Programmation Avancée

Cours 1: la programmation modulaire

Simon Forest

14 janvier 2021

Présentation

Ce cours suit celui de Vincent Padovani du premier semestre.

Enseignant : Simon Forest, ATER à l'IRIF, `sforest@irif.fr`

Séances de cours : 10h45-12h45 un jeudi sur deux à partir du 14/01 (+ séance 04/03)

Séances de TP : 13h30-15h30 chaque jeudi du 14/01 jusqu'au 15/05 (- séance 01/04)

Évaluation : contrôle continu à 50% (un projet probablement) et un Examen à 50%

Résumé des épisodes précédents

Structure minimale d'un programme en C :

```c
#include <stdio.h>
#include ...

int main()
{
  // ...
  return 0;
}
```

Résumé des épisodes précédents

Plusieurs types possibles utilisables en C :

```c
char c = 'a';
int x = -3;
unsigned int y = 3;
long long int z = 32458325869;
float u = 3.14159;
double v = 3.14159265359;
```

Résumé des épisodes précédents

On peut écrire des fonctions pour faire des calculs :

```
int f(int a, int b)
{
  return a + b + 5;
}
// ...
int x = 5;
int y = 6;
int z = f(x,y);
// ou directement
int z = f(5,6);
```

Résumé des épisodes précédents

Les définitions des fonctions peuvent en particulier être récursives :

```
int fibonacci(int n)
{
  if(n <= 1)
    return n;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Résumé des épisodes précédents

Diverses fonctions d'interaction avec l'utilisateur :

```
printf("Veuillez entrez un nombre : ");
int x;
scanf("%d",&x);
printf("Vous avez entré %d.\n",x);
```

Résumé des épisodes précédents

Concept de pointeur permettant de stocker l'adresse d'une autre variable :

```
int x = 3;
int* ptr = &x;
printf("%p",ptr); // affiche l'adresse de x
printf("%d",*ptr); // affiche la valeur de x -> "3"
*ptr = 4;
printf("%d",x); // affiche "4"
```

Résumé des épisodes précédents

Possibilité de définir des tableaux statiques (de taille fixe) :

```
int tab[5];
tab[0] = 0;
tab[1] = 1;
tab[2] = 2;
tab[3] = 3;
tab[4] = 4;
// ou plus simplement
int tab[5] = {0,1,2,3,4};
```

Résumé des épisodes précédents

Possibilité de définir des tableaux dynamiques (de taille variable) :

```c
#include <stdlib.h>
// ...
int* tab = malloc(sizeof(int) * 5);
tab[0] = 0;
tab[1] = 1;
// ...
free(tab);
tab = malloc(sizeof(int) * 10);
tab[0] = 3;
tab[1] = 4;
// ...
free(tab);
```

Résumé des épisodes précédents

Possibilité de définir des nouveaux types :

```
typedef struct {
  int x,y;
} point;
// ...
point p;
p.x = 3; p.y = 5;

typedef enum{
  Nord, Est, Sud, Ouest
} direction;
// ...
direction dir = Sud;
```

Résumé des épisodes précédents

Lecture / écriture dans les fichiers comme pour l'entrée / la sortie standard :

```
FILE* infile = fopen("entrée.txt","r");
int x;
fscanf(infile,"%d",&x);
fclose(infile);

FILE* outfile = fopen("sortie.txt","w");
fprintf(outfile,"%d",x);
fclose(outfile);
```

Aujourd'hui

On va parler de **programmation modulaire**.

Problème

Le code source d'un programme est souvent long.

Ce code peut habituellement séparé en différentes parties :

- définition de structures
- gestion des entrées / sorties
- algorithmes
- interface graphique (GUI)
- *etc.*

Problème

Tout mettre dans `main.c` ne serait pas pratique car

- il serait difficile de s'y retrouver dans cet unique fichier
- il faudrait recompiler le fichier **en entier** à chaque changement

Comment mieux faire ?

Des fichiers différents

Une solution évidente est d'organiser le code en différents fichiers. Par exemple :

- `main.c` : le code avec la fonction `main` qui lance le programme
- `algorithms.c` : le code des algorithmes du programme
- `gui.c` : le code d'une interface graphique
- *etc.*

Erreur de compilation

Mais, tel quel, cela ne suffit pas !

`algorithms.c` :

```
int calcul_complexe(int a, int b) { ... }
```

`main.c` :

```
int main()
{
  calcul_complexe(3,5) // ERREUR: fonction calcul_complex non déclarée
}
```

Il faut **annoncer** en amont l'existence de `calcul_complexe` à gcc.

Erreur de compilation

Le même problème se pose pour les types que l'on déclare.

`algorithms.c` :

```
typedef struct { int x,y; } point;

typedef enum { Nord, Est, Sud, Ouest } direction;
```

`main.c` :

```
int main()
{
  direction dir = Nord; // ERREUR: 'direction' non connu
  point p{3,4};         // ERREUR: 'point' non connu
}
```

Il faut **annoncer** en amont l'existence de `direction` et `point` à gcc.

Fichiers d'en-tête

Pour faire ces "annonces", on utilise des **fichiers d'en-tête**.

Habituellement, ces fichiers terminent en `.h` (pour ***Header*** en anglais).

On y met les différentes "annonces" que l'on a vu précédemment :

- les définitions de structures
- les **déclarations** de fonctions

Il faut maintenir une cohérence logique dans ce que l'on rassemble dans un header.

Par exemple, il faut probablement déclarer les fonctions relevant de l'interface graphique dans un fichier différent de celui des algorithmes du programme.

Structures et énumérations

Pour “annoncer” les `struct` et les `enum`, il suffit de mettre dans le `.h` ce que l'on aurait mis dans le `.c`.

`algorithms.h` :

```
typedef struct { int x,y; } point;
typedef enum { Nord, Est, Sud, Ouest } direction;
// ...
```

Fonctions

Pour les fonctions, on laisse le code dans le fichier `.c` mais on met son **prototype** dans le fichier `.h`. On distingue ainsi la **définition** dans le `.c` et la **déclaration** dans le `.h` qui utilise le prototype.

prototype : comme une définition de fonction mais sans le code `{...}` et on termine de plus par `;`.

`algorithms.c` : définition de `calcul_complexe`

```
int calcul_complexe(int a, int b) { ... }
```

`algorithms.h` : déclaration de `calcul_complexe`

```
int calcul_complexe(int a, int b);
```

Fonctions privées

Les fonctions déclarées dans le `.h` sont autorisées à être utilisées par d'autre fichier `.c` qui les définit.

Si l'on souhaite qu'une fonction reste "privée", on peut s'abstenir de la mettre dans le fichier `.h`.

Prototypes pour fonctions privées

Cependant, même si des fonctions sont privées, on peut avoir envie de les déclarer.
Pour cela, il suffit de mettre un prototype dans le `.c` avant la définition de la fonction.

Exemple pertinent : fonctions mutuellement récursives dans `calculs.c`

```c
// int g(int x);
int f(int x){
  if(x == 0) return 2;
  else return 2*g(x/2); // ERREUR: 'g' non connu
}
int g(int x)
{
  if(x == 0) return 3;
  else return 5 + f(x-1);
}
```

Prototypes pour fonctions privées

Cependant, même si des fonctions sont privées, on peut avoir envie de les déclarer.
Pour cela, il suffit de mettre un prototype dans le `.c` avant la définition de la fonction.

Exemple pertinent : fonctions mutuellement récursives dans `calculs.c`

```c
int g(int x);
int f(int x){
  if(x == 0) return 2;
  else return 2*g(x/2); // OK, fonction 'g' déclarée au-dessus
}
int g(int x)
{
  if(x == 0) return 3;
  else return 5 + f(x-1);
}
```

Directives préprocesseur

On peut mettre des déclarations de fonctions et de structures dans le `.h`.

On peut aussi y mettre des directives préprocesseurs (appelées **macros** pour faire court).

```
#define MATHS_PI 3.1415
#define MATHS_E  2.7183
#define ADD(x,y) (x+y)
// ...
```

Comme pour les fonctions et les structures, les macros mises dans le `.h` ont vocation à être partagées. Si ce n'est pas le cas, on peut les laisser dans le `.c`.

Inclusions

Une fois que les fichiers `.h` sont écrits, il faut les utiliser dans les `.c`. Pour cela, on utilise la directive préprocesseur `#include "..."`.

`algorithms.h` :

```
typedef struct { int x,y; } point;
int calcul_complexe(int a, int b);
```

`main.c` :

```
#include "algorithms.h"
// ...
int main()
{
  point p{3,4};          // OK, car déclaré par 'algorithms.h'
  calcul_complexe(3,5) // OK, car déclarée par 'algorithms.h'
}
```

Deux types d'inclusion

La syntaxe `#include "..."` permet d'inclure des `.h` **relativement au fichier courant**.

```
#include "algorithms.h"
#include "dossier-gui/gui.h" // chemin relatif
#include "../structures.h"   // chemin relatif
```

Il y a aussi la syntaxe `#include <...>` qui permet d'inclure des `.h` standards ou d'emplacements configurés à la compilation avec l'option `-I` de gcc.

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

Headers standards

Regardons le contenu d'un header standard, par exemple `stdio.h`.

Sous linux, on peut le trouver dans le dossier `/usr/include`.

On y trouve les déclarations de fonctions d'entrée-sortie courantes comme `scanf`, `printf`, `fopen`, *etc.*

```
extern int scanf (const char *__restrict __format, ...) __wur;
// ...
extern int printf (const char *__restrict __format, ...);
// ...
extern FILE *fopen (const char *__restrict __filename,
                    const char *__restrict __modes) __wur;
```

Circularité

Un fichier `.h` peut inclure d'autres `.h`.

`structures.h` :

```
typedef struct { int x,y; } point;
typedef enum { Nord, Est, Sud, Ouest } direction;
// ...
```

`algorithms.h` :

```
#include "structures.h"
int algo_complexe(direction dir, point p);
// ^ OK, car déclarés par "structures.h"
// ...
```

Circularité

Les `#include` faits par les `.h` induit un graphe de dépendance sur ces fichiers. Ce graphe peut être **circulaire** parfois.

`A.h` :

```c
#include "B.h"

typedef struct { int a; } sA;
```

`B.h` :

```c
#include "A.h"

typedef struct { int b; } sB;
```

Circularité

Dans ce cas, on risque de boucler infiniment à la compilation car l'on doit exécuter infiniment les directives `#include` .

`main.c` :

```
#include "A.h"

int main(){ ... }
```

`#include "A.h"` , `#include "B.h"` , `#include "A.h"` , `#include "B.h"` , *etc.*

On obtient alors l'erreur :

```
error: #include nested too deeply
```

Solution

Pour prévenir ce problème, on utilise une technique classique : le contenu de tout fichier `.h` doit être compris entre les directives :

```
#ifndef NOM_FICHIER_H // "si NOM_FICHIER_H" n'est pas défini
#define NOM_FICHIER_H // "définir NOM_FICHIER_H"
// contenu du .h
// (en particulier les includes)
#endif                // "fin si"
```

À la première inclusion du `.h`, `NOM_FICHIER_H` n'est pas défini donc le contenu du `#ifndef` s'exécute et, en particulier, définit `NOM_FICHIER_H`.

À la deuxième inclusion du `.h`, `NOM_FICHIER_H` est déjà défini donc on n'exécute pas le contenu de `#ifndef`. On s'arrête donc là.

Compilation multi-fichiers

Pour compiler un code constitué de plusieurs fichiers `.c` et `.h`, il y a une méthode simple : indiquer tous les fichiers `.c` (**pas les** `.h` !) dans la commande `gcc` :

```
gcc fichier1.c fichier2.c ... fichier42.c -o programme
```

Cependant, recompiler un projet avec cette méthode peut être **très** long.

En effet, pour prendre en compte un changement fait dans un fichier, il faut recommencer la compilation de 0.

Pour améliorer le processus, il faut faire des étapes intermédiaires de compilation.

Que fait gcc ?

La finalité de gcc (et de tout compilateur en général) est de convertir un fichier texte compréhensible par un humain (un fichier `.c` par exemple), en un fichier machine compréhensible par le système d'exploitation et le processeur (en **assembleur**).

Extrait d'assembleur

```
sub    $0x8,%rsp
mov    0x200c25(%rip),%rax
test   %rax,%rax
je     4003dd
callq  400420
add    $0x8,%rsp
retq
```

Processus de compilation

Voyons comment gcc procède pour exécuter la commande

```
gcc fichier1.c fichier2.c ... fichier42.c -o programme
```

Déjà, chaque fichier .c est traduit individuellement en **assembleur** (langage compris par le procésseur). Des "trous" sont laissés dans l'assembleur pour les fonctions définies dans d'autres fichiers : on parle de **fichier objet**.

```c
// fichier1.c
int f() { ... }
```

```c
// fichier2.c
int g(){
  ...
  int x = f(); // trou dans l'assembleur pour 'f' définie ailleurs
  ...
}
```

Processus de compilation

Voyons comment `gcc` procède pour exécuter la commande

```
gcc fichier1.c fichier2.c ... fichier42.c -o programme
```

Une fois que les fichiers objets ont été générés pour chaque fichier `.c`, `gcc` les rassemble et remplace les "trous" par le code adéquat : on parle d'**édition de liens** (***linking*** en anglais).

On est obligé de refaire l'édition de liens à chaque recompilation, mais on n'est pas obligé de regénérer les fichiers objets.

Compilation progressive

gcc permet de faire la compilation progressivement.

En passant l'option `-c`, on ne fait que la génération des fichiers objets :

```
gcc -c fichier1.c -o fichier1.o
gcc -c fichier2.c -o fichier2.o
gcc -c fichier3.c -o fichier3.o
...
```

On peut alors finir la compilation du programme en donnant les `.o` à gcc (sans option) :

```
gcc fichier1.o fichier2.o ... fichier42.o -o programme
```

Recompilation plus efficace

Ainsi, si l'on a fait une modification à un fichier, il suffit de regénérer son fichier objet et de faire l'édition de lien.

Par exemple, si `fichier3.c` a été modifié, pour recompiler, il suffit de faire

```
gcc fichier3.c -o fichier3.o
```

puis

```
gcc fichier1.o fichier2.o ... fichier42.o -o programme
```

Problèmes

On peut recompiler plus efficacement mais

- les commandes à écrire restent longues
- on risque de se tromper en les écrivant
- on risque de les oublier avec le temps

Pour régler tous ces problèmes à la fois, on va utiliser des `Makefiles`.

Makefile

Un `Makefile` définit la séquence de commandes à appeler dans un terminal pour obtenir un fichier **cible** à partir de **dépendances**.

Le fichier se nomme `Makefile` et est constitué d'une séquence de déclarations de la forme

```
cible: dépendance1 dépendance2 ...
	commande1
	commande2
	...
```

L'espace à gauche des commandes est une **tabulation**. Il **faut** que ce soit une tabulation et pas une séquence d'espaces (sinon erreur).

Exemple

Considérons d'abord la compilation inefficace d'un programme `prog` avec plusieurs fichiers sources. Le `Makefile`s'écrit comme suit :

```
prog: main.c algorithms.c otherfile.c
	gcc main.c algorithms.c otherfile.c -o prog
```

Ici on précise que

- `prog` dépend de `main.c`, `algorithms.c` et `otherfile.c`
- pour regénérer `prog` à partir de ses dépendances, il faut faire `gcc ...`

Exemple

Considérons maintenanc la compilation efficace décrite plus tôt. Le `Makefile`s'écrit dans ce cas comme suit :

```
main.o: main.c
	gcc -c main.c -o main.o
algorithms.o: algorithms.c
	gcc -c algorithms.c -o algorithms.o
otherfile.o: otherfile.c
	gcc -c otherfile.c -o otherfile.o
prog: main.o algorithms.o otherfile.o
	gcc main.o algorithms.o otherfile.o -o prog
```

Ici on précise que

- les fichiers `.o` dépendent des fichiers `.c` correspondant
- pour générer un `.o`, il faut appeler `gcc -c ...`

Exécuter un `Makefile`

Le contenu d'un `Makefile`peut être exécuté avec la commande `make` du terminal. Elle prend un argument qui est la `cible` voulue :

```
make cible
```

La commande `make` fera en sorte de n'exécuter que ce qui est nécessaire, en prenant en compte les modifications récéentes qui ont été faites sur les dépendances.

On peut aussi appeler `make` sans cible. Dans ce cas, c'est la première cible qui est exécutée. Par convention, on l'appelle `all` et on la précise dans le `Makefile` :

```
all: prog
```

Ici, `make` invoqué sans argument cherchera à produire la dépendance `prog` dont dépend `all`.

Exemple

Considérons maintenant la compilation efficace décrite plus tôt. Le `Makefile`s'écrit dans ce cas comme suit :

```
main.o: main.c
	gcc -c main.c -o main.o
algorithms.o: algorithms.c
	gcc -c algorithms.c -o algorithms.o
otherfile.o: otherfile.c
	gcc -c otherfile.c -o otherfile.o
prog: main.o algorithms.o otherfile.o
	gcc main.o algorithms.o otherfile.o -o prog
```

Ici on précise que

- les fichiers `.o` dépendent des fichiers `.c` correspondant
- pour générer un `.o`, il faut appeler `gcc -c ...`

Exemple d'exécution

Reprenons l'exemple simple de `Makefile`

```
prog: main.c algorithms.c otherfile.c
	gcc main.c algorithms.c otherfile.c -o prog
all: prog
```

La première fois que l'on exécute `make`

- `make` va chercher à produire `prog` dont dépend `all`
- comme `prog` n'existe pas, `make` exécute les commandes associées à `prog`
- une fois que `prog` est construit, `make` s'arrête.

Exemple d'exécution

Reprenons l'exemple simple de `Makefile`

```
prog: main.c algorithms.c otherfile.c
	gcc main.c algorithms.c otherfile.c -o prog
all: prog
```

La deuxième fois que l'on exécute `make`

- `make` va chercher à produire `prog` dont dépend `all`
- comme `prog` existe et que ses dépendances n'ont pas été modifiées, `make` s'arrête là.

Exemple d'exécution

Reprenons l'exemple simple de `Makefile`

```
prog: main.c algorithms.c otherfile.c
	gcc main.c algorithms.c otherfile.c -o prog
all: prog
```

Si l'on fait maintenant une modification à `main.c` et que l'on exécute `make`

- `make` va chercher à produire `prog` dont dépend `all`
- `make` voit que `prog` existe mais que une de ses dépendances a changé, donc `make` régénère `prog` à partir de ses règles.

Exemple d'exécution

Reprenons le `Makefile` de la compilation efficace.

```
main.o: main.c
	gcc -c main.c -o main.o
algorithms.o: algorithms.c
	gcc -c algorithms.c -o algorithms.o
otherfile.o: otherfile.c
	gcc -c otherfile.c -o otherfile.o
prog: main.o algorithms.o otherfile.o
	gcc main.o algorithms.o otherfile.o -o prog
all: prog
```

- Que se passe-t-il la première fois que l'on appelle `make` ?
- Que se passe-t-il la deuxième fois ?
- Si l'on modifie `main.c`, que va faire `make otherfile.o` ?
- Si l'on modifie `main.c`, que va faire `make main.o` ?

Variables spéciales

On remarque que certaines règles sont toujours de la même forme :

```
main.o: main.c
        gcc -c main.c -o main.o
algorithms.o: algorithms.c
        gcc -c algorithms.c -o algorithms.o
otherfile.o: otherfile.c
        gcc -c otherfile.c -o otherfile.o
```

On peut exprimer toutes ces déclarations par une seule :

```
%.o: %.c
        gcc -c $^ -o $@
```

- le % représente une variable qui sert à matcher un nom de fichier
- `$@` est une variable spéciale qui contient le nom de la cible
- `$^` est une variable spéciale qui contient le nom de la dépendance

`Makefile` partout

Les `Makefile`s n'ont rien de spécifique au `C`. On peut les utiliser dans tous les contextes informatiques :

- programmation,
- site web,
- maintenance,
- *etc.*

Il permettent de stocker et d'exécuter efficacement les commandes associées à une certaine tâche.

Ainsi, par défaut, les projets informatiques contiennent un `Makefile` permettant de "faire ce qu'il y a à faire" sans se poser de questions.