Pratique du C
Complément sur les pointeurs

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2010-2011

malloc et free

Ces fonctions nécessitent l'inclusion de l'entête stdlib.h et manipulent un segment de mémoire — appelé le tas — associé au processus.

Fonction d'allocation dynamique de mémoire :

- fonction `malloc` de la librairie standard ;
- réserve un espace mémoire dans le tas du processus ;
  - `void *malloc(size_t size)`
    réserve `size` octets dans le tas et retourne un pointeur sur la zone allouée (`NULL` en cas d'echec).

Fonction de désallocation de mémoire ;

- fonction `free` de la librairie standard ;
  - `void free(void *ptr)`
    libère une zone allouée par un précédent `malloc`.
    `ptr` doit obligatoirement être un pointeur retourné par un précédent `malloc`.

Conversion de type

Petit rappel sur le forçage de type : coercition (cast)

- force la conversion de type de la valeur d'un expression :
  (*type*) *expression*
- ne peut être une valeur gauche.

Petit rappel sur la taille d'un objet : opérateur `sizeof`

- `sizeof`(*identificateur_de_type*)
  donne la taille en octets de tout objet de type *identificateur_de_type* ;
- Avec beaucoup de précaution, on peut utiliser
  `sizeof` *expression*
  qui donne la taille en octets de son opérande `expression`. Mais attention :

```
char *ch = "Hello world" ; /*comment est-ce stock\'e~?*/
int main(void){
  char *chlocal = "Hello world" ; /* idem */
  return sizeof(ch) ; /* que retourne cette fonction~?*/
}
```

Exemple lors de l'allocation dynamique :

```
#include <stdlib.h>
struct point {
  int x, y;
};

struct point * reserve_n_cellules(int n){
  return (struct point *) malloc(sizeof(struct point)*n);
}

int main(void){
  struct point *p_point = reserve_n_cellules(10) ;
  return 0 ;
}
```

Attention : l'espace mémoire alloué dynamiquement sur — le tas — dans une fonction n'est pas détruit en fin de fonction comme l'espace associé à une variable automatique (locale).

Remarques sur les fonctions en C

Une fonction en C est

- un objet de première classe : directement manipulable ;
- avec un déclarateur postfixe `()` : `int sqr(int x) ;`
- son fonctionnement est analogue à celui des tableaux :

Déclaration d'un tableau de 5 entiers
`int ar[5] ;`
`temp = ar[i] ;`
déréférencement du pointeur d'entiers `ar` et accès à son élément `i`.
	Déclaration d'une fonction entière à valeur entière
	`int sqr(int x) ;`
	`temp = sqr(i) ;`
	déréférencement du pointeur de fonction `sqr` et appel avec le paramètre `i`.

L'identificateur d'une fonction en C est associé à un pointeur de fonction constant qui pointe sur elle même.

Plus précisément, le nom d'une fonction est un pointeur de fonction constant sur le début du code correspondant à cette fonction.

```
int                          .text
foo                          .globl foo .type foo,@function
(int bar)              foo:   .......
{                            incl     4(%esp)
   return ++bar ;            movl     4(%esp), %eax
}                            ret
                            .globl main
int                         .type    main,@function
main                   main:   ......
(void)                      pushl      $3
{                           call       foo
foo(3) ;                    addl       $16, %esp
return 0;                   movl       $0, %eax
}                           ret
```

Les pointeurs de fonctions : déclaration

- identique au prototype en rajoutant une `*` ;
- déclarer le type retourné et le type des arguments ;
- attention à la priorité : opérateur droit $<<$ opérateur gauche.

Exemple de déclaration :

- `int (*pf)(int, int)` : pointeur de fonction retournant un entier et prenant deux entiers en paramètre ;
- `int *f(int, int)` : fonction retournant un pointeur sur un entier.

```
int (*pfoo)(int) = foo ;
```

```
 /* pfoo = &foo est aussi */
 /* valide mais peu clair */
```

```
    .globl pfoo
           .data
           .align 4
           .type   pfoo,@object
           .size   pfoo,4
pfoo: .long   foo
```

Déclaration d'un synonyme (typedef)

Comme pour les autres déclarations, il est possible de déclarer un type associé aux fonctions comme suit :

```
typedef int fct_t(int) ;
```

Il est ainsi possible de déclarer des types associés aux pointeurs de fonctions

```
typedef fct_t * fctp1_t ;
typedef int (*fctp2_t)(int) ; /* sans utiliser fct_t */
```

L'utilisation de ces types se fait classiquement :

```
int fct(int par) { return par+1 ; }
fctp1_t ftcpv1 ;
fctp2_t ftcpv2 ;
fct_t * ftcpv3 ;
ftcpv1 = fct ;
ftcpv2 = fct ;
ftcpv3 = fct ;
```

Les pointeurs de fonctions : affectation

Opérations sur les pointeurs de fonctions

- affectation d'un pointeur de fonction à :
  - un nom de fonction (pointeur constant) ;
  - une variable de type pointeur de fonction ;
  - les types retournés doivent être *identiques*.
- Exemple d'assignation :

```
int sqr(int x) {
  return x*x;
}
float fsqr(float x) {
  return x*x;
}
int (*pfint1)(int), (*pfint2)(int);

pfint1 = sqr;
pfint2 = pfint1;
/* pfint2 = fsqr; ILLEGAL */
```

Les pointeurs de fonctions : appel

- appel de la fonction pointée : opérateur ()
  - déréférencer le pointeur de fonction ;
  - appeler la fonction pointée en donnant la liste des arguments entre () ;
  - l'expression est du type retourné par la fonction ;
  - le déréférencement est facultatif en C-ANSI.
- Exemple d'appel

```
/* Avec les d\'eclarations du
   transparent pr\'ec\'edent */
int i;                          /* int tab[2]={666,999} ;
                                /* int *p = tab ; */
i = sqr(12);                    /* i = p[1] ;   */
i = (*pfint1)(12);
i = pfint1(12); /* C-ANSI */
```

Les pointeurs de fonctions : utilisation

Exemples d'utilisation des pointeurs de fonction :

- calcul de l'intégrale d'une fonction quelconque de la bonne signature.

```
int sqr3(int x) { return sqr(x) * x; }

int integrale(int (*f)(int), int low, int high) {
  int i, aire = 0;
  for (i = low; i < high; i++) aire += (*f)(i);
  return aire;
}

int main(void) {
  printf("Aire de sqr  sur [1, 10]: %d\n",
         integrale(sqr, 1, 10));
  printf("Aire de sqr3 sur [1, 10]: %d\n",
         integrale(sqr3, 1, 10));
  return 0 ;
}
```

Menu de fonctions

```
   struct COMMANDE {
      char *nom ;
      void (*fun) (char *) ;
   } MENU [] = {    /* on suppose que ls est une     */
      {"ls", ls},   /* fonction d\'eclar\'ee          */
      {"cd", cd},   /* de prototype void ls(char *); */
      {"more", more} ,  /* idem pour cd, more et cat */
      {"cat",  cat},
      {0,0}
     } ;
void executer (char *commande, char *argument)
{    /* strcmp i.e. string compare */
   struct COMMANDE *p = MENU ;
   while (p->nom && strcmp (p->nom, commande)) p++ ;
   if (p->nom) {
      (*p->fun) (argument) ;
   } else fprintf (stderr, "%s : commande inconnue\n",
                           commande) ;
}
```

Fonction quicksort de la librairie standard

```
extern void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));

  typedef struct {
    char *nom;
    int note;
  } Etudiant;
  int inferieur(const void *pp1, const void *pp2) {
    Etudiant *p1 = (Etudiant *) pp1, *p2 = (Etudiant *) pp2 ;
    if (p1->note < p2->note)
      return -1;
    else
      if (p1->note == p2->note)
        return(strcmp(p1->nom, p2->nom));
      else
        return 1;
  }
  Etudiant t[250];
  qsort(t, 250, sizeof(Etudiant), inferieur);
```

Les déclarations complexes

Dans la déclaration `int *(*(*x)())[5] ;` :

- `(*x)` : x est un pointeur...
- `(*x)()` : de fonction qui retourne...
- `(*(*x)())` : un pointeur sur...
- `(*(*x)())[5]` : un tableau de 5...
- `int *(*(*x)())[5] ;` : pointeurs d'entiers.

Problème des déclarations complexes :

- l'opérateur pointeur `*` est préfixe ;
- les opérateurs tableau `[]` et fonction `()` sont postfixes ;
- l'identificateur d'une déclaration est noyé dans des opérateurs.

Pour s'en sortir, on utilise la méthode suivante :

- partir de l'identificateur d'une variable (ou d'un type) ;
- construire le type de l'intérieur vers l'extérieur ;
- en appliquant les règles suivantes :
  - les opérateurs `[]` et `()` ont une plus grande priorité que l'opérateur `*` ;
  - les opérateurs `[]` et `()` se groupent de gauche à droite, alors que les opérateurs `*` se groupent de droite à gauche.

Exemple : `struct s (*(*(*x)[3])())[5] ;`

Plus simplement, il convient d'utiliser des synonymes (typedef) pour simplifier les déclarations.

```
typedef struct s s_t       ; typedef s_t tab5_t[5]      ;
typedef tab5_t * ptrtab5_t ; typedef ptrtab5_t fct_t()  ;
typedef fct_t * ptrfct_t   ; typedef ptrfct_t tab3_t[3] ;
typedef tab3_t *ptrtab3_t  ; ptrtab3_t x                ;
```

Les paramètres de la fonction `main`

En première approximation :

- ce sont des chaînes de caractères stockées par le système dans la zone de données statiques et passés comme arguments la fonction`main` :
  `int main(int argc, char *argv[]) ...`
- `argc` : nombre d'arguments (nom de commande compris) ;
- `argv` : tableau de chaînes de caractères, correspondant aux arguments, nom de commande compris ;
- Exemple d'utilisation :

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr,"Usage: %s argument\n", argv[0]);
    return 1 ;
  }
  if (!(strcmp(argv[1], "-p")) {...} /* option -p */
  if (!(strcmp(argv[1], "-r")) {...} /* option -r */
  return 0 ;
}
```

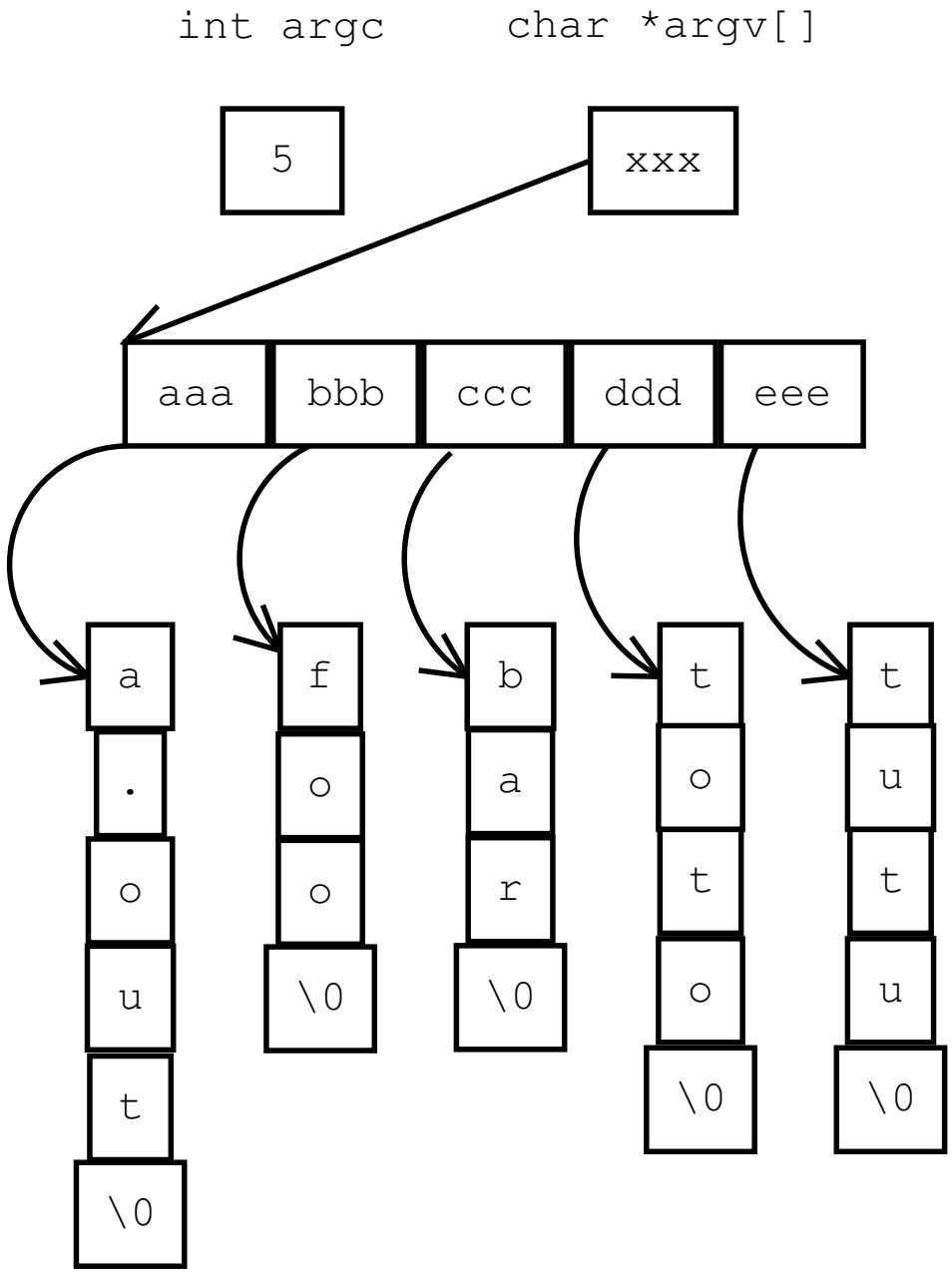
Un autre exemple d'utilisation

```
# include <stdio.h>

int main(int argc, char *argv[]){
  int i ;
  printf(" %d \n",argc) ;
  for( ; argc > 0 ; argc--){
    printf(" %d ",argc) ;
    i = 0 ;
    while(argv[argc-1][i]!=0)
      putchar(argv[argc-1][i++]) ;
    putchar('\n') ;
  }
  return 0;
}
```

```
%gcc mainPar.c
%a.out foo bar toto tutu
 5
 5 tutu
 4 toto
 3 bar
 2 foo
 1 ./a.out
```

Pourquoi écrire `char *argv[]` plutôt que `char argv[][]` ou `char **argv` ?



Les variables d'environnement

Les variables d'environnement correspondent aux variables du Shell. Elles sont :

- stockées dans la zone de données statique ;
- qui est constituée d'une suite finie de chaînes : `<nom>=<value>` ;
- accessibles par la fonction `getenv` :

  ```
  #include <stdlib.h>
  char *getenv(const char *name)
  ```

  recherche dans l'environnement une chaîne de la forme `name=value` et retourne un pointeur sur `value` si elle est présente.

Mais on peut aussi y accéder par les paramètres de la fonction main.

Forme générale des paramètres de la fonction main

Cette forme est :

```
int main(int argc, char *argv[], char **arge)
```

Le dernier paramètre `arge` étant une suite — terminées par null — de chaînes de caractères du types : `varname=value`.

Le code suivant affiche l'ensemble des variables d'environnement dont il dispose :

```
# include <stdio.h>
int main(int argc, char *argv[], char **arge) {
   while(*arge)
      printf("%s\n",*(arge++)) ;
  return 0;
}
```

On obtient entre autre :

```
PWD=/home/calforme/sedoglav/Enseignement/C/Cours/Sources
TERM=xterm
OSTYPE=linux
HOST=espoir.lifl.fr
```