Langage C - Bibliothèque et Makefile

16 mars 2010

Rappel

On utilise le prototypage pour créer des méthodes sans s'occuper de leur ordre.

```c
#include <stdio.h>

int somme(int x, int y);

int main() {
   int a = 6;
   int b = 9;
   printf("%d+%d=%d\n", a, b, somme(a,b));
   return 0;
}

int somme(int x, int y) {
   return x + y;
}
```

Rappel

Voici le cas des listes chaînées :

```c
#include <stdlib.h>

typedef struct Noeud {
  int valeur;
  struct Noeud* suivant;
} Noeud;

Noeud* ajoutTete(Noeud* teteliste, int n);
Noeud* ajoutFin(Noeud* teteliste, int n);
Noeud* supprTete(Noeud* teteliste);
Noeud* supprFin(Noeud* teteliste);

int main() {
  ...
}
// implémentation des méthodes prototypées:
...
```

Problème

Questions

- Comment réutiliser des méthodes déjà écrites dans d'autres programmes ?
- Comment intégrer dans votre application des méthodes créées par d'autres développeurs ?
- Comment mettre à disposition des méthodes C sans fournir leur code source ?

Solution : bibliothèques !

Externaliser

- le prototypage des méthodes dans un "fichier d'en-tête"
- leur implémentation dans un "fichier source" compilé vers un "fichier objet"

Le fichier d'en-tête

Le fichier d'en-tête

Il contient

- éventuellement d'autres inclusions
- des définitions de nouveaux types : énumérations, structures, unions
- des prototypages de méthodes **mais pas leur implémentation**

Un fichier d'en-tête porte l'extension .h

Exemple

```
// fichier listeChainee.h

typedef struct Noeud {
  int valeur;
  struct Noeud* suivant;
} Noeud;

Noeud* ajoutTete(Noeud* teteliste, int n);
Noeud* ajoutFin(Noeud* teteliste, int n);
Noeud* supprTete(Noeud* teteliste);
Noeud* supprFin(Noeud* teteliste);
```

Le fichier d'implémentation

Le fichier d'implémentation

- inclut le fichier d'en-tête .h
- contient l'implémentation des méthodes prototypées
- porte l'extension .c

A noter

- #include <file> : inclut des fichiers d'en-tête du système (stdio.h, stdlib.h, etc).
- #include "file" : inclut vos propres fichiers d'en-tête qui se trouvent dans le répertoire courant

Exemple

```
// fichier listeChainee.c
#include <stdlib.h>
#include "listeChainee.h"

Noeud* ajoutTete(Noeud* teteliste, int n) {
    ...
}
Noeud* ajoutFin(Noeud* teteliste, int n) {
    ...
}
Noeud* supprTete(Noeud* teteliste) {
    ...
}
Noeud* supprFin(Noeud* teteliste) {
    ...
}
```

Problème

Conflits d'inclusion

- Inclusion croisée : a inclut b, b inclut a $\Rightarrow$ boucle infinie. Erreur !
- Inclusion multiple : a inclut b et c, b inclut c $\Rightarrow$ c sera inclus deux fois dans a. Erreur !

Solution : inclusion conditionnelle

Définir un identificateur qui permet de tester si le fichier d'en-tête a déjà été inclus plus haut. Si oui, ne pas inclure ce fichier.

Inclusion conditionnelle

```
// fichier listeChainee.h amélioré
#ifndef LISTECHAINEE_H
#define LISTECHAINEE_H

typedef struct Noeud {
   int valeur;
   struct Noeud* suivant;
} Noeud;

Noeud* ajoutTete(Noeud* teteliste, int n);
Noeud* ajoutFin(Noeud* teteliste, int n);
Noeud* supprTete(Noeud* teteliste);
Noeud* supprFin(Noeud* teteliste);

#endif
```

Explications

- #ifndef LISTECHAINEE_H : teste si l'identificateur est défini (si oui, le fichier a déjà été inclus)
- #define LISTECHAINEE_H : définit l'identificateur
- #endif : fin de condition if

Encore un problème

Liste chaînée générique ?

Notre liste chaînée est conçue pour ne contenir que des entiers. Comment créer une liste chaînée capable de gérer n'importe quel type de données ?

```
// fichier listeChainee.h amélioré
#ifndef LISTECHAINEE_H
#define LISTECHAINEE_H

typedef struct Noeud {
   int valeur;
   struct Noeud* suivant;
} Noeud;

Noeud* ajoutTete(Noeud* teteliste, int n);
...
#endif
```

Encore un problème

Solution : Pointeur

En utilisant un pointeur générique comme type de la variable valeur, vous pouvez stocker n'importe quel élément dont vous possédez un pointeur.

```
// fichier listeChainee.h encore amélioré
#ifndef LISTECHAINEE_H
#define LISTECHAINEE_H

typedef struct Noeud {
   void* valeur;
   struct Noeud* suivant;
} Noeud;

Noeud* ajoutTete(Noeud* teteliste, void* n);
...
#endif
```

Compilation d'une bibliothèque

Le fichier objet

On compile les méthodes définies par

- le fichier d'en-tête .h et
- le fichier d'implémentation .c

vers un fichier "objet" binaire. Comme il n'y a pas de fonction `main`, ce fichier n'est pas exécutable. Un fichier objet porte l'extension .o

Compilation : utiliser l'option -c

```
gcc -Wall -c listeChainee.c
```

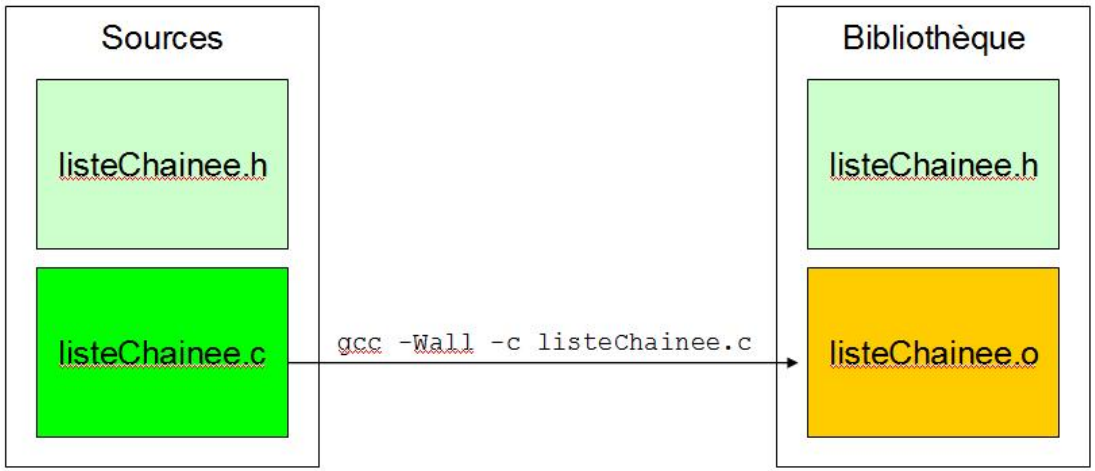
Le fichier objet résultant est "listeChainee.o"

Bilan

Voici votre bibliothèque

- le fichier d'en-tête : listeChainee.h
- le fichier objet : listeChainee.o

Elle peut être utilisée dans un programme principal.



Exemple

```c
// fichier monprog.c
#include <stdlib.h>
#include "listeChainee.h"

void main() {
  int* x = (int*)malloc(sizeof(int));
  *x = 5;
  Noeud* n = NULL;
  n = ajoutTete(n, x); // car ajoutTete(n, 5) ne marche pas
  ...
  return 0;
}
```

Comment compiler le programme entier ?

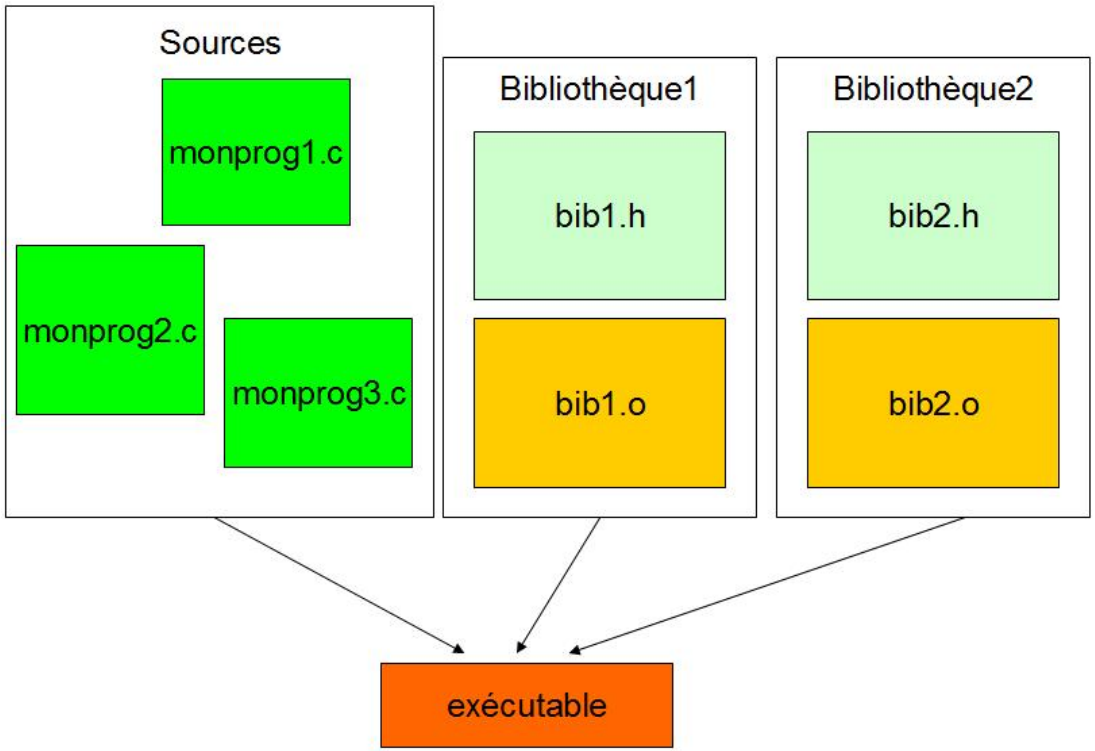
Ajouter la bibliothèque à la commande de compilation :

```
gcc -Wall monprog.c listeChainee.o -o monprog
```

Le Makefile

Idée

Automatiser la compilation d'un projet complexe. En particulier : lors de modifications, ne pas recompiler l'ensemble du projet, mais uniquement les parties concernées.



Le Makefile

Principe

- utiliser le programme `make`
- créer un fichier de configuration appelé *Makefile*

Contruction d'un makefile

Un makefile contient essentiellement des informations de la forme

```
cible : dependances
<tabulation>commande1
<tabulation>commande2
...
```

Le makefile

Voici l'exemple d'un makefile minimal. Il remplace la commande

`gcc -Wall toto.c -o monprog`

Exemple

```
monprog : toto.c
   gcc -Wall toto.c -o monprog
```

Appel

`make monprog` ou simplement `make` car par default, `make` exécute la première cible du makefile.

Les dépendances

Ce sont des règles à vérifier pour l'exécution correcte des commandes de la cible. Cela peut être :

- rien du tout : l'exécution de la cible ne souffre d'aucune précondition)
- des fichiers : ces fichiers doivent exister pour que la cible s'exécute
- d'autres cibles : elles sont alors exécutées avant la cible en cours

Exemple

```
monprog : toto.o tata.o     # créer d'abord toto.o et tata.o
   gcc -Wall monprog.c toto.o tata.o -o monprog
toto.o : toto.c          # toto.c doit exister
   gcc -Wall -c toto.c
tata.o : tata.c          # tata.c doit exister
   gcc -Wall -c tata.c
```

Important

Si l'on réexécute le makefile, `make` vérifie les timestamps de la cible et des dépendances afin d'identifier les fichiers qui ne sont pas à jour. Ainsi, seulement les fichiers modifiés seront recompilés.
Si dans l'exemple ci-dessous vous modifiez toto.c, un `make` ne recompilera pas tata.o

Exemple

```
monprog : toto.o tata.o     # créer d'abord toto.o et tata.o
   gcc -Wall monprog.c toto.o tata.o -o monprog
toto.o : toto.c           # toto.c doit exister
   gcc -Wall -c toto.c
tata.o : tata.c           # tata.c doit exister
   gcc -Wall -c tata.c
```