

LIFAPCD

CONCEPTION ET DEVELOPPEMENT D'APPLICATIONS

Université Claude Bernard Lyon 1
Licence Informatique 2^{ème} année
Nicolas Pronost

Outils pour la programmation

Outils pour la programmation

- **Compilation de fichier (GCC)**
- Compilation de projet (Makefile)
- Débogage (GDB)

GCC : compilation

- Commande de compilation

```
g++ -Wall -c fichier.cpp [-o fichier.o]
```

- **-Wall** (all Warnings) affiche tous les messages de prévention (ambigüités, type cast, oublis, etc.)
- **-c** demande la production d'un fichier objet
- **-o** donne le nom du fichier objet créé (optionnel, si omis même nom que le .cpp)

GCC : édition des liens

- Commande de création de l'exécutable

```
g++ fichier(s).o -o executable
```

- **fichier(s).o** indique le(s) fichier(s) objet(s) à utiliser
- **-o executable[.exe/out]** indique le nom du programme à créer

GCC : librairie

- Comme on souhaite le plus souvent réutiliser du code existant (optimisé et testé), vos projets vont inclure du code externe
 - Dans un ou plusieurs fichiers sources (.h ou .cpp)
`#include <NomLib.h>` (ou `"NomLib.h"`)
- Soit vous souhaitez **modifier** le code externe
 - Récupération du code source si disponible (.h et .cpp) et ajout aux commandes de compilation et d'édition des liens de tous ces fichiers
- Soit vous souhaitez uniquement **utiliser** le code externe
 - Récupération sous forme de librairie (code compilé, .h et .a/.so) et ajout de la référence à la compilation et à l'édition des liens
 - Attention à l'architecture (OS, compilateur, 32/64 bits etc.)

GCC : librairie - compilation

- A la compilation

```
g++ -Wall -IrepLib -c fichier.cpp
```

- **-IrepLib** indique qu'il faut aller chercher le .h de la librairie (**NomLib.h**) dans le répertoire **repLib**
 - Chemin absolu ou relatif
- Le répertoire **/usr/include** est automatiquement ajouté à toute compilation
 - Les headers des librairies standards y sont présents (ex. iostream, string, math etc.)

GCC : librairie – édition des liens

- A l'édition des liens

```
g++ fichier(s).o -o executable -LrepLib -lNomLib
```

- **-L** spécifie le chemin vers les librairies (.a ou .so)
 - **a** pour une librairie statique où le code sera intégré à l'exécutable
 - **so** pour une librairie dynamique où le code restera dans la librairie et il faudra donc accompagner l'exécutable de la librairie
- Le répertoire **/usr/lib** est automatiquement ajouté à toute compilation
 - Les librairies standards y sont présentes
- **-l** spécifie le nom de la librairie à lier

GCC : librairie – exemple

- Dans le .cpp

Calendrier.cpp

```
#include "Calendrier.h"    // inclusion de la déclaration de la classe
#include <string>           // inclusion de la librairie string
#include "Date.h"          // inclusion du module (librairie) Date
...
```

- Commande de compilation

```
g++ -Wall -I../extern/include -c Calendrier.cpp
```

- Commande d'édition des liens

```
g++ Calendrier.o main.o -o exec -L../extern/lib -lDate
```

Outils pour la programmation

- Compilation de fichier (GCC)
- **Compilation de projet (Makefile)**
- Débogage (GDB)

Makefile

- Ensemble de règles suivant le format:

cible: dépendances <EOL>
<tabulation>commande

- Exemple

Makefile

```
projet: main.o Vecteur.o Matrice.o Triangle.o
    g++ main.o Vecteur.o Matrice.o Triangle.o -o projet

Triangle.o: Triangle.cpp Triangle.h
    g++ -Wall -c Triangle.cpp

Vecteur.o: Vecteur.cpp Vecteur.h
    g++ -Wall -c Vecteur.cpp

...
```

Makefile : variables

- On peut aussi déclarer des variables dans un Makefile

Makefile

```
OBJS = main.o Vecteur.o Matrice.o Triangle.o
FLAGS = -ggdb -Wall

projet: $(OBJS)
    g++ $(FLAGS) -o projet $(OBJS)

...
```

- Remarque
 - **-ggdb** inclus des informations de debug pour gdb

Commande make : utilisation

- **make** tout seul produit la première règle du Makefile
 - commencer par la règle de génération du projet
- **make projet** construit la cible **projet** (i.e. exécute la commande associée)
- On peut ajouter des règles pour lancer des commandes autres que g++ (ex. génération de doc, commit forge, tests de régression, valgrind)

clean:

rm obj/*.o

doc:

doxygen doc/doxyfile

Exemple

Makefile

```
EXEC_NAME = nom_de_l_executable
OBJ_FILES = main.o maClasse.o

CC = g++
CFLAGS = -Wall -ggdb
INCLUDES = -I./extern/include
LIBS = -L./extern/lib -luneLib

all: $(EXEC_NAME)

$(EXEC_NAME): $(OBJ_FILES)
    $(CC) $(OBJ_FILES) -o $(EXEC_NAME) $(LIBS)

main.o: main.cpp maClasse.h
    $(CC) $(CFLAGS) $(INCLUDES) -c main.cpp

maClasse.o: maClasse.cpp maClasse.h
    $(CC) $(CFLAGS) $(INCLUDES) -c maClasse.cpp

clean:
    rm $(OBJ_FILES)
```

Et quand je change d'IDE...

- Beaucoup d'IDE peuvent compiler grâce à un Makefile fourni (ex. CodeBlocks, Visual Studio)
- Certains non, et ça serait bien d'avoir un outil qui gère les différences entre les formats de projet des différents IDE
- **CMake** est un système de génération de projets aux formats des IDE
- Il est configuré dans un fichier `CMakeLists.txt` à la racine du projet qui décrit comment construire le projet
 - Par convention dans un dossier `build`

Exemple

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.26)

project(monProjet)

set(SRCS ./src/main.cpp ./src/maClasse.cpp)
set(INCLUDES ./extern/include)
set(DIR_LIBS ./extern/lib)
set(LIBS uneLib)

add_executable(nom_executable ${SRCS})

target_include_directories(nom_executable PUBLIC ${INCLUDES})

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -ggdb")

target_link_directories(nom_executable PUBLIC ${DIR_LIBS})

target_link_libraries(nom_executable ${LIBS})
```


Commande CMake

- Se placer dans le répertoire du projet (où il y a le fichier `CMakeLists.txt`)
- Créer le dossier build et y aller

```
$> mkdir build  
$> cd build
```

- Générer le projet (ici un Makefile)

```
$> cmake ..
```

- Compiler le projet puis l'exécuter

```
$> make  
$> ./nom_executable
```

Commande CMake

- Pour de l'aide (dont la liste des générateurs)

```
$> cmake --help
```

- Pour générer le projet avec un IDE particulier

```
$> cmake -g "Unix Makefiles"  
$> cmake -g "MinGw Makefiles"
```

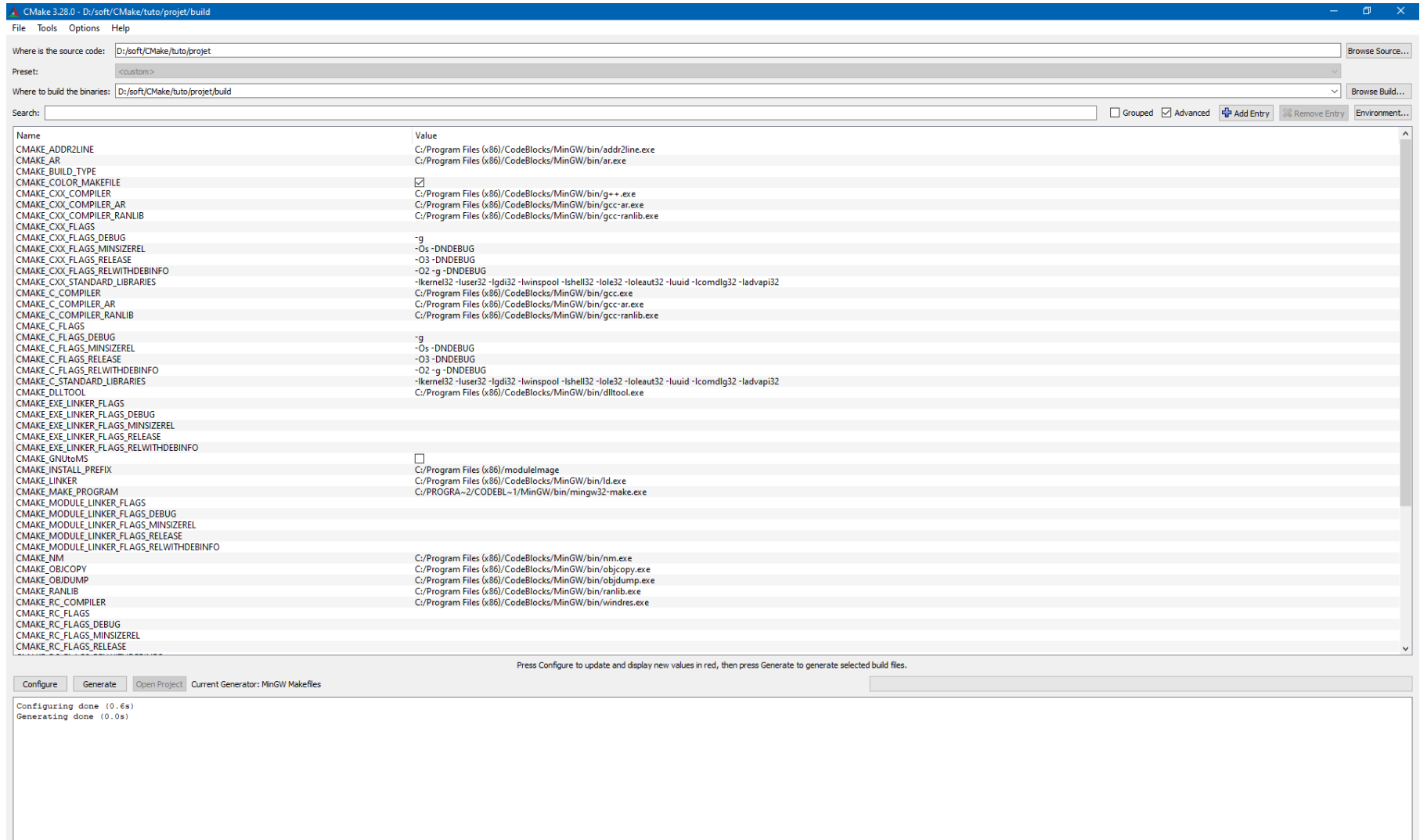
- Pour avoir le numéro de version de CMake

```
$> cmake --version
```

- Pour spécifier le dossier source et le dossier destination

```
$> cmake -S source -D destination
```

cmake-gui



Outils pour la programmation

- Compilation de fichier (GCC)
- Compilation de projet (Makefile)
- **Débogage (GDB)**

Questions

- Mon programme compile, est-ce gagné ?
- Mon programme ne marche pas, quelle solution ai-je pour le corriger ?
- Ai-je besoin de le comprendre pour le réparer ?
- Mon programme fait ceci

```
$> ./bin/prog  
Segmentation fault (core dumped)
```

Ca veut dire quoi ? D'où vient l'erreur ? Comment réparer le code ?

Débogage

- Ce n'est pas parce qu'un programme compile qu'il fait ce que l'on veut!
 - 20% du temps à écrire le code
 - 10% du temps à le faire compiler
 - 70% à le déboguer
- 2 manières principales de déboguer
 - Des `cout` (ok dans le cas d'une boucle où tous les cas nous intéressent)
 - Un **débogueur** (ou débogueur, debugger, ...)

Débogage

Segmentation fault (core dumped)

- Une **erreur de segmentation** (en anglais, parfois abrégé segfault) est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué
- Erreur à l'exécution très classique (vous en ferez...)
- **core dump** signifie que le système a sauvegardé tout l'état de la mémoire dans un fichier *./core*

Débugger / mettre au point avec GDB

- Suivi de l'exécution d'un programme: pas à pas, et/ou par arrêts spécifiques (breakpoints)
 - Inspection de la valeur des variables
 - Modification "en live" de la valeur des variables lors de l'exécution
 - Options de GCC pour utiliser gdb
 - Ajouter `-ggdb` dans les options de compilation et dans les options de la création de l'exécutable
- ```
g++ -ggdb -Wall -c main.cpp
g++ -ggdb main.o -o exec
```



# GDB : utilisation courante

- Commandes générales

```
$> gdb ./exec
(gdb) run
```

- **run** (ou juste **r**) pour lancer le programme
- **backtrace** (**bt**) pour localiser la fonction et la ligne de l'erreur (si erreur système type SegFault)
- **where** pour afficher où le programme s'est arrêté
- **list** pour afficher les lignes de code suspectes
- **quit** pour sortir de gdb

# GDB : utilisation courante

- Placer un point d'arrêt  
`break fonction` ou `break fichier:ligne`  
`tbreak` pour un point d'arrêt à 1 passage
- Reprendre l'exécution  
`next`, `step`, `continue`
- Afficher la valeur d'une variable  
`print variable`

# Déboguer dans un IDE

- Heureusement on est pas toujours obligé de passer par des lignes de commande
- La plupart des IDE fournissent un outil de débogage intégré
  - parfois juste une interface graphique avec GDB (ex. CodeBlocks)