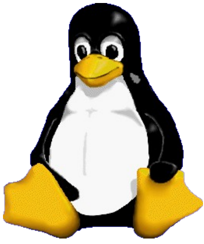


Programmation GNU/Linux

Autotools : autoconf et automake

By ShareVB

Table des matières

I.Base de la compilation....3
II.Compilation à la main....3
III.Compilation avec Makefile....3
  1.Introduction....3
  2.La commande make....3
  3.Fonctionnement....4
  4.Syntaxe d'un Makefile....4
    a)Syntaxe générale 4
    b)Les cibles....4
    c)Les dépendances....5
    d)Les actions....5
  5.Exemple de Makefile....5
IV.Compilation avec Autotools....6
  1.Pourquoi Autotools....6
  2.Objectif des autotools....6
  3.Fonctionnemement global....6
  4.Outil Autoconf....7
    a)Principe....7
    b)Fichiers nécessaires (Configure.in...)....7
    c)Mini exemple....8
  5.Outil Automake....9
    a)Principe....9
    b)Dépendances (Makefile.am...)....10
    c)Mini exemple....10
    d)Un exemple plus complet....11
    e)Les autres macros....11
    f)Génération du Makefile.in....12
  6.Gestion de la portabilité par config.h (outil Autoheader)....12
    a)Principe....12
    b)Dans configure.in....12
    c)Utilisation dans les sources....13
    d)Le fichier acconfig.h....13
  7.Outil Configure....14
    a)Principe....14
    b)Fichiers créés par configure....14
    c)Quelques options....15
      i.Généralités....15
      ii.Noms de fichiers est dossiers....15
      iii.Host Type....15
  8.Tableau récapitulatif....15
V.Compilation de library avec Autotools et Libtool....16
  1.Intégration à Autoconf....16
  2.Intégration à Automake....16
VI.FAQ & problèmes courants....16

1.Autoconf me dis quelque chose à propos de macros indéfinies 17
2.Mon Makefile ne contient aucun caractères : 17
3.Message du type « No rule to make target... needed by .deps/.P »....17
4.Impossible de refaire make....17
5.« configure: error: source directory already configured »....17
6.Problèmes de syntaxe dans le fichier configure.in 17
VII.Cross-compilation pour Windows avec Autotools, Libtools et MinGW....18
1.Pourquoi cross-compiler ?....18
2.Définition....18
3.Instalation de la plateforme sous Linux....18
a)Logiciels nécessaires....18
b)Nom de la cible....19
c)MinGW....19
d)Les binutils....20
e)GCC....20
f)Vérification de GCC....21
g)Les essais....21
i.Mode console....21
ii.Mode graphique....22
4.Intégration de la cross compilation à Autotools....22
a)Appels de configure....22
b)Appels de make et make install....23
c)Autoconf : modification dans le configure.in....23
d)Automake : modification dans les Makefile.am....24
e)Définir un MACRO suivant le type de système....24
VIII.Exemples d'utilisation des Autotools, Libtool et autres....25
1.Organisation de base d'un projet....25
2.Compilation d'une bibliothèque dynamique....25
a)Autoconf....25
i.Les bases pour compiler avec Libtool....25
ii.Ajout d'une option --enable-debug pour compiler avec informations de débogage....26
iii.Vérification de la présence d'une bibliothèque....26
iv.Vérification de la présence d'un fichier d'entête (ou plusieurs)....26
v.Vérification de la présence d'une fonction (ou plusieurs)....26
vi.Vérification de la présence et des particularités de certaines fonctions communes....26
b)Automake....27
i.Les bases....27
ii.Inclure un « .h » dans les include à l'installation....27
3.Compilation d'un exécutable utilisant le libexemple précédente....27
a)Bases....27
b)Autoconf....27
c)Automake....28
4.Utiliser Autotools avec Lex et Yacc....29
a)Autoconf....29
b)Automake....29
Bibliographie....29

I. Base de la compilation

Une compilation se fait en trois étapes :

- le prétraitement des macros (preprocessing) : cette étape consiste à traiter toutes les macros

(commençant par #, comme `#if`, `#define`) afin de remplacer les « appels » des macros par leur valeur en fonction des conditions `#if` (et autres). Le but étant de produire le code source C final à compiler qui ne contient plus de macros.

- la compilation (compiling) : cette étape consiste à traduire le code C en code objet spécifique au système sur lequel on veut faire tourner le programme. Cette étape produit un fichier objet par fichier source (sauf les entêtes)
- l'édition des liens (linking) : cette étape consiste à regouper toutes les fichiers objets et les bibliothèques de fonctions qu'utilise le programme que l'on compile. A l'issu de cette étape on obtient l'exécutable final.

II. Compilation à la main

Sous GNU/Linux, le préprocesseur/compilateur s'appelle `gcc` et le linker `ld.`

Pour compiler un programme, il suffira de taper une des séquences de commandes suivantes :

- soit en une seule fois :
  - `gcc -o` *`nom_exéctable fichier1.c fichier2.c ...`*
- soit en deux fois :
  - `gcc -c` *`fichier1.c fichier2.c ...`*
  - `gcc -o` *`nom_exécutable`* `*.o`

Si le programme utilise une library « `test` », il vous faudra rajouter dans les dernières lignes « `-ltest` »

III. Compilation avec Makefile

1. Introduction

La méthode précédente n'est pas vraiment pratique pour développer un projet ayant plusieurs dizaines de fichiers. On peut bien faire un script shell pour automatiser la compilation, mais, dans ce cas, il faut alors recompiler tous les fichiers à chaque fois que l'on fait un modification dans un seul alors que la simple recompilation du fichier suffirait. Pour cela, il existe `make` et les `Makefile.`

2. La commande make

`make` est un logiciel traditionnel des systèmes UNIX. Il sert à appeler des commandes créant des fichiers. À la différence d'un simple script shell, make exécute les commandes seulement si c'est nécessaire. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes.

`Make` fut à l'origine développé par le docteur Stuart I. Feldman, en 1977. Ce dernier travaillait pour Bell Labs à cette époque.

Depuis, plusieurs dérivés ont été développés, les plus connus et utilisés sont ceux de BSD et celui de GNU, ce dernier étant généralement celui utilisé par défaut avec les systèmes Linux. Ils diffèrent par certaines fonctionnalités, et par exemple les scripts prévus pour GNU Make peuvent ne pas fonctionner sous BSD Make.

3. Fonctionnement

Il sert principalement à faciliter la compilation et l'édition des liens puisque dans ce processus le résultat final dépend d'opérations précédentes : la compilation doit réussir pour que l'édition des liens puisse se faire.

Pour ce faire, make utilise un fichier de configuration appelé `makefile` qui porte souvent le nom de **Makefile**. Ce dernier décrit des `cibles` (qui sont souvent des fichiers, mais pas toujours), de quelles autres *cibles* elles dépendent, et par quelles `actions` (des commandes) y parvenir.

Afin de reconstruire une cible spécifiée par l'utilisateur, Make va chercher les cibles nécessaires à la reconstruction de cette cible, et ce récursivement.

Les règles de dépendance peuvent être explicites (noms de fichiers donnés explicitement) ou implicites (via des motifs de fichiers; par exemple *fichier*`.o` dépend de *fichier*`.c`, si celui-ci existe, via une recompilation).

4. Syntaxe d'un Makefile

a) Syntaxe générale

Un fichier Makefile contient donc plusieurs types de lignes :

- les lignes commençant par # sont des commentaires
- les lignes de la forme `NAME=valeur` permettent de définir des variables pour le script et les programmes appelés comme dans les scripts shels
- les lignes de règles de la forme : (<TAB> est le caractère de la touche tabulation et doit absolument être présent en tant que vrai TAB et pas simplement des espaces)

```
cible : dépendances
<TAB>action
```

b) Les cibles

`cible` peut être :

- `all` : la cible appelée par défaut si l'on ne donne pas d'arguments à make.
- un nom de fichier
- une association extension dépendance <-> extension cible sous la forme : « `.ext_dépendance.ext_cible` »
- une extension de fichier cible sous la forme « `%.ext_cible` »

c) Les dépendances

`dépendances` peut être :

- une liste de fichiers
- une extension de fichier dépendances sous la forme « `%.ext_dépendance` »

d) Les actions

Si l'action se fait en plusieurs étapes, on utilisera la syntaxe suivante :

```
cible : dépendances
<TAB>étape1 \
<TAB>étape2 \
...
<TAB>étapen-1 \
<TAB>étapen
```

5. Exemple de Makefile

Voici un exemple de makefile :

```
# Specify compiler
CC      = gcc

# Specify compiler options
CFLAGS  = -g
LDFLAGS = -L/usr/openwin/lib
LDLIBS  = -lX11 -lXext

# Needed to recognize .c as a file suffix
.SUFFIXES: $(SUFFIXES) .

# Executable name
PROG  = life

# List of object file needed for the final program
OBJS  = main.o window.o Board.o Life.o BoundedBoard.o

all: $(PROG)

# Program compilation and linking steps
$(PROG): $(OBJS)
     $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

.cpp.o:
     $(CC) $(CFLAGS) -c $*.c
```

Dans cet exemple: `.ccp.o` est une cible. Par défaut les cibles sont des fichiers, et c'est le cas ici. Cette cible ne dépend d'aucune autre puisqu'il n'y a rien d'autre sur la même ligne après *:*.

Pour parvenir à cette cible, il faut exécuter l'action, la commande `$(CC) $(CFLAGS) -c $*.c`

`all` est une autre cible qui dépend de `${PROG}` (et donc de *`life`*, qui est un fichier)

`$(PROG)` est une cible qui dépend de `$(OBJ)` (et donc de *`main.o window.o Board.o Life.o`* et *`BoundedBoard.o`*). Pour y parvenir, l'action est d'exécuter la commande `$(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)`

IV. Compilation avec Autotools

1. Pourquoi Autotools

La méthode de `makefile` est déjà beaucoup plus intéressante que la compilation manuelle. Cependant, à chaque évolution de votre projet, il est nécessaire de maintenir votre `Makefile`. De plus, il se peut qu'il faille réaliser un `Makefile` spécifique pour un système Unix particulier (en plus de modification de votre code). Maintenir un fichier `Makefile` devient vite fastidieux. On a donc inventer les Autotools afin de simplifier la maintenance des Makefile et leur portabilité.

2. Objectif des autotools

Ces outils ont deux objectifs principaux :

- simplifier le développement de programme portable. Ils permettent aux développeurs de se concentrer sur l'écriture du programme, en simplifiant les détails de portabilité entre les Unix et en lui permettant de décrire la structure de son application avec un langage plus simple que celui du `Makefile.`
- simplifier la compilation des programmes distribués sous formes de fichiers sources. Tous les programmes sont compilés en utilisant deux étapes simples et standards (*`./configure`* et *`make`*). Il n'y a pas besoin d'installer d'outils spécifiques pour l'installation.

3. Fonctionnement global

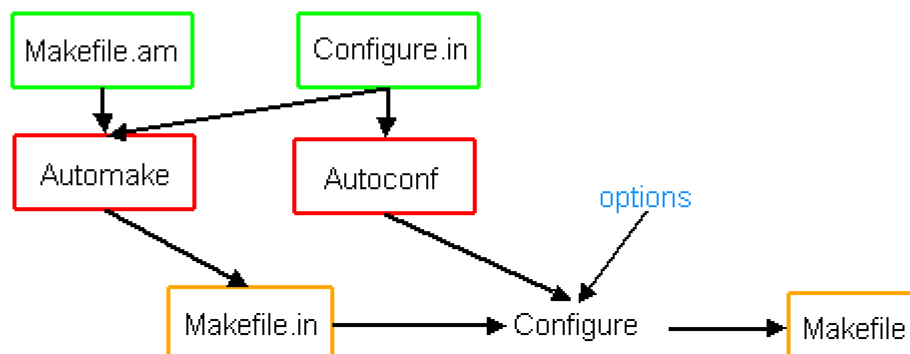
Seul les développeurs de programme doivent avoir installés les outils présentés ci-après. Les personnes qui veulent simplement installer un logiciel à partir des sources n'ont pas besoin d'outils spéciaux, excepté un shell Unix, un programme *`make`* et un compilateur C.

Voici les deux outils principaux :

- *`Automake`* : système pour décrire la structure de l'application, en permettant au développeur d'écrire un *`Makefile`* simplifié. Cet outil génère un fichier « moule » du Makefile à partir d'un série de macros.
- *`Autoconf`* : outil qui fournit une structure de portabilité, basée sur un ensemble de tests des spécificités de la machine au moment de la compilation et de l'installation.

Pour commencer à utiliser ces outils et construire un package près à l'installation, vous devez écrire au moins trois fichiers (*`configure.in, makefile.am, acconfig.h`*) et lancer certains outils pour générer les fichiers additionnels.

Voici le schéma des dépendances entre les fichiers et les outils (pour l'instant on ne prend pas en compte `acconfig.h`)



Légende :

Les deux outils possèdent la même philosophie. A partir de fichier simple écrit par le développeur, ils vont générer des fichiers "moules", qui seront complétés au dernier moment, lors de la compilation.

4. Outil Autoconf

a) Principe

Autoconf est un outil qui génère un script exécutable (*configure*) pour adapter l'installation de nos codes sources en fonction du système et de la machine d'installation. Le script *configure* est indépendant d'*Autoconf*, ainsi l'utilisateur n'a pas besoin d'avoir l'outil *Autoconf*.

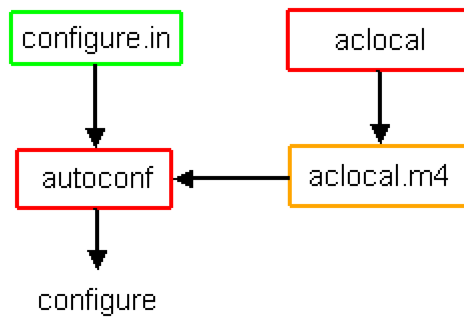
Ce script est créé à partir d'un fichier appelé *configure.in*, qui contient la description de toutes les options et spécificités que l 'on veut tester.
Ensuite, c'est le script *configure* qui effectuera réellement les tests. Il convertira le fichier "moule" *Makefile.in* en véritable *Makefile*.

b) Fichiers nécessaires (*Configure.in*...)

Le principal fichier pour la génération du script *configure* est comme vous l'avez compris le fichier *configure.in*.

- Configure.in : ce fichier contient un ensemble de macros Autoconf, décrivant les options et les caractéristiques du système à tester (comme la présence de fichiers d'en-tête ou de fonctions). De nombreuses macros existent déjà pour tester certains aspects, mais il est possible de créer ses propres macros à partir de "templates".
  Pour commencer, vous pouvez lancer la commande ***autoscan***, qui va générer un fichier nommé *configure.scan*. Ce fichier peut servir de prototype pour *configure.in*.
  Cette commande examine les fichiers sources en cherchant les problèmes de portabilité. Il ne vous reste plus qu'à renommer le fichier *configure.scan* en *configure.in*, et à le compléter par d'autres macros...
  Le langage de configure.in est procédural, c 'est à dire que chaque ligne du fichier est une commande qui est exécutée
- aclocal.m4 : nécessaire lorsque l'on utilise *Automake*, ou bien si on a défini ses propres macros. Il est possible de générer ce fichier à partir de la commande ***aclocal.*** Cet outil recherche les définitions de macros externes dans les fichiers .m4 (écrit si on redéfini des macros) et dans le fichier *Configure.in*, et créé le fichier *aclocal.m4*. Par exemple, si on utilise *Automake*, l'outil *aclocal* rendra compréhensible les macros automake (ex. AM_INIT_AUTOMAKE issue de configure.in) pour *Autoconf* (en les traduisant dans le fichier aclocal.m4).

  Ce qui donne le schéma de relation suivant :



Légende :

c) Mini exemple

Voici quelques règles pour écrire un fichier *`Configure.in`* :

- les lignes qui commencent par *`dnl`* ou *`#`* sont des commentaires et ne sont pas exécutées par `autoconf`.
- le fichier doit toujours commencer par la macro `AC_INIT()`
- il doit également toujours se terminer par `AC_OUTPUT(Makefile)`. Cette macro contient la liste des fichiers que configure doit générer. Si votre projet contient plusieurs `Makefile`, il suffit de les ajouter séparés par des espaces. Une alternative serait de définir la liste des `Makefile` à générer dans la macro `AC_CONFIG_FILES` et de laisser `AC_OUTPUT` sans argument.

Prenons un exemple de projet contenant uniquement un fichier « `main.c` » utilisant :

- les fichiers d'entête `stdlib.h, string.h, unistd.h` et les entêtes `libc` classiques
- les fonctions `malloc, getcwd, strdup` et `strstr`

Si vous exécuter `autoscan` dans le dossier de ce projet, vous obtiendrez ceci après ajout de `AM_INIT_AUTOMAKE` et retirer les `[]` autour de `config.h` dans `AC_CONFIG_HEADER` :

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
#définit les informations sur le programme
AC_INIT(nom_du_programme, version_mineur_point_majeur, votre_email)
#le fichier source
AC_CONFIG_SRCDIR([main.c])
#le fichier contenant les #define de fonctionnalités du système
AC_CONFIG_HEADER(config.h)
#initialisation d'automake pour le programme
AM_INIT_AUTOMAKE(nom_du_programme, version_mineur_point_majeur)

# Checks for programs.
#vérifier la présence d'un compilo C++
AC_PROG_CXX
#vérifier la présence d'un compilo C
AC_PROG_CC
#vérifier la présence d'un préprocesseur C
AC_PROG_CPP
#vérifier la présence du programme install (compatible BSD)
AC_PROG_INSTALL
#vérifier la présence de l'option -s de ln
```

```
AC_PROG_LN_S
#vérifier la présence de la commande make
AC_PROG_MAKE_SET
#vérifier la présence de la commande ranlib
AC_PROG_RANLIB

# Checks for libraries.

# Checks for header files.
#vérifier la présence des entêtes de la libc
AC_HEADER_STDC
#vérifier la présence de la liste de fichier en argument
AC_CHECK_HEADERS([stdlib.h string.h unistd.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.
#vérifier le comportement de malloc (en particulier, taille nulle)
AC_FUNC_MALLOC
#vérifier la présence des fonctions passés en argument
AC_CHECK_FUNCS([getcwd strdup strstr])

#fichier Makefile à générer
AC_CONFIG_FILES([Makefile])
#générer les Makefiles
AC_OUTPUT
```

Vous pouvez renommer ce fichier de `configure.scan` en `configure.in`.

Une fois le fichier *`configure.in`* écrit, il faut lancer les commandes suivantes :

```
[user]$ aclocal
[user]$ autoconf
```

La première permet d'interpréter la macro pour automake (et créé le fichier `aclocal.m4`) et la seconde génère tout simplement le script *`configure`*.

5. Outil Automake

a) Principe

Automake est un outil qui permet de minimiser les tâches de maintenance des `Makefile`, en décrivant la structure de l'application. En fait, il génère un (ou des) fichier(s) "moule" du `Makefile`, nommé *`Makefile.in`*, à partir d'une série de macros contenues dans le(s) fichier(s) *`Makefile.am`*. Cet outil est souvent associé à *`Autcoconf`*.

`Automake` se base sur le fichier *`Configure.in`* pour savoir quels sont les *`Makefile.in`* à générer. La description de ce que doit contenir un de ces fichiers est elle-même contenue dans un fichier *`Makefile.am`*.

Ensuite le fichier *`Makefile.in`* est celui qui va être transformé par l'outil *`configure`* en véritable `Makefile`.

Il faut enfin noter que le « `langage` » utilisé par `Automake` n'est pas procédural mais formel.

b) Dépendances (*Makefile.am...*)

L'outil *Automake* s'appuie sur deux fichiers pour générer le *Makefile.in*. Le principal bien sur est le fichier *Makefile.am*.

- Makefile.am : ce fichier décrit comment le code doit être construit. Il contient une série de définitions de macros Automake sous la forme :
  variable = valeur
  Il peut également contenir des définitions de "cible" de Makefike (ex. : clean, install...). Toutes ces définitions vont induire les règles du Makefile.
- Configure.in : ce fichier, décrit pour l'outil *Autoconf*, peut contenir également certaines macros Automake, par exemple pour savoir quels sont les *Makefile.in* à générer. Pour utiliser Automake, ce fichier doit contenir au moins un ligne spécifique qui est :

```
AM_INIT_AUTOMAKE(nom_du_programme,
version_mineur_point_majeur)
```

Les paramètres à la directive AM_INIT_AUTOMAKE permettent de définir le nom du package correspondant au programme, ainsi que son numéro de version.

c) Mini exemple

Voici quelques règles pour écrire des fichiers *Makefile.am* et *Configure.in* :

- pour chaque *Makefile.am*, il doit y avoir un Makefile correspondant dans la macro AC_OUTPUT ou AC_CONFIG_FILES du fichier *Configure.in*.
- il y a en général un *Makefile.am* par répertoire du projet. Pour chaque répertoire avec des sous-répertoires, le fichier *Makefile.am* doit contenir la ligne suivante :

  ```
  SUBDIRS = sous_dossier1 sous_dossier2 ...
  ```

  où chaque *sous_dossierN* est le nom d'un sous-répertoire contenant des fichiers sources du projet.
- une ligne commençant par # dans le fichier *Makefile.am* est ignorée par Automake. Il s'agit donc d'un ligne de commentaire.
- si le package doit créer un programme exécutable (pas une library), alors dans le fichier *Makefile.am* situé dans le répertoire où le programme doit être construit, il y aura une ligne du type :

  ```
  bin_PROGRAMS = nom_programme
  ```

  où *nom_programme* est le nom du programme. Ensuite, il y a une ligne pour spécifier les fichiers sources nécessaires :

  ```
  nom_programme_SOURCES = fichier1 fichier2 ...
  ```

  où chaque *fichierN* est le nom des fichiers sources liés au programme (ex. : main.c). On peut y lister les fichiers d'entêtes « .h » afin d'éviter les programmes.

d) Un exemple plus complet

Prenons un exemple plus complet :

- deux répertoires : src et doc. Ces dossiers devrait contenir (dans un vrai projet) des Makefiles pour générer leur contenu

- utilisation d'une bibliothèque statique : `libexemple.a`

Dans un premier temps, il faut donc écrire un fichier *`Configure.in`* qui contiendra pour l'instant la même chose que le précédent. Ensuite, écrire le fichier *`Makefile.am`* suivant :

```
## fichier à passer à Automake pour générer le fichier Makefile.in
SUBDIRS = src doc
bin_PROGRAMS = exemple
exemple_SOURCES = exemple.c fonction.c exemple.h
exemple_LDADD = libexemple.a
```

Et enfin quelques explications :

- `SUBDIRS` : signifie que les *`makefile`* qui sont contenus dans ces sous-répertoires doivent être générés avant de générer celui du répertoire courant.
- `bin_PROGRAMS` : nom des exécutables créés.
- *`exemple`*`_SOURCES` : liste des fichiers nécessaires à la compilation de l'exécutable *`exemple`*.
- *`exemple`*`_LDDAD` : liste des librairies statiques nécessaires à la compilation de l'exécutable *`exemple`*.

e) Les autres macros

D'autres macros non présentées dans cet exemple sont disponibles. Elles doivent être précédées du nom du fichier (les points sont remplacés par _) auquel elles s'appliquent. On peut citer :

- `_LDFLAGS` : Flags à passer au compilateur.
- `_CPPFLAGS` : Flags à passer au préprocesseur.
- `_CXXFLAGS` : options à passer au compilateur.
- `_EXTRA_DIST` : liste des fichiers à inclure au code source de distribution qui ne sont en général pas des sources (par exemple, les fichiers générés par `flex` et `bison`).

On peut aussi citer au même titre que `bin_PROGRAMS`, l'équivalent pour créer une library ou inclure un fichier dans les include du système

- `lib_LIBRARIES` : liste des librairies qui seront compilées.
- `include_HEADERS` : liste des fichiers Headers du package à installer.

f) Génération du Makefile.in

Il ne vous reste plus qu'à lancer la commande suivante pour générer le fichier *`Makefile.in`* :

```
[user]$ automake --add-missing
```

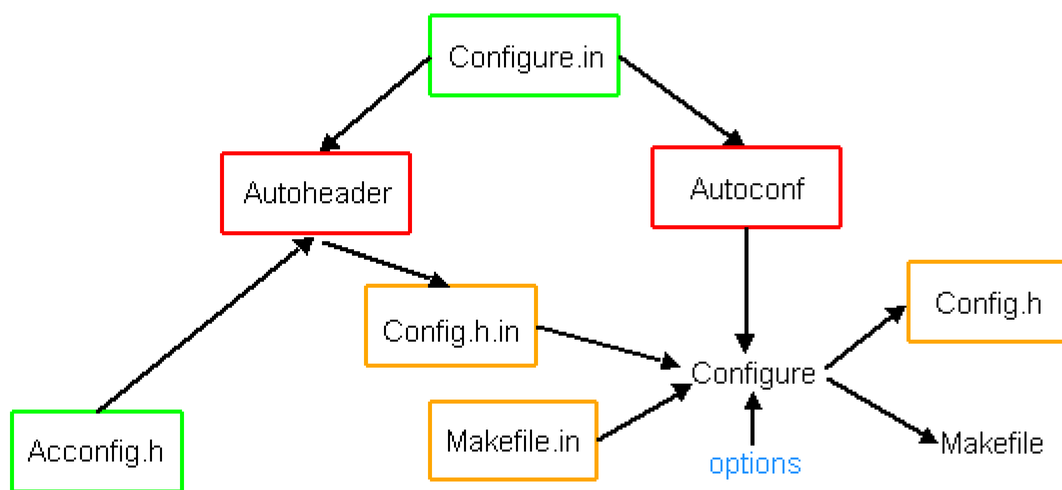
L'option passée à `Automake` l'informe qu'il doit installer les fichiers qui pourraient éventuellement manquer. En effet, `Automake` requiert certains fichiers dans certaines situations. Il est aussi possible de spécifier l'option `--`*`foreign`*, pour avoir un niveau de rigueur décontracté (par opposition à `--`

gnu, option par défaut, qui requiert les fichiers *INSTALL, NEWS, README, COPYING, AUTHORS*, et *ChangeLog* dans le répertoire source). Pour les autres options admises par Automake, voir le manuel Automake, comme d'habitude...

6. Gestion de la portabilité par config.h (outil Autoheader)

a) Principe

Cet outil est utilisé pour la gestion des variables du pré-processeur (#define, #undef), et pour la gestion des fichiers d 'en-tête et leurs fonctions (stdlib.h, fonction free...). Le but est de savoir si tel « .h » ou fonction existe sur le système sur lequel on compile. Il créer un fichier "patron" (*config.h.in*) comprenant un ensemble de #define pour l'outils *configure* à partir de *acconfig.h* et *Configure.in*. Voici un petit schéma pour mieux comprendre l'intégration de cet outil avec les autres :



Légende :

L'outils *Autoheader* scanne le fichier *Configure.in* et recherche quels sont les symboles du pré-processeur C qu'il doit définir.

b) Dans configure.in

Voici les macros que l'on peut/doit trouver dans le fichier *Configure.in* :

- AM_CONFIG_HEADER(config.h)
  Cette macro (nécessaire) spécifie le nom du fichier d'en-tête qui contiendra les définitions de macros du pré-processeur. Normalement, il s'agit de *config.h*. Cette macro peut optionnellement définir le nom du fichier créé par Autoheader (en entrée de configure). Par défaut, il s'agit de *config.h.in*. Mais ce nom n'est pas "génial" pour le système de fichier DOS. Il peut être mieux de le renommer tout simplement config.in. Ce qui donnera la macro suivante : AM_CONFIG_HEADER(config.h:config.in)
- AC_CHECK_HEADERS(unistd.h)

Permet de vérifier la présence ou non d'un (ou des) fichier(s) d'en-tête, listé(s) en argument (séparé par des espaces). Ainsi, dans le fichier *config.h.in*, on aura le code suivant :
```
/* Define as 1 if you have unistd.h. */
#define HAVE_UNISTD_H 1
```
Sur les systèmes qui ont 'unistd.h', le script configure décommentera. Sur les autres, il laissera la ligne inchangée.

- AC_CHECK_FUNCS(strstr)
  Permet de vérifier la présence ou non d'un (ou des) fonction(s), listée(s) en argument (séparé par des espaces). Ainsi, dans le fichier *config.h.in*, on aura le code suivant :
```
/* Define as 1 if you have strstr. */
#define HAVE_STRSTR 1
```
  Sur les systèmes qui ont 'strstr', le script configure décommentera. Sur les autres, il laissera la ligne inchangée.

c) Utilisation dans les sources

Pour utiliser le fichier config.h, on inclura en haut de chaque fichier source :

```
#ifdef HAVE_CONFIG_H
#include « config.h »
#endif
```

d) Le fichier acconfig.h

L'autre fichier utilisé est le fichier *acconfig.h* :

- Ce fichier est utilisé pour décrire les macros qui ne sont pas reconnues par l'outil Autoheader. Il s'agit généralement d'un fichier inintéressant, qui consiste en une collection de lignes de *#undef*.
  Si vous générer un fichier d'en-tête portable (en utilisant la macro AM_CONFIG_HEADER dans le fichier *Configure.in*), alors il faut écrire le fichier *acconfig.h*, qui contiendra les lignes suivantes :
```
/* Name of package */
#undef PACKAGE
/* Version of package */
#undef VERSION
```

L'outil Autoheader informe toujours par un message d'erreur lorsqu'il manque un élément dans le fichier *acconfig.h*.

7. Outil Configure

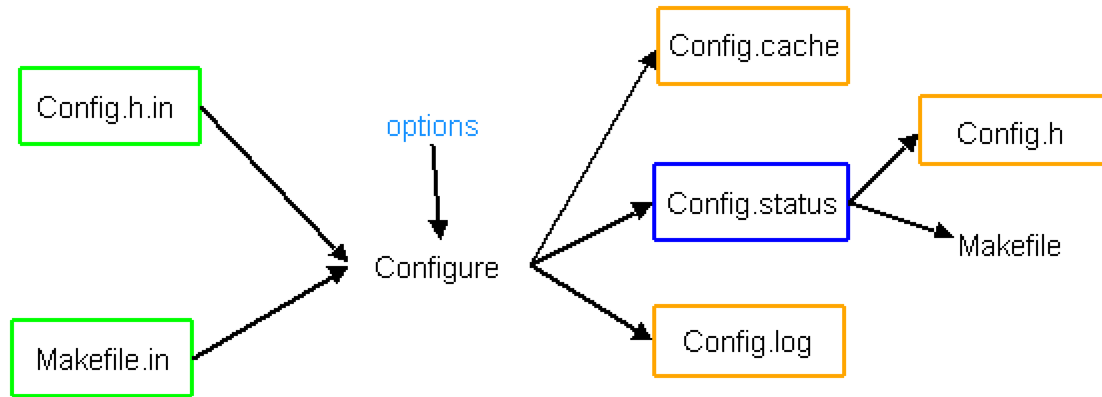
a) Principe

L'outil *Configure* est un script shell, qui va convertir les fichiers "moules" dans leur formes définitives (*Makefile.in -> Makefile*). Il effectue les tests souhaités avant la compilation et remplace les paramètres de configuration par leurs valeurs appropriées. Ce script est écrit de la manière la plus portable possible.

Ce script shell est fourni régulièrement avec le package pour l'utilisateur final. Il est créé par le développeur en lançant l'outil `Autoconf`.

b) Fichiers créés par configure

Voici le schéma représentant les fichiers qui interagissent avec *`Configure`* :



Légende :

- `config.status` : la première étape pour construire un programme est de lancer le script *`configure`*. Ce script va créer le fichier *`config.status`*, qui est lui-même un script shell chargé de sauvegarder les options que l'on a passé à `configure`.
- un ou plusieurs `Makefile` : un dans chaque sous-répertoires du projet. C'est le fichier que make va lire pour construire le programme, en compilant tous les fichiers sources et mettant à jour les binaires.
- `config.h` : ce fichier décrit les variables du pré-processeur C, sur lesquelles le code C s'appuiera pour adapter son comportement en fonction du système. Il contient les directives `#define`. Ce fichier est optionnel, créé si l'on gère un fichier d'en-tête.
- `config.cache` : ce fichier est utilisé par le script *configure* pour mettre en cache les résultats entre les différentes utilisations.
- `config.log` : il contient tous les messages produits par le compilateur, pour l'aide au débuggage si *`configure`* rencontre un problème.

c) Quelques options

i. Généralités

Il est possible de choisir le répertoire d'installation du programme. Pour cela, il faut utiliser l'option `--prefix` lors de l'utilisation de *configure*. Par défaut, ce répertoire est `/usr/local`.

On peut aussi spécifier si certaines fonctionnalités devront être activées ou désactivées :

```
--disable-FEATURE        ne pas inclure une fonctionnalité
--enable-FEATURE=no      ne pas inclure une fonctionnalité
```

```
--enable-FEATURE=yes   inclure une fonctionnalité

--with-PACKAGE=yes     utiliser PACKAGE
--without-PACKAGE      ne pas utiliser PACKAGE (idem à --with-PACKAGE=no)
```

ii. Noms de fichiers est dossiers

```
--bindir=DIR           exécutables utilisateur dans DIR [PREFIX/bin]
--sbindir=DIR          exécutables d'admin dans DIR [PREFIX/sbin]
--libexecdir=DIR       bibliothèques exec dans DIR [PREFIX/libexec]
--datadir=DIR          données dans DIR [PREFIX/share]
--sysconfdir=DIR       fichiers de config dans DIR [PREFIX/etc]
--localstatedir=DIR    données modifiables dans DIR [PREFIX/var]
--libdir=DIR           bibliothèques dans DIR [PREFIX/lib]
--includedir=DIR       fichier d'entête C dans DIR [PREFIX/include]
--infodir=DIR          documentation dans DIR [PREFIX/info]
--mandir=DIR           pages de man dans DIR [PREFIX/man]
--srcdir=DIR           trouver les sources dans DIR [configure dir ou ..]
--program-prefix=PREFIX           préfixer les noms des programmes avec PREFIX
--program-suffix=SUFFIX           suffixer les noms des programmes avec SUFFIX
```

iii. Host Type

```
--build=BUILD          configurer pour construire sur BUILD [BUILD=HOST]
--host=HOST            configurer pour construire sur l'hôte HOST [guessed]
--target=TARGET        configurer pour construire pour TARGET [TARGET=HOST]
```

8. Tableau récapitulatif

Voici un tableau qui synthétise les différents outils à utiliser dans le développement d'un package :

Quand	Commande	Actions
Au début du projet	**`autoscan`**	Génère un fichier *moule* du fichier **Configure.in**
Au début du projet		Ecrire un fichier **Makefile.am** (structure du programme)
Au début du projet	**`aclocal`**	Pour installer les outils complémentaires à `automake`
Au début du projet	**`autoheader`**	Pour déterminer les variables du pré-processeur à définir, et les sauver dans le fichier **config.h.in**
Au début, ou si configure.in a été modifié	**`autoconf`**	Génère **`configure`** à partir de `Configure.in`
Au début, ou si l'on a perdu les makefile.in	**`automake`**	Génère les fichiers **`Makefile.in`** à partir des `Makefile.am`
Pour changer de configuration	**`configure`**	Génère les fichiers **Makefile** et le fichier **config.h** en fonction des options de compilation choisies
Pour compiler les sources	**`make`**	Compile les sources ou met à jour les binaires

V. Compilation de library avec Autotools et Libtool

GNU `Libtool` est un logiciel du Projet GNU qui sert à créer des bibliothèques portables sur les systèmes UNIX (voire MacOS et Windows).

Dans le passé, si un programmeur voulait profiter des avantages des bibliothèques dynamiques, il devait écrire du code spécifique à chacune des plateformes sur lesquelles la bibliothèque était compilée. Il devait aussi écrire un système de configuration permettant à l'utilisateur qui installe le logiciel, de décider quel type de bibliothèque compiler.

Libtool simplifie la tâche du programmeur en encapsulant à la fois les dépendances par rapport à chaque plateforme, ainsi que l'interface-utilisateur, dans un seul script. Cet outil est conçu de façon que toute la fonctionnalité de chaque plateforme soit accessible via une interface générique, tout en cachant les choses obscurs au programmeur.

L'interface de Libtool vise à être cohérente. Les usagers ne sont pas supposés devoir lire de la documentation de bas niveau pour réussir à faire compiler des bibliothèques dynamiques. Ils devraient n'avoir qu'à exécuter le script *configure* (ou un équivalent), et Libtool devrait se charger des détails.

On utilise typiquement `Libtool` avec `Autoconf` et `Automake`, deux autres outils du système de compilation GNU.

1. Intégration à Autoconf

Pour utiliser `Libtool` dans un fichier `Configure.in`, il faut ajouter les directives suivantes :

- `AM_PROG_LIBTOOL` : indique que l'on utilise Libtool pour la compilation
- `AC_PROG_RANLIB` : indique que l'on doit disposer de ranlib qui est utilisé par Libtool
- `AC_PROG_MAKE_SET` : indique que l'on doit disposer de make qui est utilisé par Libtool

2. Intégration à Automake

Pour utiliser Libtool dans un fichier `Makefile.am`, il faudra :

- remplacer `lib_LIBRARIES` par `lib_LTLIBRARIES`
- remplacer les extensions `.a` par `.la`. (et aussi avec _)
- remplacer les extensions `.o` par `.lo`. (et aussi avec _)

Enfin, vous pouvez exécuter la commande libtoolize dans le dossier du projet.

VI. FAQ & problèmes courants

Voici une liste non exhaustive des problèmes que l'on peut rencontrer en utilisant ces outils.

1. Autoconf me dis quelque chose à propos de macros indéfinies

Cela signifie qu'il y a des macros dans le fichier *`configure.in`*, qui ne sont pas définies par `Autoconf`. Soit vous utilisez une vieille version `d'`*`Autoconf`*; soit le nom de la macro a été incorrectement entré. Dans le premier cas, essayer d'en installer une nouvelle version. Dans le second, vérifiez le nom des macros que vous utilisez.

2. Mon *Makefile* ne contient aucun caractères :

Cela peut signifier que vous avez essayer d'utiliser une substitution autoconf dans le *`Makefile.in`* sans avoir ajouter un appel approprié à `AC_SUBST` au scrip `configure`. Ou cela peut tout simplement signifier que vous devez reconstruire le *`Makefile`*. Pour le reconstruire depuis le *`Makefile.in`*, lancer le script shell *`config.status`* sans arguments. Si vous devez forcer *`configure`* à être relancé, lancez d'abord *`config.status --recheck`*.

3. Message du type « *No rule to make target... needed by .deps/.P* »

Les dépendances des fichiers sont stockées dans un répertoire appelé `.deps`. Si un fichier disparaît brutalement, il se peut qu'il soit encore référencé lors du `make`, ce qui conduit à ce message. Le plus simple est d'effacer le répertoire `.deps`.

4. Impossible de refaire make

Lorsque une erreur a été commise dans un *`Makefile.am`*, les mécanismes automatiques qui permettent de régénérer les `Makefile` peuvent ne plus fonctionner. Il faut alors les recréer soi-même grâce à `Automake` et `Autoconf` par la séquence de commande : « `autoheader ; aclocal ; autoconf ; automake ; ./configure ; make` »

5. « configure: error: source directory already configured »

Il est possible de compiler des sources dans plusieurs répertoires. Mais ceci n'est vrai qu'à condition de ne pas avoir compilé dans le répertoire de sources. En effet, lorsque l'on configure le répertoire de sources pour y effectuer une compilation, on créé un certain nombre de fichiers qui peuvent interférer avec ceux des autres répertoires de compilation, et engendrer des problèmes difficiles à détecter. Dans sa grande bonté, Autoconf empêche ce genre de situation.

6. Problèmes de syntaxe dans le fichier configure.in

La syntaxe des macros utilisées par Autoconf impose que les parenthèses délimitant les paramètres d'une fonction soient accolées au nom de la fonction. Donc :

```
AC_DEFINE(CFLAGS) juste

AC_DEFINE (CFLAGS) faux
```

VII. Cross-compilation pour Windows avec Autotools, Libtools et MinGW

1. Pourquoi cross-compiler ?

Imaginons que vous avez réalisé un programme qui puisse fonctionner/être utile à la fois sous Unix/Linux et Windows...cependant certaines fonctions ne sont pas disponibles, portent un autre nom ou sont présentes dans un autres « `.h` »...il vous viendra alors à l'esprit d'utiliser les `Autotools` et `Autoheader` pour inclure ou redéfinir des fonctions suivants la plateforme...

Seulement voilà, on a deux solutions pour compiler un programme pour Windows avec les Autotools et Libtool :

- utiliser MinGW sous Linux avec un version de GCC/G++ et binutils compilés pour générer du code Windows, des exécutables PE et des dlls.
- Utiliser Cygnus Cygwin et leur portage des Autotools

La seconde solutions est pratique si vous souhaitez compiler le programme directement sous Windows et que vous n'êtes pas géné d'être lié (éventuellement) à la dll d'émulation cygwin1.dll.

L'avantage de la première est que si vous développez sous Linux, vous n'aurait pas à redémarrer sous Windows pour compiler votre programme sous Windows. Il vous suffira de compiler sous Linux et de tester avec Wine...en attendant le redémarrage...Enfin, MinGW permet de ne pas avoir de liaison à une quelconque autre dll que celle de Windows (et celles que le programme utilise).

Enfin la dernière solution consisterait à porter votre code dans un fichier Visual C++/Visual Studio ce qui peut souvent s'avérer long du fait que VC++ n'utilise pas forcément les mêmes options par défaut.

2. Définition

On peut donc donner une définition de la cross compilation : c'est le fait de compiler du code sur un système pour produire du code machine pour un autre système.

3. Instalation de la plateforme sous Linux

a) Logiciels nécessaires

Pour réaliser un cross-compilateur, vous aurez besoin des logiciels suivants :

- MinGW. C'est une distribution complète d'outils pour travailler sous Windows, et parmis tout ces outils, seulement 2 paquets sont utiles pour la cross-compilation :
  - MinGW runtime en version binaire : `mingw-runtime-`*`x.x`*`.tar.gz`
  - Win32 API en version binaire : `w32api-`*`x.x`*`.tar.gz`
- les binutils. binutils-*`x.x`*`.tar.gz`
- GCC. : seuls les compilateurs C (`gcc-core`) et C++ (`gcc-g++`) ont été testés.

b) Nom de la cible

De la même manière que votre système a un identifiant pour désigner le type de plateforme, il nous en faut un pour désigner la plateforme cible. Prenons quelques exemples :

- sur un PC basé sur un AMD Athlon XP, sous Linux :

```
$ gcc -dumpmachine
i686-pc-linux-gnu
```

- sur un PC basé sur un Intel Pentium MMX, sous FreeBSD :

```
$ gcc -dumpmachine
i386-unknown-freebsd
```

Comme vous le voyez, ce nom se divise en 2 parties : la machine, puis l'OS. Nous allons donc construire un nom sur ce même modèle :

- la machine ayant toutes les chances d'être un PC, le début du nom sera `ix86-pc`. Il vous suffit de remplacer `ix86` par le bon processeur (i386 pour un 386, i586 pour un Pentium, i686 pour un Pentium II et plus).
- pour le système, on ne va pas utiliser un nom dérivé de Windows, mais un qui va rappeler la librairie que nous allons utiliser. Il existe au moins Cygwin et MinGW; ici, c'est le second qui nous interesse, donc on prendra `mingw32`.

On choisira dans le cas d'une compilation pour un Pentium II ou plus et pour Windows avec MinGW : `i686-pc-mingw32`

c) MinGW

La première chose à faire est d'installer les librairies et les headers que `binutils`, `GCC` et toutes nos cross-compilations vont réclamer (l'équivalent de la `glibc` sous `Linux`).

C'est également maintenant que vous décidez où vous voulez installer tout l'environnement, par exemple : `/usr/local/cross`.

On commence par créer le répertoire d'installation en tant que `root` :

```
[root]# mkdir /usr/local/cross; cd /usr/local/cross
[root cross]# mkdir i686-pc-mingw32; cd i686-pc-mingw32
```

Nous avons créé un répertoire portant le nom de la plateforme cible. C'est à l'intérieur que les archives vont pouvoir être décompressées :

```
[root i686-pc-mingw32]# tar zxf /chemin/vers/mingw-runtime-x.x.tar.gz
[root i686-pc-mingw32]# tar zxf /chemin/vers/w32api-x.x.tar.gz
[root i686-pc-mingw32]# chown -R root:root .
```

Le premier package, `mingw-runtime`, correspond entre autres à tous fichiers que le compilateur et le linker auront besoin pour produire des binaires pour Windows. Le second contient, comme son nom l'indique, toute l'API Win32, c'est-à-dire les headers et les librairies de base.

Maintenant, il va s'agir de compiler les `binutils`, puis `GCC`. Pour cela vous n'avez plus besoin d'être `root`.

d) Les binutils

Comme pour la plupart des compilations, tout se déroule en 3 étapes :

- lancer le script `configure`
- compiler : `make`
- installer : `make install`

Une fois que vous avez décompressé l'archive des `binutils` quelque part, vous devez créer un autre répertoire à côté, dans lequel se déroulera la compilation, parce qu'il est recommandé de ne pas travailler directement dans le répertoire décompressé :

```
[user]$ tar jxf /chemin/vers/binutils-2.14.tar.bz2
[user]$ mkdir binutils-build
[user]$ cd binutils-build
```

Maintenant, nous allons lancer le script `configure`, avec 2 paramètres :

- l'un pour donner le prefix c'est-à-dire le répertoire d'installation, dans notre cas, `/usr/local/cross`

- le nom de la cible (sans quoi les `binutils` se compileraient pour la plateforme hôte)

```
[user binutils-build]$ ../binutils-x.x/configure \
  --prefix=/usr/local/cross \
  --target=i686-pc-mingw32
```

Ensuite, compilation :

```
[user binutils-build]$ make
```

Une fois terminée avec succès, il nous reste à installer tout ceci, en tant que `root` :

```
[user binutils-build]# make install
```

C'est maintenant terminé pour les `binutils.` Il vous reste à mettre à jour votre `$PATH` si le chemin d'installation que vous avez choisi n'y est pas :

```
[user]$ export PATH="/usr/local/cross/bin:$PATH"
```

Pour tester que tout fonctionne, vous pouvez faire ceci :

```
build$ i686-pc-mingw32-ld -V
GNU ld version x.x XXXXXXXX
  Supported emulations:
   i386pe
```

Les `binutils` ont donc été compilé pour produire un format binaire `i386pe`, le `PE` étant bien le format supporté par Microsoft Windows.

e) GCC

Pour `GCC`, nous allons refaire exactement la même chose qu'avec les `binutils.` Commençons avec la décompression et la création du répertoire de travail :

```
[user]$ tar jxf /chemin/vers/gcc-core-x.x.x.tar.bz2
[user]$ tar jxf /chemin/vers/gcc-g++-x.x.x.tar.bz2
[user]$ mkdir gcc-build
[user]$ cd gcc-build
```

Ici, nous avons décompressé les archives du compilateur C et C++.

Ensuite, la ligne du `configure` est identique à celui des `binutils` :

```
[user gcc-build]$ ../gcc-3.4.0/configure \
  --prefix=/usr/local/cross \
  --target=i686-pc-mingw32
```

Cette fois-ci, il faut bien vérifier que le `configure` trouve les `binutils` installés précédemment. Il doit afficher quelque chose comme :

```
checking for i686-pc-mingw32-ar... i686-pc-mingw32-ar
checking for i686-pc-mingw32-as... i686-pc-mingw32-as
checking for i686-pc-mingw32-dlltool... i686-pc-mingw32-dlltool
checking for i686-pc-mingw32-ld... i686-pc-mingw32-ld
checking for i686-pc-mingw32-nm... i686-pc-mingw32-nm
checking for i686-pc-mingw32-ranlib... i686-pc-mingw32-ranlib
checking for i686-pc-mingw32-windres... i686-pc-mingw32-windres
```

Si au lieu de ça, il affiche des lignes comme la suivante, la compilation ne fonctionnera pas et il faut revérifier votre `$PATH` :

```
checking for i686-pc-mingw32-ar... no
```

Quand tout est bon pour le `configure`, nous passons à la compilation, qui cette fois, sera plus longue :

```
[user gcc-build]$ make
```

Nous allons conclure cette partie `GCC` avec l'installtion en tant que `root` :

```
[root gcc-build]# make install
```

Comme vous êtes passé `root`, il se peut que le `$PATH` ne soit plus à jour, et l'installation échouera peut-être. Dans ce cas, répétez la mise à jour de l'environnement, comme nous l'avons fait à la fin de l'installation des `binutils.` Après cette étape, vous pouvez revenir à votre compte normal (non privilégié).

f) Vérification de GCC

Comme contrôle basique, nous pouvons demander à GCC pour quelle plateforme il est prévu :

```
build$ i686-pc-mingw32-gcc -dumpmachine
i686-pc-mingw32
```

g) Les essais

Maintenant, vous pouvez tester, on ne sait jamais.

i. Mode console

Voici le traditionnel « `Hello World` » :

```
#include <stdio.h>

int main()
{
  printf("Hello World !\n");
  return 0;
}
```

Comme je pense que vous connaissez déjà par cœur ce code source, passons directement à la compilation :

```
build/hello$ i686-pc-mingw32-gcc -o hello.exe hello.c
build/hello$ ls
hello.c  hello.exe
```

Comme vous le voyez, `GCC` a produit `hello.exe`. Pour le lancer, vous pouvez utiliser `Wine` ou une machine sous Windows, bien évidemment. Et si « `wine hello.exe` », vous sort un magnifique « `Hello World` », vous avez réussi la compilation

ii. Mode graphique

Essayons un exemple graphique maintenant. L'objectif est d'afficher une simple boite de dialogue, ce qui nous force à utiliser le header `windows.h` et le point d'entrée `WinMain` spécifique aux applications `Win32` :

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
    LPSTR lpCmdLine, int nCmdShow)
{
  MessageBox(NULL,
    "Cette fenêtre prouve que le cross-compilateur est fonctionnel !",
    "Hello World", MB_OK);
  return 0;
}
```

Ensuite viens la compilation, identique à la précédente :

```
build/hello$ i686-pc-mingw32-gcc -o hello_ui.exe hello_ui.c
build/hello$ ls
hello_ui.c  hello_ui.exe
```

Une fois compilé, nous pouvons le tester comme précédemment avec « `wine hello_ui.exe` » qui devrait vous afficher une magnifique boîte de message « `Hello World` ».

4. Intégration de la cross compilation à Autotools

a) Appels de configure

Pour que la cross-compilation se passe bien, il est nécessaire de passer plusieurs paramètres :

- `--target i686-pc-mingw32` : indique une compilation vers exécutables Windows
- `--host i686-pc-mingw32` : indique une compilation vers exécutables Windows
- `--build i386-linux` : indique que l'on compile depuis Linux (donc cross-compilation)
- `--prefix /usr/local/cross/i686-pc-mingw32` : indique que les `dlls/exe/lib` produits vont dans les sous dossiers de l'environnement de cross-compilation et sont récupérés depuis ces mêmes emplacements.

De plus, il faut définir un certain nombre de variables comme défini dans les scripts de cross-compilation de SDL (SDL site).

On aura donc le script **`cross-configure.sh`** que l'on appelera à la place de `configure` :

```
#!/bin/sh

#cross compilation base directory
XCC_INSTALL=/usr/local/cross
#where to store aclocal.m4 and co
ACLOCAL_FLAGS="-I $XCC_INSTALL/share/aclocal"
#host (linux) gcc excutable path
HOST_CC=`which gcc`
#compil for Windows
TARGET=i686-pc-mingw32
#files installation directory
PREFIX=$XCC_INSTALL/$TARGET
#include cross compilation directories to PATH
#to find the good executables
PATH="$XCC_INSTALL/bin:$XCC_INSTALL/$TARGET/bin:$PREFIX/bin:$PATH"
export PATH
#where to store cache for configure
cache=cross-config.cache
#exec configure with necessary parameters
#and optionally given by user ($*)
sh configure --cache-file="$cache" \
        --target=$TARGET --host=$TARGET --build=i386-linux \
        --prefix=$PREFIX \
        $*
```

```
status=$?
rm -f "$cache"
exit $status
```

b) Appels de make et make install

Pour ce qui est de make, le principe est le même, dans un fichier `cross-make.sh` :

```
#!/bin/sh

#cross compilation base directory
XCC_INSTALL=/usr/local/cross
#where to store aclocal.m4 and co
ACLOCAL_FLAGS="-I $XCC_INSTALL/share/aclocal"
#host (linux) gcc excutable path
HOST_CC=`which gcc`
#compil for Windows
TARGET=i686-pc-mingw32
#files installation directory
PREFIX=$XCC_INSTALL/$TARGET
#include cross compilation directories to PATH
#to find the good executables
PATH="$XCC_INSTALL/bin:$XCC_INSTALL/$TARGET/bin:$PREFIX/bin:$PATH"
export PATH
#exec make with parameters optionally given by user ($*)
exec make $*
```

On appelera **`./cross-make.sh`** à la place de `make`, et **`./cross-make.sh install`** à la place de `make install`

c) Autoconf : modification dans le configure.in

Pour ce qui est de `configure.in` :

- Dans tous les cas :
  - ajouter `AC_CANONICAL_SYSTEM` juste après `AM_INIT_AUTOMAKE`, afin d'activer la prise en charge de `--target`, `--host` et `--build`.
- Dans le cas d'un **`exécutable`** :
  - Ajouter `AC_EXEEXT` après `AC_CANONICAL_SYSTEM` afin de faire prendre en charge les extensions « `.exe` » automatiquement.
- Dans le cas d'une « **`dll`** » :
  - Ajouter `AC_LIBTOOL_WIN32_DLL` avant l'appel à `AM_PROG_LIBTOOL` ou `AC_PROG_LIBTOOL`

d) Automake : modification dans les Makefile.am

Pour ce qui est de `Makefile.am` :

- Dans le cas d'un **`exécutable`** : rien à faire de particulier
- Dans le cas d'une « **`dll`** »
  - Ajouter `lib`*`nom_library`*`_la_LDFLAGS = -no-undefined` : indique que tous les symboles doivent pouvoir être résolus pour que la compilation réussisse.
  - Ajouter ceci à la fin du fichier, pour fournir un règle création de « dll » :

```
%.dll : lib%.la
	$(DLLWRAP) --output-def $*.def --driver-name=$(CXX) -o $@ `ar t
.libs/lib$*.dll.a` $(LDFLAGS) $(LIBS) \
	$(DLLTOOL) --dllname $@ --def $*.def --output-lib .libs/lib$*.dll.a
```

e) Définir un MACRO suivant le type de système

On peut définir des macros `LINUX`, `WINDOWS`...en fonction du système sur/pour lequel on compile afin de faire de la compilation conditionnelle (`include`, `define`, `fonctions` différentes en fonction de l'OS).

```
dnl  Check the operating system
dnl
dnl  Define LIBPREFIX and LIBEXT so that dynamic libraries are
dnl  named libFOO.so in Linux and FOO.dll in Win32.
dnl
case "x${target_os}" in
  x)
      SYS=unknown
      ;;
  xlinux*)
      SYS=linux
      SYSGROUP=unix
      LIBEXT=".so"
      LIBPREFIX="lib"
      AC_DEFINE(LINUX,[],[Compile on Linux])
      ;;
  x*cygwin*)
      CYGWIN=Yes
      SYS=cygwin
      SYSGROUP=win32
      LIBPREFIX="lib"
      LIBEXT=".so"
      AC_DEFINE(WINDOWS,[],[Compile on Windows])
      ;;
  x*mingw*)
      SYS=mingw
      SYSGROUP=win32
      LIBPREFIX=""
      LIBEXT=".dll"
      AC_DEFINE(WINDOWS,[],[Compile on Windows])
      ;;
esac
```

VIII. Exemples d'utilisation des Autotools, Libtool et autres

1. Organisation de base d'un projet

L'organisation d'un projet classique `GNU` se compose de :

- le dossier du projet contenant les fichiers :
  - `INSTALL` : décrit la procédure d'installation
  - `NEWS` : décrit les nouveautés du projet
  - `README` : donne des informations sur le projet
  - `AUTHORS` : liste les auteurs du projet
  - `ChangeLog` : changements/corrections de bugs dans le projet
  - `COPYING` : licence du projet
- un dossier `src` contenant les sources pouvant être elles mêmes réparties en sous dossiers
- un dossier `doc` pour la documentation

Il y aura donc un `configure.in` dans le dossier du projet et un fichier `Makefile.am` par sous dossier.

2. Compilation d'une bibliothèque dynamique

a) Autoconf

i. Les bases pour compiler avec Libtool

Pour compiler avec `Libtool`, il faudra au moins mettre ceci dans le `configure.in` :

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

#version prérequise de autoconf
AC_PREREQ(2.59)
#initialisation du paquetage
AC_INIT([nom_projet_exemple], [version_majeure_point_mineure],
[email_mainteneur])
#chemin du dossier des sources
AC_CONFIG_SRCDIR([src/un_fichier_source])
#fichier de config autoheader
AM_CONFIG_HEADER(config.h)
#initialise automake
AM_INIT_AUTOMAKE([nom_projet],[version_majeure_point_mineure])

# Checks for programs.
#teste pour C++
AC_PROG_CXX
#teste pour C
AC_PROG_CC
#teste le préprocesseur C
AC_PROG_CPP
#teste pour libtool
AM_PROG_LIBTOOL
#teste l'existence de make et autres
AC_PROG_MAKE_SET
#teste l'existence de ranlib
AC_PROG_RANLIB
#teste l'existence de la command install
AC_PROG_INSTALL
#teste l'existence de ln -s
AC_PROG_LN_S

#makefile à générer
AC_CONFIG_FILES([Makefile
                 src/Makefile])
#génère-les
AC_OUTPUT
```

ii. Ajout d'une option --enable-debug pour compiler avec informations de débogage

La ligne suivante mise dans `configure.in` permet de compiler avec les options de déboggage :

```
AC_ARG_ENABLE(debug,
[  --enable-debug          compiles with debugging info],
[if test x$enableval = xyes; then
```

```
    CFLAGS="$CFLAGS -ggdb"
    CPPFLAGS="$CPPFLAGS -ggdb"
        fi])
```

iii. Vérification de la présence d'une bibliothèque

La ligne suivante permet de tester l'existence sur le système d'une bibliothèque. Si la bibliothèque s'appelle libshell, alors on écrira [shell] :

```
AC_CHECK_LIB([nom_biblio_sans_lib], [une_fonction])
```

iv. Vérification de la présence d'un fichier d'entête (ou plusieurs)

La ligne suivante permet de tester l'existence dun fichier d'entête C :

```
AC_CHECK_HEADERS([liste_à_espace_de_nom_fichier_h])
```

v. Vérification de la présence d'une fonction (ou plusieurs)

La ligne suivante permet de tester l'existance d'un fonction C :

```
AC_CHECK_FUNCS([liste_à_espace_de_nom_fonction])
```

vi. Vérification de la présence et des particularités de certaines fonctions communes

Les lignes suivantes permettent de tester les fonctionnalités et particularités de malloc, realloc et stat :

```
AC_FUNC_MALLOC
AC_FUNC_REALLOC
AC_FUNC_STAT
```

b) Automake

i. Les bases

Le fichier `Makefile.am` du dossier du projet contient la liste des sous dossiers :

```
SUBDIRS = src doc
```

Dans le dossier `src`, le fichier `Makefile.am` contiendra si par exemple, le dossier `src` contient un sous dossier `test`, contenant les sources réparties en principal et bibliothèque interne :

```
SUBDIRS = test

lib_LTLIBRARIES = libexemple.la
libexemple_la_SOURCES = fichier.c fichier2.c
libexemple_la_LIBADD = \
        test/libtest.la
```

Et enfin dans le sous dossier `test`, contenant une sous bibliothèque interne :

```
noinst_LTLIBRARIES = libtest.la
libtest_la_SOURCES = \
        fichier.c \
```

```
        fichier.h
```

ii. Inclure un « .h » dans les include à l'installation

Dans le fichier `Makefile.am` du dossier contenant le « `.h` » :

```
include_HEADERS = fichier.h
```

3. Compilation d'un exécutable utilisant le libexemple précédente

a) Bases

On suppose que votre exécutable utilisateur de `libexemple` n'a qu'un seul dossier de source, et donc qu'il contient un fichier `configure.in` et `Makefile.am` dans le dossier des sources.

b) Autoconf

Dans le fichier `configure.in` :

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

#version minimal d'autoconf utilisable
AC_PREREQ(2.59)
#initialisation
AC_INIT([nom_paquet],[version],[email])
#un fichier source
AC_CONFIG_SRCDIR([main.c])
#fichier config.h généré par autheader
AM_CONFIG_HEADER(config.h)
#initialise automake pour le projet
AM_INIT_AUTOMAKE([nom_paquet],[version])

#génére les variables de détermination du système hôte
AC_CANONICAL_SYSTEM
#gère les extensions des exécutables
AC_EXEEXT

#permet de choisir si on ajoute des informations de déboggage
#par l'option –enable-debug de configure
AC_ARG_ENABLE(debug,
[  --enable-debug          compiles with debugging info],
[if test x$enableval = xyes; then
    CFLAGS="$CFLAGS -ggdb"
    fi])

# Checks for programs.
#
#voir plus haut
AC_PROG_CC
AC_PROG_CXX
AC_PROG_CPP
AC_PROG_MAKE_SET
AC_PROG_INSTALL
AC_PROG_LN_S
AC_PROG_RANLIB
```

```
# Checks for libraries.
#teste si notre library est disponible
AC_CHECK_LIB([exemple],[une_fonction])

# Checks for header files.
#vérifie la présence des fichiers d'entêtes C
AC_HEADER_STDC
#et de string.h et unistd.h
AC_CHECK_HEADERS([string.h unistd.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.
#vérifie la présence de la fonction strstr
AC_CHECK_FUNCS([strstr])

#produit une fichier Makefile
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

c) Automake

Dans le `Makefile.am` :

```
bin_PROGRAMS = prog_exemple
mozhistory_SOURCES = main.c
```

4. Utiliser Autotools avec Lex et Yacc

a) Autoconf

Pour pouvoir compiler un programme utilisant au moins un fichier Flex/Bison ou Lex/Yacc, il faut contrôler la présence de `flex` et `bison`, de la library `fl`.

Dans le fichier `configure.in` :

```
AM_PROG_LEX
AC_PROG_YACC

AC_CHECK_LIB(fl, yywrap)
```

b) Automake

Si vous utilisez plusieurs fichiers « `.y` » ou « `.l` », il est nécessaire de préfixer les fonctions `yyparse` et autres de sorte qu'elles deviennent `yy_prefix_parse` et que chaque module puisse utiliser son propre parser sans interférer avec les autres modules (sans quoi la compilation réussi mais pas l'édition des liens). De plus il est important que si vous utilisez plusieurs grammaires, de les séparer dans des sous dossiers car flex/bison génèrent toujours des fichiers dont le nom est identique : `lex.yy.c` pour `lex/flex` et `y.tab.c` (et `y.tab.h`) pour `yacc/bison`.

Dans le fichier `Makefile.am` :

```
#indique le préfixe à utiliser pour distinguer deux parser
YACC=@YACC@ -d -p 'yy_un_prefix_'
LEX=@LEX@ -o 'lex.yy.c' -P 'yy_un_prefix_'

#indique les fichiers « .y » et « .l » à utiliser
#il est important de placer le fichier « .y » avant le « .l »
```

```
# car le .l nécessite le .c généré à partir du .y
#pour pouvoir se compiler
machin_SOURCES = \
        fichier_yacc.y fichier_lex.l

#fichiers à supprimer lors de make distclean
CLEANFILES = fichier_yacc.c fichier_lex.c
#fichier à ajouter parmi les sources
EXTRA_DIST = fichier_yacc.h
```

Bibliographie

La documentation officielle :

- http://www.gnu.org/manual/autoconf-2.13/html_mono/autoconf.html
- http://www.gnu.org/manual/automake-1.4/html_mono/automake.html

Autres :

- http://www.amath.washington.edu/~lf/tutorials/autoconf
- http://sourceware.org/autobook/autobook/autobook_toc.html
- http://www-igm.univ-mlv.fr/~dr/XPOSE/Breugnot/

Livres :

- Managing Projects with make, by Andrew Oram & Steve Talbott (O'Reilly)

  Un livre sur l'outil Make.

- Programming with GNU Software, by Mike Loukides & Andy Oram (O'Reilly)

  Un autre livre, sur l'ensemble des outils GNU