Introduction à la compilation

Anne Cadiou, Laurent Pouilloux,
Francis Chavanon, Gilles Galboud, **Laurent Carrel**

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et Séminaires Pour l'Informatique et le Calcul Scientifique
PMCS2I
Lundi 25 janvier 2021

CC BY SA

Pourquoi utiliser un compilateur ?

Un compilateur est un programme permettant de traduire un code écrit dans un langage de programmation en autre programme en langage machine.

Le langage machine est composé d'instructions et de données codées en binaire (composé de 0 et de 1). Chaque type de processeur possède son propre jeu d'instruction, donc est spécifique au processeur sur lequel on travaille.

Le compilateur doit donc

- analyser les sources du programme
- générer l'objet binaire correspondant
- assembler éventuellement différents objets en un exécutable binaire

Un peu d'histoire

- 1950-1970 Premiers compilateurs
  - 1957 Langage FORTRAN *FORmula TRANslator*
- 1970-1990 Processeurs électroniques
  - 1971 Intel 4004 pour PC
  - 1972 Langage C
  - 1983 Langage C++
  - 1983 premier compilateur open source GCC 1.0 (génère un binaire sémantiquement proche du programme source, peu d'optimisaion)
  - 1984 MIPS, architecture RISC
- 1990-2000 Développement de nouveaux concepts architecturaux (pipelines, processeurs superscalaires, mémoire caches, machine parallèles...)
  - Compilation à la volée (*just-in-time*, JIT) par opposition à sa réalisation classique, en avance de phase (*ahead-of-time, AOT*), utilisée dans LLVM

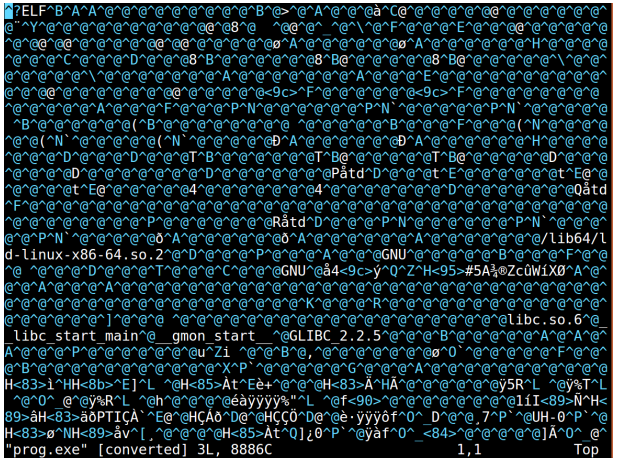
```
int main()
{
    int a=97;

    return 0;
}
```

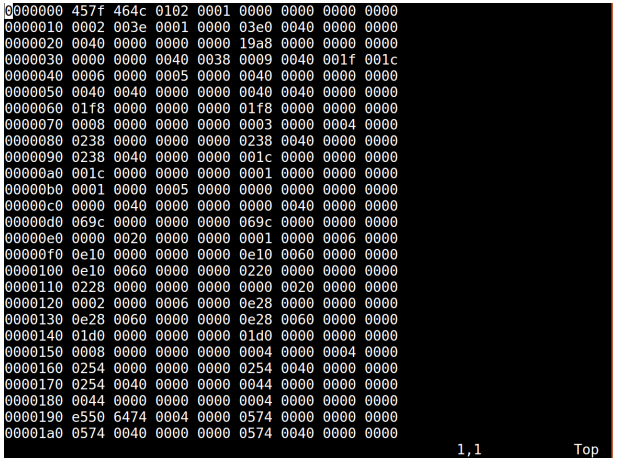
Programme C++ `prog.cpp`

```
  .file "prog.cpp"
  .text
  .globl main
  .type main, @function
main:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq %rsp, %rbp
  .cfi_def_cfa_register 6
  movl $97, -4(%rbp)
  movl $0, %eax
  popq %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size main, .-main
  .ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0
        20160609"
  .section .note.GNU-stack,"",@progbits
```

Programme correspondant en assembleur



Code compilé affiché en ASCII



Code compilé affiché en hexadécimal

Qu'attend-on d'un compilateur ?

Détection de erreurs :

- erreur typographique, parenthèses non fermées, etc.
- constructions syntaxiques incorrectes
- identificateur non déclaré
- expressions mal typées
- références non instanciées
- etc.

Les erreurs détectées à la compilation sont des erreurs **statiques**.
Les erreurs détectées à l'exécution sont des erreurs **dynamiques** (division par zéro, dépassement mémoire etc.)

Qu'attend-on d'un compilateur ?

Efficacité :

- il doit être le plus rapide possible
- il doit produire un code qui utilisera efficacement les ressources disponibles (et sera aussi rapide que possible)

Généralement, le compilateur produit en premier lieu un code intermédiaire, constitué d'une sére d'instructions bas niveau, sans optimisation. Il doit être optimisé pour produire un code exécutable.

La traduction binaire dépend

- du processeur utilisé
- des conventions utilisées par le système d'exploitation utilisé

L'optimisation du code intermédiaire se fait

- en diminuant la taille du code (réduction du nombre de variables, d'opérations, etc.)
- 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 d'un compilateur dépend du jeu d'instructions du processeur.

Opérations réalisées par un compilateur

- Prétraitement (preprocessing) : inclue des directives, insère des macros et permet la compilation conditionnelle
- Analyse lexicale (lexer) : identifie les lexèmes (unités lexicales) du langage (mot-clé, identifiant etc.)

```
float x = #;
```

```
error: stray '#' in program
```

- Analyse syntaxique (parsing) : trouve la structure syntaxique (arbre) et teste l'appartenance au langage

```
(x > 0.) x=+1;
else x=-1;
```

```
error: 'else' without a previous
       'if'
```

- Analyse sémantique : vérifie la cohérence des instructions reconnues

```
float y;
y = x + 1.0;
```

```
error: 'x' was not declared
       in this scope
```

- Génération du code : encodage en assembleur, optimisations et allocation des registres, traduction en code objet.

Lorsque qu'un code compilé sur une machine peu s'exécuter sur une autre, on parle de compilation croisée (cross compiler)

Après la compilation

- La compilation d'un programme se fait souvent de manière **modulaire**. Les dépendances par rapport aux autres parties du programme ou aux bibliothèques sont faites par l'**édition de liens**.
- La compilation peut être intégrée à un **runtime** (environnement d'éxecution), qui fournit des services indépendants du programme (accès systèmes, I/O, gestionnaires de mémoire, etc.). Le code natif est éxecuté par le compilateur, alors que le code objet l'est par le runtime. Pour les langages interprétés, le runtime est l'interpréteur.

Phase de pré-processing

Le pré-processeur procède à des transformations du code source. Il enlève les commentaires et exécute les instructions commençant par un dièse.
Le plus utilisé est celui de C et C++. Actions possibles :

- **inclusion** d'autres fichiers
- **définition** de constantes symboliques et de macros
- **compilation conditionnelle** de morceaux de programme
- **exécution conditionnelle** des directive du précompilateur

En fortran les inclusions sont dépréciées au profit de l'usage des modules. Le préprocesseur est automatiquement appelé sur les fichiers d'extension .F90, .F95, etc. (doit être spécifié par une option de compilation sinon)

Directives de pré-compilation

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

```
#include <iostream>
```

`#define const` : permet de définir des constantes

```
#define PI 3.14159265358979323846
```

`#define macro` : permet de définir des macros avec ou sans arguments

```
#define PRINTA(a) { if (myid==0) std::cout << a; }
```

En C++, les macros sont remplacées par les modèles et fonctions inline

```
inline int min(int a,int b)
{
   return (a<b)? a:b;
}
```

Elles sont définies jusqu'à la fin du fichier où elles sont déclarées ou jusqu'à ce qu'elles soient supprimée par la directive `#undef`

```
#undef PI
```

Macros pré-définies

`__LINE__`	numéro de ligne en cours dans le code source
`__FILE__`	nom du fichier source
`__DATE__`	date de compilation
`__TIME__`	heure de compilation
`__STDC__`	index de compatibilité avec la norme ANSI

Exemple d'usage :

```
#define XYC_Only(vec)\
{\
  if (!(vec).XYC())\
    {\
      std::cout << __FILE__ << ", line "\
                << __LINE__ << ", compiled "\
                << __DATE__ << " at " << __TIME__ <<" ***" << std
                    ::endl;\
      ERREUR("Vecteur3D is not XYC()");\
     }\
}
```

On note que le symbole \ permet d'écrire sur plusieurs lignes.
https://gcc.gnu.org/onlinedocs/cpp/

Autres directives

`#error :` permet de faire du diagnostic en affichant un message d'erreur

```
#if !defined(FOO) && defined(BAR)
#error "BAR requires FOO."
#endif
```

`#pragma :` provoque une action définie au niveau du compilateur

```
#ifdef __BORLANDC__
#pragma message You are compiling using Borland C++ version
    __BORLANDC__.
#else
#pragma message ("This compiler is not Borland C++")
#endif
```

```
#pragma omp for
for (int i=0; i<N; i++) buff1[i] *= buff2[i];
```

`#line :` provoque la numérotation des lignes de code source

```
#line 10 myfunc.C
```

Autres directives

 # : conversion d'une série de caractères en une chaîne de caractères entourée de guillements

```
#define BONJOUR( x ) std::cout << " Bonjour , " #x << sdt::endl
```

L'appel à `BONJOUR(Jean)` sera développé en

```
sdt::cout << " Bonjour , " " Jean " << std::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("STOP") ;
  AFFICHE(1) ;
  AFFICHE(2) ;
  AFFICHE(3) ;
}
```

Compilation conditionnelle

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

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

`#ifdef #ifndef` : raccourcis pour `#if define(KEY)` et `#if !define(KEY)`

- évite d'include le même fichier plusieurs fois
- aide par exemple au débogage en permettant d'insérer des portions de codes à ne compiler qu'en mode debug

Exemple classique d'utilisation en en-tête des fichiers à inclure

```
#ifndef parallel_H
#define parallel_H
  ...
#endif
```

Exemple d'utilisation en mode debug
Compilation avec l'option -DDEBUG

```
#ifdef DEBUG
  MPI_Comm_set_errhandler(
      MPI_COMM_WORLD,
      MPI_ERRORS_RETURN);
#endif
```

Principales options de compilation

Pour `gcc`

- `-c` : compilation et assemblage sans édition de liens ; génère un fichier objet d'extension `*.o`.
- `-S` : génère un fichier en assembleur d'extension `*.s`, sans faire l'assemblage.
- `-E` : stoppe après l'étape de pré-processing. le code généré est envoyé sur la sortie standard.
- `-o file` : renvoie le fichier compilé dans `file` ; par défaut, l'exécutable s'appelle `a.out`.
- `-v` : sort les différentes commandes exécutées pendant les étapes de compilation (mode `verbose`.
- `-ansi` : impose une compatibilité avec les standards iso.

Contôle des warnings, options de débogage

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

Optimisation d'optimisation

- `-O0` : aucune optimisation
- `-O1` : optimisation visant à accélérer le code, en particulier quand il ne contient pas beaucoup de boucles
- `-O2` : O1 et déroulage de boucles ; augmente le temps de compilation
- `-O3` : O2 et transformation des boucles, des accès mémoire
- `-march=cpu_type` : génère les instructions adaptées au jeu d'instructions du processeur spécifié

Certaines optimisations peuvent modifier l'ordre des opérations et donc affecter les résultats

Options du pré-processeur, option du link

- `-DKEY ou -DKEY=valeur` : affectation d'une valeur à une macro
- `-UKEY` : annule toute affectation à la clef `KEY`
- `-ldirectory` : 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 spécfiés
- `-M` : définit les règles 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++
- `-static` : force le linkage statique avec les autres librairies

Outils de construction de programmes

- encapsulés dans les IDE,
- autotools, automake,
- Makefile,
- CMake,
- scons, etc.

Compilation séparée, édition de lien

- séparer le source en plusieurs fichiers
- création de plusieurs fichiers objets
- édition de liens : assemblage de ces fichiers en un exécutable

`bibliothèque statique` : include dans l'exécutable. 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 `/etc/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.

Makefile

- automatise le 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

```
cible: dependances
       commandes
```

```
exe: vector.h main.cpp
     g++ main.cpp -o exe
```

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
exe : main.o
      g++ main.cpp -o exe

# Regle 1
main.o : vector.h main.cpp
      g++ -c main.cpp

# Regle 2 :
clean :
  rm main.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.

Variables

- déclaration : NOM = VALEUR
- utilisation : $NOM

  variables 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 :

```
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'inférence

Ce sont de 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 des fichiers source

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 dufichier

```
OBJ=$(notdir \$(patsubst %.cpp,%o,$(SRC)))
```

Utilisation de conditions

Utilisation :

```
make MODE_DEBUG=yes
```

Exemple :

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

Projet simple

Programme seul, sans dépendances.

```
#include <iostream>

int main()
{
    std::cout << "'Hello, world!'\n";
}
```

Structure du projet

```
project/
    └── src/
        └── prog.cpp
```

Objectif : compiler le code, l'installer, le tester.

Compilation manuelle

en utilisant les compilateurs de GNU

```
cadiou@moon:~/project/src$ g++ prog.cpp
```

Génération de l'exécutable

```
cadiou@moon:~/project/src$ ls
cadiou@moon:~/project/src$ a.out prog.cpp
cadiou@moon:~/project/src$ ./a.out
'Hello, world!'
```

Création manuelle d'un Makefile

```
project/
    ├── src/
    │    ├── prog.cpp
    │    └── Makefile
    └── bin/
```

```
cadiou@moon:~/project/src$ make -n
g++ -c prog.cpp
g++ -o a.out prog.o
cadiou@moon:~/project/src$ make
cadiou@moon:~/project/src$ make install
mv a.out ../bin/.
cadiou@moon:~/project/src$ cd ../bin
cadiou@moon:~/project/bin$ ./a.out
```

Exemple de Makefile

```
CC = g++
CLINKER = g++

OBJS = prog.o

.SUFFIXES: .o .cpp
.cpp.o:
  $(CC) -c $(CCLAGS) $<

EXE = a.out

all : $(EXE)

$(EXE): $(OBJS)
  $(CLINKER) $(LDFLAGS) -o $@ $(OBJS)

clean:
  rm -f $(OBJS) $(EXE)
install:
  mv $(EXE) ../bin/.
```

Pourquoi ne pas se contenter d'appeler le compilateur ?

Dans les codes de calculs, la construction de l'éxecutable est une tâche parfois délicate. Elle dépend

- d'un **grand nombre de fichiers**
- avec de nombreuses **dépendances**
- du cas à traiter (compilation **conditionnelle**, ...)
- de nombreux **éléments externes** (bibliothèques, ...)
- et de la manière dont ils sont installés
- de la **plateforme**
- de l'**environnement** sur le plateforme

La plupart du temps, on ne contrôle pas ces dépendances.

Résultats

Résolution d'un système linéaire par le méthode S.O.R. $64 \times 64 \times 64$ (2002)

Système	Cpu	Compilateur	Options	Time
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Intel(R) Fortran Compiler for 32-bit applications Linux Version 5.0.1 Build	-O3 -tpp7 -axW -ipo	61s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Absoft f90	-O -B100	84s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	GNU g77	-s -mcpu=pentiumpro -march=pentiumpro -mpentiumpro -O6 -frerun-cse-after-loop -fno-defer-pop -fschedule-insns -fomit-frame-pointer -fstrength-reduce -fforce-mem -fforce-addr -funroll-loops -freduce-all-givs -Wall	85s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Portland Group pgf90	-fast	88s
Alpha Server Compaq DS20 OSF 4.0F	Mémoire : 1.2 Gb Cache Prim. : ? Cache Second. : 4 Mb	DIGITAL f90	-O4 -tune ev6 -arch ev6 -fast	99s
Alpha Server Compaq 8200 OSF 4.0F	Mémoire : 2 Gb Cache Prim. : ? Cache Second. : 4 Mb	DIGITAL f90	-O4 -tune ev6 -arch ev6 -fast	150s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	GNU g77	.	173s
Linux 2.4.2-2 (Redhat 7.2)	Pentium III Coppermine 993.33Mhz Memory : 512Mb Cache : 8K L1, 256K L2	Portland Group pgf90	-fast	366s
Linux 2.4.2-2 (Redhat 7.2)	Pentium III Coppermine 993.33Mhz Memory : 512Mb Cache : 8K L1, 256K L2	GNU g77	.	466s

Références

Certaines planches de cette présentation sont directement reprise des cours suivants (dont la lecture vous est recommandée !) :

1. **Compilation, debogage**, Violaine Louvet, École doctorale MathIf, 2009
2. **Introduction à la compilation**, Christine Paulin-Mohring, Université Paris-Sud, 2011

Merci à leurs auteur(e)s respectives.