13. Fichiers sources, compilation séparée, visibilité

1 Introduction

Les fichiers contenant les instructions des programmes C sont nommés « fichiers sources » . Un programme [1] peut, ou non, être contenu dans un fichier source unique. Lorsqu'une même fonction est utilisée par des programmes différents, il est pratique de la mettre dans un fichier à part, ce qui évite d'avoir à la recopier dans le fichier de chaque programme l'utilisant. Un programme sera alors réparti dans plusieurs fichiers sources. Cette façon de procéder permet aussi :

— de rendre une variable globale entre certaines fonctions seulement et non entre toutes
— de faire qu'une fonction puisse être connue de certaines fonctions seulement et non de toutes.

2 Programme réparti dans plusieurs fichiers sources

Dans la suite `type` symbolisera un type quelconque (`int`, `double`, ...) et `x` une variable quelconque.
Soit par exemple le programme suivant inclus dans un fichier unique nommé *source_1.cpp* :

```
#include<iostream>
using namespace std;
//-------------------------
type f(type x) {            /* en-tete de la fonction f */
  ...
}
//-------------------------
int main() {
  f(...);                   /* appel de la fonction f */
}
```

Il comprend une fonction et le `main`. On peut placer la fonction dans un fichier à part, nommé par exemple *f.cpp*, qui contient uniquement :

```
type f(type x) {
  ...
}
```

Le programme principal est placé seul dans un autre fichier nommé *source_2.cpp* :

```
#include<iostream>
using namespace std;
type f(type x);
//-------------------------
int main() {
  f(...);
}
```

Il est identique à celui contenu dans *source_1.cpp* mais la déclaration [2] de la fonction est maintenant obligatoire. Quand on travaillera avec plusieurs fichiers sources, contrairement à ce qui a été fait jusqu'à présent avec un fichier source unique, on écrira explicitement tous les prototypes de fonctions. [3] Pour simplifier l'application de cette règle et éviter des redites on utilise la directive `#include`.
De façon générale, la directive `#include "mon_fich.h"` a pour effet d'insérer, avant la compilation, le contenu du fichier *mon_fich.h* par exemple, là où elle se trouve.
On place le ou les prototypes dans un fichier à part, qu'on nomme, par exemple, *proto.h*, qui, dans le cas de l'exemple précédent, contient uniquement :

```
type f(type x);             /* prototype de f */
```

et, dans le fichier contenant le programme principal, seule la troisième ligne est modifiée :

1. C'est à dire des directives, un `main` et des fonctions.
2. C'est à dire le prototype. En fait pour un prototype il suffit d'écrire `type f(type);`, c'est à dire sans le nom `x`. Mais si la fonction a beaucoup d'arguments il est souvent plus lisible d'inclure également les noms.
3. On rappelle que, dans le cas d'un fichier source unique, le prototype n'est pas obligatoire pour les fonctions dont l'en-tête précède l'utilisation dans le fichier.

```
#include<iostream>
using namespace std;
#include "proto.h"
//-------------------------
int main() {
  f(...);
}
```

et tout se passe exactement comme si on avait mis la ligne :

```
type f(type x);
```

à la place de la ligne :

```
#include "proto.h"
```

Le programme est maintenant réparti dans trois fichiers au lieu d'un, c'est à dire :

source_2.cpp, *f.cpp* et *proto.h*

au lieu de :

source_1.cpp

La compilation doit être faite par :

g++ -lm f.cpp source_2.cpp

l'ordre n'ayant pas d'importance. [4]

Désormais, tout programme, placé dans un fichier nommé par exemple *source_qqc.cpp*, contenant la directive :

```
#include "proto.h"
```

et compilé par :

g++ -lm f.cpp source_qqc.cpp

peut utiliser la fonction `f`.
f.cpp et *source_qqc.cpp* sont d'abord compilés séparément, puis reliés entre eux.

Remarque

La différence entre l'utilisation de < > et de " " dans les `#include` est qu'avec " " le compilateur cherche le fichier *.h* d'abord dans le même répertoire que le répertoire du fichier *.cpp* qui contient le `#include` (normalement le répertoire courant), tandis qu'avec < > il cherche directement dans les répertoires système.

3 Visibilité des variables globales entre fichiers sources

3.1 Rappel dans le cas d'un fichier source unique

3.1.1 Variables locales

Toutes les variables déclarées :

à l'intérieur du `main`
à l'intérieur ou en argument muet des fonctions

sont locales.

3.1.2 Variables globales

Les variables déclarées en dehors du `main` et des fonctions sont globales pour toutes les fonctions (y compris le `main`) suivant la déclaration dans le fichier source. Une exception arrive lorsqu'une variable locale porte le même nom qu'une variable globale : cette dernière est alors invisible dans la fonction contenant la variable locale.

3.2 Cas de plusieurs fichiers sources

3.2.1 Variables locales

Rien n'est changé.

4. L'option *-lm* est seulement nécessaire si l'un (ou plusieurs) des fichiers utilise `math.h`.

3.2.2 Variables globales

Soit par exemple les deux fichiers sources :
source_f.cpp

```
#include<iostream>
using namespace std;
#include "proto.h"
type p, q;
//-------------------------
int main() {
  ...
}
//-------------------------
type f1(...) {
  ...
}
//-------------------------
type f2(...) {
  ...
}
...
```

et :
source_g.cpp

```
#include "proto.h"
type a, b;
type g1(...) {
  ...
}
type g2(...) {
  ...
}
...
```

les prototypes des fonctions `f1`, `f2`, ..., `g1`, `g2`, ..., étant placés dans le fichier `proto.h`.
Pour ce qui va être dit dans la suite, les deux fichiers sources *source_f.cpp* et *source_g.cpp* doivent être vus comme complètement symétriques. En effet l'un contient le `main` mais celui-ci peut être considéré comme une fonction ordinaire.
Les variables `p`, `q`, ... sont globales pour toutes les fonctions du fichier source *source_f.cpp*, mais pas pour les fonctions du fichier source *source_g.cpp*.
Les variables `a`, `b`, ... sont globales pour toutes les fonctions du fichier source *source_g.cpp*, mais pas pour les fonctions du fichier source *source_f.cpp*.
Si l'on veut que `a` soit globale pour l'ensemble des deux fichiers, il faut la redéclarer dans *source_f.cpp* par :

```
extern type a
```

et symétriquement si on veut que `p` soit aussi globale pour les deux fichiers.

Si l'on veut empêcher que `b`, par exemple, puisse être rendue globale dans *source_f.cpp* par un :

```
extern type b
```

on la déclare :

```
static type b
```

dans *source_g.cpp*.

Remarque

le mot clé `static` employé dans ce contexte a une signification différente de celle qu'il a quand on veut signifier qu'une variable locale est statique. En effet `b` n'est pas locale puisqu'elle est globale et donc déjà statique.

Ainsi, même s'il existe une variable globale nommée `b` dans un troisième fichier source qu'on a rendue globale avec *source_f.cpp* en la déclarant dans ce dernier par :

```
extern type b
```

elle n'a rien à voir avec la variable globale `b` du fichier *source_g.cpp* qui n'est globale qu'à l'intérieur de ce dernier et il n'y a donc pas de conflit.
On voit donc qu'on peut déclarer des variables, globales au sein d'un fichier source, mais locales par rapport à d'autres fichiers source. Cette possibilité permet d'écrire des fichiers sources disposant en leur sein des avantages des variables globales, tout en pouvant être utilisés comme des boîtes noires, c'est à dire en ayant uniquement connaissance de ce qui entre et de ce qui sort.

Remarques

Les variables globales sont dites « privées » par rapport aux fichiers sources dont elles ne sont pas connues et « publiques » par rapport aux autres.

On rappelle que, de toutes façons, les variables globales ne doivent être utilisées qu'en dernier recours, car en établissant des liens entre des fonctions, elles rendent plus difficile de contrôler les conséquences des appels de ces fonctions.

4 Visibilité des fonctions entre fichiers sources

On se place toujours dans le cadre de l'exemple vu à la section précédente. Contrairement au cas des variables globales, les fonctions sont, à priori, toutes connues les unes des autres, on dit qu'elles sont « publiques » . Cela peut être un inconvénient pour assurer l'indépendance des fichiers sources. Supposons, par exemple, qu'on veuille créer de nouvelles fonctions dans *source_f.cpp* ou dans un nouveau fichier source. Il faut s'assurer en particulier que ces nouvelles fonctions ne portent pas le nom de fonctions déjà existantes dans *source_g.cpp*. Si, dans *source_g.cpp* seule la fonction `g1` est appelée depuis l'extérieur cela représente une contrainte inutile en ce qui concerne les fonctions autres que `g1`. Pour simplifier cela, l'attribut `static` donne la possibilité de rendre certaines fonctions connues uniquement à l'intérieur de leur fichier source[5]. Il suffit d'écrire l'en-tête de chaque fonction concernée :

```
static type g2(...);
```

`g2` est alors dite fonction « privée » du fichier source *source_g.cpp*.

5 Méthode générale pour constituer une bibliothèque avec *Linux*

Soit, par exemple, les trois fonctions suivantes, écrites respectivement dans les fichiers *combi.cpp*, *binomiale.cpp* et *eq_3d.cpp* : [6]

combi.cpp :

```
/* Calcul de C(n,k), pas d'avertissement en cas de dépassement */
#include "ma_bibli.h"
int combi(int n, int k) {
  int c, j;
  if(k > n-k)
    k = n-k;
  for(c = 1, j = 1; j <= k; j++)
    c = c*(n-j+1)/j;
  return c;
}
```

binomiale.cpp :

```
/* Fournit la valeur de la loi binomiale pour une proba p, un nombre d'épreuves n
   et un nombre de réalisations k */
#include<math.h>
#include "ma_bibli.h"
double binomiale(double p, int n, int k) {
  int i;
  double b, q = 1.-p;
  b = pow(q,n);
```

5. Comme dans le cas des variables globales.
6. On peut éviter la répétition des `#include` autre que `#include "ma_bibli.h"` dans chaque fichier en les mettant dans `ma_bibli.h`.

```
  for(i = 1; i <= k; i++)
    b = b*(n-i+1)/i*p/q;
  return b;
}
```

eq_ 3d.cpp :

```
/* Calcule les racines de l'équation du troisième degré ax^3+bx^2+cx+d=0 */
#include<iostream>
#include<math.h>
#include "ma_bibli.h"
using namespace std;
static double rac(double x) {
  double y;
  y = pow(fabs(x),1./3.);
  if(x < 0.)
    y = -y;
  return y;
}
int eq_3d(double a, double b, double c, double d, double* x) {
  double r, s, t, p, q, u, v, phi, rr, rrr, rs3, dps3;
  dps3 = 2*M_PI/3;
  if(a == 0.) {
    cout << "Equation de degre < 3" << endl;
    return -1;
  }
  r = b/a; s = c/a; t = d/a; p = s-r*r/3; q = 2*r*r*r/27 - r*s/3 + t; rs3 = r/3;
  u = q*q/4 + p*p*p/27;
  if(u > 0) {
    x[0] = rac(-q/2+sqrt(u)) + rac(-q/2-sqrt(u)) - rs3;
    return 1;
  }
  if(u == 0.) {
    v = 2*rac(-q/2);
    x[0] = v-rs3; x[1] = x[2] = -v/2-rs3;
    return 2;
  }
  rr = sqrt(-p*p*p/27); rrr = 2*sqrt(-p/3); phi = acos(-q/2/rr);
  x[0] = rrr*cos(phi/3)-rs3;
  x[1] = rrr*cos(phi/3+dps3)-rs3;
  x[2] = rrr*cos(phi/3-dps3)-rs3;
  return 3;
}
```

On veut mettre ces fonctions dans une bibliothèque, utilisable dans tout programme.
On place les trois fichiers *combi.cpp*, *binomiale.cpp* et *eq_ 3d.cpp* dans un répertoire quelconque, dont le nom complet est par exemple */home/lulu/mes_fonctions*. Dans ce même répertoire on écrit le fichier suivant, nommé obligatoirement *Makefile* :

```
CC=-c -Wall --pedantic
ma_bibli.ar : combi.o binomiale.o eq_3d.o
        ar -r ma_bibli.ar combi.o binomiale.o eq_3d.o
combi.o : combi.cpp
        g++ $(CC) combi.cpp
binomiale.o : binomiale.cpp
        g++ $(CC) binomiale.cpp
eq_3d.o : eq_3d.cpp
        g++ $(CC) eq_3d.cpp
```

Attention : au début des lignes commençant par `ar` et `g++` il ne faut pas mettre des blancs mais un caractère de

tabulation (appuyer une fois sur la touche *Tab*).
Toujours dans le même répertoire écrire le fichier nommé *ma_bibli.h* contenant les prototypes de chaque fonction :

```
#ifndef MA_BIBLI_H
#define MA_BIBLI_H
int combi(int, int);
double binomiale(double, int, int);
int eq_3d(double, double, double, double, double*);
#endif
```

Ensuite dans ce même répertoire taper la commande :

make

Toutes les fonctions sont alors compilées et, s'il n'y a pas d'erreur à la compilation, un fichier *ma_bibli.ar* est créé. La bibliothèque est alors prête à l'emploi. Pour utiliser ces fonctions dans un programme quelconque nommé *mon_prog.cpp* placé dans n'importe quel répertoire il faut y mettre la directive `#include<ma_bibli.h>` et compiler par :

g++ -lm -Wall -I/home/lulu/mes_fonctions mon_prog.cpp /home/lulu/mes_fonctions/ma_bibli.ar

(c'est ce qui est fait dans la commande *ccc*).
Si on modifie une fonction déjà existante de la bibliothèque il faut refaire *make*, cela ne recompilera que la fonction modifiée.
Si on ajoute une fonction il faut compléter les fichiers *Makefile* et *ma_bibli.h* puis faire *make*, seule la nouvelle fonction sera compilée.
Toute fonction de la bibliothèque peut appeler toute autre fonction de cette bibliothèque.