Ecole Doctorale

Compilation, debogage

Violaine Louvet [1]

[1] ICJ - CNRS

Année 2012-2013

Objectifs de ce cours

Compilation

- Comprendre le rôle et le fonctionnement d'un compilateur.
- Connaître les principales directives et options de compilation.
- Savoir faire de la compilation séparée.
- Savoir utiliser un outil de compilation : CMake.

Debogage

- Connaître le principe du debogage.
- Savoir utiliser les debuggers et certains outils associés.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Sommaire

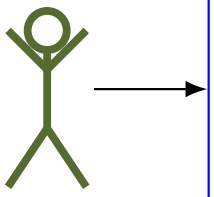
1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Du programme au hardware

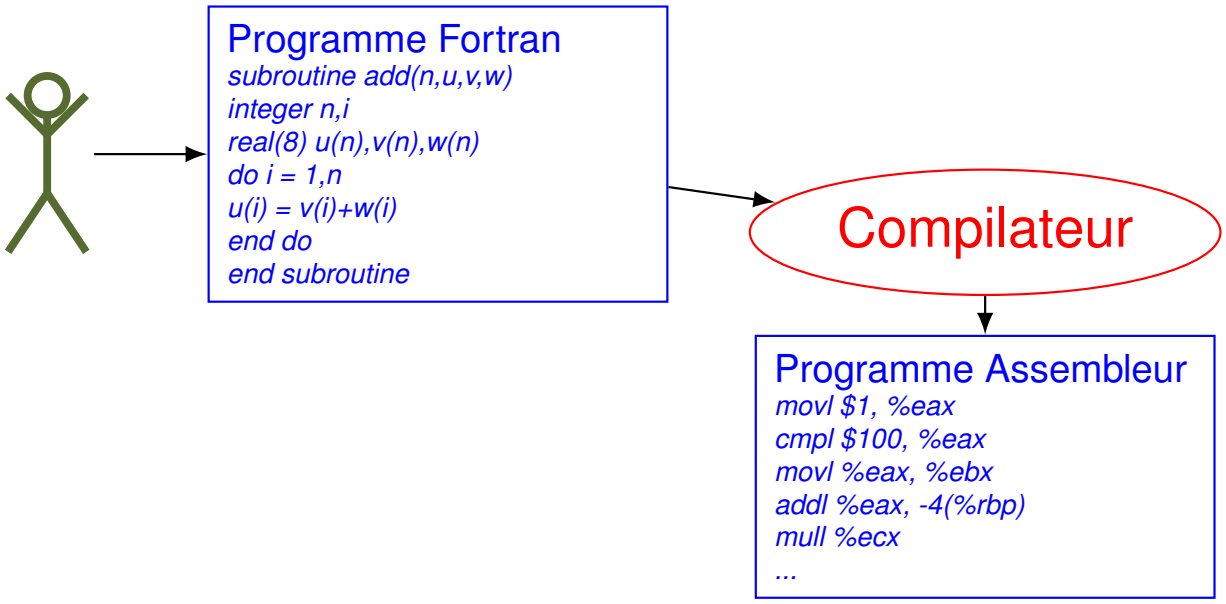
Du programme au hardware



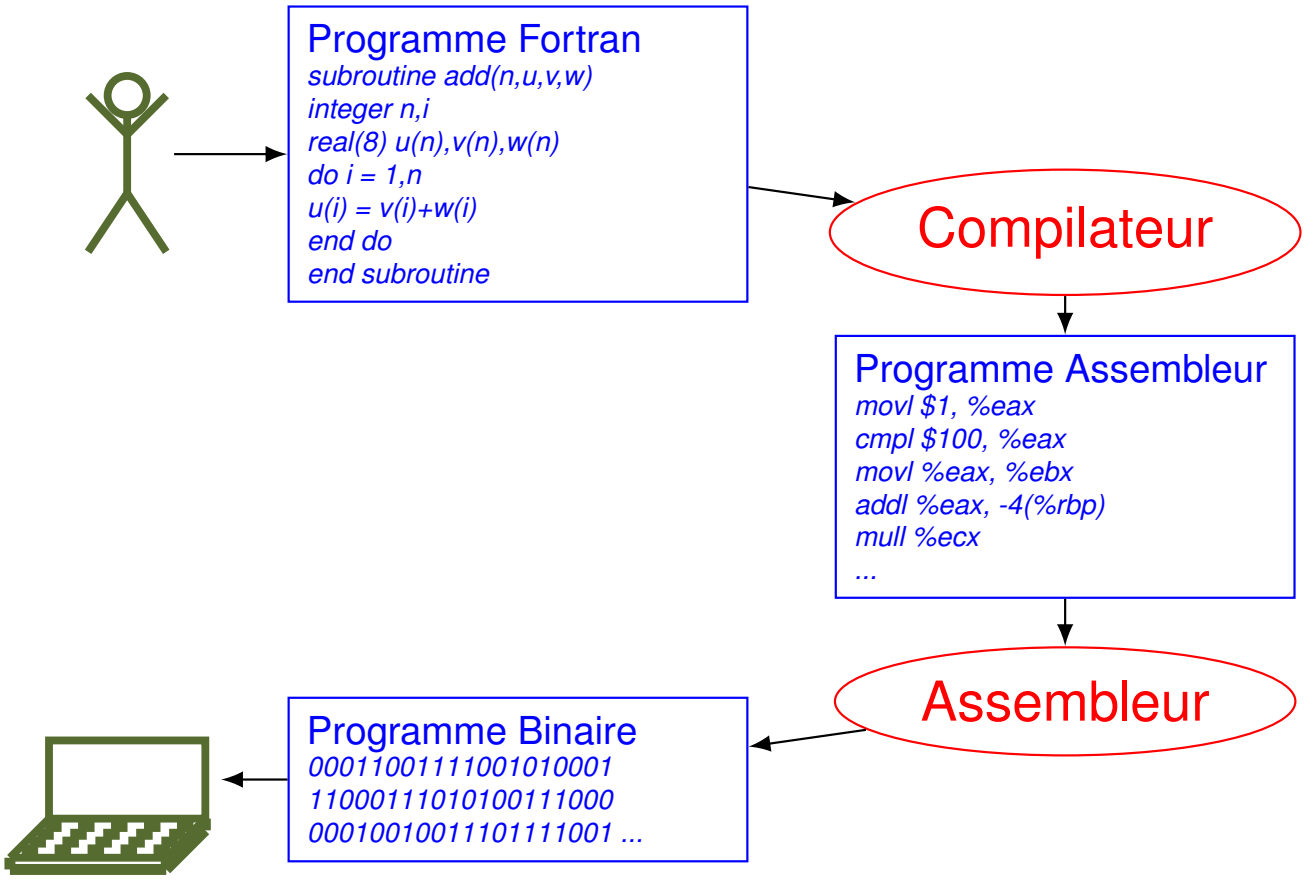
Programme Fortran

```
subroutine add(n,u,v,w)
integer n,i
real(8) u(n),v(n),w(n)
do i = 1,n
u(i) = v(i)+w(i)
end do
end subroutine
```

Du programme au hardware



Du programme au hardware



Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Rôle du compilateur

- Langage compilé : traduit en instructions lisibles par la machine une fois pour toutes. Fortran, C, C++ ...
- Langage semi-interprété : il existe un compilateur traduisant le programme non pas en « langage-machine » mais en un code intermédiaire assez analogue à de l'assembleur (pseudo-code, p-code, Byte Code). Java, Python ...
- Langage interprété : converti en instructions exécutables par la machine au moment de son exécution.PHP, Perl ...

Rôle du compilateur

Traduction d'un langage source (en pratique un code source) vers un langage cible (en pratique un code objet ou un code binaire exécutable par la machine).

Etapes de la compilation

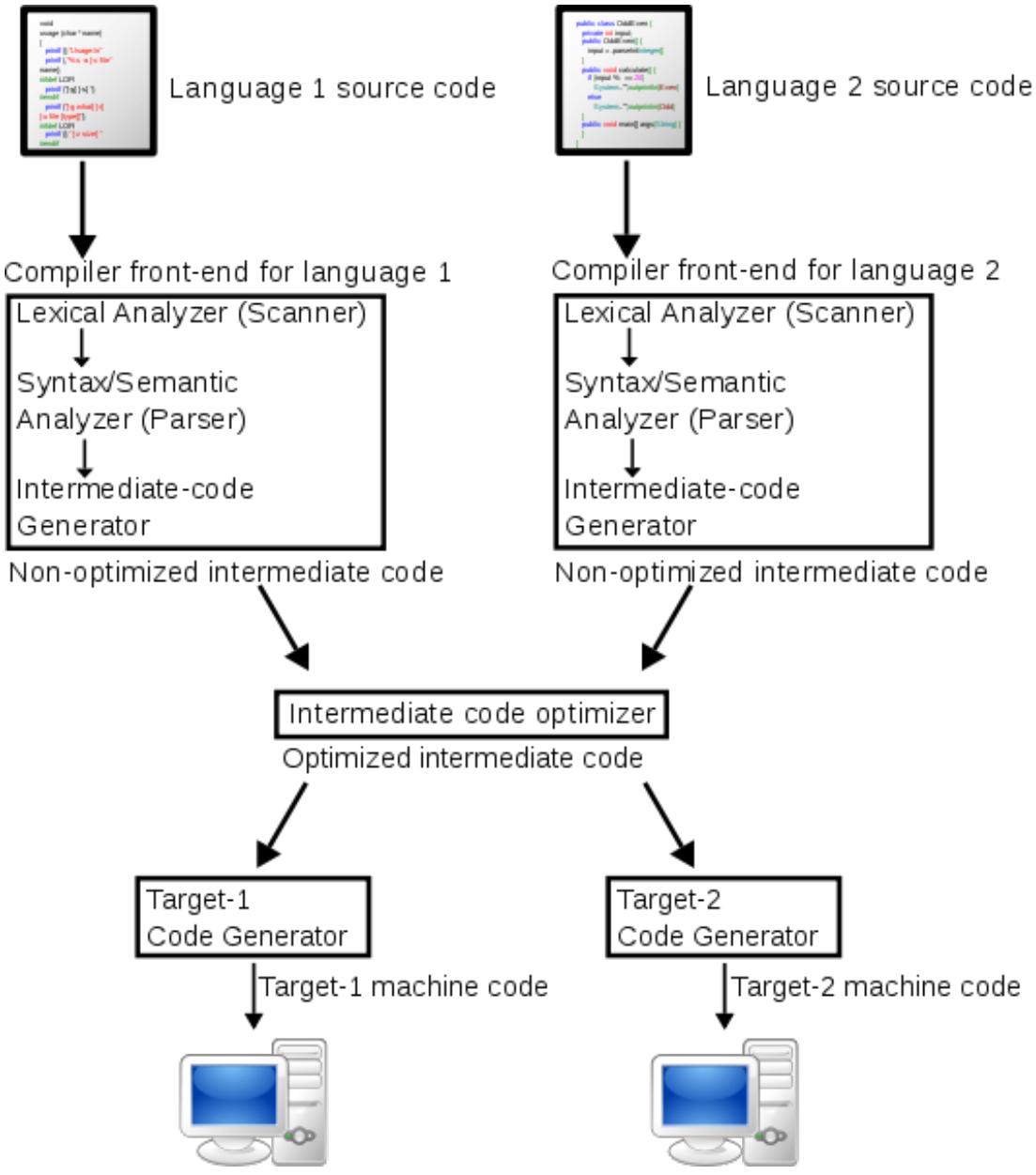
- Analyse lexicale : identifie les lexèmes (unités lexicales) du langage : mot-clé, identifiant, ...

  Code source *int x = @ ;*
  Compilation *error : stray '@' in program*
- Analyse syntaxique : trouve la structure syntaxique (arbre) et teste l'appartenance au langage.

  Code source *else sans if*
  Compilation *error : 'else' without a previous 'if'*
- Analyse sémantique : vérifie la cohérence des instructions reconnues.

  Code source *variable x utilisée mais non déclarée*
  Compilation *error : 'x' was not declared in this scope*
- Génération du code : encodage en assembleur, optimisations et allocations des registres, traduction en code objet. Phases dépendantes de la machine cible.

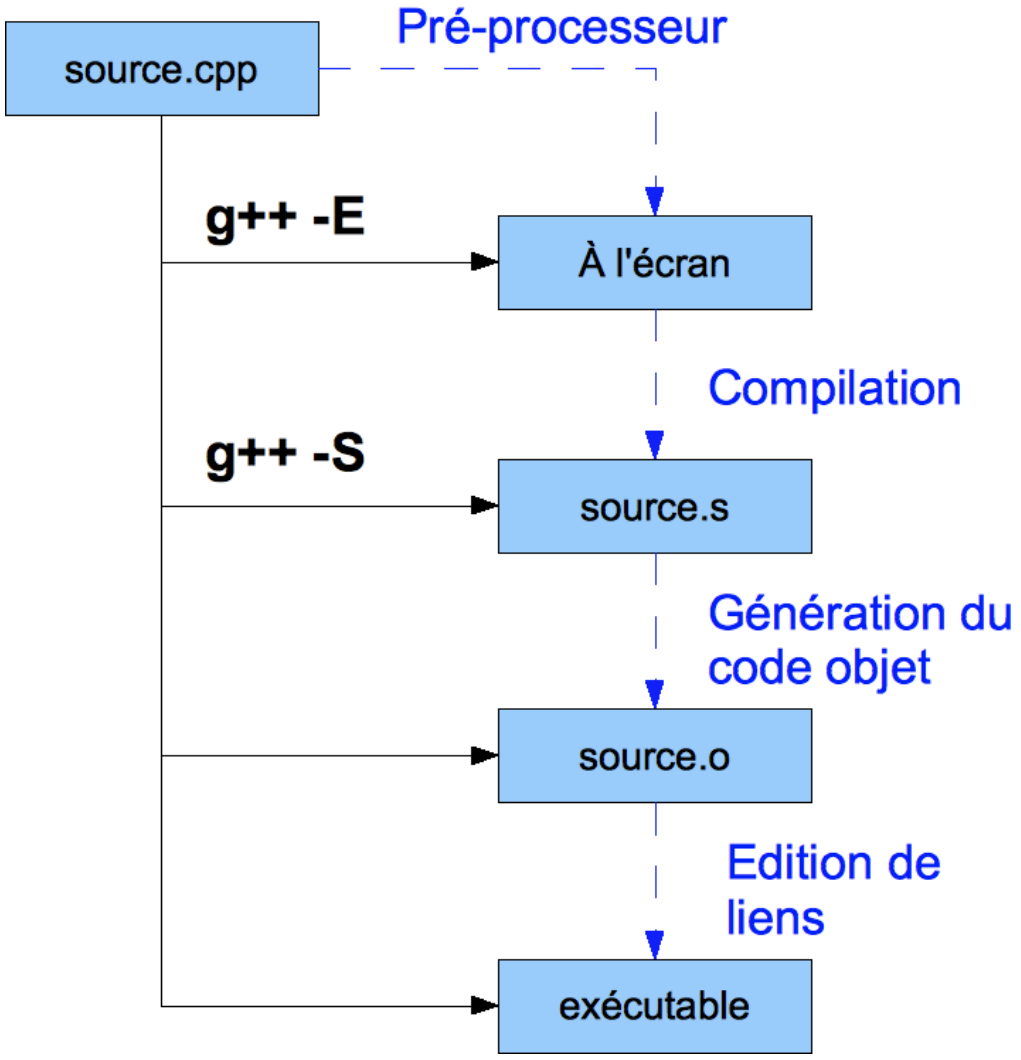
Etapes de la compilation



Travail du compilateur

- Le code intermédiaire est une série d'instructions bas niveau, produit sans optimisation. Il doit être optimisé pour produire un exécutable efficace sur la machine cible.
- La traduction binaire dépend :
  - du processeur utilisé
  - des conventions de codage utilisées par le système d'exploitation
- L'optimisation du code intermédiaire se fait :
  - En diminuant la taille du code (travail sur les boucles : réduction du nombre de variables et du nombre d'opérations)
  - En ordonnançant les instructions en fonction des dépendances.
  - En utilisant au mieux les registres du processeur.
  - En déroulant les boucles pour profiter des unités vectorielles des processeurs.
- La quantité de travail du compilateur dépend du jeu d'instructions du processeur.

En pratique


source.cpp
Pré-processeur
g++ -E
À l'écran
Compilation
g++ -S
source.s
Génération du
code objet
source.o
Edition de
liens
exécutable

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Rôle du pré-processeur

Le pré-processeur enlève les commentaires et exécute les instructions commençant par un dièse.

Actions possibles

- Inclusion d'autres fichiers dans le fichier à compiler.
- Définition des constantes symboliques et des macros.
- Compilation conditionnelle de code de programme
- Exécution conditionnelle des directives du précompilateur.

Directives de pré-compilation

#include : Provoque l'insertion d'une copie d'un fichier spécifié à la place de la directive.

```
#include <iostream>
```

#define constantes : Permet de créer des constantes représentées comme des symboles.

```
#define PI 3.14159
```

Peut être utilisé seulement dans le fichier dans lequel ces constantes sont définies.
En C++, on préfère utiliser des variables const car elles ont un type de donnée spécifique et sont visibles par leur nom pour un débogueur.

#define macros sans argument : Définition de macros.

```
#define newline() (cout << endl;)
```

En C++, les macros ont été remplacées par les modèles et les fonctions inline.

Directives de pré-compilation

#define macros avec arguments : Définition de macros.
Aucune vérification de type de donnée n'est effectuée sur les arguments de macro.

```
#define AIRE_DU_CERCLE(x) (PI*(x)*(x))
```

- Si le texte de remplacement est plus long que le reste de la ligne, une barre oblique inverse (\ ) placée à la fin de la ligne indique que ce texte continue sur la ligne suivante.
- On peut supprimer les constantes symboliques et les macros à l'aide de la directive de précompilation *#undef*.
- Portée d'une constante symbolique ou d'une macro : de sa définition jusqu'à la suppression de la définition par #undef ou jusqu'à la fin du fichier.

Compilation conditionnelle

#if ... #endif : évalue une expression entière constante qui détermine si le code doit être compilé.

```
#if !defined(compiler)
  #define compiler 0
#endif
```

Si *compiler* est défini, *!defined(compiler)* est 0 et la directive *#define* est omise. Autrement, *defined(compiler)* s'évalue à 0 et compiler est défini.

#ifdef #ifndef : raccourcis pour *#if defined(nom)* et *#if !defined(nom)*.

- Evite d'inclure le même fichier plusieurs fois.
- Aide au débogage permettant de mettre des portions de code en commentaire pour éviter que ce code soit compilé.

```
#if 0
  code omis a la
  compilation
#endif
```

```
#ifdef DEBOGAGE
 cerr << "␣Variable␣x␣=␣"
  << x << endl;
#endif
```

Directives de pré-compilation

#error : permet d'afficher un message. La précompilation s'arrête, le programme n'est pas compilé.

```
#if !defined(__cplusplus)
#error Un compilateur C++ est requis.
#endif
```

#pragma : provoque une action définie au niveau du pré-compilateur.

```
#pragma message("Compiling " __FILE__ )
#pragma message("Last modified on " __TIMESTAMP__ )
```

Permet d'afficher le nom du fichier à compiler, la date et le moment où ce fichier a été modifié pour la dernière fois.

Directives de pré-compilation

#line : provoque la renumérotation des lignes de code source qui suivent la valeur constante entière spécifiée.

```
#line 100 "FichierA.cpp"
```

Démarre la numérotation à 100 à partir de la prochaine ligne de code source. En plus, le nom du fichier "FichierA.cpp" apparaîtra dans tout message du compilateur.

Constantes symboliques prédéfinies

*_LINE_* : numéro de ligne en cours du code source.

*_FILE_* : nom du fichier source

*_DATE_* : date de compilation

*_TIME_* : heure de compilation

*_STDC_* : Constante entière 1 : pour vérifier la compatibilité avec la norme ANSI.

Opérateurs de pré-compilation

\# : provoque la conversion d'une série de caractères du texte de remplacement en une chaîne de caractères entourée de guillemets.

```
#define BONJOUR( x ) cout << "␣Bonjour,␣" #x << endl
```

BONJOUR(Jean) sera développé en :

```
cout << "␣Bonjour,␣" "Jean" << endl
```

\## : provoque la conversion d'une série de caractères du texte de remplacement par une série de caractères passée en paramètre.

```
#define AFFICHE(n) (cout << endl << s##n << endl)
using namespace std;
void main()
{
 string s1, s2("OUF"), s3("p");

 AFFICHE(1);
 AFFICHE(2);
 AFFICHE(3);
}
```

Assertions

- La macro *assert*, définie dans le fichier en-tête *<assert.h>*, teste la valeur d'une expression.
- Si la valeur de l'expression vaut 0 (false) alors *assert* affiche un message d'erreur et appelle la fonction *abort* de la bibliothèque d'utilitaires généraux *<stdlib.h>* pour terminer l'exécution du programme.

```
#include ''assert.h''
...
assert( x <= 10);
```

Si x > 10 alors un message d'erreur contenant le numéro de ligne et le nom du fichier est affiché et le programme se termine.

- Si la constante symbolique NDEBUG est définie,

```
#define NDEBUG
```

les assertions qui suivent seront ignorées. Cela évite d'enlever les assertions après la mise au point.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Contrôle du type de sorties

-c : compilation et assemblage sans linkage. Génère un fichier objet d'extension **.o*.

-S : arrête après l'étape de compilation sans faire l'assemblage. Génère un fichier en assembleur d'extension **.s*

-E : arrête après l'étape de pré-processing. Le code généré est envoyé sur la sortie standard.

-o file : fichier de sortie dans *file*. Par défaut, l'exéutable s'appelle *a.out*.

-v : sort les différentes commandes exécutées pendant les étapes de compilation.

-ansi : impose pour gcc une compatibilité avec les standards iso.

Contrôle des warnings, options de debogage

-Werror : Transforme les warnings en erreur.

-pedantic : émet des warnings prévus par les normes ISO appliquées de façon très strictes.

-Wall : Active tous les avertissements indispensables.

-Wextra : Active quelques avertissements supplémentaires par rapport à *Wall*.

-Wstrict-aliasing : vérifie le strict aliasing (deux types indépendants ne peuvent pas pointer vers la même mémoire). S'utilise avec *-fstrict-aliasing*.

-g : Activation de la génération des informations pour le déboggage. En particulier, *-ggdb* pour l'utilisation de *gdb*.

-pg : Instrumentation et production des informations pour l'outil de profiling gprof.

–coverage, -fprofile-arcs, -ftest-coverage : options utilisées pour l'analyse de couverture d'un code avec *gcov* (*–coverage* équivalent à *-fprofile-arcs -ftest-coverage*).

Options d'optimisation

-O0 : aucune optimisation

-O1 : optimisations visant à accélérer le code, en particulier quand il ne contient pas beaucoup de boucles.

-O2 : *O1* + déroulage de boucles (considère notamment un aliasing strict). Augmente sensiblement le temps de compilation.

-O3 : *O2* + transformation des boucles, des accès mémoire.

Certaines optimisations « agressives » peuvent modifier la précision des résultats (ordre des opérations par exemple).

-march=cpu_type : génère les instructions adaptées au jeu d'instructions du processeur spécifié.

Options du pré-processeur, option du link

-DMON_SYMBOL ou -DMON_SYMBOL=valeur : affectation d'une valeur à une macro.

-UMON_SYMBOL : Annule toute définition précédente associée à *MON_SYMBOL*.

-Idirectory : indique le chemin de recherche pour trouver les fichiers inclus dans les macros *#include*. Un certain nombre de répertoires sont prédéfinis et n'ont pas besoin d'être spécifiés.

-M : définit les régles de dépendances pour le fichier donné, adapté à l'utilitaire *make*. L'option *MM* fait le même travail sans référencer les headers système.

-llibrary : Cherche la bibliothèque *library* au moment de la génération de liens. Le linker cherche et lie les fichiers objet et bibliothèques dans l'ordre spécifié. La bibliothèque est cherché sous la forme *liblibrary.a*.

-Ldirectory : chemin de recherche pour les bibliothèques.

-static-libstdc++ : force le linkage statique avec la librairie standard C++. Option *-static* pour les autres bibliothèques.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

TP : ex1

- Travail du pré-processeur :
  - A partir du fichier *preproc.cpp*, rajouter les lignes permettant de choisir à la compilation le type de vecteur utilisé.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Compilation séparée, édition de lien

- Séparer le source en plusieurs fichiers.
- Création de plusieurs fichiers objets.
- Edition de liens : assemblage de ces fichiers en un exécutable.

Bibliothèque statique : incluse dans l'exécutable. Exécutable auto-suffisant. Nécessite la recompilation du code à chaque changement de la bibliothèque. Copie multiple de la bibliothèque si plusieurs codes l'utilisent.

Bibliothèque dynamique : l'appel de la bibliothèque est géré à l'exécution du code par le chargeur dynamique (*ld.so*). Nécessite un environnement utilisateur bien configuré :

- Chemin de recherche standard défini dans */et/ld.so.conf*
- Variable d'environnement *LD_LIBRARY_PATH*
- Visualisation des dépendances d'un exécutable : *ldd*

Création et utilisation de bibliothèques

- Par défaut, utilisation de bibliothèques dynamiques. Options *-Wl -Bstatic* pour préciser un lien statique :
  g++ object1.o object2.o -Wl,-Bstatic -lapplejuice -Wl,-Bdynamic -lorangejuice -o binary
- Bibliothèque statique : *ar rcs libfile.a file1.o file2.o file3.o*
  Créé une bibliothèque avec un index (option *s*).
- Bibliothèque dynamique : création à l'aide de l'option *-shared* du compilateur :
  g++ -o libfile.so -shared file1.o file1.o file3.o
- L'ordre dans l'édition des liens est important : les symboles d'une bibliothèque ne sont pris en compte que s'ils sont utilisés par un objet présent avant dans la ligne de commande : si b dépend de a, a devra figurer après b. Possibilité de dépendances circulaires : certaines bibliothèques peuvent figurer plusieurs fois.

nm : liste l'ensemble des symboles contenus dans un fichier objet ou un exécutable (fonctions, constantes ...).

objdump, readelf : donne différentes informations sur des fichiers objets.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Makefile

- Automatise la compilation des fichiers.
- Basé sur un fichier *Makefile* et sur l'appel de la commande *make*.
- Le fichier *Makefile* contient une liste de règles et de dépendances utilisées pour construire des cibles.
- Chaque commande est précédée d'une tabulation.

Forme des règles

```
cible : dependances
        commandes
```

1er exemple de *Makefile* simplissime

```
exmake : vector.hpp exmake.cpp
        g++ exmake.cpp -o exmake
```

Fonctionnement de make

- La cible de la première règle est la principale.
- Make ne recompile que ce qui a été modifié.
- Make regarde si les dépendances sont satisfaites :
  - Si elles ne le sont pas, prend la première dépendance pour cible.
  - Si elles le sont, exécute les commandes associées à la règle.
- En terme d'exécution, on peut voir make comme un programme récursif.

Exemple

```
# Regle principale
exmake : exmake.o
	g++ exmake.cpp -o exmake
# Regle 1
exmake.o : vector.hpp exmake.cpp
	g++ -c exmake.cpp
# Regle 2 : nettoyer les .o
clean :
	rm exmake.o
```

Cibles classiques

all : génération de tous les exécutables.

.PHONY : les dépendances sont toujours reconstruites.

clean : suppression des fichiers intermédiaires.

mrproper : suppression des fichiers intermédiaires et des exécutables.

Exemple

```
# Regle principale
all : exmake
# Regle 1
exmake : exmake.o
	g++ exmake.cpp -o exmake
# Regle 2
exmake.o : vector.hpp exmake.cpp
	g++ -c exmake.cpp
# Regle 3
.PHONY : clean mrproper
# Regle 4
clean :
	rm exmake.o
# Regle 5
mrproper : clean
	rm -rf exmake
```

Variables personnalisées

- Déclaration : *NOM = VALEUR*
- Utilisation : *$NOM*
- Prédéfinies :

CFLAGS	: options de compilation pour le C
CPPFLAGS	: options de précompilation
LDFLAGS	: options d'édition de liens
CXXFLAGS	: options de compilation pour le C++
CC ou *CXX*	: désignent les compilateurs

Exemple

```
# Declarations de constantes
CC = gcc
CFLAGS = -O4 -Wall
LD = gcc
LDFLAGS = -s
```

Variables internes

- $ @ représente la cible.
- $ ^ représente la liste des dépendances.
- $ < représente la première dépendance.
- $? représente la liste des dépendances plus récentes que la cible.
- $ * représente le nom de la cible sans suffixe.

Exemple

```
# $@ == rk4 et $^ == main.o rk4.o
rk4 : main.o rk4.o
	$(CC) -o $@ $^
main.o : main.cpp rk4.hpp
	$(CC) -o $@ -c $<
rk4.o : rk4.cpp rk4.hpp
	$(CC) -o $@ -c $<
```

Règles d'inférences

Ce sont des règles génériques qui sont appelées par défaut.

Exemple

```
rk4 : main.o rk4.o
	$(CXX) -o $@ $^ $(LDFLAGS)

%.o :%.cpp
	$(CXX) -o $@ -c $< $(CXXFLAGS)

main.o : rk4.hpp
rk4.o : rk4.hpp
```

Manipulation des noms de fichiers sources

wildcard : Récupération des noms de fichiers.
*SRC=$(wildcard *.cpp)*

patsubst : Remplacement d'une expression par une autre.
OBJ=$(SRC :.cpp=.o)
OBJ=$(patsubst %.cpp,%.o,$(SRC))

notdir : Extraction du nom de fichier.
OBJ=$(notdir $(patsubst %.cpp,%.o,$(SRC)))

Utilisation de conditions

Exemple

```
MODE_DEBUG=no
ifeq ($(MODE_DEBUG),yes)
      CXXFLAGS=-Wall -ansi -g
      LDFLAGS=-Wall -ansi -g
else
      CXXFLAGS=-Wall -ansi
      LDFLAGS=-Wall -ansi
endif
```

Utilisation

make MODE_DEBUG=yes

Recherche de dépendances

Variable *VPATH* : modifie les chemins de recherche.

```
VPATH= src :../headers
rk4.o: rk4.cpp
```

- *rk4.cpp* sera recherché dans le répertoie courant ainsi que dans le répertoire *headers* et dans le répertoire *src* du répertoire courant.

Directive *vpath* : chemin de recherche pour une classe de fichiers.

```
vpath %.h ../headers
rk4.o: rk4.cpp
```

- Les fichiers d'entête seront recherchés dans le répertoire *headers* du répertoire courant.

Appel imbriqué de Makefile

Makefile appelant (dans le répertoire racine)

```
export CXX=g++
export CXXFLAGS=-Wall
export LDFLAGS=-Wall
SRC_DIR=src
EXEC=$(SRC_DIR)/prog

all : $(EXEC)

$(EXEC) :
	cd $(SRC_DIR) && $(MAKE)

.PHONY: clean mrproper $(EXEC)

clean :
	cd $(SRC_DIR) && $(MAKE) $@

mrproper : clean
	cd $(SRC_DIR) && $(MAKE) $@
```

Makefile appelé (dans répertoire *src*)

```
EXEC=prog
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all : $(EXEC)

prog : $(OBJ)
	$(CXX) -o $@ $^ $(LDFLAGS)

%.o :%.c
	$(CXX) -o $@ -c $< $(CXXFLAGS)

.PHONY: clean mrproper

clean :
	rm -rf *.o

mrproper : clean
	rm -rf $(EXEC)
```

A noter : ce pattern semble maintenant comme une mauvaise pratique, et il faudrait privilégier la directive *include*. Référence : http ://aegis.sourceforge.net/auug97.pdf

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Autres outils de construction de programme

autotools : génération automatique de Makefile configurés automatiquement selon la plateforme.

CMake : construction portable de programmes.

scons : construction portable de programmes écrit en python.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Introduction

CMake ?

Ensemble d'outils permettant de

- compiler un projet pour différentes plate-formes,
- faire des tests
- créer des packages pour différents systèmes.

Utilisé dans de nombreux projets tels que KDE et MySQL. Bonne alternative aux autotools.

Composants

- cmake : création de Makefile et compilation de projets
- ctest : mise en place de tests sur vos projets
- cpack : création de package

Mode Opératoire

- Création d'un fichier *CMakeLists.txt* dans chaque répertoire du projet
- Le fichier *CMakeLists.txt* du répertoire principal pilote l'ensemble.
- Génération des fichiers pour la compilation par la commande *cmake*
  - Bonne pratique : construction hors des sources du projet
- Variables utilisées par CMake (compilateurs, options de compilation ...) mises en cache dans le fichier *CMakeCache.txt*

Exemple minimal

Projet/main.cpp

```
#include <iostream>
int main (int argc,char** argv) {
  std::cout << "Hello World "
            << std::endl;
  return 0;
}
```

Projet/CMakeLists.txt

```
project(hello CXX)
add_executable(hello main.cpp)
```

```
$ cd Projet
$ mkdir build ; cd build
$ cmake .. ; make
```

Syntaxe du CMakeLists.txt

- Commentaires : *#*
- Commandes :*COMMAND(arg1 arg2 ...)*
- Listes : *A ; B ; C*
- Variables : *$VAR*, créées par la commande *SET(nom_var contenu)*
- Tests pour contrôler le flux d'exécution :
  - *if(expr) ... else(expr) ... endif(expr)*
  - *foreach(varloop) ... endforeach(varloop)*
  - *while(expr) ... endwhile(expr)*

Syntaxe du CMakeLists.txt

Quelques commandes

- *project (nom language)* : nom et langage du projet
- *add_executable(nom_exec src1 src2 ...)* : définit un exécutable à partir des sources src1 ...
- *add_library(nom_lib SHARED src1 src2 ...)* : définit une bibliothèque partagée
- *target_link_libraries(nom_exec nom_lib)* : spécifie que l'exécutable a besoin de la bibliothèque
- *add_subdirectory(nom_ss_rep)* : indique qu'il y a des choses à faire dans le sous-répertoire qui doit contenir un fichier *CMakeLists.txt*

Gestion des dépendances

- La commande *find_package* permet de chercher des programmes sur votre système et les informations relatives (comme par exemple le répertoire des include ou des librairies).
- CMake fournit un ensemble de modules avec l'extension .cmake permettant de trouver certains programmes sur votre système.
- Ces fichiers se trouvent dans */usr/share/cmake-*/Modules*
- Vous pouvez également créer vos propres modules.

Exemple

```
project(myproject Fortran CXX)
cmake_minimum_required(VERSION 2.8)

# add cmake directory to find SuperLU
set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake/" ${CMAKE_MODULE_PATH})

# search SuperLU
FIND_PACKAGE(SuperLU REQUIRED)
include(${SUPERLU_INCLUDES})

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_BINARY_DIR}/src)
add_subdirectory(src)
```

En pratique

- Privilégiez l'arborescence des sources :
  - Répertoire *src* pour les sources
  - Répertoire *test* pour les programmes principaux
  - Répertoire *build* pour la compilation
- Ecrire un fichier *CMakeLists.txt* dans chacun des répertoires où la compilation est nécessaire.
- Ecrire le fichier *CMakeLists.txt* principal
- Créer le répertoire *build*
- Dans *build*, *cmake ..*
- Dans *build*, *make*
- On retrouve alors dans *build* la même arborescence que dans le répertoire principal, avec un placement analogue des fichiers générés.
- Ne pas hésiter à utiliser la doc de cmake, l'outil est très puissant donc très complet ...

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

TP Makefile : ex2

- Ecriture d'un *Makefile* :
  - Ecrire le Makefile correspondant à l'exemple précédent en définissant les cibles associées à l'utilisation des deux types de vecteurs.

    make clean vector : compilation du code avec la classe *vector*.
    make clean std : compilation du code avec les vector de la bibliothèque standard c++.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

TP CMake : ex3

- A partir d'un code complet fourni, écriture de l'ensemble des fichiers nécessaires à sa compilation :
  - Créer une bibliothèque à partir des fichiers du répertoire *src*
  - Créer un exécutable à partir du fichier du répertoire *test*
  - Créer le fichier principal *CMakeLists.txt*
  - Compiler et exécuter le code.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Principes du debogage

« Ce n'est pas un bug, c'est une fonctionnalité non documentée ! »

« Tout programme non trivial possède au moins un bug. »

Pourquoi déboguer ?

- Programme produisant un résultat incorrect.
- Plantage du programme.
- Problème de gestion mémoire : fuite mémoire, erreur de segmentation, dépassement de tampon, de pile ...

Débogueurs

Débogueur

Le débogueur permet d'analyser l'état d'exécution d'un logiciel à un instant donné, les opérations en cours, les informations en mémoire, les fichiers ouverts, etc.

Débogueur en ligne : permet de suspendre l'exécution du logiciel à tout moment, d'analyser l'état, puis de continuer les traitements.

Débogueur post-mortem : permet d'analyser l'état d'exécution d'un logiciel après un crash. L'analyse se fait sur la base d'un fichier qui contient la copie du contenu de la mémoire au moment du crash. Fichier appelé core dump sur les systèmes d'exploitation Unix.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

Outils de debogage

Débogueur en mode texte

gdb : debogueur GNU en mode texte. **Le** debogueur !

Electric Fence : débogage de la gestion mémoire.

DUMA : Detect Unintented Memory Access, fork d'Electric Fence.

strace : visualiser les appels systèmes effectués par le programme tracé.

ltrace : visualise les appels aux fonctions de bibliothèques.

valgrind : debuggage mémoire.

Outils de debogage

Pour ceux qui préfèrent les clickodromes

ddd : Data Display Debugger, interface graphique interfaçable avec plusieurs debogueurs dont gdb.

Kdevelop : environnement de développement basé sur KDE offrant entre autre une interface graphique à gdb.

emacs : éditeur de texte ultra puissant qui offre aussi une interface à gdb (gdb-mode).

kdbg : interface graphique à gdb basée sur KDE.

xxgdb : interface graphique à gdb pour X.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

En pratique : gdb

- Compilation du programme avec l'option *-g* en désactivant les options d'optimisation.
- Lancement du debogueur :

  gdb ./mon_prog : analyse de l'exécution.
  gdb ./mon_prog core : analyse post-mortem à partir du fichier *core*.
  gdb ./mon_prog 2345 : analyse d'un programme en cours d'exéution (de pid *2345*).

- *break [fichier_source :]fonction* : point d'arrêt au début de la fonction spécifiée. On peut directement passer un numéro de ligne à la place du nom de la fonction.
  - *info break* permet de connaître les arrêts définis.
  - *delete numéro_break* supprime un arrêt (sans argument = effacement de tous les arrêts).
  - *clear numéro_ligne* ou *clear nom_fonction* permet aussi d'effacer les arrêts.
  - *enable numéro_break* et *disable numéro_break* active et déactive des points d'arrêt.

En pratique : gdb

- *condition numéro test* : pose une condition sur un point d'arrêt. *condition 1 i==10* : activation du point d'arrêt numéro 1 que dans le cas où la variable *i* vaut 10.
- *run [liste de paramètres]* : démarre l'exécution.
- *bt* : backtrace : affiche le contenu de la pile du programme.
- *up, down* : permet de remonter ou de redescendre dans la liste de la pile.
- *print expression* : affiche la valeur d'une expression.
- *display expression* : afficher à chaque pas la valeur d'une expression.
- *watch expression* : créée des points d'arrêt lorsque le contenu de l'expression change de valeur. L'arrêt peut être conditionnel : *watch i==10*.

En pratique : gdb

- *cont* : reprise de l'exécution après un arrêt.
- *next* : exécute la ligne suivante dans un programme suspendu.
- *step* : idem mais trace aussi les fonctions appelées.
- *set [option][paramètres]* : permet de changer les paramètres de gdb : *environment* pour une variable d'environnement du programme exécuté, *language* pour le langage source, *variable* pour changer à la volée le contenu d'une variable.
- *whatis variable* : type de la variable.
- *help [nom]* : informations sur la commande donnée en argument.
- *quit* : quitte gdb.

Sommaire

1 Compilateurs, options et directives
- Rôle et fonctionnement
- Directives de pré-compilation
- Principales options de compilation (compilateurs gnu)
- TP

2 Outils de construction de programmes
- Compilation séparée, édition de lien
- Makefile
- Autres outils de construction de programme
- CMake
- TP Makefile
- TP CMake

3 Debogage
- Principes du debogage
- Outils de debogage
- En pratique : gdb
- TP Debug

TP Debug : ex4

- Deboguer le code *debug.cpp*.
  Remarque : il contient 2 erreurs.