STRUCTURES DYNAMIQUES PROGRAMMATION MODULAIRE

2012-12-03

© Emmanuel Chieze
Département d'informatique, UQAM
INF3135

Plan

- Structures dynamiques
- Modules en C
- Modularité des programmes

Listes

3

- Liste
  - suite d'éléments de même type
  - structure fondamentalement dynamique
    - => éléments pas nécessairement stockés de façon contiguë
- On accède aux éléments d'une liste par sa tête
- On ajoute de nouveaux éléments au début de la liste

Listes

4

- Liste = structure autoréférentielle
- Liste = [Tete|Queue] où
  - Tete = élément
  - Queue = Liste (éventuellement vide)

Listes d'un type donné

5

- Implémentation en C : exemple de listes d'entiers
- cf. exemple6.2.c
- Que faudrait-il changer pour implémenter :
  - une liste de chaînes de caractères
  - une liste de vecteurs de 10 entiers
  - une liste de structures …
- cf. exemple6.3.c : listes de chaînes de caractères

Listes génériques

6

- Généralisation à des listes de n'importe quel type
  - utiliser le seul type générique en C : void *
  - nécessite des casts explicites
  - cf. exemple6.4.c

Autre utilisation de void *

7

- Pointeurs vers des fonctions
  - pour des fonctions ayant des arguments de type variable
  - Aucune perte d'information lors de conversions de/vers void *
  - Utiliser des cast explicites
  - Voir exemple5.11.c

Plan

8

- Structures dynamiques
- Modules en C
- Modularité des programmes

Module de listes

9

- Étape logique suivante : définir un module de gestion de listes
- Peut être appelé depuis n'importe quel programme, par inclusion de l'en-tête
  - exemple6.listes.h: en-tête du module
    - incluse dans le module lui-même
    - garantit que les déclarations sont les mêmes dans le module et dans les programmes appelants
  - exemple6.listes.c : code du module
  - exemple6.5.c : programme faisant appel au module

Module de listes

10

- Le module et le programme appelant peuvent être compilés :
  - ensemble
    - gcc -o exemple6.5 exemple6.listes.c exemple6.5.c
  - ou séparément (par génération des fichiers objets : suffixe .o)
    - gcc -c exemple6.listes.c
    - gcc -c exemple6.5.c
    - gcc -o exemple6.5 exemple6.listes.o exemple6.5.o (édition des liens)
    - gcc -o exemple6.5 exemple6.listes.o exemple6.5.c (pas recommandé)

Module de listes

11

- Approche modulaire
  - Permet de ne recompiler que les modules modifiés
- NB : les extensions ne sont pas pertinentes pour UNIX. Elles le sont pour gcc :
  - .c : code source en C
  - .C, .cc : code source en C++
  - .i : code source en C, prétraité
  - .s : code source en assembleur
  - .o : fichier objet
  - .a : fichier archive …

Make

12

- Utilitaire permettant d'automatiser la compilation
- make effectue la compilation minimale nécessaire
- Utilise par défaut un fichier makefile (ou Makefile sinon) spécifiant
  - les dépendances entre les fichiers,
  - les actions permettant de construire les fichiers
- `make -f <makefile>` pour un autre nom
- Généralement : un seul exécutable par makefile

Makefile

13

- Structure :
  - Assignations de valeurs à des variables prédéfinies
  - Définition de variables additionnelles
  - Définition des dépendances entre fichiers
    - optionnellement, spécification des actions permettant la construction du fichiers cible
    - les actions commencent par une tabulation
- Contenu d'une variable accédé par $(VAR)

Makefile

14

```
# Variables predefinies
CC = gcc
CFLAGS = -g -W -Wall
# Variables additionnelles

# "dependance" formelle pour lancer la compilation de programmes
   independants. Par defaut, make (sans arguments) ne traite que
   la premiere dependance rencontree. Aucune action par defaut
   ici, car gcc ne "sait" pas comment traiter ces dependances
all : exemple6.6
```

Makefile

15

```
# Dependances simples : gcc "sait" comment traiter les fichiers
   .c, et "comprend" d'apres le nom des fichiers cibles qu'il
   s'agit d'executables. Commande de compilation inutile.
exemple6.6 : exemple6.6.o exemple6.listesABS.o
# On peut specifier une action de compilation (Pour l'exemple
   ici, non necessaire dans ce cas-ci)
$(CC)  $(CFLAGS) exemple6.6.o exemple6.listesABS.o -o exemple6.6
exemple6.6.o : exemple6.6.c exemple6.listesABS.h
exemple6.listesABS.o : exemple6.listes.c exemple6.listesABS.h

# Nettoyage du repertoire (fichiers objets et executables)
# make doit etre appelee avec clean comme argument, pour que
   cette "dependance" soit traitee
clean :
   rm exemple6.? exemple6*.o
```


Fonctionnement de make

16

- Par défaut, make ne traite que la première dépendance rencontrée
  - parcours récursif des sous-dépendances
  - ne recompile que le minimum nécessaire :
    - fichiers cibles absents
    - fichiers cibles dont l'un des fichiers sources est plus récent
- `make <dépendance>` pour traiter une autre dépendance que la première du fichier
- utiliser `touch <fichier>` pour forcer une recompilation

Module de listes

17

- Problème avec l'approche précédente :
  - la structure d'une liste est visible des programmes appelants
  - on ne peut la modifier sans impacter ces derniers
  - aucune validation possible des données
  - il s'agit d'un "type abstrait" non opaque

Masquage de l'implémentation

18

- Étape logique suivante : masquer l'implémentation des listes
  - séparer
    - l'interface du module (visible du monde extérieur)
    - de son implémentation (interne au module)
  - encapsuler les données du module
    - les données ne sont accessibles que par les fonctions du module
  - permet :
    - la maintenabilité et l'extensibilité du code
    - la validation des données

Masquage de l'implémentation

19

- Le type abstrait devient opaque (une boîte noire)
  - Attention : dans d'autres contextes (e.g. Haskell), la notion de type abstrait implique l'opacité du type. Un type abstrait non opaque est appelé type algébrique.

Masquage de l'implémentation

20

- L'interface du module (fichier .h) définit
  - un pointeur vers la structure représentant les listes
  - les fonctions permettant d'accéder à la structure via son pointeur
    - fonction de création d'une nouvelle structure de données
      - retourne un pointeur vers une nouvelle instance de la structure, créée dynamiquement.
    - fonction de suppression d'une instance de la structure
    - le premier paramètre des autres fonctions de l'interface est le pointeur vers la structure de données traitée.

Masquage de l'implémentation

21

- La mise en œuvre du module (fichier .c)
  - définit la structure représentant les listes (inaccessible hors du module)
  - les fonctions auxiliaires définies dans le module n'apparaissent pas dans l'interface.
- Permet de rajouter facilement des contrôles lors de la création de nouveaux objets
  - exemple : code postal canadien = LCL CLC où L est une lettre, C un chiffre

Comparaison entre le C et l'OO

22

- Fichier source = implémentation d'une classe (données et traitements)
- Fonctions définies dans l'en-tête =
  - fonction de création d'un nouvel objet
  - et méthodes de la classe
- Appel à la fonction de création d'une nouvelle structure = instanciation d'un nouvel objet de la classe
- Pas d'implémentation possible de l'héritage

Masquage de l'implémentation

23

- Exemple : type abstrait opaque mettant en œuvre des listes
  - exemple6.listesABS.h : en-tête du module
    - incluse dans le module lui-même
    - garantit que les déclarations sont les mêmes dans le module et dans les programmes appelants
  - exemple6.listesABS.c : code du module
  - exemple6.6.c : programme faisant appel au module

Masquage de l'implémentation

24

- Exemple d'implémentation variable d'une même classe
  - Fichier source de la classe = exemple6.compteur1.c ou exemple6.compteur2.c
  - Fichier de l'interface = exemple6.compteur.h
  - Programme utilisant la classe = exemple6.7.c
  - NB : les noms des fonctions apparaissant dans l'interface commencent tous par le nom du module, pour éviter tout conflit avec d'autres modules

Plan

25

- Structures dynamiques
- Modules en C
- Modularité des programmes

Objectifs

26

- lisibilité du code
- maintenabilité du code
  - facilité de corriger les erreurs sans en créer d'autres ailleurs
- portabilité du code
- extensibilité du code
  - pouvoir modifier l'implémentation d'une fonctionnalité
  - pouvoir rajouter des fonctionnalités
- réutilisabilité du code

Définition

27

- un programme est modulaire lorsqu'il est scindé en composantes, les modules, satisfaisant aux objectifs précédents
- un module regroupe des fonctions associées à un même traitement

Conception de modules

28

- approche dirigée par le contrôle et analyse fonctionnelle descendante
  - on décompose un problème en sous-problèmes plus simples, de façon récursive
  - approche usuelle en programmation procédurale
  - ne permet pas toujours la réutilisabilité et l'extensibilité
- approche dirigée par les données ou approche ascendante
  - exemple : analyse par objets

Types de modules

29

- 1er type : type abstrait
  - Deux sortes de types abstraits
    - collection de valeurs
      - les entités modélisées n'évoluent pas dans le temps
      - fournir une ou plusieurs fonctions de création de valeurs, mais aucune fonction de modification des valeurs
      - exemples : dates, fractions, nombres complexes

Types de modules

30

- 1er type : type abstrait
  - Deux sortes de types abstraits
    - classe d'objets
      - les entités modélisées évoluent dans le temps (tout en conservant leur identité)
      - on distingue 3 types de fonctions
        - créateur : fonctions créant un nouvel objet
        - mutateur : fonctions modifiant l'état d'un objet existant
        - observateur : fonctions ne modifiant pas l'état d'un objet
      - Exemple : compteurs

Types de modules

31

- 2ème type : machine abstraite
  - modéliser l'état d'un objet par des variables globales statiques (donc invisibles de l'extérieur)
  - modéliser les méthodes comme précédemment, en omettant le premier argument (pointeur vers l'objet)
  - inconvénient majeur : on ne peut avoir qu'une instance de l'objet à la fois
  - cf. l'exemple de la pile (Tremblay, 2005, p.9)

Types de modules

32

- 3ème type : bibliothèque de routines
  - aucune donnée n'est partagée entre les routines
  - exemple : bibliothèque mathématique math.h

Types de modules

33

- Deux applications, A et B, gèrent des factures
  - A : une facture ne peut pas être corrigée, et doit être payée intégralement (exemple : facture d'épicerie)
  - B : une facture peut être corrigée et on lui rattache un solde restant à payer
- Quels types de données sont utilisés pour implémenter les factures ?

Cohésion

34

- Détermine
  - l'indépendance d'une fonction ou d'un module
  - la force d'association entre ses composantes
- En théorie, on distingue différents niveaux de cohésion (principalement liés aux fonctions)
  - cohésion fonctionnelle : une seule activité est réalisée, au complet
  - cohésion séquentielle, communicationnelle, procédurale, temporelle, logique …

Cohésion

35

- En pratique :
  - peut-on nommer la fonction au moyen d'un seul verbe ?
  - le module est-il centré sur une seule activité ?
- Attention :
  - une seule activité != activité élémentaire

Couplage

36

- Détermine la force des liens (l'interdépendance) entre deux modules
- Niveaux de couplage
  - Nul
    - modules complètement indépendants
  - Faible
    - couplage par données simples : les modules échangent des données simples (types de base, tableaux) par l'intermédiaire de paramètres
    - couplage par objets : les modules échangent des types abstraits (fonctionnellement cohésifs)
    - et que seules les données nécessaires sont échangées

Couplage

37

- Niveaux de couplage
  - Fort (à proscrire en général)
    - modules communiquant par l'intermédiaire de variables globales
      - couplage fort entre fonctions d'un module mettant en œuvre une machine abstraite OK, car variables globales internes au module
    - module passant un indicateur de contrôle à un autre pour lui dire quoi faire
    - module accédant à la zone mémoire d'un autre module

Niveau de modularité

38

- Un programme est d'autant plus modulaire que
  - le niveau de cohésion de chacune de ses composantes est élevé
  - et que le niveau de couplage entre ses composantes est faible

Exemples

39

- Soient
  - un type personne contenant une date de naissance de type date
  - Et une fonction age calculant l'âge d'une personne
- `int age(personne)`
  - couplage trop fort
  - la fonction pourrait prendre en compte d'autres caractéristiques de la personne, voire les modifier
  - risque élevé de problèmes de maintenance
- `int age(date)`
  - couplage acceptable
  - la fonction devient plus générale et donc réutilisable
  - la fonction est plus facile à tester

Exemples

40

- Soient
  - un type point représentant un point dans l'espace à 2 dimensions
  - Et une fonction distance calculant la distance entre 2 points
- `float distance(float x1, float y1, float x2, float y2)`
  - couplage trop fort
  - car adopte une certaine représentation du point
- `float distance(point p1, point p2)`
  - couplage acceptable
  - laisse la possibilité d'un changement de représentation interne des points

Modularité

41

- Chaque fonction doit
  - Être autonome
    - Contre-exemple : une fonction de validation des arguments de main()
  - Correspondre à un calcul précis
  - Permettre d'éviter la redondance du code
- Un module doit
  - Être réutilisable en d'autres circonstances
  - Ajouter une fonctionnalité par rapport à des modules existants

Qualité de l'interface

42

- Quels problèmes cette interface pose-t-elle ?

```
typedef void * definition;
typedef struct dictionnaire * dictionnaire;
typedef enum statut {OK, ERR_ECHEC_ALLOCATION,
  ERR_INSERE_ENTREE_EXISTANTE, ERR_DICTIONNAIRE_VIDE
  } statut;

dictionnaire dictionnaire_cree(void);
dictionnaire dictionnaire_insere_entree(struct
  dictionnaire *, char *, void *);
int dictionnaire_supprime_entree(char *, definition
  *, dictionnaire *);
statut dictionnaire_lit_entree(dictionnaire, char *,
  definition *);
```

Qualité de l'interface

43

- Utiliser des synonymes adéquats
  - pour clarifier le rôle des arguments
  - de façon uniforme
  - `Element` plutôt que `void *`
- Ordre des arguments
  - mettre en premier le type manipulé
  - uniformiser l'ordre des arguments

Qualité de l'interface

44

- Utiliser des noms de paramètres significatifs et constants
- Préfixer le nom des fonctions et des synonymes du nom du module
- Utiliser un niveau d'indirection constant
  - `Element` partout ou `Element *` partout
- Retourner un statut explicite plutôt qu'une valeur spéciale indiquant une erreur

Références

- Tremblay, G. 2005. "Notions de base de conception de logiciels"
  - document complémentaire aux acétates
  - disponible sur la page
    http://www.labunix.uqam.ca/~tremblay/INF3135/
  - à lire absolument