Langage C, compilation, aspects organisationnels

DUT 1ère année

Module SE

christophe.cerin@iutv.univ-paris13.fr

(version au 12 février 2009)

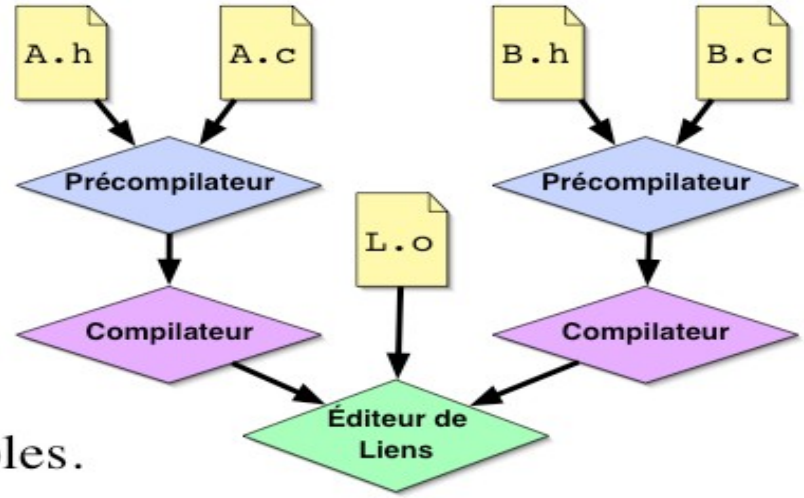
Introduction

- Le problème : vous devez gérer un projet de plusieurs milliers de lignes développé par plusieurs personnes (éventuellement depuis des systèmes différents)
- Très ambitieux (outils comme CVS, ANT…)
- On se contentera de quelques bases

Introduction

Compilation C

- Plusieurs phases.
  - Précompilation.
  - Compilation.
  - Édition de liens.
- Permet
  - Système flexible.
  - Maintenir outils simples.
  - Supporter langages différents.



Compilation

- Transformer un texte dans un langage haut niveau vers un exécutable
- GCC permet via l'option -S de générer uniquement l'assembleur (que l'on peut retoucher… mais c'est du « haut vol »)
- GCC permet via l'option -c de générer le code objet (un *fichier objet* est un fichier intermédiaire intervenant dans le processus de compilation. Ce fichier est censé contenir du code machine. Le fichier contient d'autres informations nécessaire à l'édition de liens (symboles) et lors du déboguage.

Code avec de l'assembleur (1/2)

- Ne cherchez pas à vouloir optimiser votre code avec de l'assembleur
- Voir l'exemple sous le lien http://www.lipn.fr/~cerin/SE/mysqrt_mylog2.c
- Il s'agit d'un code qui vise a reutiliser l'instruction assembleur SQRT et celui de calcul du logarithm
- Il faut d'abord penser à l'algorithme !

Code avec de l'assembleur (2/2)

- Le compilateur est capable de réaliser de manière automatique des optimisations élaborées : dépliage de boucle, lecture anticipée en mémoire
- Dans un TP, on vous demande d'insérer dans l'assembleur des instructions de lecture anticipée, puis de reprendre la compilation
- Prefetching : opération non bloquante qui va pré-charger dans le cache des données
- http://www.lipn.fr/~cerin/SE/f1.s est un exemple (si vous enlevez ces instructions, le code fonctionne quand même)

Exemple de Prefetching

```
for (i = 0; i < N; i++)
      ip = ip + a[i]*b[i];
```

(a)

```
for (i = 0; i < N; i++){
      fetch( &a[i+1]);
      fetch( &b[i+1]);
      ip = ip + a[i]*b[i];
}
```

(b)

```
for (i = 0; i < N; i+=4){
      fetch( &a[i+4]);
      fetch( &b[i+4]);
      ip = ip + a[i]*b[i];
      ip = ip + a[i+1]*b[i+1];
      ip = ip + a[i+2]*b[i+2];
      ip = ip + a[i+3]*b[i+3];
}
```

(c)

```
fetch( &ip);
fetch( &a[0]);
fetch( &b[0]);

for (i = 0; i < N-4; i+=4){
      fetch( &a[i+4]);
      fetch( &b[i+4]);
      ip = ip + a[i]  *b[i];
      ip = ip + a[i+1]*b[i+1];
      ip = ip + a[i+2]*b[i+2];
      ip = ip + a[i+3]*b[i+3];
}
for ( ; i < N; i++)
      ip = ip + a[i]*b[i];
```

(d)

Ici on a du dépliage de boucle : éviter des instructions 'branch' qui gèlent le pipeline

La distance de prefetching (de combien d'itérations on doit déplier) dépend des caractéristiques du cache.

Compilation séparée

- Concept inévitable !
- Idée : on a décrit des « bibliothèques » répondant a des spécifications. La résolution du problème passe par la compilation de l'ensemble de nos sources.
- Permet de ne pas recompiler TOUT un projet
- Rappel : dans la vraie vie, on n'écrit pas un seul beau et gros fichier !

Compilation séparée

- Et c'est là que les problèmes commencent : imaginons deux fichiers (parmi 1000 fichiers) avec une même fonction int foo(); mais qui ne réalisent pas exactement la même chose.
- Lorsque l'on compile le projet, quelle est la fonction foo() qui sera utilisée ? Est-on capable de détecter le problème ?

Compilation séparée

- Réponse : les compilateurs détectent que l'on utilise deux fois le même nom de fonction !
- Exemple de travail : le fichier f1.c contient une définition de fonction et f2.c contient le main() et qui fait un appel à la fonction du main()

Compilation séparée

```
/* f1.c */
#include<stdio.h>

int sum(int x, int y)
{
  return(x+y);
}
```

```
/* f2.c */
#include<stdio.h>

main(int argc, char *argv[], char *arge[])
{
  int x=3, y=4;

  printf("somme de %d et %d = %d\n",x,y,sum(x,y));
}
```

```
$ gcc f1.c f2.c
$ gcc -c f2.c
$ gcc -c f1.c
$ gcc f2.o f1.o
$ gcc f2.c f1.c
```

3 instructions

Remarque : mon compilateur est insensible à l'ordre des fichiers présentés... C'est pas toujours le cas !

Compilation

- Options d'optimisation -O4, …, -O1, -O => permet de faire des choses très poussées comme du dépliage de boucle…
- Option -Wall pour une liste détaillée des warnings

- Pour le déverminage compiler avec l'option -g puis utiliser gdb ou ddd qui est un outil graphique pour Linux (on peut mettre des points d'arrêt, visualiser des valeurs de variables… en pointant avec la souris)

Edition de lien

- C’est la dernière phase lancée par GCC
- Elle permet de compléter la résolution des références aux variables à l’aide de la table des symboles ; d’établir l’accès aux variables ; produire l’exécutable (à partir des fichiers objets de l’utilisateur, des bibliothèques de fonctions, des routines de lancement et de terminaison des programmes)

Edition de lien

- Il y a plusieurs formats de fichiers exécutables mais le plus utilisé est maintenant le format ELF pour le monde PC

```
cerin@weblipn:~$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.0, dynamically linked (uses shared libs),
not stripped
cerin@weblipn:~$

Ordinateur-de-Christophe-Cerin:~ cerin$ file a.out
a.out: Mach-O executable ppc
Ordinateur-de-Christophe-Cerin:~ cerin$
```

Utilitaires pour les fichiers binaires

- nm : affiche les symboles à partir de la table des symboles
- Principaux symboles : U (undefined dans le fichier mais défini dans une librairie dynamique), A (absolute), T (text section symbol), D (data section symbol), B (bss section symbol), C (common symbol), S (symbol in a section other than those above)
- Utilisation : repérer ce qui est défini localement ou de manière externe (dans une librairie externe)

```
         U _free
         U _mach_init_routine
00001ec0 T _main
00003014 d
   _pointer_to__darwin_gcc3_preregister_frame_info
00003010 d _pointer_to_objcInit
         U _printf
000030b8 S _receive_samples
00001f38 T _sum
```

Utilitaires pour les fichiers binaires

- **objdump** : donne l'assembleur correspondant au fichier binaire.
- On peut alors modifier l'assembleur… puis générer le nouveau binaire.
- Outil dans le monde Linux x86 : pas disponible sur Mac PPC (par exemple).

Edition de lien

- Les problèmes apparaissent lorsqu'il faut spécifier les librairies avec lesquelles lier le programme. Utiliser GCC avec des options particulières :
  - `-lpthread` pour lier avec la librairie de gestion des processus pthread (tout est collé !)
  - `-L/home/cerin/lib` pour ajouter un nouveau chemin ou GCC ira chercher des librairies. La variable d'environnement LD_LIBRARY_PATH sert aussi a cela.

```
$ gcc -v : mode verbose, pour de l'information sur
ce qui se passe à la compilation
```

Archivage

- Archive : collection de fichiers regroupés organisée pour retrouver les fichiers originels
- Par exemple, les bibliothèques statiques (elles contiennent « tout ce qu'il faut ») sont organisées comme archive
- Note : pour compiler de manière statique avec gcc : option -static. Permet de porter l'exécutable sur une machine avec éventuellement d'autres ou pas de librairies. C'est « self content »

Compilation statique

```
$ gcc -static f2.c f1.c
ld: can't locate file for: -lcrt0.o
```

Le problème est dû au fait qu'on ne dispose pas ici de la librairie statique correspondant au printf (librairie crt0)… on n'a qu'une version dynamique sur la machine de compilation !

Compilation statique (sur une architecture x86)

```
cerin@taipei:~$ gcc -static f1.c f2.c
cerin@taipei:~$ ./a.out
somme de 3 et 4 = 7
cerin@taipei:~$ ldd ./a.out
        not a dynamic executable
cerin@taipei:~$ file ./a.out
./a.out: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.2.0,
statically linked, not stripped
cerin@taipei:~$ nm ./a.out | grep " U"
cerin@taipei:~$
cerin@taipei:~$ ls -al a.out
-rwxr-xr-x  1 cerin ocad 490358 2005-11-02 12:49 a.out
cerin@taipei:~$
```

Création de l'archive

```
$ gcc -c f1.c f2.c
$ ar -cr f.a f1.o f2.o
$ ls -al f*
-rw-r--r--  1 cerin  cerin    1384  2 Nov 13:00 f.a
-rw-r--r--  1 cerin  cerin      60 28 Oct 16:44 f1.c
-rw-r--r--  1 cerin  cerin     392  2 Nov 12:59 f1.o
-rw-r--r--  1 cerin  cerin     159  2 Nov 12:43 f2.c
-rw-r--r--  1 cerin  cerin     716  2 Nov 12:59 f2.o

Utilisation de l'archive :
$ gcc f3.c -L./ -lf
```

Bibliothèque partagée

Les bibliothèques partagées ne sont chargées que lorsqu'un programme les appellent, ce qui permet de gagner de la place en mémoire. Pour savoir à quelles bibliothèques partagées sont liées un programme, utiliser la commande `ldd`. Par exemple pour le programme `/usr/bin/X11/xterm` :
`ldd /usr/bin/X11/xterm` Le résultat devrait ressembler à ca :

```
libXaw.so.6 => /usr/X11R6/lib/libXaw.so.6 (0x4000b000)
libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6 (0x40042000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6 (0x40054000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x40096000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x4009f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x400b4000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x400bf000)
libc.so.5 => /lib/libc.so.5 (0x4015e000)
```

La ligne `libc.so.5 => /lib/libc.so.5 (0x4015e000)` par exemple indique la bibliothèque C standard de version 5.

Bibliothèque partagée

L'éditeur de liens dynamiques `ld` recherchera les bibliothèques dans les répertoires `/lib`, `/usr/lib`, et dans les répertoires spécifiés dans le fichier `/etc/ld.so.conf` (vérifier le nom svp) et dans la variable d'environement `$LD_LIBRARY_PATH` (utile pour ses propres bibliothèques confidentielles, par exemple). I

Il faut par ailleurs invoquer la commande `ldconfig` après chaque modification du fichier `/etc/ld.so.conf`.

Les bibliothèques partagées ont une extension *.so* et se trouvent en général dans le répertoire `/lib`.

L’utilitaire make

- Gestionnaire de projets (très général ; pas limité à la compilation de fichiers C)
- Permet de spécifier :
  - Dépendances d’un fichier
  - Comment les compiler
- Make décide des différences phases de la compilation

L'outil make et les fichiers Makefile

Makefile – bases

- Fichier qui décrit la compilation
- Syntaxe:

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

Un fichier dépend de tout fichier utilisé pendant sa génération

- Exemple:

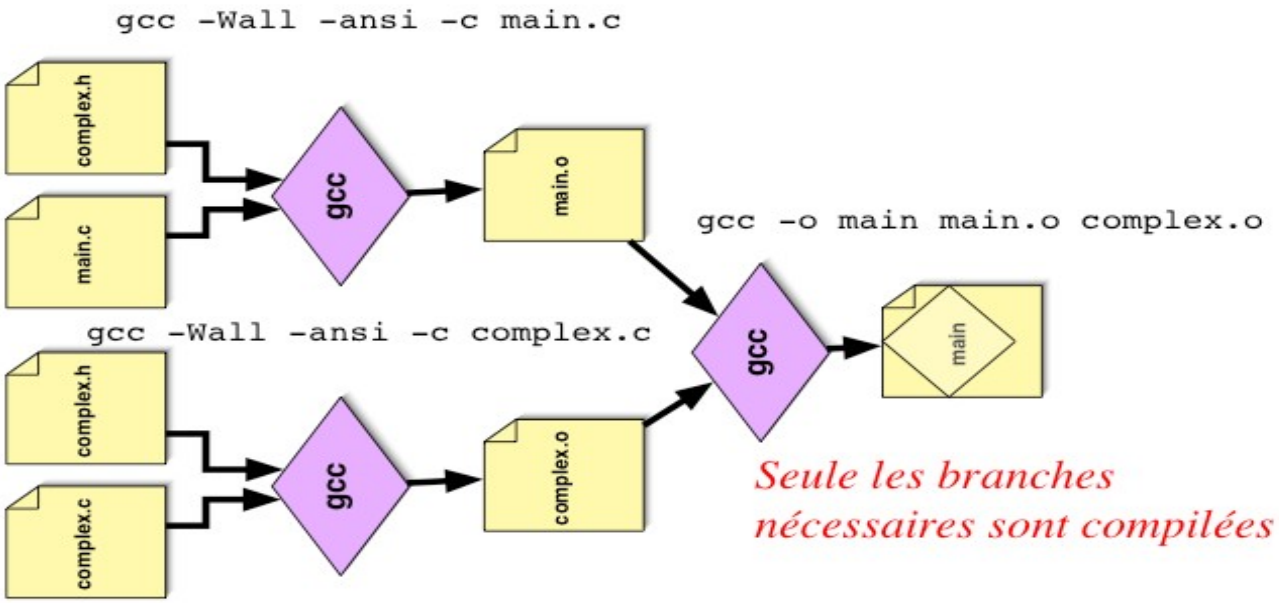
Objet principal (1er)

```
main: main.o complex.o
  gcc -o main main.o complex.o
main.o: main.c complex.h
  gcc -Wall -ansi -c main.c
complex.o: complex.c complex.h
  gcc -Wall -ansi -c complex.c
```

Exécution du ficher Makefile

Make – exécution

- Si on tape `make main`



Variables dans les fichiers Makefile

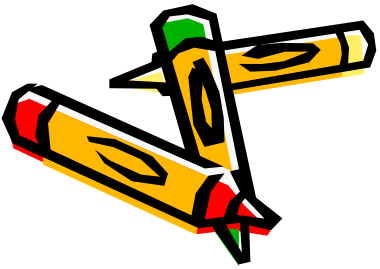
- Idée : du texte est répété plusieurs fois dans le Makefile => grouper le texte commun et le ranger dans des variables

Variables prédéfinies

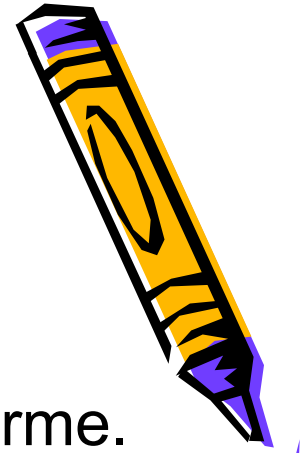
```
CC = gcc
CFLAGS = -Wall -O4
TARGET = main
OBJS = main.o complex.o

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $TARGET $(OBJS)
```

Pour aller plus loin…

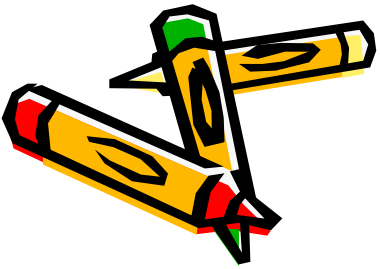
- Dans la documentation de make, vous allez trouver comment fabriquer des règles générales, par exemple pour produire un .o à partir d'un .c
- Notations spéciales : %, $< (cible), $@ (nom complet du fichier en cours d'examen)
- Note : make n'est pas *cross-platform*. C'est un outil du monde Unix.



Pour aller plus loin…



- L'outil ant (http://ant.apache.org) est cross-platforme.
- Les buildfiles (c.à.d les Makefiles) sont écrits en XML => très verbeux
- On retrouve des notions de make, par exemple les cibles => on peut avoir une cible pour compiler, pour installer, pour créer une distribution
- on peut spécifier la dépendance entre cibles (si une condition/propriété est vérifiée)



Example Buildfile

```
<project name="MyProject" default="dist" basedir=".">
   <description>
      simple example build file
   </description>
   <!-- set global properties for this build -->
   <property name="src" location="src"/>
   <property name="build" location="build"/>
   <property name="dist"  location="dist"/>

   <target name="init">
      <!-- Create the time stamp -->
      <tstamp/>
      <!-- Create the build directory structure used by compile -->
      <mkdir dir="${build}"/>
   </target>
   <target name="compile" depends="init" description="compile the source " >
      <!-- Compile the java code from ${src} into ${build} -->
      <javac srcdir="${src}" destdir="${build}"/>
   </target>
   <target name="dist" depends="compile" description="generate the distribution" >
      <!-- Create the distribution directory -->
      <mkdir dir="${dist}/lib"/>
      <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
      <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
   </target>
   <target name="clean" description="clean up" >
      <!-- Delete the ${build} and ${dist} directory trees -->
      <delete dir="${build}"/>
      <delete dir="${dist}"/>
   </target>
</project>
```

En couleur, exemples d'attributs - syntaxe « à la » HTML (balises)