Programmation et structures de données en C cours 5: Généricité (libC)

Jean-Lou Desbarbieux, Stéphane Doncieux
et Mathilde Carpentier
LU2IN018 Sorbonne Université 2020/2021

Sommaire

Pointeurs sur fonctions

Est-il possible de manipuler des fonctions comme des variables ?

Oui, au travers d'un pointeur de fonction. Intérêt :

- passer une fonction en argument à une autre fonction (exemple : "map" de SCHEME ou PYTHON)
- regrouper une donnée avec les fonctions associées dans une même structure (premier pas vers des objets...)

Pointeurs sur fonctions

Est-il possible de manipuler des fonctions comme des variables ?
Oui, au travers d'un pointeur de fonction. Intérêt :

- passer une fonction en argument à une autre fonction (exemple : "map" de SCHEME ou PYTHON)
- regrouper une donnée avec les fonctions associées dans une même structure (premier pas vers des objets...)

Pointeurs sur fonctions

Déclaration :

```
type_valeur_de_retour (* nom_de_variable)(type1 arg1, ...);
```

Initialisation :

```
nom_de_variable = & nom_de_fonction;
nom_de_variable = nom_de_fonction; /* ecriture possible */
```

Utilisation :

```
(* nom_de_variable)(arg1, ...);
nom_de_variable(arg1, ...); /* ecriture possible */
```

Exemple :

```
#include <stdio.h>

void f(int n) {
  printf("n=%d\n", n);
}

int main(void) {
  void (*pf)(int); /* declaration de pf */
  pf=f; /* initialisation de pf */
  pf(3); /* affiche: n=3 */
  return 1;
}
```

Pointeurs sur fonctions : applications

Parcours des arbres ou des listes : donner à la fonction de parcours de la liste le pointeur sur la fonction à appliquer (équivalent au “map” de SCHEME ou PYTHON...).
Exemple : application d’une fonction à chaque élément d’un tableau d’entiers :

```
typedef void (*fonctionSurEntier)(int);
void map (fonctionSurEntier f, int *tableau, int taille) {
  unsigned int i;
  for (i=0;i<taille;i++) {
     f(tableau[i]);
  }
}

void print_int(int i) {
  printf(''Element : %d\n'',i);
}

int main(void) {
  int tab[10]={0,1,2,3,4,5,6,7,8,9};
  map(print_int,tab,10);
}
```

Exemple d’utilisation : bibliothèque générique de listes

Principe

Manipulations de listes toujours les mêmes (insérer, chercher, détruire, ...).
Beaucoup de fonctions de manipulation des listes ne dépendent que peu des données.
Les besoins spécifiques aux données sont pour :

- dupliquer
- copier
- détruire
- afficher
- comparer
- lire
- écrire

Si on associe à nos listes les fonctions permettant de faire ces manipulations sur les données, on va pouvoir écrire une bibliothèque de listes réutilisable !

Principe

Dans la suite, présentation d'un exemple de liste générique (on peut faire de différentes façons...).

On utilise un pointeur générique pour représenter la donnée...
Prototypes de fonctions :

```
void *dupliquer(const void *src);
void copier(const void *src, void *dst);
void detruire(void *data);
void afficher(const void *data);
int compare(const void *a, const void *b);
int ecrire(const void *data, FILE *f);
void *lire(FILE *f);
```

Structure associant donnée et fonctions...

```
typedef struct _element *PElement;
typedef struct _element {
  void *data;
  PElement suivant;
} Element;

typedef struct _liste *PListe;
typedef struct _liste {
  PElement elements;
  void *(*dupliquer)(const void *src);
  void (*copier)(const void *src, void *dst);
  void (*detruire)(void *data);
  void (*afficher)(const void *data);
  int (*comparer)(const void *a, const void *b);
  int (*ecrire)(const void *data, FILE *f);
  void *(*lire)(FILE *f);
} Liste;
```

Insertion en début de liste

```
void inserer_debut(PListe pliste, void *data) {
  PElement newe=malloc(sizeof(Element));
  if (newe==NULL) {
    affiche_message("Erreur d'allocation");
  }
  newe->data=pliste->dupliquer(data);
  newe->suivant=pliste->elements;
  pliste->elements=newe;
}
```

Chercher dans la liste

```
PElement chercher_liste(PListe pliste, void *data) {
  PElement tmp=pliste->elements;
  while(tmp) {
    if (pliste->comparer(data, tmp->data)==0)
      return tmp;
    tmp=tmp->suivant;
  }
  return NULL;
}
```

Afficher/détruire

```
void detruire_liste(PListe pliste) {
  PElement tmp=pliste->elements;
  PElement tmp2;
  while (tmp) {
    tmp2=tmp->suivant;
    pliste->detruire(tmp->data);
    free(tmp);
    tmp=tmp2;}
  free(pliste);}

void afficher_liste(PListe pliste) {
  PElement tmp=pliste->elements;
  while (tmp) {
    pliste->afficher(tmp->data);
    printf("\n");
    tmp=tmp->suivant; }
}
```

Lire/écrire

```
int ecrire_liste(PListe pliste, const char *nom_fichier) {
  FILE *f=fopen(nom_fichier,"w");
  if (f==NULL) {
    printf("Erreur lors de l'ouverture du fichier %s\n",nom_fichier);
    return 0;}
  PElement tmp=pliste->elements;
  while (tmp) {
    pliste->ecrire(tmp->data,f);
    fprintf(f,"\n");
    tmp=tmp->suivant; }
  fclose(f);
  return 1;}

int lire_liste(PListe pliste, const char * nom_fichier) {
  FILE *f=fopen(nom_fichier, "r");
  if (f==NULL) {
    printf("Erreur lors de l'ouverture du fichier %s\n",nom_fichier);
    return 0;}
  void *data=pliste->lire(f);
  while(data!=NULL) {
    inserer_fin(pliste,data);
    free(data);
    data=pliste->lire(f); }
  fclose(f);
  return 1; }
```

Exemple sur des entiers

Exemple de fonction de manipulation d'entiers :

```
void *dupliquer_int(const void *src) {
  int *isrc=(int *)src;
  int *idst=malloc(sizeof(int));
  if (idst==NULL) {
    affiche_message("Erreur d'allocation");
    return NULL;
  }
  *idst=*isrc;
  return (void *)idst;
}
```

Exemple sur des entiers

Exemple de fonction de manipulation d'entiers :

```
void copier_int(const void *src, void *dst) {
  int *isrc=(int *)src;
  int *idst=(int *)dst;
  *idst=*isrc;
}

void detruire_int(void *data) {
  free(data);
}

void afficher_int(const void *data) {
  int *idata=(int *)data;
  printf("%d",*idata);
}
```

Exemple sur des entiers

Exemple de fonction de manipulation d'entiers :

```
int comparer_int(const void *a, const void *b) {
  int *ia=(int *)a;
  int *ib=(int *)b;
  return (*ia>*ib)-(*ia<*ib);
}
```

Exemple sur des entiers

Exemple de fonction pour des entiers :

```
int ecrire_int(const void *data, FILE *f) {
  const int *idata=(const int *)data;
  return fprintf(f, "%d", *idata);
}

void * lire_int(FILE *f) {
  int i;
  int r=fscanf(f," %d",&i);
  if (r<1) return NULL;
  int *pi=(int *)malloc(sizeof(int));
  *pi=i;
  return pi;
}
```

Exemple sur des entiers

```
int main(void) {
  int n;
  PListe pl=malloc(sizeof(Liste));
  if (pl==NULL) {
    printf("Erreur d'allocation");
    return 1; }
  pl->elements=NULL;
  pl->dupliquer=dupliquer_int;
  pl->copier=copier_int;
  pl->detruire=detruire_int;
  pl->afficher=afficher_int;
  pl->comparer=comparer_int;
  pl->ecrire=ecrire_int;
  pl->lire=lire_int;
  int *a=(int *)malloc(sizeof(int));
  int *b=(int *)malloc(sizeof(int));
  int *c=(int *)malloc(sizeof(int));
  *a=12; *b=24; *c=35;
  inserer_debut(pl, (void *)a);
  inserer_debut(pl, (void *)b);
  inserer_debut(pl, (void *)c);
  afficher_liste(pl);
  detruire_liste(pl);
  return 0;
}
```

libC : bibliothèque standard du C

libC : introduction

- Ensemble des fonctions de base fournies avec le compilateur
- Elle est normalisée ! Norme internationale ISO (90 avec mises à jour en 1995, 1999 et 2011)...
- ... mais il y a plusieurs implémentations (parfois avec des ajouts)
- Elle est maintenant souvent fournie avec l'OS :
  - sous Linux : glibc
  - sous windows : Microsoft C run-time library (Microsoft Visual C++)
  - ...

libC : contenu

- `<assert.h>` : macro de test
- `<complex.h>` (C99) : pour gérer les nombres complexes
- `<ctype.h>` : pour tester des caractères (`isalnum`, `isspace`...)
- `<float.h>` : macro donnant des constantes liées à l'implémentation (FLT_MAX, FLT_MIN, ...)
- `<stdio.h>` : entrées/sorties
- `<stdlib.h>` : allocation, générateurs pseudo-aléatoires...
- ...

libC : exemples

```c
#include <stdlib.h>
int atexit(void (*function)(void));
```

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

void ensortant(void) {
  printf("Le programme se termine...\n");
}

int main(void) {
  atexit(ensortant);
  printf("Voila le programme...\n");
  return 1;
}
```

libC : exemples

```c
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
  time_t maintenant = time(NULL);
  struct tm *tnow=localtime(&maintenant);
  char buff[100];

  strftime(buff, sizeof(buff), "%x ---- %c", tnow);
  printf("Par defaut: %s\n",buff);

  setlocale(LC_TIME, "fr_FR.UTF-8");
  strftime(buff, sizeof(buff), "%x ---- %c", tnow);
  printf("En Francais: %s\n",buff);
  return 1;
}
```

Retour sur le tri

Retour sur le tri

Principe du tri à bulle :
Parcours du tableau du premier au dernier élément en permutant toutes les paires de deux éléments consécutifs non ordonnés.
Après le premier passage, l'élément maximum est en dernière position du tableau et après k passages, les k derniers éléments sont bien placés.

L'algorithme de tri est indépendant du type de données triées !

De quoi a-t-on besoin pour le rendre générique ?

Retour sur le tri

1. D'une fonction de comparaison adaptée aux données,
2. De la capacité à déplacer les données.

Retour sur le tri

1. D'une fonction de comparaison adaptée aux données
   $\rightarrow$ à définir pour chaque type de donnée
2. De la capacité à déplacer les données
   $\rightarrow$ possible d'écrire du code générique

Tri à bulle avec des pointeurs sur fonction

```
typedef int (*cmp_function)(const void *a, const void *b);

void triBulle(void *gtab, int nb, int taille, cmp_function cmp )
{
  int i, j;
  char *tab=(char *)gtab;
  char *tmp=(char *)malloc(taille*sizeof(char));

  for (i = nb; i > 1; i--) {
    for (j = 0; j < i - 1; j++) {
      if ((*cmp)(tab+(j + 1)*taille,tab+j*taille)<1) {
        memcpy(tmp,tab+j*taille, taille);
        memcpy(tab+j*taille, tab+(j+1)*taille, taille);
        memcpy(tab+(j+1)*taille, tmp, taille);
      }
    }
  }
  free(tmp);
}
```

Tri à bulle avec des pointeurs sur fonction

Tri d'entiers :

```
int compare_int (const void *a, const void *b)
{
  const int *da = (const int *) a;
  const int *db = (const int *) b;

  return (*da > *db) - (*da < *db);
}

  int t[20]={13, 3, 12, 2, 15, 9, 14, 16, 4, 5, 17, 10, 7, 8,
             19, 1, 11, 18, 6};
  int i;
  printf("=====␣Tri␣d'entiers␣====\n");
  printf("***␣Avant␣***\n");
  for (i=0;i<20;i++) printf("%d\n",t[i]);
  triBulle(t,20,sizeof(int),&compare_int);
  printf("***␣Apres␣***\n");
```

Tri à bulle avec des pointeurs sur fonction

Tri de chaînes de caractères :

```
int compare_str(const void *a, const void *b) {
  const char **sa=(const char **)a; /* ATTENTION... */
  const char **sb=(const char **)b; /* ATTENTION... */
  return strcmp(*sa,*sb);
}


  printf("=====␣Tri␣de␣chaines␣de␣caracteres␣====\n");
  char *ts[4]={"car", "voiture", "avion", "bateau"};
  printf("***␣Avant␣***\n");
  for (i=0;i<4;i++) printf("%s\n", ts[i]);
  triBulle(ts,4, sizeof(char *),&compare_str);
```

LibC et tri : `qsort`

Tri rapide. Prototype :

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

Exemple d'utilisation :

```
#include <stdlib.h>

  ...

  int tab[10]={3,1,0,4,2,8,6,9,7,5};
  qsort(tab,10, sizeof(int),compare_int);
```

Arguments de la fonction main

Généricité d'un programme

Un programme doit pouvoir s'adapter aux besoins de l'utilisateur.
→ utilisation d'arguments en ligne de commande

Exemple : ls

- -l : liste détaillée
- -t : trié par ordre de date
- -a : tout lister
- -d : lister les répertoires comme des fichiers simples
- ...

Arguments de la fonction main

Prototype :

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

- `argc` : nombre d'arguments
- `argv` : arguments, tableau de `char *`

Arguments de la fonction main

```
int main(int argc, char *argv[]) {
  unsigned int i;
  for (i=0;i<argc;i++) {
    printf(``Argument %d : %s\n'',i,argv[i]);
  }
  return 0;
}
```

```
bash$ ./main UE LU2IN018
Argument 0 : ./main
Argument 1 : UE
Argument 2 : LU2IN018
bash$
```

LibC et arguments de la fonction main : getopt

```c
#include <unistd.h>

extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int    getopt(int argc, char * const argv[], const char *optstring);
```

LibC et arguments de la fonction main : getopt

```c
#include <unistd.h>
#include <stdio.h>

int main (int argc, char ** argv) {
  int c;
  while ((c=getopt(argc,argv,"abcd:"))!=-1) {
      if(c=='?')
         printf("Option inconnue !\n");
      else {
         printf("Option %c\n",c);
         if (c=='d')
            printf("  argument: %s\n",optarg);
      }
   }
}
```

```
bash$ ./getopt -a -c -d coucou
Option a
Option c
Option d
  argument: coucou
```

Arguments de fonctions : stdarg

Objectifs

Pas toujours possible de définir *a priori* l'ensemble des arguments d'une fonction.

Exemple typique printf :

```c
int printf(char *fmt, ...);
```

Fonctions à nombre variable d'arguments

La bibliothèque `stdarg.h` de la libC permet de le gérer

Fonctions "variadiques" :

```
type f(type arg1, ...);
```

Gestions de ces fonctions au travers d'un type `va_list` et de 3 macros.

va_list

```
va_list
```

Ce type correspond à un pointeur générique sur un argument.

va_start et va_end

```
void va_start (va_list ap, last);
```

Initialise ap pour les utilisations ultérieures de va_arg et va_end, et doit donc être appelée en premier.

`last` est le dernier paramètre avant la liste d'argument variable.

```
void va_end (va_list ap);
```

va_arg

```
type va_arg (va_list ap, type);
```

La macro va_arg se développe en une expression qui a le type et la valeur de l'argument suivant de l'appel.

ap est la va_list initialisée par va_start.

Chaque appel de va_arg modifie ap pour que l'appel suivant renvoie l'argument suivant.

va_end

```
void va_end (va_list ap);
```

A chaque invocation de va_start doit correspondre une invocation de va_end dans la même fonction. Après l'appel va_end(ap) la variable ap est indéfinie. Plusieurs traversées de la liste sont possible, à condition que chacune soit encadrée par va_start et va_end. va_end peut être une macro ou ue fonction.

exemple d'utilisation de stdarg.h

```
#include<stdio.h>
#include<stdarg.h>
void mini_printf(char *fmt, ...) {
  va_list pa;
  char *p,*vals;
  int vali;
  va_start(pa, fmt);
  for( p=fmt ; *p ; p++) {
    if (*p != '%') {putchar(*p);continue;
    }
    switch (*++p) {
      case 'd' : vali = va_arg(pa, int);
                 ecritentier(vali);
                 break;
      case 's' : for( vals = va_arg(pa, char *); *vals; vals++)
                    putchar(*vals);
                 break;
      default  : putchar(*p);
                 break;
    }}
  va_end(pa);}
```

exemple d'utilisation de stdarg.h

```
void ecritentier(int n) {

  int m=n%10;

  if (n>=10) {
    ecritentier(n/10);
  }

  putchar(m+'0');

}


int main() {
  mini_printf("debut format entier %d puis chaine %s\n",
              3, "la chaine");
  return 0;
}
```

C'est tout pour aujourd'hui !