Esto quiere decir que no es necesario ejecutar `meson build` si hacemos cambios en nuestros archivos `meson.build`, es más, si lo hacemos se producirá un error.

46. Prácticas en grupo:

COMPARATIVA MAKE/NINJA

1. Echa un vistazo a este artículo donde se comparan Make y Ninja.
2. ¿Porqué Make recursivo es lento? ¿Qué hace a Ninja tan rápido?

47. Prácticas individuales:

ELIGE UNA PRÁCTICA DE CUALQUIER ASIGNATURA, COPIALA EN DOS DIRECTORIOS DISTINTOS:

1. En uno de ellos, prepárala para ser configurada/compilada con *Autoconf + Automake*
   - Crea una versión de la misma usando `Automake` recursivo.
   - Crea otra versión de la misma usando `Automake` **no** recursivo.

2. Y en el otro prepárala para ser configurada/compilada con *Cmake*. En este caso echa un vistazo a CPack y trata de añadir un objetivo "`dist`" a "`CMakeLists.txt`" para que se pueda distribuir el código fuente en formato `TGZ` cuando tecleemos "`make dist`".

Ten en cuenta que:

`CPack crea un objetivo llamado "package_source"...puedes ayudarte en este caso de la orden de "cmake": "add_custom_target".`

OPCIONAL: EXISTEN OTROS SISTEMAS DE CONFIGURACIÓN.

- Es el caso de premake. Prepara el código anterior para ser configurado/compilado con premake.

Si no está instalado:

`Puedes instalarlo de este modo: sudo su; pacman -Sy premake; exit.`

ENTREGA:

- La práctica se entregará en pracdlsi en las fechas allí indicadas.

48. Aclaraciones

EN NINGÚN CASO ESTAS TRANSPARENCIAS SON LA BIBLIOGRAFÍA DE LA ASIGNATURA.

- Debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.