TECHNIQUES DE PROGRAMMATION

Johann PAVIS (selon source : A. ZEMMARI)

Table des matières

Chapitre 1

LE PRÉPROCESSEUR

Le préprocesseur traite le fichier source avant le compilateur .

Il ne sait manipuler que des chaînes de caractères :

1. il retire tous les commentaires (entre `/*` et `*/` ou après `//`)
2. il prend en compte les lignes commençant par `#` pour créer le code que le compilateur analyse

Les possibilités sont :

- inclure des fichiers en-tête (`<fichier.h>`)
- définir des `macros-constantes` et des `MACROS-FONCTIONS`
- inclusion et compilation conditionnelle
- pilotage des messages d'erreur

Liste des directives :

```
define , include , error
if , else , elif
ifdef , ifndef , endif
undef , line , pragmas
```

Remarque : Le résultat du préprocesseur (pré-traitement) peut-être visualisé au moyen de l'option `-E` du compilateur `gcc`.

1.1 Définition des macros-fonctions

1.1.1 Les macros-constantes

C'est un symbole défini au moyen de la directive `#define`. Sa valeur est une chaîne de caractères.

Syntaxe

```
#define ident chaîne
```

⇒ lors de la précompilation, chaque occurrence de `ident` est remplacée par `chaîne` dans le texte du programme.

Exemple

```
#define TailleMax 100
```

`int tab[TailleMax] ;` est alors récrit en `int tab[100] ;`

Une définition de macro-constante reste valide jusqu'à la fin du fichier source, ou bien jusqu'à ce que cette définition soit annulée par la directive `undef`.

On écrit en général les macros-constantes en *majuscules* afin de (mieux) les distinguer des autres identificateurs.

Le symbole d'une macro-constante **doit commencer par une lettre**, le caractère '_' étant considéré comme une lettre.

Il est aussi possible de définir une macro constante lors de la phase de compilation, au moyen de l'option `-D` du compilateur `gcc` :

```
gcc -D TailleMax 100 main.c
```

Macros prédéfinies

`__FILE__` : nom du fichier courant
`__LINE__` : numéro de la ligne courante
`__DATE__` : date de la compilation
`__TIME__` : heure de la compilation
`__STDC__` : =1 si le compilateur est conforme à la norme `ANSI`

1.1.2 Les macros-fonctions

Le mécanisme de substitution de `define` est paramétrable , *i.e.* on peut définir des paramètres de la macro-constante : les `macros-fonctions`.

Syntaxe

```
#define ident(p[\dots]) corps
```

Lors de la substitution, toute occurrence d'un paramètre dans le corps de la macro-fonction est remplacée par le paramètre correspondant.

Exemple macro-fonction retournant le plus grand de 2 nombres

```
#define PLUS\_GRAND (x,y) x>y ? x:y
```

Une macro-fonction peut, par exemple, calculer la valeur absolue :

```
#define ABSOLUE (x) x>=0 ? x:-x
```

La substitution se fait en considérant les paramètres comme des chaînes de caractères. Il apparaît alors la nécessité de parenthéser les arguments :

```
ABSOLUE (x+y) → x+y>=0 ? x+y:-x+y
```

1.2 Construction générique : le préprocesseur

Exemple tableau contenant des éléments

Cette structure va être manipulée par des primitives, telles que l'initialisation, l'ajout ou la suppression.

```
#define TAILLE_MAX 100
#define DECLARER_TABLEAU (type , ident) \
        type ident[TAILLE_MAX]
#define INSERER_ELEMENT (ident , e , n) \
        ((ident)[n++]=(e))
#define SUPPRIMER (ident , n) \
        ((ident)[n--])
```

Exemple

```
int main(void)
   {
      DECLARER_TABLEAU (int tab-1);
      DECLARER_TABLEAU (double , tab-2);
      int tab-1=0;
      int tab-2=0;

      for (i=0 ; i<10 ; i++)
         {
            INSERER_ELEMENT (tab-1 , i , taille_1);
            INSERER_ELEMENT (tab-2 , 1.5 taille_2);
         }

      for (i=0 ; i<10 ; i++)
        {
            printf(``%d ième élément %d'',i,
                        SUPPRIMER(tab-1,taille_1));
            printf(``%d ième élément %f'',i,
                        SUPPRIMER(tab-2,taille_2));
        }
   }
```

Chapitre 2

POINTEURS DE FONCTION

En C, une fonction elle-même n'est pas une variable, mais il est possible de définir des pointeurs de fonction que l'on peut affecter, faire retourner par retourner par des fonctions, *etc.*

Un pointeur de fonction correspond à l'adresse du début du code de cette fonction.

Un pointeur sur une fonction est de prototype : `type fonction (type1 , type2 , ...) ;` ou bien de prototype : `type (*fonction) () ;`

Attention à l'utilité des parenthèses :

`int * f() ;` → `f` retourne un **pointeur sur entier**

`int (*f) () ;` → `f` est un **pointeur de fonction** qui retourne un entier

Exemple Fonction opérateur binaire prenant deux entiers en paramètres (`a` et `b`)

`int op_binaire (int a,int b,int (*f)(int,int));` (définition)

`int op_binaire (int,int,int (*f)(int,int));` (déclaration)

Pour pouvoir utiliser cette fonction, on utilisera comme troisième argument l'identificateur de la fonction utilisée. Par exemple, si la fonction `somme` est de prototype `int somme(int,int)`, on pourra utiliser l'appel :

`op_binaire(a,b,somme(a,b));`

Le code de la fonction `op_binaire` est le suivant :

```
int op_binaire(int a,int b,int (*f) (int a,int b))
   {
       return ((*f)(int a,int b));
   }
```

Exemple Fonction prenant en arguments deux entiers séparés par la chaîne `plus` ou `fois` et qui retourne la somme ou le produit de ces deux entiers

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

void usage(char *);
int somme(int,int);
int produit(int,int);
int op_binaire(int,int,(*f)(int,int));

int usage(char * code)
   { printf(``\nUsage : %s[fois/plus] int\n'',code); }

int somme (int a,int b)
   { return (a+b); }

int produit (int a,int b)
   { return (a*b); }

int op_binaire (int a,int b,int (*f)(a,b))
   { return ((*f)(a,b)); }

int main(int argc,char * argv[])
   {  int a,b;
      if (argc != 4);
         { printf(``\nNombre d'argument incorrect\n'');
           usage(``argv[0]'');
           retunr EXIT_FAILURE; }
      a = atoi(argv[1]);
      b = atoi(argv[3]);
      if (!strcomp(argv[2] , ``plus''))
         { printf(``\n%d\n'',op_binaire(a,b,somme));
           return EXIT_SUCCESS; }
      if (!strcomp(argv[2] , ``fois''))
         { printf(``\n%d\n'',op_binaire(a,b,produit));
           return EXIT_SUCCESS; }
      printf(``\nErreur : arguments invaldes'');
      usage(argv[0]);
      return EXIT_FAILURE; }
```

Exemple Fonction `qsort` de la librairie `<stdlib.h>`
Le prototype de la fonction est :

```
void qsort (void * tableau,size_t nbr_element,
            size_t taille_element,
            int (*comp)(const void*,const void*))
```

Elle permet de trier les `nbr_element` premiers éléments du tableau `tableau`. Le paramètre `taille_element` donne la taille des éléments du tableau.

La fonction `qsort` est paramétrée par la fonction de comparaison `comp` de prototype `int comp (void * a,void * b)`, où `a` et `b` sont des pointeurs ***génériques*** (de type void). Ils correspondent à des adresse dont le type n'est pas déterminé.

⇒ `qsort` permet de trier un tableau d'éléments en utilisant une fonction adéquate de comparaison.

On définit donc les fonctions de comparaison suivantes :

pour les chaînes de caractères

```
int comp_str(char * s1,char * s2)
   { return (srtcmp(*s1 , *s2)); }
```

pour les entiers

```
int comp_int (int * a,int * b)
   { return (*a - *b); }
```

Ce qui donne pour la fonction `main` :

```
int main (int argc,char * argv[])
   {
      char * tableau_char[NB_ELEMENT] =
           {``toto'',''titi'',''auto'',''a'',''b''};
      qsort (tableau_char,
         NB_ELEMENT,sizeof(tableau_char[0]),comp_str);
   }
```

Exemple Recherche d'un élément dans un tableau trié :

```
void * bsearch (const void * clef,const void * tab
         size_t nb_elt,(*comp)(const void *,const void *))
```

Elle recherche dans le tableau trié `tab` un élément qui est égal à l'élément d'adresse `clef`.

Chapitre 3

FONCTIONS A NOMBRE VARIABLE DE PARAMÈTRES

La fonction `printf`, en C, prend un nombre variable de paramètres. Il est possible de définir **soi même** de telles fonctions en C.

Une fonction possédant un nombre variable de paramètres doit **posséder au moins un paramètre formel fixe** (pff).

La notation ...à ***la fin*** des paramètres indique que la fonction possède un nombre variable de paramètres, éventuellement de types différents, en plus des paramètres formels fixes.

Exemple 1 Fonction `concaténer` prenant en argument plusieurs chaînes de caractères et retournant leur concaténation.

Le prototype d'une telle fonction est le suivant :

```
char * concatener (char * , ...)
```

On pourra par exemple lancer l'appel : `concaténer(S1,S2,NULL)`, ou encore celui-ci : `concaténer(S1,S2,S3,NULLL)`.

Le pointeur `NULL` utilisé à la fin de la liste des arguments permet de marquer la fin de cette liste.

N.B. : Ce choix est purement arbitraire.

Pour accéder à la liste des paramètres , on utilise les macros définies dans le fichier en-tête standard `<stdarg.h>` :

`void va_start(va_liste liste,dernier_paramètre) ;` pour initialisé la liste de paramètres

`type va_arg(va_liste,type) ;` pour accéder aux éléments de la liste

`void va_end(va_liste liste) ;` pour indiquer que l'on a terminé d'utiliser la liste des paramètres

Il faut donc tout d'abord déclarer **une variable pointant sur la liste des paramètres**. Cette variable est de type `va_liste`.

Par exemple : `va_liste liste_paramètres`

Les éléments de la liste des paramètres sont obtenus en **itérant** les appels à la macro-fonction `va_arg` prenant en paramètre l'objet de type `va_liste` (précédemment initialisé à l'aide de la macro-fonction `va_start`) et le type de l'élément à récupérer :

```
va_arg(liste_paramètres , type)
```

La fonction `va_end` libère la liste des paramètres une fois la récupération terminée.

C'est **l'utilisateur lui-même** qui doit gérer le **nombre d'arguments** de la liste. Pour cela, en général on utilise un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Exemple

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

int somme (int nbr , ...)
   { int i; int res = 0;
     va_liste liste_nbr;
     va_start(liste_nbr);
     for (i=0 ; i<nbr ; i++)
        { res++ = va_arg(liste_nbr,int);}
     va_end(liste_nbr);
     return res; }
```

```
int main ()
   { printf(‘‘Somme(2,5)=%d\n’’,somme(2,2,5));
     printf(Somme(4,7,8,9)=%d\n’’,somme(4,4,7,8,9));
     return EXIT_SUCCESS; }
```

Chapitre 4

LA MODULARITÉ

Cela consiste à *découper* un programme en plusieurs petits *composants* indépendants.

Quelques principes à respecter :

1. la lisibilité : le code doit être lisible et facile à comprendre
2. la maintenabilité : une erreur qui se produit est *facilement* détectable, sa correction ne doit pas induire de nouvelles erreurs dans d'autres parties du programme
3. la portabilité : plus d'indépendance de l'environnement et de l'architecture
4. la réutilisabilité
5. l'extensibilité

Comment faire ?

1. **abstraction** des constantes
   Les instructions telles que `int tab[100] ;` nuisent à l'extensibilité du programme et telles que `fopen(''/home/.../fichier'',''r'')` sont une entrave à la portabilité.
2. **factorisation** du code
3. **masquage** de l'implémentation
4. sémantique par référence

Il faut séparer l'interface et l'implémentation

Dans l'approche modulaire, il est nécessaire de séparer l'interface de l'implémentation. Si les modules sont orientés par les données, un module représente un type d'objets et les opérations possibles sur cet objet.

Si on utilise en plus la sémantique par référence, alors les instances des objets sont représentés par des pointeurs.

On doit clairement séparer la partie code de la partie destinée aux clients du module : les clients ne sont pas sensés connaître l'implémentation et ne doivent avoir accès qu'aux services offerts par le module.

En C, la séparation se fait en mettant l'interface dans un fichier `.h` et l'implémentation dans un fichier `.c`

L'approche sémantique par référence uniforme (Uniform Reference Semantics)

C'est une approche qui consiste à manipuler les objets par leur référence. Cette approche est implicite dans les langages tel Lisp, Java. En C, on peut l'implémenter grâce aux pointeurs.

Avantages

- le principal avantage est l'**indépendance** par rapport à l'implémentation (masquage) car on manipule les références des objets, ce qui cache la manière dont ils sont implémentés.
  Par exemple, on peut implémenter une pile soit par un tableau soit par des listes chaînées. Le client, lui, n'aura accès qu'à l'interface : le changement de l'implémentation n'affectera en rien son application.
- le second avantage est la **généricité**. Le fait de ne considérer que des pointeurs permet de manipuler des tests génériques.
- quand on recopie un objet, on recopie sa référence. On occupe **moins de mémoire** et les programmes sont **plus rapides**.

Inconvénients

- quand un objet est référencé plusieurs fois, il devient difficile de savoir quel objet supprimer.
- supprimer un objet non référencé
  $\Rightarrow$ `core dumped`
- ne jamais supprimer un objet
  $\Rightarrow$ on consomme trop de mémoire

Exemple Implémentation d'une pile

pile.h

```
#ifndef PILE_H
#define PILE_H
struct pile;
typedef struct pile * pile;
```

```
pile creer_pile(void);
void empiler_pile(pile p,void * elt);
void depiler_pile(pile p);
void * sommet(pile p);
int est_vide(pile p);

#endif
```

pile.c

```
#include ``pile.h''

struct pile {void * elt;struct pile* next;};

pile creer_pile(void)
   { return NULL; }

void empiler_pile(pile p,void * elt)
   { pile tmp = (pile)malloc(sizeof(struct pile));
     assert(tmp != NULL);
     tmp->elt = elt;
     tmp->next = p;
     p=tmp; }

void depiler_pile(pile p)
   { pile tmp = p;
     if (!est_vide(p))
        { p = p->next;
          free(tmp); }
   }

void * sommet(pile p)
   { if (!est_vide(p))
       { return (p->elt); }
     return NULL; }

int est_vide (pile p)
   { return (p == NULL); }
```

Chapitre 5

LE MAKEFILE

La syntaxe

```
<but> : <dépendances>
<tab>   commande shell
```

Exemple Problème : il est trop long

```
main : main.o pile.o
       gcc -o main main.o pile.o
main.o : main.c pile.h
       gcc -c main.c
pile.o : pile.h pile.c
       gcc -c pile.c
clean :
       rm -f *.o
run : main
       main
```

Solution possible

```
CC = gcc
CFLAGS = -g -Wall
OBJ = main.o pile.o
main : $(OBJ)
     $(CC) $(OBJ) $(CFLAGS) -o main
main.o : main.c pile.h
pile.o : pile.h pile.c
clean :
     rm -f *.o
run : main
     main
```

Chapitre 6

LES ENTRÉES / SORTIES STANDARD (de la bibliothèque standard)

Ensemble des opérations de lecture depuis une unité périphérique vers la mémoire et d'écriture depuis la mémoire vers un terminal, une unité de disque, ...

Les opérations d'E/S peuvent se faire sur des unités de natures diverses. L'approche du C est de manipuler toutes ces unités en tant que fichier, sans distinction.

Le principe de manipulation d'un fichier est le suivant :

1. Ouverture du fichier
2. Lecture, écriture, déplacement dans le fichier
3. Fermeture du fichier

Remarque Toutes les fonctions de manipulation de fichiers se trouvent dans `<stdio.h>`

6.1 Les E/S standard

A l'exécution d'un programme, trois pseudo-fichiers sont ouverts par l'environnement de programmation :

`stdin` fichier d'**entrée standard**, associé par défaut au clavier

`stdout` fichier de **sortie standard**, associé par défaut à l'écran

`stderr` fichier d'**erreurs standard**, associé par défaut à l'écran

6.1.1 Échanges caractère par caractère

Deux fonctions principales :

`int getchar()` → pour lire un caractère
`int puchar()` → pour écrire un caractère

Exemple

```
#include <stdio.h>
int main ()
   { int c;
     while ((c = getchar()) != EOF)
          { putchar(c); }
     return EXIT_SUCCESS; }
```

6.1.2 Échange ligne par ligne

Deux fonctions principales :

`char * gets(char *)` → lire une ligne de `stdin`
`int puts (char *)` → écrire une ligne sur `stdout`

Exemple

```
#include <stdio.h>
int main()
   { char buffer[256]
     while (gets(buffer) != NULL)
        { puts(buffer); }
     return EXIT_SUCCESS; }
```

6.1.3 Échanges avec format

On utilise à cet effet les fonctions `printf` et `scanf`, qui permettent respectivement d'écrire sur `stdout` et de lire sur `stdin`.

6.2 Ouverture d'un fichier

Un flot est codé par une structure identifiée : l'identificateur de type `FILE`.

L'ouverture d'un fichier s'effectue au moyen de la fonction `fopen` de prototype :

```
FILE * fopen (const char *,const char *)
```

La première chaîne de caractère correspond au chemin du fichier, la seconde au mode d'ouverture.

Les modes d'ouverture possibles sont les suivants :

`''r''` → fichier ouvert en lecture ; erreur si le fichier n'existe pas
`''w''` → fichier ouvert en écriture ; erreur si le fichier n'existe pas
`''a''` → fichier ouvert en ajout

6.3 Fermeture d'un fichier

Elle se fait au moyen de la fonction `fclose` de prototype :

```
int fclose(FILE *)
```

Tout fichier ouvert doit être FERME avant la fin du programme.

6.4 Accès au contenu d'un fichier

6.4.1 Accès caractère par caractère

Généralisation de `getchar` et `putchar` :

```
int fgets(FILE *)
int fputc(FILE *)
```

6.4.2 Accès ligne par ligne

On utilise les fonctions suivantes :

```
char * fgets (char *,FILE *)
int fputs (const char *,FILE *)
```

Exemple

```
#include <stdio.h>
int main (int argc , char * argv [])
   { FILE * f;
     char b[256];
     if (argc != 2)
       { return EXIT_FAILURE; }
     if ((f = fopen(argv[1],''r'')) == NULL)
       { printf(''erreur de lecture'');
         return EXIT_FAILURE; }
     while(fgest(b,sizeof(b)) != NULL)
       { fputs(b,stdout); }
     fclose(f); }
```

6.4.3 Accès par enregistrement

Permet de lire et d'écrire des objets structurés dans un fichier. Pour ce type d'accès, le fichier **doit** être ouvert en mode binaire. L'accès se fait par le biais des fonctions suivantes :

```
size_t fread(void * zone,size_t taille,size_t nbr,FILE * f)
size_t fwrite(void * zone,size_t taille,soze_t nbr,FILE * f)
```

Exemple

```
#include <stdio.h>
struct automobile {int age;char couleur[20],numero[10],
                      type[10],marque[10];} parc_auto[20];

int main(int argc,char * argv[])
   { FILE * f;
     int i;
     if ((f = fopen(``fichier'',''rb+'')) == NULL)
        { printf(``''erreur de lecture'');
          return EXIT_FAILURE; }
     for (i=0 ; i<20 ; i++)
        { if ((fread(&parc_auto[i],sizeof(struct automobile),
                               1,f)) != 1)
            { printf(``erreur'');
              return EXIT_FAILURE; }
          fclose(f); }
   }
```

6.5 Déplacement dans les fichiers

Les fonctions déjà vues modifient le automatiquement le pointeur courant dans le fichier correspondant.

Cependant, il est possible de **connaître** et **modifier** la valeur de cette position. Pour cela on utilise :

`int fseek (FILE *,long,int)`

où `long` indique le déplacement dans le fichier et `int` la position initiale

`SEEK_SET` → déplacement par rapport au début du fichier
`SEEK_CUR` → déplacement par rapport à la position courante
`SEEK_END` → déplacement par rapport à la fin du fichier

`int fgetpos(FILE *,fpos_t *)` → indique la position courante
`int fsetpos(FILE *,const fpos_t)` → change la position courante

Exemple Modifier l'âge des voitures dans le fichier `parc`. On réalisera la modification d'un enregistrement dans un fichier n procédant comme suit :

1. On lit un enregistrement du fichier dans une zone mémoire
2. On modifie la zone mémoire
3. On replace le pointeur courant du fichier sur le début de l'enregistrement pour pouvoir récrire cet enregistrement
4. On récrit la zone mémoire dans le fichier

```
struct automobile {int age;char couleur[20],numero[10],
                      type[10],marque[10];} une_auto;

int main (void)
   { FILE * f;
     int i;
     if ((f=fopen(‘‘fichier_parc’’,’’rb+’’)) == NULL)
        { ... }
     for (i=0 ; i<20 ; i++)
        { if (fread(&une_auto,sizeof(une_auto),1,f) != NULL)
            { ... }
          if (fseek(f,sizeof(une_auto),SEEK_CUR) == 0)
            { ... }
          if (fwrite(f,&une_auto,sizeof(une_auto),1) != 1)
            { ... }
        }
     fclose(f);
     return EXIT_SUCCESS; }
```

Chapitre 7

LES SAUTS NON LOCAUX
(`setjump` / `longjump`)

7.1 Contexte d'exécution

Chaque appel de fonction se traduit par l'empilement de son contenu dans la pile d'exécution.

Un **contexte** contient l'adresse de retour de la fonction, l'emplacement de ces variables, la copie des paramètres de l'appel et une référence au contexte précédent.

La sauvegarde et la restauration du contexte met en oeuvre ce principe : une première fonction permet de sauvegarder dans un tampon le contexte local et une seconde permet de le restaurer

7.2 Les fonctions `setjump` et `longjump`

Ces deux fonctions se trouvent dans la bibliothèque `<setjump.h>`. Elles sont de prototypes suivants :

```
int setjump (jump_buf contexte)
void longjump (jump_buf contexte,int valeur_retour)
```

`setjump` sauvegarde le contexte courant et retourne la valeur 0 et `longjump` restaure le contexte.

Exemple

```
#include <setjump.h>
int setjump (jump_buf contexte);
void longjump (jump_buf contexte,int valeur_retour);

(*)  if (setjump(contexte) == 0)
        instruction 1
     else
        instruction 2

(**) longjump(contexte,1);
```

(*) : la fonction `setjump` retourne 0. L'instruction 1 est donc exécutée et on arrive sur (**).

(**) : `longjump(contexte,1)` restaure le contexte

⇒ on peut se retrouver une deuxième fois en (*). Cette fois cependant, `setjump` renverra 1

⇒ on exécute l'instruction 2.

Exemple

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

int main (int argc,char * argv[])
   { FILE * f;
     jump_buf contexte;
     if (setjump(contexte))
        { fprint(stderr,''Erreur d'ouverture\n'');
          return EXIT_FAILURE; }
     if ((f = fopen(argv[1],''r'')) == NULL)
        { longjump(contexte,1);
          ...
     fclose(f);
     return EXIT_SUCCESS; }
```

Chapitre 8

TRAITEMENT DES EXCEPTIONS

Types d'exception

``Failure detection'' , Epuisement des ressources , Autres

8.1 Failure detection

Une telle exception se produit en général à cause d'une erreur de programmation. La meilleure solution, c'est de terminer correctement le programme.

L'outil standard pour détecter une telle erreur est la macro `assert`.

Cette macro est aussi un outil pour la programmation "par contrat", *i.e.* un accord entre le module et le client qui l'utilise.

Exemple

```
        void fonction (type1 param1,...)
           { assert(condition sur les paramètres);
             ...
             assert(condition sur le résultat); }

long factorielle (unsigned long n)
   { unsigned long i,f;
     assert((n < MAX_ULONG) && (n>0));
     f = 1;
     for (i=2 ; i<n ; i++)
        { assert(f < MAX_ULONG);
          f* = i; }
     assert(f < MAX_ULONG); return f; }
```