Cours de C - IR1
2007-2008

Petits secrets du C
&
programmation avancée

Sébastien Paumier

Affectations étendues

- **a+=expression ≈ a=a+expression**
- si l'évaluation de **a** contient un effet de bord, il n'est effecté qu'une fois:

```
int main(int argc,char* argv[]) {
int i=0,a[2];
a[0]=a[1]=7;
a[i++]+=2;
printf("i=%d a[0]=%d a[1]=%d\n",i,a[0],a[1]);
i=0; a[0]=7;
a[i++]=a[i++]+2;
printf("i=%d a[0]=%d a[1]=%d\n",i,a[0],a[1]);
return 0;
}
```

```
$>gcc -Wall -ansi test.c
test.c:14: warning:
operation on `i' may be
undefined
$>./a.out
i=1 a[0]=9 a[1]=7
i=2 a[0]=9 a[1]=7
```

Affectations étendues

- Mêmes règles de conversion de types que pour `a=a+expression`

  `+=` `-=` `*=` `/=` `%=`

  `<<=` `>>=`

  `&=` `^=` `|=`

- Pas de `&&=` ni de `||=`, parce que `&&` et `||` sont paresseux
- Pas de `~=` ni de `!=`, parce que `~` et `!` sont unaires

Affectations étendues

- Opérateurs non utilisables pour des initialisations (c'est logique)

```
int main(int argc,char* argv[]) {
    int y+=45;
    printf("y=%d\n",y);
    return 0;
}
```

```
$>gcc -Wall -ansi test.c
test.c: In function `main':
test.c:48: syntax error before '+=' token
test.c:49: `y' undeclared (first use in this function)
test.c:49: (Each undeclared identifier is reported only once
test.c:49: for each function it appears in.)
```

Labels et sauts

- label=identificateur suivi par `:`
- on peut y sauter avec `goto label;`

```
int main(int argc,char* argv[]) {
    int d;
    /* ... */
    if (d==0) goto null_error;
    /* ... */
    null_error:
    /* ... */
    return 0;
}
```

Labels et sauts

- le **goto** et le label doivent être dans la même fonction, mais pas forcément dans le même bloc
- le label peut être avant le **goto**
- le label doit précéder une *instruction*

```
int main(int argc,char* argv[]) {
    start:
    int d;
    /* ... */
    goto start;
    /* ... */
    return 0;
}
```

```
$>gcc -Wall -ansi labels.c
labels.c: In function `main':
labels.c:9: parse error before "int"
```

Labels et sauts

- "Formellement, le **`goto`** n'est jamais indispensable, et en pratique, on peut facilement s'en passer en général."

Kernighan & Richie, *Le langage C*

- Règle d'or à suivre absolument!
- Seul 1 cas sera toléré

L'unique exception

- Gestion d'erreurs multiples:

```
int main(int argc,char* argv[]) {
   if (argc!=3) goto end;

   FILE* f=fopen(argv[1],"r");
   if (f==NULL) goto end;

   FILE* f2=fopen(argv[2],"w");
   if (f2==NULL) goto close_f;

   int* buffer=(int*)malloc(N*sizeof(int));
   if (buffer==NULL) goto close_f2;

   if (!test(f2)) goto free_buffer;
   /* do the job */
   free_buffer: free(buffer);
   close_f2: fclose(f2);
   close_f: fclose(f);
   end: /* ... */
   return 0;
}
```

Les fonctions imbriquées

- Interdites en C ISO

```
int main(int argc,char* argv[]) {
   void hello_world() {
      printf("Hello world !\n");
   }
   hello_world();
   return 0;
}
```

```
$>gcc -Wall -ansi --pedantic-errors nested.c
nested.c: In function `main':
nested.c:9: ISO C forbids nested functions
```

- N'apportent rien, donc à proscrire 

Le vrai visage de main

- **`int main(int argc,char* argv[], char* env[]) { ... }`**
- **`env`**=tableau contenant toutes les variables d'environnement
- dernière chaîne à **`NULL`**

```
int main(int argc,char* argv[],
         char* env[]) {
   int i;
   for (i=0;env[i]!=NULL;i++) {
      printf("%s\n",env[i]);
   }
   return 0;
}
```

→

```
$>./a.out
LESSKEY=/etc/.less
LC_PAPER=fr_FR
LC_ADDRESS=fr_FR
LC_MONETARY=fr_FR
TERM=xterm
SHELL=/bin/bash
LC_SOURCED=1
HISTSIZE=1000
...
```

Accéder à une variable

- **`char* getenv(const char* name);`** (**`stdlib.h`**)
- renvoie une chaîne ou **`NULL`** si la variable n'est pas trouvée

```
int main(int argc,char* argv[]) {
   char* login=getenv("USER");
   if (login!=NULL) {
      printf("Hello %s !\n",login);
   }
   return 0;
}
```

→

```
$>./a.out
Hello paumier !
```

Invoquer une commande

- **`int system(const char* cmd);`** (**`stdlib.h`**)
- démarre un shell qui lance la commande **`cmd`** et attend sa fin complète
- retourne le code de retour de **`cmd`**

```
int main(int argc,char* argv[]) {
   printf("exit value=%d\n",
          system("ls"));
   return 0;
}
```

→

```
$>./a.out
f1      dir/
exit value=0
```

Mettre des crochets d'arrêt

- `int atexit(void (*f)(void));`
- exécute `f` à la fin du `main` ou quand `exit` est invoquée
- crochets empilés ⇒ ordre inversé
- ex: dump en urgence d'un gros calcul

```
void end1() {printf("end 1\n");}

void end2() {printf("end 2\n");}

int main(int argc,char* argv[]) {
   atexit(end1);
   atexit(end2);
   return 0;
}
```

→

```
$>./a.out
end 2
end 1
```

Mettre des crochets d'arrêt

- ne marche pas si le programme est tué "sauvagement"

```
void end1() {printf("end 1\n");}

void end2() {printf("end 2\n");}

int main(int argc,char* argv[]) {
    atexit(end1);
    atexit(end2);
    fgetc(stdin);
    return 0;
}
```

→

```
$>./a.out
q
end 2
end 1
$>./a.out
<CTRL+c>
```

Allocation sur la pile

- quand on a besoin de mémoire de taille `n` juste le temps d'une fonction, on peut éviter l'allocation dynamique en C99:

```
type nom[n];
```

- `n`=expression entière quelconque
- condition: `n` doit être raisonnable, car sinon la pile peut exploser
- éviter la fonction `alloca` qui n'est pas portable et varie selon les compilateurs

Allocation sur la pile

- exemple: fonction palindrome en UTF8

```
int palindrome2(int* s) {
   int i=0,n=u_strlen(s)-1;
   while (i<n) {
      if (s[i++]!=s[n--]) return 0;
   }
   return 1;
}

/* Ã©tÃ© = été en UTF8, c'est donc un palindrome */
int palindrome_utf8(char* s) {
   int tmp[strlen(s)+1];
   decode_utf8(s,tmp);
   return palindrome2(tmp);
}
```

variable temporaire, allocation dynamique inutile

getopt

- sert à analyser les arguments d'un programme
- permet d'ignorer leur ordre
- standard sous UNIX/Linux (`unistd.h`)
- mal porté sous Windows (cygwin) :(

Les règles

- si un argument commence par `-` (autre que `-` ou `--`), c'est une option
- option courte=1 caractère
- option longue=chaîne de caractères
- si on a `-xyz`, `x`, `y` et `z` désignent des options courtes

Les règles

- **optstring** définit les options courtes autorisées
- **x:** indique que l'option **x** a un paramètre
- **x::** indique que l'option **x** a un paramètre optionnel

```
int main(int argc,char* argv[]) {
    /* -h -> help
     * -i input
     * -o [output] */
    char* optstring="hi:o::";
    /* ... */
    return 0;
}
```

Les règles

- le paramètre d'une option doit être introduit soit par un espace, soit par rien

  `-i xxx` ou `-ixxx` ⇒ arg = `xxx`

  `-i=xxx` ⇒ arg = `=xxx`

- dans le cas d'un paramètre optionnel, seule la forme `-ixxx` fonctionne

Les règles

- si `optstring` commence par `:`, getopt renverra `?` en cas d'option inconnue et `:` en cas d'argument manquant
- la variable externe `int optopt` contient le caractère d'option courant

Utilisation de getopt

- `int getopt(int argc,char* const argv[],const char* optstring);`
- un caractère d'option? `getopt` le renvoie, sinon:
  - `EOF` si plus d'option
  - en cas d'erreur, cf. transparent prédécent

Utilisation de getopt

- Exemple:

```
int main(int argc,char* argv[]) {
   /* -h -> help
    * -i input
    * -o [output] */
   const char* optstring=":hi:o::";
   int val;
   while (EOF!=(val=getopt(argc,argv,optstring))) {
      switch (val) {
         case 'h': printf("help\n"); break;
         case 'o': printf("output %s\n",optarg); break;
         case 'i': printf("input %s\n",optarg); break;
         case ':': printf("arg missing for option %c\n",optopt); break;
         case '?': printf("unknown option %c\n",optopt); break;
      }
   }
   return 0;
}
```

Utilisation de getopt

- on peut grouper plusieurs options
- ordre sans importance

```
$>./getopt -hi
help
arg missing for option i

$>./getopt -o -h -i tutu
output (null)
help
input tutu

$>./getopt -i -y -z -oh
input -y
unknown option z
output h
```

Utilisation de getopt

- **getopt** réorganise les paramètres
- quand il renvoie **EOF**, les paramètres autres qu'options sont rangés entre **optind** et **argc** (l'ordre est conservé)

```
int main(int argc,char* argv[]) {
const char* optstring=":hi:o::";
int val;
while (EOF!=(val=getopt(argc,argv,
                        optstring)));
while (optind!=argc) {
   printf("%s\n",argv[optind++]);
}
return 0;
}
```

```
$>./a.out aa -h bb -i cc dd
aa
bb
dd
```

cc n'y est pas!

Les options longues

- `int getopt_long(int argc,char* const argv[],const char* optstring, const struct option* longopts, int *longindex);`
- accepte des noms longs introduits par `--`
- les arguments optionnels peuvent être introduits soit avec `=`, soit avec un espace: `--input=xxx` ou `--input xxx`

Les options longues

```
struct option {
   const char* name;
   int has_arg;
   int* flag;
   int val;

};
```

- **has_arg**: 0=**no_argument**,
  1=**required_argument**,
  2=**optional_argument**
- **flag**: si **NULL**, **getopt_long** renvoie **val**; sinon, renvoie 0 et ***flag** vaut **val**
- **val**: valeur à renvoyer

Les options longues

- `getopt_long` accepte aussi les options courtes

```
const char* optstring=":hi:o::";
const struct option lopts[] = {
   {"help", no_argument, NULL, 'h'},
   {"input", required_argument, NULL, 'i'},
   {"output", optional_argument, NULL, 'o'},
   {NULL, no_argument, NULL, 0}
};
```

Les options longues

- `getopt_long` renvoie le caractère d'option
- si `longindex` est non `NULL` et si une option longue est trouvée, alors `*longindex` vaut l'indice de l'option dans le tableau

Les options longues

```
const char* optstring=":hi:o::";
const struct option lopts[] = {
   {"help", no_argument, NULL, 'h'},
   {"input", required_argument, NULL, 'i'},
   {"output", optional_argument, NULL, 'o'},
   {NULL, no_argument, NULL, 0}
};
int val,index=-1;
while (EOF!=(val=getopt_long(argc,argv,optstring,lopts,&index))) {
   char msg[32];
   if (index==-1) sprintf(msg,"short option -%c",val);
   else sprintf(msg,"long option --%s",lopts[index].name);
   switch (val) {
      case 'h': printf("%s\n",msg); break;
      case 'o': printf("%s arg=%s\n",msg,optarg); break;
      case 'i': printf("%s arg=%s\n",msg,optarg); break;
      case ':': printf("argument missing for option %c\n",optopt); break;
      case '?': printf("unknown option %c\n",optopt); break;
   }
   index=-1; /* penser à réinitialiser index! */
}
```

Les options longues

- s'il n'y pas d'ambiguïté, les options longues peuvent être abrégées

```
$>./a.out --help --he --h -h --i=toto --i tutu --output
long option --help
long option --help
long option --help
short option -h
long option --input arg=toto
long option --input arg=tutu
long option --output arg=(null)
```

Calculs sur les flottants

- coût beaucoup plus élevé que pour des entiers
- erreurs de calcul
- ⇒ à éviter si possible 
- 2 méthodes:
  - mettre des valeurs en cache
  - simuler des flottants avec des entiers

Calculs avec cache

- exemple trigonométrique

```
#define PI 3.14159265358979323846

void test() {
   double x,y,cosinus[360];
   int i;
   for (i=0;i<360;i++) {
      double angle=i*PI/180.0;
      cosinus[i]=cos(angle);
   }
   int before=time(NULL);
   x=0;
   for (i=0;i<100000000;i++) {
      y=cos(i*PI/180.0); x=x+y*y;
   }
   int middle=time(NULL);
   printf("%d seconds\nx=%f\n",middle-before,x);
   x=0;
   for (i=0;i<100000000;i++) {
      y=cosinus[i%360]; x=x+y*y;
   }
   int end=time(NULL);
   printf("%d seconds\nx=%f\n",end-middle,x);
}
```

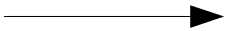
```
$>./a.out
27 seconds
x=50000000.325323
3 seconds
x=49999995.586343
```

plus rapide, mais légère perte de précision

Un calcul bizarre...

```c
#include <stdio.h>

int main(int argc,char* argv[]) {
   int i;
   float sum=0.f;
   for (i=0;i<999999997;i++) {
      sum=sum+0.001f;
   }
   printf("sum=%f\n",sum);
   return 0;
}
```



```
$>./a.out
sum=32768.000000
```

- Pourquoi pas 999999.997 ?
- Parce qu'au-delà de 32768, 0.001 est négligeable par rapport à la mantisse

Simuler des flottants

- Solution: simuler des flottants avec des entiers:
- Exemple: *n* de 0 à 1000000 permet de représenter des flottants à 2 décimales entre 0 et 10000
- Avantages:
  - précision
  - vitesse
- Inconvénient:
  - limite des valeurs

Simuler des flottants

```c
/* A type for manipulating positive floats with 3 decimals */
typedef unsigned int MyFloat;

MyFloat new_MyFloat(float f) {
   if (f<0 || f>1) return -1;
   return (int)(f*1000);
}

float to_float(MyFloat f) {
   return f/1000.0;
}

MyFloat add(MyFloat a,MyFloat b) {
   return a+b;
}

MyFloat mult(MyFloat a,MyFloat b) {
   return a*b/1000; /* Ne pas oublier la division!! */
}
```

Simuler des flottants

```
int main(int argc,char* argv[]) {
   int i;
   int a=time(NULL);
   float sum=0.f;
   for (i=0;i<999999997;i++) {
      sum=sum+0.001f;
   }
   printf("sum=%f\n",sum);
   int b=time(NULL);
   printf("%d seconds\n",b-a);
   MyFloat sum2=0;
   MyFloat inc=new_MyFloat(0.001);
   for (i=0;i<999999997;i++) {
      sum2=add(sum2,inc);
   }
   printf("sum=%d.%d\n",sum2/1000,sum2%1000);
   int c=time(NULL);
   printf("%d seconds\n",c-b);
   return 0;
}
```

faux et lent

```
$>./a.out
sum=32768.000000
11 seconds
sum=999999.997
6 seconds
```

rapide et juste

Simuler des flottants

```
int main(int argc,char* argv[]) {
   int i;
   int a=time(NULL);
   float sum=0.f;
   for (i=0;i<999999997;i++) {
      sum=sum+0.001f;
   }
   printf("sum=%f\n",sum);
   int b=time(NULL);
   printf("%d seconds\n",b-a);
   MyFloat sum2=0;
   MyFloat inc=new_MyFloat(0.001);
   for (i=0;i<999999997;i++) {
      sum2=add(sum2,inc);
   }
   printf("sum=%d.%d\n",sum2/1000,sum2%1000);
   printf("sum=%f\n",sum2/1000.0);
   printf("sum=%f\n",to_float(sum2));
   int c=time(NULL);
   printf("%d seconds\n",c-b);
   return 0;
}
```

```
$>./a.out
sum=32768.000000
11 seconds
sum=999999.997
sum=999999.997000
sum=1000000.000000
6 seconds
```

entier **sum2** stocké avec plus de précision dans le processeur

conversion

⇒ perte de précision

Bien allouer

- allocation n'est pas forcément synonyme de `malloc`, qui a un coût mémoire
- ne pas utiliser `malloc` quand:
  - plein de petits objets (<32 octets)
  - plein d'allocations, mais pas de désallocation
- exemple de solution: gérer sa propre mémoire avec un tableau d'octets

Mauvais malloc

plein de malloc qui se cumulent
=87572Ko

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

#define SIZE_STRING 84
#define N_STRINGS 1000000

int main(int argc,char* argv[]) {
   int i;
   for (i=0;i<N_STRINGS;i++) {
      malloc(SIZE_STRING);
   }
   getchar();
   return 0;
}
```

```
$>./a.out&
$>top
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
19929 paumier   20   0   87572  84m   260 T  0.0  9.6   0:00.20 a.out
```

Bonne allocation personnelle

utilisation d'un tableau
=83540Ko

économie
=87572-83540
=4032Ko
≈4Mo

```
$>./a.out&
$>top
  PID USER      VIRT  RES  SHR
19952 paumier  83540  300  252
```

```c
#define SIZE_STRING 84
#define N_STRINGS 1000000
#define MAX N_STRINGS*SIZE_STRING

char memory[MAX];
int pos=0;

char* my_alloc(int size) {
   if (pos+size>=MAX) return NULL;
   char* res=&(memory[pos]);
   pos=pos+size;
   return res;
}

int main(int argc,char* argv[]) {
   int i;
   for (i=0;i<N_STRINGS;i++) {
      my_alloc(SIZE_STRING);
   }
   printf("%d\n",pos);
   getchar();
   return 0;
}
```

Bonne allocation personnelle

- même comparaison avec des objets de 8 octets au lieu de 84 (par exemple, une structure avec 2 champs entiers)

```
$>top
  PID USER      VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21944 paumier  17216  15m  260 T  0.0  1.8   0:00.08 a.out
21999 paumier   9324  300  252 T  0.0  0.0   0:00.01 a.out
```

- économie=17218-9324=7894Ko≈7,7Mo

Bien réallouer

- si on doit utiliser **realloc**, il faut éviter de le faire trop souvent
- mauvais exemple:

```
/* Adds the given value to the given array,
 * enlarging it if needed */
void add_int(struct array* a,int value) {
if (a->current==a->capacity) {
   a->capacity=a->capacity+1;
   a->data=(int*)realloc(a->data,a->capacity*sizeof(int));
   if (a->data==NULL) {
      fprintf(stderr,"Not enough memory!\n");
      exit(1);
   }
}
a->data[a->current]=value;
(a->current)++;
}
```

Bien réallouer

- bonne conduite: doubler la taille, quitte à la réajuster quand on a fini de remplir le tableau

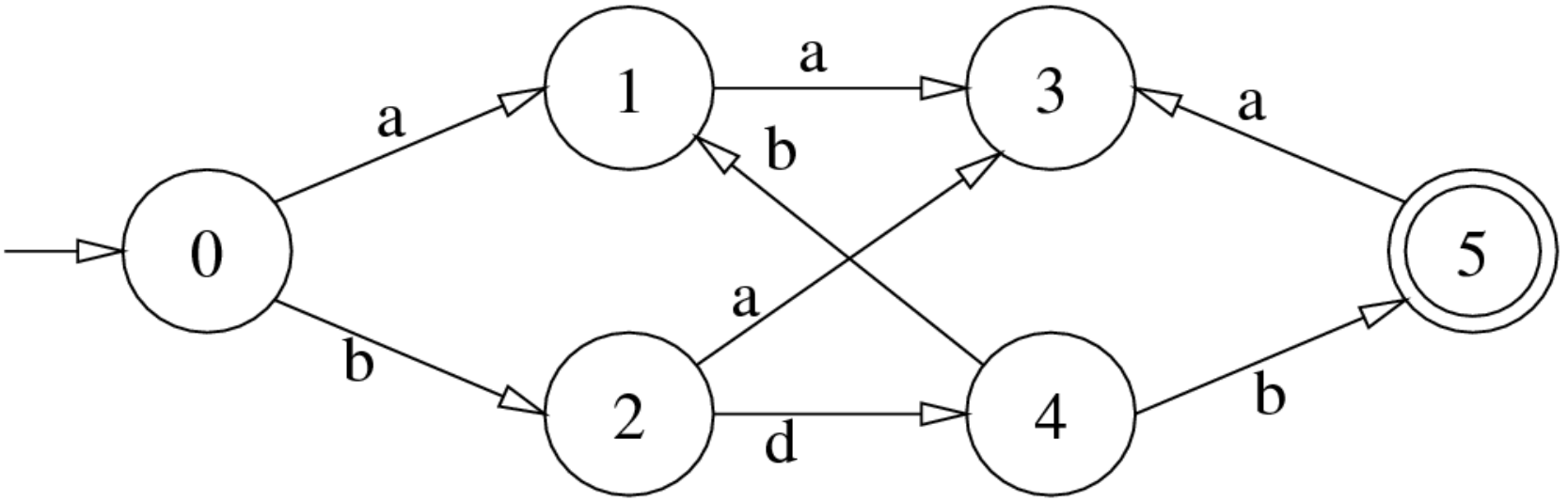
```
/* Adds the given value to the given array,
 * enlarging it if needed */
void add_int(struct array* a,int value) {
if (a->current==a->capacity) {
   a->capacity=a->capacity*2;
   a->data=(int*)realloc(a->data,a->capacity*sizeof(int));
   if (a->data==NULL) {
      fprintf(stderr,"Not enough memory!\n");
      exit(1);
   }
}
a->data[a->current]=value;
(a->current)++;
}
```

Bien désallouer

- on ne doit jamais invoquer `free` plus d'une fois sur un même objet mémoire
- d'où: problème de libération si on a plusieurs pointeurs sur un même objet
- solutions:
  - ne pas le faire
  - comptage de référence
  - table de pointeurs
  - garbage collector

Exemple problématique

- comment représenter un automate ?



Solution 1: éviter le problème

- chaque état est une structure
- automate=tableau de structures
- état=indice dans le tableau

- une seule allocation/libération:

  le tableau de structures
- très bonne solution

Solution 2: comptage de réf.

- chaque état est une structure
- ajouter à cette structure un compteur
- à chaque fois qu'on fait pointer une adresse sur un état, on augmente son compteur
- à chaque fois qu'on déréférence un état, on décrémente son compteur et on le libère quand on atteint 0

Solution 2: comptage de réf.

- lourd à mettre en œuvre
- risque d'erreur (oubli de mise à jour du compteur)
- à éviter

Solution 3: table de pointeurs

- chaque état est identifié par son indice
- automate=tableau de pointeurs
- pour accéder à l'état *n*, on passe par *tableau[n]*
- pour libérer l'état *n*, on libère *tableau[n]* et on le met à `NULL`, s'il n'était pas déjà à `NULL`
- à n'utiliser que si la solution 1 n'est pas possible (c'est rare)

Solution 4: garbage collector

- allocation explicite, mais pas de libération
- tâche de fond ou périodique qui vérifie toute la mémoire
- comment faire:
  - http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html
  - faire du Java :)