MODULARITÉ

Éléments de modularité

- fichiers d'en-têtes
- fichiers d'inclusion
  permettant d'utiliser des bibliothèques écrites en C
- unités de traduction à compilation séparée
- modules objets
  permettant d'utiliser des bibliothèques d'unités compilées, en particulier :
  - des ensembles de déclarations et fonctions prédéfinies en C
  - des modules objets provenant d'autres langages
- fonctions avec paramètres et variables locales
- macro-instructions (macros)

Exécutable

unité … unité

pré-traitement, compilation

modules de bibliothèques

objet … objet objet … objet

édition des liens

exécutable

Module-objet

fichiers d'en-têtes et d'inclusions

$\boxed{\text{unité}}$ $\boxed{\text{inclusion}}$ … $\boxed{\text{inclusion}}$

pré-traitement

$\boxed{\text{unité complète}}$

compilation

$\boxed{\text{objet}}$

NOTE : les noms des fichiers d'en-têtes sont souvent suffixés par "`.h`", ceux des autres fichiers sources par "`.c`".

Structure d'un programme C

- un ou plusieurs fichiers en C : les unités de traduction
- une seule unité contient "`main`", le point d'entrée
- la communication entre unités se fait par objets globaux :
  - variables communes ("`extern`")
  - appels de fonctions
- entre unités, les déclarations et définitions doivent être cohérentes
- une variable ou une fonction ne peut être définie qu'une seule fois (mais déclarée plusieurs fois)

Méthodologie

Les déclarations de variables globales et fonctions d'un programme peuvent être réunies dans un fichier en-tête.

Une bonne méthode de programmation consiste à déterminer pour chaque unité `f.c` un fichier en-tête `f.h` qui constitue son *interface*, et qui contient les déclarations des variables et fonctions accessibles depuis le reste du programme (au moyen de "`extern`").

L'unité `f.c` constitue l'*implémentation* de l'interface et peut contenir des fonctions et des variables privées (préfixées par "`static`"), inaccessibles du reste du programme.

Exemple

`/* fichier som.c */`	*commentaire*
`#include <stdio.h>`	*macro-inclusion*
`#define PI 3.14159`	*macro-définition*
`float a,b;`	*définition de variables globales*
`float som(float, float);`	*déclaration de fonction*
`float dif();`	*déclaration de fonction*
`main(){`	*définition de "main"*
`    float c,d;`	*définition de variables locales*
`    a = PI; b = 1 ;`	*initialisations*
`    c = som(a,b);`	*appel de fonction*
`    d = dif();`	*appel d'une fonction de* `dif.c`
`    printf("%f %f",c,d);`	*fonction E/S*
`}`	
`float som(float x, float y){`	*définition de fonction*
`    return x+y;`	
`}`	

`/* fichier dif.c */`	*commentaire*
`extern float a,b;`	*déclaration de variables externes*
`float dif(){`	*définition de fonction*
`    return a-b;`	
`}`	

Unités et variables globales, Blocs et variables locales

- Unité :

```
/* fichier */
liste de déclarations et
de définitions de fonctions
```

- Variable globale = variable déclarée hors de tout bloc

- Bloc :

```
{
        liste de déclarations
        liste d'instructions
}
```

- Variable locale = variable déclarée dans un bloc

Définitions

L'opération qui réserve de la place en mémoire pour les variables et fonctions s'appelle la *définition*.

Une variable ou une fonction doit être *définie une fois et une seule* dans un programme, ceci dans n'importe quelle unité.

Ceci entraîne en particulier qu'une variable globale ou une fonction ne peut figurer dans un fichier inclus dans plusieurs autres.

- Définition d'une variable (initialisation)

```
int annee = 1994;
```

- Définition d'une fonction

```
int max(int a, int b, int c){
   if (a > b) return (a > c) ? a : c;
   else (b > c) ? b : c;
}
```

Déclarations

La *déclaration* décrit des propriétés des variables et des fonctions (leur prototype).

Une définition vaut déclaration.

Une variable ou une fonction peut être *déclarée plusieurs fois* dans un programme (ou une unité).

MAIS ATTENTION dans une unité, l'ensemble des déclarations d'une variable globale constitue sa définition (avec initialisation à 0)

- Déclarations de variables

  ```
  int annee;
  extern double taux;
  ```

- Déclarations possibles d'une fonction

  ```
  int max(int a, int b, int c);
  int max(int, int, int);
  int max();
  ```

  Dans le dernier cas, il n'y a pas nécessairement de contrôle sur les arguments.

Règle de déclaration

Dans une unité,
un identificateur doit être déclaré
avant toute utilisation,
sauf exception ci-dessous

Déclaration implicite des fonctions :

lorsqu'un appel de fonction,

```
fct(...)
```

n'est pas précédé de déclaration (ni de définition), la fonction est considérée comme étant déclarée par

```
extern int fct();
```

Dans ce cas il n'y a pas nécessairement de contrôle sur les arguments.

Portée des identificateurs

Les identificateurs se partagent en plusieurs *espaces de noms* indépendants déterminés par les objets qu'ils représentent :

1. variables, fonctions, noms de types, constantes d'énumération
2. labels (pour les branchements)
3. labels de structures, unions, et énumérations
4. membres de chaque structure ou union individuellement

- dans une unité (resp. bloc), un identificateur est utilisable depuis sa déclaration jusqu'à la fin du fichier (resp. bloc), sauf dans les blocs (resp. sous-blocs) où il est redéclaré dans le même espace de noms
- la définition d'un identificateur masque celle du même identificateur du même espace de noms d'un sur-bloc ou de l'unité
- une occurrence d'un identificateur fait référence à sa déclaration dans le plus petit bloc (ou l'unité) qui les contient

Classes d'allocation

- Zone mémoire utilisateur :

zone texte (code du programme)
zone données (variables globales)
pile (données temporaires)
tas (mémoire dynamique)

- Classes d'allocation des variables
  - **statique** : variables implémentées en zone données
  - **automatique** : variables implémentées dans la pile
  - **dynamique** : variables implémentées dans le tas, allouées à la demande (voir "Pointeurs")

Allocations

- par défaut

	variable globale	variable locale
allocation	statique à la compilation	automatique à l'exécution du bloc
durée de vie	celle du programme	celle du bloc
initialisation	à zéro	aucune

- "`static`" rend statique une variable locale : ses valeurs deviennent persistantes

```
int suivant(){
     static int compteur = 0;
     return compteur++;
}
```

- "`register`" demande l'allocation d'une variable automatique dans un registre machine (si possible) : pour une variable très sollicitée

  Note : pas d'adresse accessible (avec l'opérateur &)

```
for(i=0; i<n; i++)
     for(j=0; j<n; j++){
         register char temp;
         temp = ...
     }
```

Constantes

- Les constantes symboliques (souvent en majuscules) sont définies par

  `#define` *identificateur texte*

```
#define TAILLE_MAXI 1000
#define EOF (-1)
```

- "`const`" indique que les valeurs des variables du type ainsi qualifié ne seront pas modifiées : permet certaines optimisations du module exécutable

```
...
const double pi = 3.14159;
...
int longueur(const char chaine[]);
...
```

- "`volatile`" indique le contraire : supprime certaines optimisations éventuelles sur les variables
- "`const`" et "`volatile`" s'appliquent à un type

Visibilité

Les fonctions et variables globales d'une unité sont, par défaut, *publiques*, c'est-à-dire accessibles depuis les autres unités du programme.

- "`extern`" indique une déclaration sans définition : permet de faire référence à une variable globale qui est définie ailleurs, en dehors de l'unité.
- "`static`" rend *privée* à son unité une variable globale ou une fonction : celle-ci devient inaccessible depuis une autre unité (même avec "`extern`").

```
...
int sommet(void);
void depiler(void);
boolean empiler(int);
boolean vide(void);

static char pile[TAILLE];
static int ind_sommet = -1;
...
```

Récursivité

- Appel récursif : une fonction peut faire appel à elle-même.
- Attention à la condition d'arrêt des appels récursifs !

```
/* affichage de la valeur */
/* d'une variable entière */
void ecrire_entier(int n){
     if (n < 0){
         putchar('-'); n = -n;
     }
     ecr_ent(n);
}
```

```
/* la valeur est positive */
void ecr_ent(int n){
     if (n / 10)
         ecr_ent(n / 10);
     putchar(n % 10 + '0');
}
```

Macro-instructions

"**Macros**" : substitutions traitées par le préprocesseur

- Macro-définitions

```
#define FOREVER for(;;) /* boucle infinie */
#define NEWLINE putchar('\n')
```

- Avec paramètres : pseudo-fonction.

  `#define` *identificateur(paramètres) définition*

```
#define TAUX(x, y) (((y) - (x)) / (x))
...
x = TAUX(a*b, a+3);
```

  produit après pré-traitement

```
x = (((a+3) - (a*b)) / (a*b));
```

- **Avantages** : rapidité, pas de typage, généricité (définition de structures génériques)

  `#define TABLE(type,ident,taille) type ident[taille]`

- **Inconvénients** : pas de contrôle de types, duplication possible de code (`a*b` calculé deux fois), etc ....

Attention

- pas d'espace entre le nom et la parenthèse (sinon il s'agit d'un définition de constante)

```
#define ABS (x) ((x) > 0 ?  (x) :  -(x))
...
b = 2*ABS(a);
...
```

produit après pré-traitement

```
...
b = 2*(x) ((x) > 0 ?  (x) :  -(x))(a);
...
```

- définition sur une seule ligne, ou utilisation de '\' pour la couper

```
#define TABLE(type, ident, taille) \
     type ident [taille]
```

- parenthésage important

```
#define CARRE(x) x*x
...
b = CARRE(a+b);
```

produit après pré-traitement

```
b = a+b*a+b;
...
```

- éviter les effets de bords sur les arguments

```
#define TAUX(x, y) (((y) - (x)) / (x))
...
t = TAUX(a++, b);
...
```

produit après pré-traitement

```
...
t = (((b) - (a++)) / (a++));
...
```

À l'exécution, la variable “a” est incrémentée deux fois !

Bibliothèques

L'inclusion de fichier se fait par l'une des instructions

- `#include <stdio.h>`
  recherche de "`stdio.h`" dans les répertoires standards
- `#include "entete.h"`
  recherche de "`entete.h`" dans le répertoire de travail, puis dans les repertoires standards

Il existe 15 fichiers qui font partie de l'environnement C standard. Les principaux sont

`float.h` : définitions de pseudo-constantes liées aux types réels (ex : `DBL_EPSILON`)

`limits.h` : définitions de pseudo-constantes liées aux types entiers (ex : `LONG_MIN` et `LONG_MAX`)

`math.h` : prototypage de fonctions mathématiques standards (ex: `sin, sqrt, log`)

`stddef.h` : définitions de types divers (ex : `size_t` utilisé par `sizeof`)

`stdio.h` : entrées/sorties

`stdlib.h`: prototypes de fonctions de la bibliothèque standard, conversions de chaînes numériques (`atof, atoi,...`)

`string.h` : interface avec fonctions de manipulation de chaînes (ex : `strlen, strcpy, strcat`)

EXERCICES

1. **Puissance** : implémenter la fonction `double puiss(double x, int n)` qui calcule x^n ($x^0 = 1$) :

   - itérativement,
   - récursivement par la formule $x^n = x^{n-1} * x$, si $n > 0$,
   - récursivement par les formules $x^n = (x * x)^{n/2}$, si $n > 0$ est pair, $x^n = (x * x)^{n/2} * x$, si $n > 0$ est impair.

   Combien de multiplications flottantes sont-elles utilisées dans chaque cas pour calculer 2^{10} ?

2. **Capitalisation** : écrire une fonction qui donne le capital obtenu en plaçant une somme s au taux de $t\%$ pendant n années, en appliquant la fonction précédente.

3. **Maximum** : écrire une macro-instruction pour le maximum de trois nombres.

4. **Pile** : écrire une unité qui implémente une structure de pile de caractères en accord avec ces déclarations :

```
int sommet(void);
void depiler(void);
boolean empiler(int);
boolean vide(void);
static char pile[TAILLE];
static int ind_sommet = -1;
```

Puissance

```
double puissance1 (double x, int n) {
   double p = 1; int i;
   if(n < 0) { n = -n; x = 1/x;}
   for (i = 0; i < n; i++)
      p *= x;
   return p;
}

double puissance2 (double x, int n) {
   if(n < 0) { n = -n; x = 1/x;}
   if (n == 0)
      return 1;
   else if (n % 2 == 0)
      return puissance2(x*x, n/2);
   else
      return puissance2(x*x, n/2) * x;
}

double puissance3 (double x, int n) {
   double p = 1; int i;
   if(n < 0) { n = -n; x = 1/x;}
   while (n) {
      if (n & 1) p *= x;
      x *= x;  n >>= 1;
   }
   return p;
}
```

Capitalisation

```
# include <stdio.h>

double puissance(double x, int n){
   if(n == 0)
      return 1;
   else if(n % 2 == 0)
      return puissance(x*x, n/2);
   else
      return puissance(x*x, n/2) * x;
}

double capital(double somme, double interet, int annees){
   return somme * puissance(1+interet/100, annees);
}

main(){
   float s, i; int n, lu;
   printf("Somme Interet Annees ? ");
   scanf("%f%f%d", &s, &i, &n);
   printf("%f %f %d\n", s, i, n);
   printf("Capital acquis = %f\n", capital(s, i, n));
}
```

```
Somme Interet Annees ?  10000 10 7
10000.00      10.00        7
Capital acquis = 19487.17
```

Maximum

```
# include <stdio.h>

# define max(x, y, z) \
(x)>(y) ? ((x)>(z) ? (x) : (z)): ((y)>(z) ? (y) : (z))

main(){
   printf("max de 1, 2, 3 = %d\n",
     max(1, 2, 3));
   printf("max de 'c', 'b', 'a' = %c\n",
     max('c', 'b', 'a'));
   printf("max de 'c'+1, 'b'+11, 'a'+1 = %c\n",
     max('c'+1, 'b'+11, 'a'+1));
}
```

```
max de 1, 2, 3 = 3
max de 'c', 'b', 'a' = c
max de 'c'+1, 'b'+11, 'a'+1 = m
```

Pile

```
/************************* fichier "pile_entete.h" */

typedef enum {FAUX, VRAI} boolean;

char sommet(void);
void depiler(void);
boolean empiler(int);
boolean vide(void);

/********************************** unite "pile.c" */

# include <stdio.h>
# include "pile_entete.h"

# define TAILLE 20

static int pile[TAILLE];
static int ind_sommet = -1;

void affiche_pile(void){
   int i;
   printf("PILE : ");
   for(i=0; i<=ind_sommet; i++)
      printf("%c ", pile[i]);
   putchar('\n');
}
```

```c
int sommet(void){
   if(ind_sommet == -1) printf("PILE VIDE\n");
   else return pile[ind_sommet];
}

void depiler(void){
   if(ind_sommet == -1) printf("PILE VIDE\n");
   else ind_sommet--;
}

boolean empiler(int x){
   if(ind_sommet == TAILLE-1){
      printf("PILE PLEINE\n");
      return FAUX;
   } else {
      pile[++ind_sommet] = x;
      return VRAI;
   }
}

boolean vide(void){
   return ind_sommet == -1;
}
```

Test de “pile.c”

```
/********************** unite qui contient "main" */

# include <stdio.h>
# include "pile_entete.h"

main(){
   int car;
   printf("MENU : q s d e? v\n");
   while((car = getchar()) != 'q'){
   switch(car){
   case 's' : printf("Sommet = %c\n", sommet());
      affiche_pile(); break;
   case 'd' : depiler();
      affiche_pile(); break;
   case 'e' : car = getchar();
      if(empiler(car) == FAUX) printf("Pile pleine\n");
      affiche_pile(); break;
   case 'v' : if(vide() == VRAI) printf("Pile vide\n");
      else printf("Pile non vide\n");
      affiche_pile(); break;
   default  : if(car != '\n')
      printf("Commande inconnue\n");
   }
   }
}
```