IN3R11-2 – C

Cours 2

Damien MASSON
massond@esiee.fr
http://www.esiee.fr/~massond/Teaching/I3FRProg/

21 novembre 2011

Boucles et structures conditionnelles

for, while, do while et if, else, switch

Comme en Java (ou plutôt l'inverse).

`for(){}`, `while(){}` ou `do{}while()` ?

- comme vous voulez !
- celle qui permet de rendre le code le plus lisible possible
- for($init_1$, $init_2$, ..., $init_n$;continuation ;$incr_1$, $incr_2$...) : permet de ne regarder qu'une seule ligne pour comprendre la boucle
- `while(cond)` : equivalent plus joli de `for(;cond;)`
- `do while` :
  - permet de faire toujours un passage sans traiter le premier cas à part ;
  - peut souvent être évitée en initialisant judicieusement les variables ;
  - on ne lit la condition de continuation qu'à la fin, pas toujours lisible...

Booléens, opérateurs logiques

- pas de type `boolean` en C (vous le savez déjà)
- convention C, 0 : faux, $\neq 0$: vrai
- c'est l'inverse de la convention UNIX pour le retour des processus (0 : OK, $\neq 0$: code d'erreur)

```
int main(int argc,char* argv
    []) {
   printf("4␣AND␣6\t=␣%d\n"
          "4␣AND␣0\t=␣%d\n"
          "4␣OR␣6\t=␣%d\n"
          "0␣OR␣0\t=␣%d\n"
          "NOT␣17\t=␣%d\n"
          "NOT␣0\t=␣%d\n",
          4&&6, 4&&0,
          4||6, 0||0,
          !17, !0);
   return 0;
}
```

```
$>./a.out
4 AND 6 = 1
4 AND 0 = 0
4 OR 6 =  1
0 OR 0 =  0
NOT 17 =  0
NOT 0  = 1
```

Évaluation paresseuse

- opérateurs && et || “paresseux”
- évaluation de gauche à droite
- s’arrête dès que possible :

```
(a=0 && b=c) /* b n'est jamais affecte */
(a=1 || b=c) /* b n'est jamais affecte */
```

Les tableaux

Éléments contigus de même type

- déclaration et allocation statique : *type nom*[*taille*] ; où *taille* est connue à la compilation :

```
int t[N];
double cosinus[360];
```

- avec initialisation :

```
char vowel1[6]={'a','e','i','o','u','y'};
/* la taille devient optionnelle */
char vowel2[]={'a','e','i','o','u','y'};
```

- référence : pointeur sur le premier élément

```
int t[]={1,1,3};
int* t2 = t;
int t3[] = t2;
t2[1]=2;
printf("%d\n",t3[1]);/* 2 */
```

- les tableaux statiques sont sur la pile

Taille des tableaux

- un tableau ne connait pas sa taille (comme en Java)
- trois solutions :

taille connue grâce à une constante

```
void print1(int t
    []) {
  int i;
  for (i=0;i<N;i++)
      {
    printf("%d\n",t
        [i]);
  }
}
```

taille passée en paramètre

```
void print2(int t
    [], int n) {
  int i;
  for (i=0;i<n;i++)
      {
    printf("%d\n",t
        [i]);
  }
}
```

utilisation d'un marqueur de fin

```
void print3(int t
    []) {
  int i;
  for (i=0;t[i]!=Z;
      i++) {
    printf("%d\n",t
        [i]);
  }
}
```

Aucun contrôle de débordement ! ! !

Tableaux à n dimensions

```
int t[100][16][45];
```

- chaque $t[i][j]$ est un tableau de 45 int
- si on passe un tableau à une fonction, on doit mettre toutes les dimensions sauf la première :

```
void foo(int t[][M]);
```

- *int t*[2][3] ;

t[0][0]	t[0][1]	t[0][2]	t[1][0]	t[1][1]	t[1][2]
@+0	@+4	@+8	@+12	@+16	@+20

- attention à l'ordre de parcours ! !

dans l'ordre

```
for (i=0;i<2;i++)
  for (j=0;j<3;j++)
    ...t[i][j]...
```

pas dans l'ordre !

```
for (j=0;j<3;j++)
  for (i=0;i<2;i++)
    ...t[i][j]...
```

Les pointeurs

Première passe : pointeurs et tableaux

On peut en C manipuler des adresses mémoire

- ces adresses se stockent dans une variable, appellée pointeur
- un pointeur n'est pas seulement une adresse, il est également associé à un type
- `type * nom;` : déclaration d'une variable `nom` qui contient l'adresse mémoire d'une variable de type `type`
- ca vous rappelle les tableaux ? c'est normal, un tableau n'est ni plus ni moins qu'un pointeur sur son premier élément

```
int tab[]={1,2,3};
int * t = tab;
/* t est un int* : un pointeur */
/* *t est un int : la valeur pointee */
*t = -1;
/* on peut faire de l'arithmetique sur les pointeurs ! */
*(t+1) = -2;
*(t+2) = -3;
printf("%d␣%d␣%d\n", tab[0], tab[1], tab[2]);
/* -1 -2 -3 */
```

tableaux sur la pile : et alors ?

- taille importante = stack overflow
- durée de vie associée à la fonction !

```c
#include <stdio.h>

int * function(){
  int tab[] = {1,2,3};
  return tab;
}

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

```
dm@pc4206b:$ gcc tabstatfunc.c
tabstatfunc.c: In function
    func:
tabstatfunc.c:5: attention :
    cette fonction retourne l
    adresse d une variable
    locale
dm@pc4206b:$ ./a.out
1
-1075103544
-1209200652
```

- solution : allocation dynamique (voir prochain cours)

Les structures

- objets regroupant plusieurs données appelées "champs"
- à définir hors d'une fonction
- définition :

```
struct nom {
    type_champ1 nom_champ1;
    type_champ2 nom_champ2;
    ...
};
```

- déclaration d'une variable :

```
struct nom_type nom_var;
```

- accès aux champs :

```
nom_var.nom_champ
```

- tout ce dont le compilateur connait la taille peut être un champs
- l'ordre des champs est important

Les structures

exemple

```
struct complex {
    double real;
    double imaginary;
    /* struct complex c; /* interdit ! */
};

struct autre {
     struct complex complex; /* OK, et pas d'ambiguite */
};

int main(int argc,char* argv[]) {
    struct complex c;
    c.real=1.2;
    c.imaginary=6.3;
/*
    struct complex c = {1.2,6.3}; /* OK
*/
    printf("%f+%f*i\n",c.real,c.imaginary);
    return 0;
}
```

Structures et tableaux

Les structures sont plus ou moins similaires à des tableaux où les éléments ne sont pas tous du même type.

- adresse de la structure = adresse du premier élément
- adresse du deuxième élément = dépend du compilateur ! Il tente d'obtenir :
  - des adresses multiples de la taille des données
  - des adresses multiples de la taille des pointeurs
  - autre...
- en conséquence, la taille de la structure va dépendre de l'ordre :

sizeof(str) = 12

```
struct str{
    char a;
    int b;
    char c;
    char d;
};
```

sizeof(str) = 8

```
struct str{
    int a;
    char b;
    char c;
    char d;
};
```

Points communs entre structures et tableaux

- affectation des valeurs lors de la déclaration possible,
- recopie par affectation possible,

```
struct to{
  int a;
  char b;
  char c;
  char d;
};

int main(int argc,char* argv[]) {
   struct to z={'a',145,'y','u'};
   struct to t=z;
   printf("%c␣%d␣%c␣%c\n",t.a,t.b,t.c,t.d);
   return 0;
}
```

- stockage sur la pile,
- sauf si déclaration d'un pointeur puis allocation dynamique : à utiliser si la taille est importante ou si le tableau (resp. la structure) doit survivre à la fonction.

Unions

Déclaration d'une zone mémoire vu soit comme `type1`, soit...

```
union toto {
    type1 nom1;
    type2 nom2;
    ...
};
```

- s'utilisent comme les structures
- la taille est celle du plus grand champs
- au programmeur de savoir quel champs utiliser :

```
union toto {
    char a;
    char s[16];
};
int main(int argc,char* argv[]) {
    union toto t;
    strcpy(t.s,"coucou");
    t.a='$';
    printf("%s\n",t.s); /* $oucou */
    return 0;
}
```

Unions

- économise de la mémoire si deux informations sont exclusives :

```
union student {
    char login[16]
    int id;
};
```

- on peut déclarer une union anonyme dans une structure :

```
struct student {
    char name[256];
    union {
        char login[16];
        int id;
    };
};
```

- on peut aussi déclarer une structure anonyme dans une union :

```
union color {
    /* RGB representation */
    struct {
        unsigned char red, blue, green;
    };
    /* 2 colors: 0=black 1=white */
    char BandW;
};
```

Comment éviter de se tromper de champ ?

En encapsulant dans une structure avec un champ d'information :

```
struct color {
    /* 0=RGB 1=black & white */
    char type;
    union {
        /* RGB representation */
        struct {
            unsigned char red, blue, green;
        };
        /* 2 colors: 0=black 1=white */
        char BandW;
    };
};
```

Enumérations

- `enum nom {id1, id2, ..., idn}`
- une variable de type `enum nom` peut alors prendre pour valeurs `id1, id2 ... idn`
- ... mais pas seulement ! ! ! (pas de contrôle du compilateur)
- en fait, les valeurs sont des `int` commencant à 0 et incrémenté de 1 en 1 (id1=0, id2=1, ..., idn=n-1)

exemple (avec un passage par adresse)

```
enum gender {male,female};
void init(enum gender *g,char c) {
  if(c=='m')*g=male;
  else *g=female;
}
```

Enumérations

On peut modifier les valeurs par défaut pour avoir plusieurs fois les mêmes :

exemple

```
enum color {
    blue=45, BLUE=blue, Blue=blue,
    green/* =46 */, GREEN=green, Green=green
};
int main(int argc,char* argv[]) {
    printf("%d␣%d␣%d␣%d␣%d␣%d\n",
           blue,BLUE,Blue,green,GREEN,Green);
    return 0;
}
```

Enumérations : pas de contrôle

- Contrairement à Java : ni warning ni erreur

```
enum gender {male='m',female='f'};
enum color {blue,red,green};
int main(int argc,char* argv[]) {
    enum gender g='z';
    enum color c=g;
    /* ... */
    return 0;
}
```

- Une seule exception : Warning avec `-Wall`

```
enum color {blue,red,green,yellow};
void foo(enum color c) {
    switch (c) {
        case blue: /* ... */ break;
        case red: /* ... */ break;
        /* pas de case default: */
    }
}
```

Enum et constantes

On peut utiliser une `enum` anonyme pour déclarer des constantes :

```
enum {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
    Sunday};
char* names[]={"Monday","Tuesday","Wednesday","Thursday","
    Friday",
                "Saturday","Sunday"};
void print_day(int day) {
   printf("%s\n",names[day]);
}
int main(int argc,char* argv[]) {
   print_day(Saturday);
   return 0;
}
```

Union et Enum

Pour la lisibilité du code, il est conseillé d'utiliser une `enum` pour décrire le champ d'information d'une union :

```
enum cell_type {EMPTY, BONUS, MALUS, PLAYER, MONSTER};
struct cell {
    enum cell_type type;
    union {
        Bonus bonus;
        Malus malus;
        Player player;
        Monster monster;
    };
};
```

Typedef le retour

En fait `typedef` ne définit jamais un nouveau type, mais permet de créer des alias.

```
enum cell_type {EMPTY, BONUS,
    MALUS, PLAYER, MONSTER};

typedef enum cell_type
    CellType;

struct cell {
   CellType type;
   union {
      Bonus bonus;
      Malus malus;
      Player player;
      Monster monster;
   };
};

typedef struct cell Cell;
```

```
typedef enum {EMPTY, BONUS,
    MALUS, PLAYER, MONSTER}
    CellType;

typedef struct {
   CellType type;
   union {
      Bonus bonus;
      Malus malus;
      Player player;
      Monster monster;
   };
} Cell;
```

Appel de fonction : passage par copie

- lorsqu'on appelle une fonction, copie des paramètres
- attention à la persistance !
- cas des tableaux/unions/structures/enumération ?

Passage par adresse

```c
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char ** argv)
{
  int a = 5,b = 10;
  swapt(a,b);
  printf("a = %d, b = %d",a,b);
  /* a = 5, b = 10 */
}
```

```c
void swap(int * a, int * b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char ** argv)
{
  int a = 5,b = 10;
  swapt(&a,&b);
  printf("a = %d, b = %d",a,b);
  /* a = 10, b = 5 */
}
```