- Langage C avancé : Séance 6

- Outils d’aide à la compilation
- Outils d'aide à la correction d'erreurs

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Outils d’aide à la compilation

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Comme exemple, on utilisera le code source réparti dans les différents sous-répertoires et fichiers du répertoire Code:

main.c:	contient le programme principal qui utilise les types vecteur et matrice pour résoudre un système linéaire (par la méthode du gradient conjugué), inclut les fichiers matrice.h, vecteur.h et operations.h
matrice.c:	contient le code source du type matrice et les fonctions associées, inclut matrice.h
vecteur.c:	contient le code source du type matrice et les fonctions associées, inclut vecteur.h
operations.c:	contient les fonctions qui calculent des opérations avec des vecteurs et matrices (exemple: produit matrice vecteur), inclut matrice.h, vecteur.h et operations.h

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

On compile les fichiers de code source, pour produire un exécutable code.exe

- soit en une étape:

```
gcc main.c types/vecteur.c types/matrice.c
    types/operations.c -o code.exe
```

- soit en plusieurs étapes:

```
gcc -I types -c types/vecteur.c -o types/vecteur.o
gcc -I types -c types/matrice.c -o types/matrice.o
gcc -I types -c types/operations.c -o types/operations.o
gcc -I types -c main.c -o main.o
gcc main.o types/vecteur.o types/matrice.o
    types/operations.o -o code.exe
```

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Pour ne pas devoir taper toutes les commandes, on fournit un fichier de commandes shell : build.sh

Pour tout (re)compiler, il suffira de taper dans un terminal:

```
./build.sh
```

Inconvénients de ce type de compilation

- Tout est (re)compilé même si un seul des fichiers est modifié.
- Si on change les options de compilation (debug/release/…) ou le compilateur utilisé (gcc, clang, icx, cl.exe, …), il faut modifier toutes les commandes.
- Les fichiers binaires (intermédiaires et le code final) est mélangé au code source.
- Pour ajouter un nouveau fichier source, ou enlever un ancien au code, il faut modifier avec soin les commandes de compilation.

Il est possible d'adapter le fichier build.sh pour éviter les inconvénients ci-dessus, mais cela peut être complexe pour éviter les oublis et les erreurs dans la procédure de compilation.

On préfère presque toujours utiliser les outils d'aide à la compilation présentés dans la suite.

Compilation partielle

On suppose que le code a déjà été compilé et certains des fichiers sont modifiés, par exemple:

1. Dans le fichier operations.c, on modifie l'intérieur de la fonction gradconj (mais pas ses paramètres).
   Le seul fichier à recompiler est operations.c **et il faut refaire l'édition de liens finale**:

```
~~gcc -I types -c types/vecteur.c -o types/vecteur.o~~          (inutile)
~~gcc -I types -c types/matrice.c -o types/matrice.o~~          (inutile)
gcc -I types -c types/operations.c -o types/operations.o
~~gcc -I types -c main.c -o main.o~~                            (inutile)
gcc main.o types/vecteur.o types/matrice.o types/operations.o
        -o code.exe
```

2. On modifie la fonction printVecteur en lui ajoutant un paramètre (pour pouvoir au choix, écrire dans un fichier ou afficher à l'écran).
   Les fichiers à recompiler sont ceux qui définissent ou utilisent la fonction printVecteur

```
gcc -I types -c types/vecteur.c -o types/vecteur.o
gcc -I types -c types/matrice.c -o types/matrice.o          (inutile)
gcc -I types -c types/operations.c -o types/operations.o    (inutile)
gcc -I types -c main.c -o main.o                            (inutile)
gcc main.o types/vecteur.o types/matrice.o types/operations.o
        -o code.exe
```

Dépendances entre les fichiers

Dans cet exemple, on a donc les règles suivantes:

- code.exe dépend de main.o, vecteur.o, matrice.o et operations.o
- main.o dépend directement de main.c et, indirectement, de operations.h, matrice.h et vecteur.h (inclus dans main.c)
- operations.o depend directement de operations.c et, indirectement de operations.h (inclus dans operations.c), vecteur.h et matrice.h (inclus dans operations.h)
- …

Dépendances entre les fichiers : remarques

1. On ne tient pas compte des dépendances entre les fichiers système (stdio.h, string.h …) et les fichiers du code source

   Parce que les fichiers système ne sont modifiés que quand on change de compilateur, ou de version du système d'exploitation (linux, windows, …), etc.

2. Un fichier .c peut directement dépendre d'un fichier .h ou indirectement, et parfois les deux.

   Exemple dans le code : main.c dépend directement de vecteur.h et aussi indirectement par l'itermédiaire de operations.h

Pour éviter les problèmes d'inclusions multiples

Si un fichier .h est inclus plusieurs fois dans un même fichier .c, certains compilateurs refusent de le compiler, parce que les structures C et les fonctions définies dans le .h sont déclarées plusieurs fois.

C'est pour cela que tous les fichiers .h doivent être encadrés par des « include guards »:

f.h

```
#ifndef XXX
#define XXX
 void f(int x);
 …
#endif
```

Où XXX est un mot quelconque mais unique entre les différents fichiers .h inclus par le code source

Inclusions multiples sans les « include guards »

Exemple :

f.h

```
struct S1 {…};
```

g.h

```
#include "f.h"
struct S2 {…};
```

Fichier d'origine (main.c)

```
#include "f.h"
#include "g.h"

int main()
{ … }
```

Code intermédiaire généré par préprocesseur

```
struct S1 {…};
struct S1 {…};
struct S2 {…};

int main()
{ … }
```

Erreur : la structure S1 est définie 2 fois !!!

Inclusions multiples avec les « include guards »

Exemple :

f.h

```
#ifndef _F_H
#define _F_H
struct S1 {…};
#endif
```

g.h

```
#ifndef _G_H
#define _G_H
#include "f.h"
struct S2 {…};
#endif
```

Fichier d'origine (main.c)

```
#include "f.h"
#include "g.h"

int main()
{ … }
```

Code intermédiaire généré par préprocesseur

```
struct S1 {…};
struct S2 {…};

int main()
{ … }
```

La structure S1 est définie une seule fois

Pour déterminer les compilations utiles

Les outils qui suivent vont utiliser les dates de modification des fichiers pour déterminer quels sont les compilations nécessaires.
Si le fichier main.c contient

```
#include "f.h"
#include "g.h"

int main()
{ … }
```

Le fichier main.c sera (re)compilé si:

- Le fichier binaire main.o n'existe pas
- Le fichier binaire main.o existe mais est plus ancien que main.c
- Un des fichiers f.h ou g.h est plus récent que main.c
- Un fichier inclus par f.h ou g.h est plus récent que main.c

Outil Make

Outil d’aide à la compilation : make

L’outil utilisé ici est la commande make (il en existe d’autres: par exemple nmake sous windows)

On crée un fichier dont le nom est Makefile et où sont décrites les règles dépendances et de construction.

La commande make utilise ce fichier. Si on donne un autre nom au fichier de règles, il faut taper

```
make -f nom_du_fichier_de_règles
```

Syntaxe du fichier Makefile

Le fichier Makefile contient une liste de règles de la forme (chaque commande est précédée d'une tabulation):

```
Cible: Liste_de_dépendances
	Commande1
	Commande2
	…
```

Make détermine qu'il faut exécuter commande1, commande2,… si « Cible » n'existe pas ou au moins une dépendance dans « Liste_de_dépendances » est plus récente que la cible

Quand la cible d'une règle ne correspond à aucun fichier et que les commandes ne créent pas de fichier ayant le nom « cible », la règle est toujours exécutée

Syntaxe « make » (2)

Quand on tape « make », l'outil examine les dépendances de la première cible.

Si une dépendance n'est pas un fichier existant ou est plus récente que la cible, make cherche dans le fichier Makefile si une autre règle existe dont la cible est la dépendance.

> Si aucune règle n'est disponible, make affiche un message d'erreur

Quand on tape « make nom_de_cible », l'outil examine les dépendances de la cible « nom_de_cible »

Exemple

Dans le code source pris comme exemple, le fichier Makefile peut s'écrire :

```
Code.exe: main.o types/vecteur.o types/matrice.o \
          types/operations.o
    gcc main.o types/vecteur.o types/matrice.o \
          types/operations.o -o code.exe
main.o: main.c vecteur.h matrice.h
    gcc -c main.c -o main.o
matrice.o: matrice.c matrice.h
    gcc -c matrice.c -o matrice.o
…
```

Fonctionnalités supplémentaires

Plusieurs fonctionnalités permettent de simplifier ou d’écrire un makefile plus générique

- Variables utilisateur
- Commandes
- Variables spéciales
- Modèles de règle

Variables utilisateur

Si un mot est utilisé plusieurs fois dans plusieurs règles et/ou plusieurs commandes, on peut définir une variable
Exemple (la variable Code contient le nom de l'exécutable construit par make) :

```
Code = code.exe
${Code}: main.o …
	gcc main.o … -o ${Code}
clean:
	rm -f ${Code} *.o
```

Fonctions standards dans un makefile

Make propose des fonctions utilitaires, deux exemples

wildcard : crée une liste de tous les fichiers suivant un modèle

exemple : liste des fichiers C dans le répertoire du fichier Makefile

```
sources = $(wildcard *.c)
```

patsubst : à partir d'une liste de mots, crée une seconde liste où les mots de la première liste sont transformés avec une règle de substitution

exemple : on crée une liste de fichiers binaires (terminés par .o) à partir de la liste dans sources :

```
binaires = $(patsubst %.o, %.c, ${sources})
```

Fonctions standards dans un makefile (2)

On peut ensuite compiler un code qui contiendra tous les binaires correspondant aux fichiers sources C :

```
Exec1 : ${binaires}
    gcc ${binaires} -o Exec1
```

Variables spéciales

Ces variables sont automatiquement définies pour chaque règle et doivent être utilisée seulement dans les commandes associées à cette règle, elles ne sont pas modifiables.

$@ contient la cible d'une règle

$< contient l'ensemble des dépendances

$^ contient la première dépendance

Exemples représentent

```
Exec1 : main.o A.o B.o
    gcc $< -o $@
```

```
Exec1 : main.o A.o B.o
    gcc main.o A.o B.o -o Exec1
```

```
A.o : A.c A.h
    gcc -c $^ -o $@
```

```
A.o : A.c A.h
    gcc -c A.c -o A.o
```

Modèles de règle

Plusieurs règles peuvent être remplacées par un modèle de règle si les cibles et dépendances ont une forme similaire

Exemples :

```
%.pdf: %.tex
    pdflatex @<
```

La règle s'applique pour tous les noms de fichier se terminant en .tex et construisent le fichier dont le nom se termine en .pdf

```
%.o : %.c
    gcc -c $^ -o $@
```

Aide du compilateur

Les compilateurs peuvent souvent aider à écrire un makefile en générant les règles de dépendance. En utilisant gcc, la commande:

```
gcc -I types -MM main.c types/operations.c \
          types/vecteur.c types/matrice.c
```

affiche à l'écran:

```
main.o: main.c types/vecteur.h types/matrice.h types/operations.h \
 types/vecteur.h types/matrice.h
operations.o: types/operations.c types/operations.h types/vecteur.h \
 types/matrice.h
vecteur.o: types/vecteur.c types/vecteur.h
matrice.o: types/matrice.c types/matrice.h
```

On peut utiliser ceci comme point de départ d'un nouveau Makefile.

Pour plus de fonctionnalités de make, voir le manuel de référence (de la version GNU) de make :

https://www.gnu.org/software/make/manual/make.pdf

ou

https://www.gnu.org/software/make/manual/html_node/index.html

Exercice : dans le répertoire Code_Make se trouve une copie du code source d'origine. Ecrivez un fichier Makefile et utilisez le pour compiler le code.

Outils de plus haut niveau

L’écriture des fichiers Makefile n’est pas simple dans le cas de codes source complexes.
D’autre part, dans les exemples précédents, on a mis « en dur » dans les Makefiles des informations système, par exemple le nom du compilateur.

Des outils supplémentaires ont été proposés pour utiliser le système de compilation sur plusieurs machines, systèmes et avec des compilateurs différents.

Si on se trouve dans un environnement machine / système / compilateur spécifique, ces outils génèrent des fichiers Makefile adaptés.

Outils autools : automake, autoconf, ...

Outils autotools

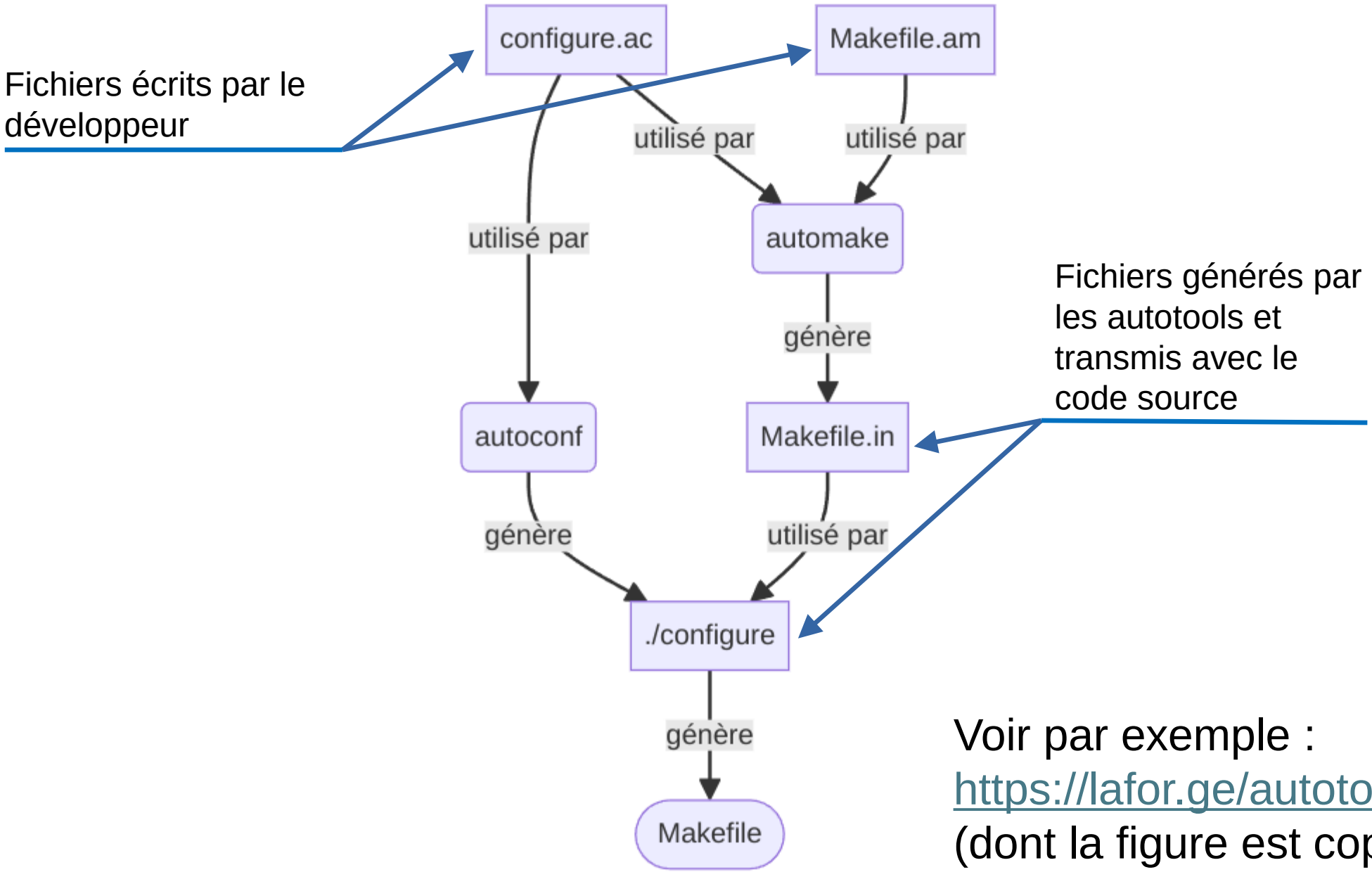
La famille d’outils m4, autoconf, automake, … ont été conçus pour générer les Makefiles et des outils de configuration.

Le processus complet se fait en 2 parties:

- *une première partie à faire par ceux/celles qui écrivent le code source*
- *une seconde partie à faire par ceux/celles qui compilent le code source et installent le code compilé.*

Un fichier principal de configuration « configure » est créé par les autotools, il teste le compilateur et les autres outils pour savoir s’ils fonctionnent, quelles sont leur options, ...

Outils autotools



Voir par exemple :
https://lafor.ge/autotools/
(dont la figure est copiée ici)

Exemple autotools

Dans le répertoire Code_Autotools/src se trouvent le code source et les fichiers Makefile.am et configure.ac (et des fichiers qui décrivent le code, les auteurs, etc.).
Les développeurs tapent les commandes

```
aclocal
automake --add-missing
autoconf
```

Puis envoient le répertoire src aux installateurs/utilisateurs.

Exemple autotools

Ceux/celles qui reçoivent le code source (répertoire src) installent en tapant les commandes :

```
mkdir <build_dir>
cd <build_dir>

<source_dir>/configure --prefix <install_dir>
make
make install
```

Exemple concret de code géré par les autotools : compilateurs de la suite GNU (gcc, g++, gfortran, etc.)

Outils autotools

Les autotools sont utilisés par un grand nombre de codes C (et C++) existants. Il est donc utile de connaitre ces outils.

Pour plus d'information, les manuels de ces outils sont disponibles :

https://www.gnu.org/software/autoconf/manual/autoconf.pdf

https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.72/html_node/index.html

On conseille cependant d'utiliser l'outil présenté aux pages suivantes, plus récent et plus puissant.

Outil CMake

CMake

cmake est un outil qui génère des procédures de configuration et de compilation.
Pour utiliser cmake, il faut écrire un fichier CMakeLists.txt dans tous les répertoires qui contiennent des fichiers sources.

cmake est un « vrai » langage de programmation avec :

- des boucles (foreach … endforeach),
- des instructions conditionnelles (if … else … endif)
- des variables définies par cmake ou par l'utilisateur,
- des fonctions,
- des générateurs (pour créer des makefiles ou d'autres systèmes de compilation, par exemple des projets visual studio (sous Windows), ou des projets Xcode (sous macOS))

Cmake Exemple

Dans le répertoire Code_Cmake/src, on trouvera une copie du code source.
On crée un fichier CMakeLists.txt dans src contenant les lignes suivantes :

```
cmake_minimum_required(VERSION 3.10)
project(ExempleCmake C)
include_directories(types)
add_executable(code.exe
   main.c
   types/vecteur.c
   types/matrice.c
   types/operations.c)
install(TARGETS code.exe DESTINATION .)
```

Cmake Exemple

Dans le répertoire Code_Cmake, tapez les commande suivantes:

```
cmake -S src -B build_dir -DCMAKE_INSTALL_PREFIX=install_dir
cmake --build build_dir
cmake --install build_dir
```

1. La première commande crée le répertoire build_dir et génère, dans ce répertoire, les « makefiles » ou correspondants dans d'autres systèmes de compilation (visual studio ou Xcode)

   c'est l'équivalent de automake-autoconf-configure dans l'exemple avec les autotools.
2. La seconde commande lance la compilation dans le répertoire build_dir
3. La troisième copie le fichier exécutable dans install_dir

Avantages de Cmake

- Cmake possède un nombre important de fichiers de configuration qui facilitent
  - L'utilisation de différents langages de programmation (C, C++, Java, Python, Cuda, Fortran, …)
  - L'utilisation de différents systèmes de compilations (makefiles, ninja, projet visual studio, Xcode, …)
  - L'utilisation de différents compilateurs
  - La connexion avec des librairies externes
- On peut facilement générer des versions optimisées, debug, etc.
- On peut séparer proprement le code source, les fichiers intermédiaires produits pendant la compilation, et les fichiers à fournir aux utilisateurs. Et laisser le code source inchangé à la fin de la compilation.

Inconvénient de Cmake

Cmake facilite beaucoup la gestion des étapes de compilation, mais il est parfois difficile de comprendre ce qui se passe quand on est face à des compilations qui se passent mal.
Il est possible, mais pas toujours facile, de voir quelles sont les appels exacts aux compilateurs.

Cmake Exemple 2

L’exemple 2 avec cmake illustre la façon habituelle de procéder : mettre un fichier CMakeLists.txt dans chaque répertoire où se trouve du code source.

On illustre aussi l’utilisation de libraries et de boucles.

Examiner le répertoire src et les fichiers CMakeLists.txt qui s’y trouvent.

Outils d’aide à la compilation

Erreurs dans un code C (1)

Un code source peut contenir des erreurs qui provoquent des messages d'avertissement ou erreurs de compilation, des arrêts prématurés de l'exécution du code binaire, ou une exécution complète mais qui fournit des résultats faux.

On a déjà vu que le compilateur propose des options qui affichent des messages d'avertissement sur du code dont la syntaxe est correcte mais qui risquent des arrêts prématurés, de produire d'autres résultats que ceux attendus: -Wall –Wextra …
Ces options repèrent certaines des erreurs les plus habituelles mais ne garantissent pas l'absence d'erreurs.

Il est, de toute façon, fortement conseillé d'utiliser ces options.

Erreurs et messages d'avertissement de compilation dans un code C (1)

- Les erreurs de compilation sont dues à des erreurs de syntaxe C, des fonctions ou des variables globales utilisées mais non définies, des fonctions ou des variables globales définies plusieurs fois, etc.
  Ce sont les plus faciles à détecter et à corriger: le compilateur refuse de générer un code binaire en affichant des messages qui donnent en général assez d'information pour corriger ce type d'erreurs.

Les messages d'avertissements et d'erreurs sont différents suivant les compilateurs, la version du langage C, etc.
Pour les codes destinés à être diffusés, il peut être intéressant de compiler le code source avec plusieurs compilateurs et sur plusieurs systèmes différents.

Erreurs et messages d'avertissement de compilation dans un code C (2)

- Dans le sous-répertoire erreurs_compilation/err1, compiler en tapant :

```
gcc main.c –o code
```

Corriger les erreurs éventuelles jusqu'à ce que le code compile.
Exécutez le code binaire.

- Recompiler avec les options de vérification, en tapant :

```
gcc –Wall –Wextra main.c –o code
```

Modifier le code source pour tenir compte des messages d'avertissement.
Exécutez le code binaire

Erreurs et messages d'avertissement de compilation dans un code C (3)

- Premier enseignement : il peut y avoir plusieurs façons de corriger les erreurs

Exemple:

```
double f(int n) {
  …
}

int main() {
  double x = f(10, 3.4);
}
```

?

```
double f(int n, double x) {
  …
}

int main() {
  double x = f(10, 3.4);
}
```

```
double f(int n) {
  …
}

int main() {
  double x = f(10);
}
```

Les 2 modifications donnent une syntaxe correctes.
Pour choisir, il faut comprendre ce qu'est censé faire le code

Erreurs et messages d'avertissement de compilation dans un code C (4)

- Deuxième enseignement : avoir une syntaxe correcte ne suffit pas, le code peut se compiler (éventuellement avec des messages d’avertissement) mais s’exécuter en fournissant des résultats faux ou en s’arrêtant sur une erreur.

Donc, il faut utiliser toutes les options vérifications à la compilation et tenir compte des messages d’avertissement.

On verra souvent qu’il faut aussi faire des vérifications à l’exécution (voir plus loin).

Erreurs dans l'appel des fonctions ou dans l'utilisation des variables globales

- Dans le sous-répertoire erreurs_compilation/err2, compiler en tapant :

```
gcc main.c f.c –o code
```

Le compilateur refuse de créer un binaire exécutable parce que la variable globale m est définie 2 fois. Modifier f.c en ajoutant « extern » devant la définition de m dans f.c:

```
double m;
```

```
extern double m;
```

Le message du compilateur signifie que la variable globale m ne peut pas être définie séparément dans plusieurs fichiers, il faut la définir dans un seul ficher et utiliser « extern » dans les autres.

Erreurs dans l'appel des fonctions ou l'utilisation des variables globales

- Recompiler et exécuter le code en tapant ./code
  Le code s'exécute mais affiche des valeurs incohérentes, sur ma machine:
  > f: x = 0 m = 2.47033e-323
  > f: r = 0
  > main: f(10) = 9

- Le problème vient de ce fait le compilateur quand on tape:

```
gcc main.c f.c –o code
```

  - Compilation séparée de main.c (fichier caché main.o)
  - Compilation séparée de f.c (fichier caché f.o)
  - Rassemblement des fichiers main.o et f.o dans l'exécutable code

Le compilateur vérifie que toutes les fonctions et les variables globales utilisées sont définies une et une seule fois, mais pas que les utilisations des fonctions ou variables globales sont conformes à leur définition.

Exploration des fichiers binaires

Pour voir ce qui se passe, recompiler les fichiers en plusieurs étapes:

```
gcc –c main.c
gcc -c f.c
gcc main.o f.o –o code
```

Et utiliser la commande objdump qui montre le contenu des fichiers compilés (l'exécutable final ou les fichiers objets .o):

```
objdump –t <nom de fichier>
```

On peut aussi utiliser la commande nm qui affiche les mêmes informations sous une forme différente:

```
nm <nom de fichier>
```

Exploration des fichiers binaires

```
objdump -t main.o
```

```
…
0000000000000000 g     O .bss   0000000000000004 m
0000000000000000 g     F .text  0000000000000049 main
0000000000000000         *UND*  0000000000000000 f
0000000000000000         *UND*  0000000000000000 printf
…
```

```
objdump -t f.o
```

```
…
0000000000000000 g     F .text  0000000000000074 f
0000000000000000         *UND*  0000000000000000 m
0000000000000000         *UND*  0000000000000000 printf
…
```

UND : signifie que le symbole (fonction, variable globale) est utilisé dans le fichier .o mais sa définition est à l'extérieur.

Exploration des fichiers binaires

```
objdump -t code
```

(extraits)

```
0000000000001192 g     F .text  0000000000000074    f
0000000000000000       F *UND*  0000000000000000    printf@GLIBC_2.2.5
0000000000004014 g     O .bss   0000000000000004    m
0000000000001149 g     F .text  0000000000000049    main
```

Tous les symboles sont bien définis (f, m, main), c.-à-d. le compilateur les a trouvés dans un des .o.
Seule la fonction printf est non définie dans le binaire mais l'exécutable utilise la fonction printf du système.
Utilisez la commande ldd pour voir ce que l'exécutable utilise dans la machine:

```
ldd code
```

```
linux-vdso.so.1 (0x00007fff391f7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff50749f000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff5076cc000)
```

Exploration des fichiers binaires

L’information qui nous intéresse ici est que les symboles (fonctions, variables globales) sont enregistrés dans les .o avec leur nom, mais pas avec leur type (variables globales), pas avec leur type de retour ou leurs types et nombre de paramètres (fonctions).

Ce qui fait que le compilateur ne voit pas qu’il y des différences entre les symboles de même nom mais dans des .o différents.

Ceci est particulier au C !!

Correction de l'erreur

Pour corriger ce type d'erreur, on procède généralement comme suit, pour chaque variable globale:

- On met la déclaration de la variable globale dans un fichier d'entête (se terminant souvent par .h) avec le mot clef « extern »
- On remplace la déclaration de cette variable globale dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la variable globale), on définit la variable globale (sans le mot clef « extern ») – C accepte de déclarer une variable globale 2 fois (avec et sans « extern »)

On peut réutiliser le même fichier d'entête pour plusieurs variables globales.

Correction de l'erreur

Pour chaque fonction:

- On met le prototype de la fonction dans un fichier d'entête (se terminant souvent par .h) avec ou sans le mot clef « extern » (qui est ajouté par défaut)
- On remplace le prototype de cette fonction dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la fonction), on définit la fonction (avec le code interne de la fonction)

On peut réutiliser le même fichier d'entête pour les prototypes de plusieurs fonctions.

Cela permet au compilateur C de contrôler la cohérence des fonctions et variables globales.

Voir le répertoire symboles_globaux pour un exemple incomplet à compléter.

Un code avec une syntaxe correcte qui compile peut contenir des erreurs

Quand on exécute un code binaire C (produit de la compilation d'un code source), il peut se produire des erreurs à l'exécution ou des résultats faux.
Dans ce cas, il y a plusieurs moyens pour les repérer et les corriger:

- Réviser le code source
  En général, valable pour un petit code et si on a une bonne idée du problème
- Utiliser printf pour afficher des variables au cours du calcul
- Utiliser la fonction assert pour contrôler des conditions au cours du calcul
- Utiliser un outil de débug
  Il s'agit outils qui permettent d'exécuter un code instruction par instruction et d'afficher des variables du code pendant l'exécution.
- Utiliser un outil de test de la mémoire dynamique
  Si on soupçonne un problème avec l'utilisation de la mémoire dynamique.

Afficher des valeurs intermédiaires pendant le calcul

Afficher des valeurs intermédiaires avec printf

Quand on soupçonne que des variables prennent des valeurs non correctes pendant le calcul, on peut afficher ces valeurs en plusieurs endroits du code source.

Avantages:

- Ce moyen est souvent celui qui reste disponible, si aucun autre outil de recherche d'erreur ne l'est.
- Si on a une idée précise de la variable et de l'endroit qui pose problème, c'est un moyen de contrôle simple.

Inconvénients:

- C'est un processus qui peut-être long, surtout si un code source est de grande taille et qu'on n'a pas d'idée précise où se situe le problème.
- Ajouter des instructions au code (même des printf), le modifie et peut changer son comportement.

  Dans certains cas, un code peut s'arrêter sur une erreur sans printf et s'exécuter jusqu'à la fin avec des printf

Exemple

Dans le répertoire racine, un fichier C (main.c) contient une fonction qui calcule la racine carrée d'un nombre par une méthode de type Newton.

Le programme principal demande à l'utilisateur de rentrer un nombre et affiche la racine carrée calculée de ce nombre.

Compiler et exécuter plusieurs fois ce code en rentrant des valeurs quelconques positives, nulles et négatives (2, 3, 0, -1, -6).
Quel est le comportement du code ?

Afficher les valeurs successives de r dans la fonction racine.

Utilisation de la fonction assert

Tester des conditions dans le code source avec assert

C propose une fonction « assert » qui teste une condition.
Si la condition est vraie, l'exécution continue, sinon l'exécution s'arrête en affichant un message d'erreur.
Ne pas oublier d'inclure le fichier assert.h.
Voir le répertoire tests/racine2.

Il est possible de désactiver les asserts en compilant le code avec l'option NDEBUG (no debug):

```
gcc -DNDebug main.c -o racine
```

Dans ce cas, les instructions asserts seront ignorées.

Si la variable qui peut poser problème, est une variable initialisée par l'utilisateur, à la place de assert (ou en plus de assert), il faut plutôt ajouter un test explicite:

```
scanf("%lg", &x);
if (x < 0) { printf("entrer un nombre positif ou nul"); exit(-1); }
```

Conseils avec les instructions « assert »

En général, on procède comme suit :

- pendant la phase d'écriture du code, on compile en mode debug et on active les asserts et les tests sur le données d'entrée

```
gcc –g main.c –o racine
```

- quand le code a été testé et avant de le fournir aux utilisateurs, on compile en mode optimisé et on désactive les assert mais on garde les tests sur les données d'entrée

```
gcc –O2 –DNDebug main.c –o racine
```

La fonction assert est plutôt destinée aux développeurs de code et pas aux utilisateurs.

Préconditions avec les instructions « assert »

Dans chaque fonction du code source, on ajoute des "assert" au début de la fonction pour contrôler la validité des paramètres de cette fonction (si cela à un sens)

Exemple: produit de matrices A, B avec résultat la matrice C (le type matrice utilisé est celui vu à la séance précédente)

```
void produit(const Matrice *A, const Matrice *B, Matrice *C) {

  assert(A.n() == C.n());
  assert(B.m() == C.m());
  assert(A.m() == B.n());

  for (int i=0; i<C.n(); i++)
    for (int j=0; j<C.m(); j++)
      for (int k=0; k<A.m(); k++)
        C.c[i*C.m() + j] += A.c[i*A.m() + k]*B.c[k*B.m() + j];
}
```

Postconditions avec les instructions « assert »

Dans chaque fonction du code source qui produit des résultats, on ajoute des "assert" en fin de la fonction pour contrôler la validité des résultats de cette fonction (si cela à un sens)

Exemple: calcul par une formule de Taylor de sin(x)

```
double sinus(double x) {
  double s = x – x*x*x/6 + …;
  assert(s>-1 && s<1);
}
```

L'utilisation de pré- et post-conditions est parfois appelée "programmation par contrat" ou "programmation défensive"

Utilisation d’un outil de debug : exemple avec gdb

Utilisation d'un outil de debug

Les outils logiciels de debug permettent :

- d'exécuter instruction par instruction un code binaire,
- d'afficher la valeur d'une variable du code à un instant
- d'exécuter le code jusqu'à une instruction donnée
- d'exécuter le code tant qu'une condition est vraie
- d'exécuter le code tant qu'une variable ne change pas de valeur
- …

C'est donc un outil très puissant pour examiner ce qui se passe pendant l'exécution d'un code.

Il existe plusieurs outils de debug (on dit aussi debugger): gdb (linux), lldb (linux), codeview (intégré à visual studio dans windows), …

Gdb est souvent à la base des outils de débug disponibles dans les environnements de développement (codeblocks, visual code, eclipse,…)

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un débugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de l'une des commandes

```
gcc main.c –o code
gcc –O2 main.c –o code
gcc –g –O2 main.c –o code
```

Il faut utiliser ou mieux:

```
gcc –g main.c –o code
gcc –g –O0 main.c –o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake –S srcDir –B buildDir –DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un débugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de l'une des commandes

```
gcc main.c –o code
gcc –O2 main.c –o code
gcc –g –O2 main.c –o code
```

Il faut utiliser ou mieux:

```
gcc –g main.c –o code
gcc –g –O0 main.c –o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake –S srcDir –B buildDir –DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb : utilisation (1)

Pour exécuter le code binaire « code » sous le contrôle de gdb, il faut taper:

```
gdb code
```

Pour arrêter gdb, taper ctrl-d (et répondre y pour confirmer).

Si le message suivant (ou similaire) s'affiche :

```
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n])
```

Quitter gdb et tapez la commande (dans le terminal):

```
echo "set debuginfod enabled on" >> $HOME/.gdbinit
```

Debugger gdb : utilisation (2)

Quand on lance gdb (avec le nom du code binaire exécutable), gdb se met en attente d'une commande.

Les principales commandes sont:

- ctrl-x a: permet de voir le code source en même temps qu'on tape des commandes de gdb (ctrl-x a à nouveau pour sortir)
- run : lance l'exécution (jusqu'à la fin ou une erreur d'exécution ou un point d'arrêt (voir ci-dessous)
- quit : sort de gdb (confirmer en tapant y)
- b xxx.c:n (break) : met un point d'arrêt au début d'une instruction à la ligne nunéro n du fichier xxx.c

  Gdb affiche un numéro de point d'arrêt qui permettra de le supprimer, de le désactiver, de le réactiver, …

- p v : affiche la valeur de la variable v

Debugger gdb : utilisation (3)

- del k : supprime le point d'arrêt numéro k
- disable k : si le point d'arrêt numéro k est actif, le désactiver
- enable k : si le point d'arrêt numéro k est inactif, le réactiver
- cont (continue) : si l'exécution est arrêtée à un point d'arrêt, continuer jusqu'à la fin ou au point d'arrêt suivant
- n (next) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, « sauter au-dessus de l'appel)
- s (step) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, gdb se met au début du code de la fonction)
- w x (watch) : où x est le nom d'une variable simple (pas une structure), si on tape cont ou run à la suite de la commande watch, l'exécution se poursuit tant que la variable x ne change pas de valeur

Debugger gdb : utilisation (4)

- `b xxx.c:n if « condition »` : pose un point d'arrêt conditionnel à la ligne n du fichier xxx.c (l'exécution s'arrêtera au point d'arrêt seulement si la condition est vraie)

  Exemple: extrait du fichier « toto.c »:

```
21: for (i=0; i<1000; i++)
22:     a[i] = 2*b[i]-b[i-1]-b[i+1];
```

Si on tape : `b toto.c:22 if (i==500)`
l'exécution s'arrêtera à l'itération numéro 501 de la boucle sur i

- `bt` : affiche la pile d'appels (la liste des lignes par où est passé le code pour arriver à la ligne courante)

Debugger gdb : utilisation (5)

Il existe (beaucoup) d'autres commandes, voir, par exemple, le manuel de gdb (https://sourceware.org/gdb/current/onlinedocs/gdb.html)

Pour se rappeler les commandes de gdb, le mieux est de s'exercer sur des codes:

- Recompiler le code dans le répertoire sqrt en mode debug et utiliser gdb pour exécuter le code, mettre des points d'arrêts, afficher des variables, etc.

Comprendre les erreurs d’utilisation de la mémoire dynamique : l’outil valgrind

Erreurs d’exécution : utilisation de la mémoire dynamique

Un grand nombre d’erreurs d’exécution sont dues à une mauvaise utilisation de la mémoire dynamique:

- Utilisation d’une mémoire dynamique non (ou pas encore) réservée
- Utilisation d’un pointeur en dehors de la mémoire dynamique réservée
- Utilisation d’une mémoire dynamique après qu’elle ait été rendue au système.

L’outil valgrind

valgrind est un logiciel conçu à l’origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l’environnement KDE dans linux.
Il s’est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- valgrind crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, valgrind peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l’efficacité de l'outil, est que le temps d’exécution est (considérablement) allongé.

L’outil valgrind

Valgrind est un logiciel conçu à l’origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l’environnement KDE dans linux.
Il s’est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- valgrind crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, valgrind peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l’efficacité des vérifications, est que le temps d’exécution est (considérablement) allongé.

Utilisation de valgrind

Utiliser valgrind est simple, il suffit de taper valgrind suivi du nom du code binaire:

```
valgrind code
```

valgrind affiche à l'écran, en plus des affichages normaux du code, tout ce qu'il a noté dans l'utilisation de la mémoire dynamique.

Compte-tenu du (potentiellement) grand nombre de message affiché, il est préférable de rediriger les affichages écran dans un fichier qu'on examinera à la fin de l'exécution :

```
valgrind code >& log
```

(crée un fichier de nom log qui contient les messages de valgrind et les affichages du code)

Exemple d'utilisation de valgrind

Se mettre dans le répertoire memoire_dynamique/exemple2,
compiler en mode debug et exécuter la commande

```
valgrind ./code >& log
```

On examinera en séance le fichier log avec les messages valgrind.

Comprendre les erreurs d’utilisation de la mémoire dynamique : l’outil sanitizer

L’outil sanitizer

Plusieurs outils ont été développés chez google, puis intégrés dans les compilateurs (d’abord clang, puis gcc et icx (le compilateur Intel))

C’est un fichier binaire (une librairie) à ajouter à la compilation. Il contient des fonctions qui interceptent toutes les utilisations de la mémoire dynamique.
Pour compiler le code, il faut ajouter une option –fsanitize=… et, éventuellement, une librairie (exemples pour gcc et clang) :

```
gcc –g –fsanitize=address main.c –static–libasan –o code

clang –fsanitize=memory –fno–omit–frame–pointer –g main.c –o code
```

Exemple d'utilisation des sanitizers

Le répertoire memoire_dynamique/exemple2 contient plusieurs versions avec différents types d'erreurs possibles.

Un Makefile compile chaque fichier séparément pour générer des codes, avec et sans les options sanitizer.

- Utiliser valgrind avec les exécutables compilés sans les options sanitizer
- Exécuter directement, les exécutables générés avec les options sanitizer

Interpréter les sorties.

La librairie sanitizer est peut-être un futur outil standard.
Cet outil n'est pas encore aussi efficace que valgrind mais est beaucoup plus rapide.
Par contre, il faut recompiler tout les fichiers sources.

Dans la pratique

Aucun des outils présenté ici n'est capable de trouver toutes les erreurs possibles.

Il faut donc, en général, les combiner

Cas des vecteurs de taille fixe – débordement de tableau

Compiler et exécuter le code contenu dans le répertoire « vecteurs-statiques ».

Expliquer l'erreur dans ce qui est affiché.

Dans la séance, on utilisera gdb pour repérer la cause de l'erreur.

Ce genre d'erreur (débordement de tableau) est souvent utilisé par les virus informatiques qui parviennent à écraser des zones mémoires normalement inaccessibles dans un code et à modifier le comportement du code.
Il est donc important de corriger ce type d'erreur.

Malheureusement, les outils de debug arrivent difficilement à trouver la cause de l'erreur (dans l'exemple, on a repéré la zone écrasée en affichant toutes les variables en plusieurs points du code).