./configure; make; make install

Cet article, j'avais pensé l'appeler "autoconf/automake", mais en fait, si l'article porte sur cela, les exemples illustreront non seulement ces outils, mais plusieurs autres déjà traités dans d'autres numéros. L'article sera structuré de telle manière que chaque partie supposera un niveau de connaissance supérieur. La première partie ne nécessite que les connaissances de base en C.

Introduction

Quand j'écris un programme, ou que j'en télécharge un depuis Internet, c'est d'abord un simple programme avec quelques fichiers en .c et en .h, avec souvent un fichier *Makefile*. Ce fichier *Makefile* sert à simplifier la compilation: il suffit de taper `make`. Mais dans un projet un peu plus important, le fichier *Makefile* contient des paramètres à modifier "avec votre éditeur de texte favori", ainsi qu'on le lit couramment dans les fichiers *README*.
La première partie de l'article partira d'un simple programme helloworld sur lequel on ajoutera la couche permettant de le compiler avec `./configure; make; make install`.
La seconde partie, un peu courte, présentera un peu plus en détail `autoconf`, `automake` et les outils associés.
Dans la troisième partie, on compliquera l'exemple afin de voir comment s'en sortir dans les situations habituelles
Quatrième partie, l'exemple sera un gnome-exemple, et il faudra tenir compte des librairies GNOME.
Et pour finir, partie V, un peu d'internationalisation de programmes.

Partie I: démarrer avec autoconf/automake.

L'exemple d'abord

Partons d'un exemple simple: helloworld.c. Pour le compiler, c'est facile: `gcc helloworld.c -o helloworld`. Pour compiler ce programme simple, on n'a pas besoin de se compliquer la vie. Exemple suivant: helloworld, mais sur deux fichiers source.

main.c
```
#include <stdio.h>
#include "helloworld.h"
int main(int argc, char *argv[]) {
  hello();
  exit(0);
}
```

helloworld.h
```
void hello();
```

helloworld.c
```
#include <stdio.h>
void hello() {
  printf("Bonjour le monde\n");
}
```

Pour le compiler, voici trois solutions:

1.
```
gcc helloworld.c main.c -o helloworld
```
2.
```
gcc -c helloworld.c
gcc -c main.c
gcc main.o helloworld.o -o helloworld
```
3.
```
make
```

La solution 1 convient encore pour un si petit exemple. Mais dès qu'on arrivera à un projet plus conséquent, il faudra écrire un script de compilation. Idem pour la solution 2 qui simplifie un peu ce script.
La solution 3 est bien plus élégante, car plutôt que d'utiliser un script de compilation, make va utiliser un fichier *Makefile* qui gère les dépendances entre fichiers. Par exemple, si le fichier main.o est déjà à jour, il est inutile de le recompiler.

Eh bien, nous n'avons plus qu'à écrire ce fichier Makefile. Pour ceux qui m'auraient pris au sérieux, un bon bouquin sur Makefile fera l'affaire. Mais j'ai mieux à vous proposer. Faisons générer notre Makefile à un programme. Ce programme, il s'appelle `configure`. Ça y est, on s'y retrouve ?

./configure, cela crée un fichier *Makefile*. `make`, cela compile le programme en utilisant le *Makefile* généré ainsi. Et `make install`, c'est comme d'habitude. Et ce configure, d'où vient-il ? Il est généré grâce à autoconf. Et automake génère d'autres fichiers nécessaires au bon fonctionnement de configure.

Comment utiliser autoconf et automake ?

Avant de commencer, il semble que la plupart des projets contiennent leurs fichiers sources dans un sous-répertoire *src*. Nous allons faire de même en créant ce répertoire (`mkdir src`), puis en y déplaçant nos trois fichiers sources. J'appellerai *racine du projet* le répertoire qui contient *src*.

Maintenant, il faut un fichier de configuration qui s'appelle *configure.in*. La méthode la plus simple est d'utiliser l'outil `autoscan`, puis au fur et à mesure que le projet s'améliorera, on éditera le fichier *configure.in* à la main pour y ajouter les lignes supplémentaires.

Le programme `autoscan` génère un seul fichier *configure.scan*. On le lance depuis la racine du projet. Puis il faut le renommer en *configure.in*, et l'éditer. Voici mon *configure.in* après édition:

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

```
AC_INIT(src/helloworld.c)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(helloworld, 0.0.1)

dnl Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL

dnl Checks for libraries.

dnl Checks for header files.
AC_HEADER_STDC

dnl Checks for typedefs, structures, and compiler characteristics.

dnl Checks for library functions.

AC_OUTPUT([Makefile src/Makefile])
```

dnl, cela signifie commentaire. On est habitués à lire #, // ou /* */. Ici, c'est dnl. C'est comme ça et on ne discute pas.

Que manque-t-il maintenant ? On n'a pas dit quels fichiers compiler. Cela se fait dans le fichier *src/Makefile.am*. Sa structure dans notre cas simple est la suivante:

src/Makefile.am
```
bin_PROGRAMS = helloworld
helloworld_SOURCES = main.c helloworld.c helloworld.h
```

La première ligne indique dans `bin_PROGRAMS` le nom des binaires à générer. Dans notre cas, on ne générera que le binaire helloworld.
La seconde ligne indique les fichiers sources qu'on compilera pour générer le binaire helloworld. Notons au passage que le nom `helloworld_SOURCES` n'est pas choisi au hasard. C'est bien [nom du binaire]_SOURCES.
Si l'on voulait créer deux binaires, on aurait le src/Makefile.am suivant:

```
bin_PROGRAMS = bin1 bin2
bin1_SOURCES = source1.c
bin2_SOURCES = source2.c source2.h
```

Mais revenons à nos moutons, il ne reste plus qu'un fichier à créer. C'est le fichier *Makefile.am* à la racine du projet. Lui contiendra la liste des sous-répertoires dans lesquels on trouve les autres *Makefile.am*. On y trouve simplement:

Makefile.am
```
SUBDIRS=src
```

Notons aussi que la syntaxe des *Makefile.am* est la même pour tous les répertoires. Seulement, à la racine du projet, il n'y a pas de fichiers sources. Et dans le répertoire src, il n'y a pas de sous-répertoires. Sinon, on aurait bien sur mélangé les deux types de contenus de *Makefile.am* de ci-dessus.

Ça y est, on peut utiliser `autoconf` et `automake`. Commençons par exécuter `aclocal`, sans arguments. On génère ainsi un fichier *aclocal.m4* qui contient des macros nécessaires pour la suite. Puis `autoconf`, sans arguments non plus génère le tant attendu script *configure*. Ensuite, on peut utiliser `autoheader` pour créer le fichier de configuration *config.h.in* nécessaire pour *config.h*. Et pour terminer, `automake -a -c`.

Et là, ça se complique car il y a des messages d'erreurs et des warnings. Pour les messages d'erreur, sur ma configuration, ce n'est pas grave. En effet, on m'indique que certains fichiers n'ont pas pu être copiés, mais ils l'ont tout de même été. Pour les warnings, ils indiquent que des fichiers manquent. Chez moi, ce sont *NEWS*, *README*, *AUTHORS* et *ChangeLog*.

Il faut les créer, et un simple `touch NEWS` suffit. Il en est de même pour les trois autres fichiers. Cependant, il est plus sympathique de mettre un nom et un e-mail dans *AUTHORS*, et de mettre une première ligne dans *ChangeLog* afin de signaler quand le projet est parti. En ce qui concerne *NEWS*, j'ai l'habitude d'y mettre un résumé du *ChangeLog*. Ainsi, les développeurs iront voir *ChangeLog*, et les utilisateurs se contenteront de *NEWS* qui leur est plus parlant. Quant au *README*, chacun y met ce qu'il veut. J'utilise aussi un fichier *TODO* qui contient une sorte de mémento de ce qu'il me reste à faire, qui indique aux autres développeurs là où ils peuvent participer; et aux utilisateurs, il signale que la fonctionnalité qui leur manque tellement, elle est prévue.

A ce stade, tous les fichiers sont créés. Relançons `automake -a` pour tenir compte des fichiers qu'on a ajoutés. Et là, il n'y a plus de messages d'erreur ou de warning. Et c'est fini!

Pour la peine, exécutons `./configure`. Miracle, ça fait comme les autres programmes. Puis `make`. Et là, on voit les lignes de compilation, qui sont un peu plus complexes que celles qu'on aurait écrites dans un *Makefile* fait "à la main". Pas de surprise si l'on fait `make install`: on n'a pas les droits d'écriture dans /usr/local. Il faut être root pour cela. Le résultat serait d'installer le binaire *helloworld* dans */usr/local/bin*.

Partie II: utilisation courante d'autoconf/automake

`autoconf`, `automake` et `./configure` ne se limitent pas à générer des fichiers au début puis basta. A chaque modification du fichier *configure.in*, il faut exécuter `autoconf`. Et à chaque modification d'un fichier *Makefile.am*, il faut exécuter `automake -a` pour prendre en compte les changements.

Les changements sont de plusieurs natures. La plus évidente est le changement de version. Il suffit d'éditer le début de *configure.in*, puis de relancer `autoconf`. Parallèlement, il devient indésirable de coder le numéro de version du programme dans les sources. Il vaut mieux utiliser la constante `VERSION` qui contient toujours le numéro de version indiqué dans *configure.in*. De même pour la constante `PACKAGE` qui contient le nom du projet.

Un autre changement pourrait être un ajout d'une librairie. Dans ce cas, il faut ajouter un test dans *configure.in*, et signaler la librairie dans les *Makefile.am* concernés. Nous allons traiter ce cas dans la partie III.

Qu'apportent `autoconf`/`automake` par rapport à un simple *Makefile* ? Ces outils apportent bien sur les tests de l'environnement de compilation. Mais on peut aussi utiliser `make` avec des cibles autres que la cible par défaut (`make`), la cible install (`make install`) ou même la cible clean (`make clean`).

Pour nettoyer, il y a donc `make clean` qui se contente de supprimer les objets et binaires créés lors de la précédente compilation. Mais il y a aussi `make distclean` qui supprime tous les fichiers ne faisant pas partie du package. Ainsi, les *Makefile* seront supprimés puisque c'est le rôle de `configure` de créer ces fichiers. Les fichiers de cache seront aussi supprimés. C'est avec `make distclean` que l'on supprime tous les fichiers non essentiels au projet. Attention, avant de lancer le premier `make distclean`, faites une copie de sauvegarde de votre projet. En effet, si un fichier essentiel au projet n'est pas listé dans *configure.in* ou dans un *Makefile.am*, ou si une erreur s'est glissée quelque part dans ces fichiers, le fichier qui était essentiel risque d'être supprimé. Je le sais, ça m'est déjà arrivé. Il est toujours bon d'avoir une sauvegarde...

Pour faire le tarball contenant le projet, l'habitude était de faire un `tar cvzf projet-version.tar.gz`, précédé d'un `make clean` ou maintenant `make distclean`. Il y a plus simple: `make dist` créera ce tarball *projet-version.tar.gz*, dans notre cas *helloworld-0.0.1.tar.gz*. Et ce tarball contiendra tout ce qu'il faut et rien de plus.

Partie III: utilisons une librairie

Pour ne pas réinventer le monde, on utilise toujours le travail des autres, en l'occurrence via les librairies que les autres laissent à notre disposition. Nous allons en utiliser une pour voir comment adapter le projet à l'utilisation de cette librairie. J'ai choisi d'utiliser zlib car j'ai déjà écrit un article à ce sujet (LMAG 22), mais je n'avais pas mis d'exemple car cela aurais été peu illustratif et éventuellement rébarbatif. Alors maintenant, c'est l'occasion: notre programme helloworld va lire ce qu'il doit afficher, dans un fichier compressé. Un clone simple de zcat.

main.c
```
#include <stdio.h>
#include "helloworld.h"

int main(int argc, char *argv[]) {
  hello(argc, argv);
  exit(0);
}
```

helloworld.h
```
void hello(int argc, char *argv[]);
```

helloworld.c
```
#include <stdio.h>
#include <zlib.h>

void hello(int argc, char *argv[]) {
  char buffer[1024];
  gzFile *FH;

  FH = gzopen(argv[1], "rb");
  while(gzgets(FH, buffer, 1024) != Z_NULL) {
    printf("%s",buffer);
  }
  gzclose(FH);
}
```

Remarque importante: n'utilisez cet exemple qu'à titre d'illustration de l'article. Il est truffé de bugs. Ainsi, on ne vérifie pas la présence du fichier fourni en argument. L'argument lui-même n'est pas testé....

Le changement majeur dans le projet est l'utilisation de la zlib, ce qui se voit dans *helloworld.c*

Répercutons ce changement avec `automake` et `autoconf`. Commençons par éditer *configure.in*.
La première chose que l'on peut faire est de changer le numéro de version, et mettre 0.0.2 à la place de 0.0.1.
Puis il faut ajouter une ligne pour tester la présence de la librairie zlib:

```
dnl Checks for libraries.
AC_CHECK_LIB(z, gzopen)
```

Explication de texte: la macro `AC_CHECK_LIB` vérifie la présence de la librairie donnée en premier argument. `z` signifie libz. Si on avait voulu tester libtruc, on aurait mis truc. Et pour tester la présence de cette librairie, `AC_CHECK_LIB` a besoin d'un nom d'une fonction contenue dans cette librairie. J'ai choisi `gzopen`. J'airai pu prendre `gzclose` ou même `uncompress`.

Et c'est tout! C'est tout car on utilise une librairie simple à utiliser. Cela n'est pas le cas pour gnome ou glib qui sont plus complexes à mettre en oeuvre. Mais on y viendra plus tard.

Voyons si vous avez suivi: que dois-je faire maintenant ? Je dois lancer `autoconf` pour que la modification de *configure.in* soit prise en compte. Puis `./configure; make` comme d'habitude.

Ajoutons maintenant la librairie pthread. Aucune difficulté supplémentaire: je vous le laisse en exercice. Pierre Ficheux a d'ailleurs écrit un article sur cette librairie dans LinuxMag il y a déjà un certain temps. A vos archives!

Partie IV: Autoconf et automake avec gnome

Les fichiers à créer

`Autoconf` et `automake` avec gnome, ce sujet a déjà été abordé dans le numéro 10 de votre revue favorite. Je vais reprendre le sujet et le traiter plus en détail ici. D'autre part, je ne mettrai pas de listing d'exemple ici car un programme gnome est bien plus long qu'un helloworld, et cet article ne traite pas de la programmation gnome. Limitons-nous au cas général (et gardons gimp-1.4.0 pour quand gimp-1.2.0 sera sorti :-).

Les fichiers et répertoires à créer sont les suivants:

- *po/*
- *po/POTFILES.in*
- *intl/* et son contenu
- *configure.in*
- *Makefile.am*
- *src/Makefile.am*
- *acconfig.h*

po/ et intl/

Un programme gnome parle plusieurs langues. Il faut donc créer un répertoire *po/* et un répertoire *intl/*. C'est le programme `gettextize -c` qui va s'en occuper. Puis il faut mettre dans le fichier *po/POTFILES.in* tous les fichiers susceptibles de contenir des chaînes de caractères à traduire. Si on reprend l'exemple helloworld, *POTFILES.in* contiendrait:

POTFILES.in
```
src/main.c
src/helloworld.c
```

configure.in

Dans *configure.in*, il faut ajouter les lignes suivantes, avant la dernière ligne:

```
ALL_LINGUAS=""
AM_GNU_GETTEXT
```

Dans la dernière ligne, on ajoute les fichiers suivants aux fichiers existants du `AC_OUTPUT`: `intl/Makefile po/Makefile.in`. Voici ce que cela peut donner. J'inclus aussi les modifications liées à GNOME et j'ai laissé la ligne concernant zlib:

configure.in
```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(src/main.c)
AM_INIT_AUTOMAKE(gnomeprog, 0.1.0)
AM_ACLOCAL_INCLUDE(macros)
GNOME_INIT

dnl Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL
AC_PROG_MAKE_SET
AM_PROG_LIBTOOL
AC_ISC_POSIX

dnl Checks for libraries.
AC_CHECK_LIB(z, gzopen)

dnl Checks for header files.
AC_HEADER_STDC

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

dnl Checks for library functions.
GNOME_COMPILE_WARNINGS
GNOME_X_CHECKS

CFLAGS="$CFLAGS -Wall"

ALL_LINGUAS=""
AM_GNU_GETTEXT

localedir=${datadir}/locale

AC_SUBST(localedir)
AC_SUBST(CFLAGS)
AC_SUBST(CPPFLAGS)
AC_SUBST(LDFLAGS)

AC_OUTPUT([intl/Makefile macros/Makefile po/Makefile.in src/Makefile Makefile])
```

J'ai aussi ajouté l'option `-Wall` au compilateur, afin qu'il signale tout ce qui n'est pas propre. C'est aussi pour montrer comment ajouter des options au compilateur.

Makefile.am

Le *Makefile.am* de la racine ne bouge pas. Par contre, le *Makefile.am* du répertoire *src* doit être largement modifié. En voici un modèle:

src/Makefile.am
```
DEFS=@DEFS@ -DLOCALEDIR=\"${localedir}\"
INCLUDES = -I$(top_srcdir) -I$(includedir) \
           -DG_LOG_DOMAIN=\"gnomeprog\" \
           -DGNOMELOCALEDIR=\""$(datadir)/locale"\" \
           `gnome-config --cflags gnome` \
           -I../intl -I$(top_srcdir)/intl \
           -DPREFIX=\"${prefix}\" -DDATA_DIR=\"${datadir}\"

bin_PROGRAMS = gnomeprog

gnomeprog_SOURCES = main.c interface.c interface.h
gnomeprog_LDADD = @INTLLIBS@ @GTK_LIBS@ @GNOMEUI_LIBS@
gnomeprog_LDFLAGS = $(LDFLAGS) @GNOME_LIBDIR@ @GNOME_INCLUDEDIR@
```

Notez la ligne `INCLUDES` qui contient plusieurs variables intéressantes comme `PREFIX`.

acconfig.h

Un autre fichier à créer est *acconfig.h*. D'après le manuel de `gettext` (`info gettext`), on peut le récupérer dans les fichiers du package gettext. Mais en voici un exemplaire:

acconfig.h
```
#undef ENABLE_NLS
#undef HAVE_CATGETS
#undef HAVE_GETTEXT
#undef HAVE_LC_MESSAGES
#undef HAVE_STPCPY
#undef HAVE_LIBSM
#undef HAVE_POPT
#undef TARGET
```

Ce fichier contient des initialisations de variables qui iront dans le fichier *config.h*.

macros/

Et pour terminer, il manque le répertoire *macros/*. Ceci est l'aspect le moins documenté de la programmation extra gnome. Il semble qu'il faille créer ce répertoire, puis y copier les macros venant soit d'un répertoire contenant de la documentation sur gnome, soit d'un programme existant. Le premier cas n'étant pas toujours valable, rabattons-nous sur gnome-hello que l'on peut trouver ici:
ftp://ftp.gnome.org/pub/GNOME/stable/sources/GnomeHello/GnomeHello-0.1.tar.gz

Cette solution a aussi l'avantage de montrer un exemple simple de programme qui marche aux débutants.

Passons à l'action

Maintenant, passons à l'action avec les commandes que l'on commence à connaître:

```
autoconf
automake
./configure
make
```

Et cela passe comme une lettre à la poste. Que reste-t-il à faire ? Il reste à traduire la version française et à savoir que faire pour ajouter une nouvelle langue, ce qui ne manquera pas d'arriver au fur et à mesure que votre programme acquérira ses lettres de noblesses.

Partie V: internationalisons

Au stade ou nous en sommes, le programme est facilement internationalisable. Je supposerai que vous savez comment faire un programme qui supporte l'internationalisation avec gnome. J'entends par cela les initialisations et le fait de mettre `_("string")` plutôt que`"string"`. Sinon, retour à LinuxMag 10. Je n'aborderai que l'aspect qui concerne `autoconf`/`automake` ici.

Pour ajouter une nouvelle langue, il faut d'abord un fichier *xx.po*. Vous allez créer le *fr.po*, et d'autres gens sur internet vous enverrons leurs fichiers dans leur langue. Prenons le cas *fr.po*.

Première étape : copier le fichier *po/monprog.pot* en *fr.po*. Puis éditer de fichier. Le début est simple à comprendre: il faut compléter les trous. La suite est toujours faite ainsi:

```
msgid "a string"
msgstr ""
```

I faut mettre la traduction du texte `msgid` dans `msgstr`: cela devient:

```
msgid "a string"
msgstr "une chaîne de caractères"
```

Et ainsi de suite.

Seconde étape : le fichier *fr.po*, ou n'importe quelle autre traduction, il faut le placer dans le répertoire *po/*. Puis il faut éditer le fichier *configure.in* et ajouter la bonne langue dans la variable `ALL_LINGUAS`. Nous avions `ALL_LINGUAS=""`. Cela devient `ALL_LINGUAS="fr"`.

Puis, si un Allemand nous envoie son *de.po*, on mettra ce *de.po* dans le répertoire *po/*, puis on éditera à nouveau *configure.in* pour y mettre `ALL_LINGUAS="fr de"`.

Maintenant bien sur, la séquence que vous devez connaître par coeur maintenant:

```
autoconf
automake
./configure
make
```

Pour éditer les fichiers **.po*, n'importe quel éditeur de texte convient. Mais il faut tout de même indiquer que Emacs est bien adapté à cette tâche (ce n'est pas de la pub, je n'utilise pas Emacs!).

Conclusion

J'ai essayé d'aborder quelques points qui sont habituellement survolés car n'étant pas de la programmation pure. J'espère que cet article vous permettra à partir d'un simple programme de pouvoir utiliser `./configure; make; make install`, et si c'est un programme gnome, de pouvoir faire parler le programme en plusieurs langues. Je n'ai pas cherché à m'étendre sur certains points comme la zlib ou la programmation GNOME. D'autres articles sont là pour cela. Je n'ai pas abordé non plus d'autres outils nécessaires à la bonne gestion d'un projet, comme utiliser CVS, répondre rapidement aux mails ou encore l'intérêt de diffuser le projet le plus souvent possible. Cet article avait pour but d'aborder un maximum de ce qui interfère avec automake et autoconf. J'espère que j'y suis parvenu, et que ceux qui n'ont pas craqué avant la fin sauront aller aussi loin que leurs projets le nécessitent.

Yves Mettier
Ingénieur Systèmes et réseaux chez Admiral
Coauteur de gtktalog
http://gtktalog.sourceforge.net

Références:

Liens

autoconf
http://www.gnu.org/manual/autoconf/index.html

automake
http://www.gnu.org/manual/automake/index.html

gettext
http://www.gnu.org/manual/gettext/index.html

Programmes utilisés lors de la réalisation de l'article

GnomeHello
ftp://ftp.gnome.org/pub/GNOME/stable/sources/GnomeHello/GnomeHello-0.1.tar.gz

gtktalog
http://gtktalog.sourceforge.net