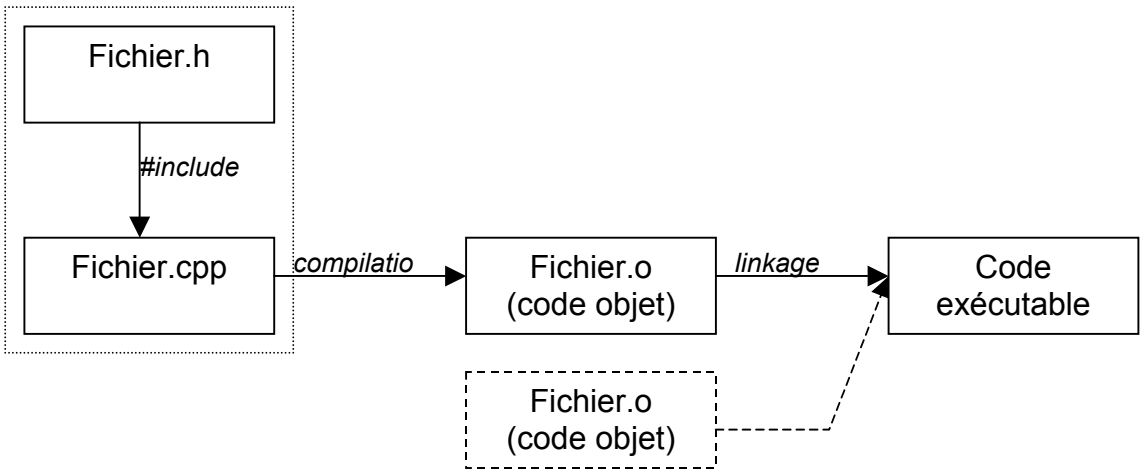
# La programmation C/C++ sous Linux

## 1. Rappel : la création d'un exécutable

Les différentes étapes pour arriver à la création d'un exécutable à partir d'un fichier source écrit en C, C++ ... sont les suivantes :




## 2. Les outils fournis par Linux

Linux propose plusieurs outils de développement. Ils sont fournis avec toutes les distributions. On retrouve entre autres outils les célèbres compilateurs gcc et g++, versions GNU des compilateurs C et C++ issus du monde Unix.

Ces « compilateurs » sont aussi capables d'assurer le linkage et de produire des exécutables, librairies... à partir d'un ou plusieurs fichiers sources, objets...

Le compilateur C : cc
Le compilateur C : g++

### 2.1. Les options principales de ces deux compilateurs

| Option | Rôle |
|---|---|
| **-g** | Génère les informations de débugage pour gdb |
| **-o** | Permet de spécifier le nom du fichier à produire |
| **-c** | Réalise simplement la compilation. Ne produit que le fichier .o |

*Exemples :*

```
cc    hello.c -o hello    ➔ Crée un exécutable appelé hello à partir du fichier c hello.c
g++   -c tClasse.cpp      ➔ Crée le fichier tClasse.o (code objet)
```

### 2.2. Création d'exécutables à partir de plusieurs fichiers

Ces compilateurs sont capables de travailler à partir de plusieurs fichiers. Il suffit de préciser la liste des différents fichiers :

*Exemple :*
Soit un projet composé de :

| | | |
|---|---|---|
| test.cpp | : | Fichier contenant le programme de test d'une classe |
| tClasse.cpp | : | Fichier contenant la classe à tester |
| tService.o | : | Fichier « bibliothèque » fourni sous forme de code objet |

Création de l'exécutable TestClasse :

**g++ -g test.cpp tClasse.cpp tService.o -o TestClasse**

## 3. L'utilitaire make

Lorsque l'on travaille avec des projets comprenant plusieurs fichiers (sources, librairies, objets...) il est fastidieux de taper à chaque fois la (les) lignes permettant de générer les fichiers finaux.
Un utilitaire permet de regrouper tous les travaux et de les lancer en fonction des besoins de manière séquentielle. Cet utilitaire s'appelle **make**.
Il utilise des fichiers appelés **makefile**. L'utilitaire make n'est pas un simple script qui permettrait de lancer à la suite les différentes commandes de compilation. Il est capable de ne travailler que sur les fichiers qui ont été modifiés, de gérer des dépendances entre plusieurs fichiers, de nettoyer automatiquement les fichiers inutiles...

### 3.1. Syntaxe d'un fichier makefile

On trouvera dans un fichier makefile plusieurs choses :

- des variables
- des règles de génération
- des règles de dépendance

Pour assurer une souplesse maximale, on pourra utiliser des variables pour indiquer le compilateur à utiliser, les options de compilation...
Par exemple : CC=g++
CFLAGS=-g –c

Les règles de dépendance permettent à make de déterminer les fichiers nécessaires à la production d'un autre fichier.
Par exemple : test : test.o tClasse.o indique que les fichiers nécessaires à la production de test sont test.o et tClasse.o
Test.o : test.cpp indique que le fichier test.o sera généré à partir de test.cpp
TClasse.o : tClasse.cpp tClasse.h Il faudra que les deux fichiers tClasse.cpp et tClasse.h soient présents pour générer tClasse.o
Les règles de génération vont réaliser les appels réels aux utilitaires (compilateurs...) une telle règle doit se trouver sur une ligne et débuter **après une tabulation** (pas des espaces). Si ceci n'est pas respecté, make va générer une erreur. Attention donc au comportement des éditeurs vis-à-vis des tabulations.

### 3.2. Exemple de fichier makefile

L'exemple présenté ici montre comment générer un fichier exécutable appelé test à partir d'un fichier source test.cpp. Le but de cet exécutable est de tester une classe tClasse. Elle est décrite dans deux fichiers tClasse.cpp et tClasse.h.
Les lignes débutant par # sont des commentaires.

```
#Compilateur
CC=g++

#Options de compilation
DEBUG=-g
COMPILATION_SEULE=-c

#Fichier final – règles de dépendance
test : test.o tClasse.o
#Fichier final, règles de génération
	$(CC) $(DEBUG) test.o tClasse.o -o test

#Fichier test.o, dépendances et règles de production
test.o : test.cpp
	$(CC) $(DEBUG) $(COMPILATION_SEULE) test.cpp

#Fichier tClasse.o
tClasse.o : tClasse.cpp tClasse.h
	$(CC) $(DEBUG) $(COMPILATION_SEULE) tClasse.cpp

#Nettoyage après génération
clean :
	rm -f *.o
```

Explications :

L'appel de make seulement va générer le fichier test. Si les fichiers nécessaires ne sont pas disponibles, ils vont être générés d'après les règles définies dans le fichier makefile.
L'appel : make tClasse.o va simplement générer le fichier tClasse.o

Remarque : si le fichier tClasse.o est à jour, il ne sera pas régénéré.

L'appel : make clean va exécuter la commande rm -f *.o c'est à dire la suppression de tous les fichiers objet qui ne sont plus nécessaires une fois que l'exécutable est créé.

## 4. Le débugueur intégré gdb

Linux propose en plus des compilateurs, un débugueur en ligne de commande : **gdb**.
Pour qu'il soit possible de tracer un programme avec cet outil, il faut avoir compilé le(s) fichier(s) source(s) avec l'option **-g**.

### 4.1. Lancement du débugueur

En général, on lance le débugueur en lui passant comme argument le nom de l'exécutable à tracer : **gdb TestClasse**.

### 4.2. Utilisation de gdb

| Commande | Raccourci | Rôle |
|---|---|---|
| Run | r | Lancer le programme |
| Step | s | Pas à pas dans le programme |
| Next | n | Idem s mais sans passer par les fonctions |
| Continue | c | Reprendre le cours normal du programme |
| Breakpoint | b | Placer un point d'arrêt |
| Print | p | Afficher le contenu d'une variable |
| list | l | Afficher le listing des prochaines lignes du programme |
| help | | Afficher l'aide sur gdb |

## 5. Création de librairies partagées

On ne trouve pas sous Linux les célèbres fichiers DLL. Il existe cependant l'équivalent : les librairies partagées (fichier so).

### 5.1. Règles de nommage des librairies partagées

Les bibliothèques partagées ont deux noms :

- Le nom propre, qui est enregistré dans la bibliothèque et est recherché par le chargeur dynamique lors de l'exécution du programme.
- Le nom physique, son utilité se limite aux chemins d'accès.

Cette distinction permet de faire coexister plusieurs versions différentes de la même librairie sur un seul système.

Les conventions de nommage sont :

**lib***nom*.**so**.*majeur.mineur*

Prenons un exemple : une librairie permettant de gérer les nombres complexes dont le nom physique est **libcomplexe.so.1.0**.
Son nom propre est **libcomplexe.so.1**.

Pour un fonctionnement en accord avec la philosophie générale des bibliothèques partagées, on pourra placer le fichier produit dans un répertoire standard : /usr/lib par exemple.
Il faudra ensuite créer des liens symboliques vers ce fichier : libcomplexe.so.1 → libcomplexe.so.1.0 et libcomplexe.so → libcomplexe.so.1.
Ce travail pourra être réalisé par l'utilitaire **ldconfig**.

### 5.2. Règles d'incrémentation du mineur/majeur

Lorsque le développeur corrige des erreurs dans la librairie, ou lorsqu'il ajoute des fonctions, il sera nécessaire de recompiler cette bibliothèque.
Il faudra alors incrémenter le numéro mineur. Pour des modifications qui altèrent le fonctionnement des logiciels existants, il faudra incrémenter le numéro majeur.
Par exemple : libcomplexe.so.1.0 après modifications majeures, va devenir libcomplexe.so.2.0, son nom propre deviendra libcomplexe.so.2. Le lien libcomplexe.so va devoir être recréé.
Si les logiciels utilisent comme nom de librairie libcomplexe.so, ils utiliseront toujours la dernière version présente sur le système.

### 5.3. Création d'une librairie partagée

La première chose à faire lorsque l'on désire créer une librairie partagée est de compiler le source. On va devoir utiliser une option particulière de g++ : -fPIC (Position Independant Code).

➔ **g++ -fPIC -c libtest.cpp** (production de libtest.o à partir de libtest.cpp)

Il faudra ensuite générer « l'exécutable », c'est à dire la librairie en elle-même. Pour cela, c'est une édition de liens (avec g++) avec des options particulières :

➔ **g++ -shared -Wl,-soname,libtest.so.1 -o libtest.so.1.0 libtest.o**

On génère ainsi un exécutable partagé (-shared) avec un nom propre libtest.so.1 (-Wl,-soname,.....). Le fichier produit s'appelle libtest.so.1.0 (-o .....) et le fichier objet à utiliser est libtest.o .

On peut, bien sur, si on le désire ajouter des options de compilation (-g par exemple...).

### 5.4. Utilisation de fonctions d'une bibliothèque partagée

On retrouve ici le principe déjà décrit dans l'utilisation de fonctions de DLL de manière dynamique sous Windows. A savoir : chargement de la librairie en mémoire, récupération des adresses des fonctions et utilisation à travers des pointeurs de fonctions.
Les fonctions permettant ce travail sont prototypées dans dlfcn.h, qu'il ne faudra pas oublier d'inclure à notre projet utilisant des librairies partagées.

Les fonctions utiles sont :

- dlopen pour se lier à la bibliothèque
- dlsym pour récupérer les adresses des fonctions
- dlerror En cas d'erreur
- dlclose Pour décharger la bibliothèque

*Remarque :*

*Pour utiliser une bibliothèque partagée, il faut signifier au compilateur (linkeur) que l'on utilisera des fonctions situées dans une telle bibliothèque. Pour cela, on utilisera l'option de compilation -ldl de g++.*

# 6. Exemple complet pour l'utilisation d'un fichier makefile

Fichier de prototypage de la classe tClasse : **tClasse.h**

```
class tClasse
{
private:
char* Nom;

public:
    tClasse(char* apNom);
    ~tClasse();
void DireBonjour(void);
void DireAuRevoir(void);

};
```

Fichier source correspondant : **tClasse.cpp**

```
#include <string.h>
#include <stdio.h>
#include "tClasse.h"

tClasse::tClasse(char* apNom)
{
Nom=new char[strlen(apNom)+1];
strcpy(Nom,apNom);
}

tClasse::~tClasse()
{
delete Nom;
}

void tClasse::DireBonjour(void)
{
printf("Bonjour %s\n",Nom);
}

void tClasse::DireAuRevoir(void)
{
printf("Au Revoir %s\n",Nom);
}
```

Fichier de test : **test.cpp**

```
#include "tClasse.h"

void main(void)
{
tClasse Objet("Toto");

Objet.DireBonjour();
Objet.DireAuRevoir();

}
```

Fichier makefile correspondant :

```
CC=g++

test: test.o tClasse.o
	$(CC) -g test.o tClasse.o -o test
test.o: test.cpp
	$(CC) -g -c test.cpp
tClasse.o: tClasse.cpp tClasse.h
	$(CC) -g -c tClasse.cpp
clear:
	rm -f *.o
```

## 7. Exemple complet pour la génération et l'utilisation d'une librairie partagée

Fichier de prototypage des fonctions : **libtest.h**

```
extern "C" void Hello (void);
extern "C" void Bonjour(char* Nom);
extern "C" int Somme(int a,int b);
extern "C" float DonnerPi(void);
```

Fichier source correspondant : **libtest.cpp**

```
#include <stdio.h>
#include "libtest.h"

//-------------------------------------------------------------------------
extern "C" void Hello(void)
{
printf("Hello World\n");
}
//-------------------------------------------------------------------------
extern "C" void Bonjour(char* Nom)
{
printf("Bonjour %s\n",Nom);
}
//-------------------------------------------------------------------------
extern "C" int Somme(int a,int b)
{
return (a+b);
}
//-------------------------------------------------------------------------
extern "C" float DonnerPi(void)
{
float Pi=3.14;
return(Pi);
}
//-------------------------------------------------------------------------
```

Fichier source de test des fonctions : **test.cpp**

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

typedef void (*Hello_t)(void);
typedef void (*Bonjour_t)(char* Nom);
typedef int (*Somme_t)(int a,int b);
typedef float (*DonnerPi_t)(void);
```

```
void main(void)
{
void* Handle;
Hello_t     Hello=NULL;
Bonjour_t Bonjour=NULL;
Somme_t Somme=NULL;
DonnerPi_t DonnerPi=NULL;

int Res=0;
float Pi;

Handle=dlopen("libtest.so.1.0",RTLD_LAZY);
if (Handle==NULL)
    {
    printf("Erreur ouverture librairie\n");
    exit (-1);
    }
// Recup adresses fonctions
Hello = (Hello_t) dlsym (Handle,"Hello");
if (Hello==NULL)
    {
    printf("Erreur connexion avec Hello\n");
    exit (-1);
    }
Bonjour = (Bonjour_t) dlsym (Handle,"Bonjour");
if (Bonjour==NULL)
    {
    printf("Erreur connexion avec Bonjour\n");
    exit (-1);
    }
Somme = (Somme_t) dlsym (Handle,"Somme");
if (Somme==NULL)
    {
    printf("Erreur connexion avec Somme\n");
    exit (-1);
    }
DonnerPi = (DonnerPi_t) dlsym (Handle,"DonnerPi");
if (DonnerPi==NULL)
    {
    printf("Erreur connexion avec DonnerPi\n");
    exit (-1);
    }

// Essais
(*Hello)();
(*Bonjour)("Alexis");
Res = (*Somme)(2,3);
printf("Somme : %d\n",Res);
Pi=(*DonnerPi)();
printf("Pi = %f\n",Pi);

}
```

Fichier makefile commun :

```
CC=g++

libtest: libtest.o
      $(CC) -shared -Wl,-soname,libtest.so.1 -o libtest.so.1.0 libtest.o
libtest.o: libtest.cpp libtest.h
      $(CC) -fPIC -c libtest.cpp
test: test.o
      $(CC) -ldl test.o -o test
test.o: test.cpp

      $(CC) -c test.cpp
```