

Language C

rappels et compléments

Laurent Réveillère

Enseirb
Département Télécommunications

`reveillere@enseirb.fr`
`http://www.enseirb.fr/~reveille/`

Rappels sur les opérateurs bit à bit

- ◆ Variable de type entier vue comme un vecteur de bits de taille fixe

- ◆ Opérateurs

- « & » *et* booléen
- « | » *ou* booléen
- « ^ » *ou exclusif*
- « ~ » *complémentaire booléen*
- « << » « >> » *décalage de bits à gauche ou à droite*

- ◆ Exemples

```
flags = 0x0a;          /* flags vaut 00001010      */
f5 = fags & (1 << 5) /* valeur du bit 5          */
flags |= 0x20;        /* positionne le bit 5 à 1 */
flags &= ~(1 << 5)    /* positionne le bit 5 à 0 */
```

Affectations combinées

◆ Raccourcis spécifiques pour les opérateurs binaires

- Arithmétiques

- Bit à bit

`var = var op expr; ⇔ var op= expr;`

◆ Exemples

```
i = 1;
i += 2            /* i vaut 3        */
i -= 1            /* i vaut 2        */
i *= 3            /* i vaut 6        */
i |= 0xf0         /* i vaut 246      */
```

Pré et post incrémentations

- ◆ Opérateurs unaire spécifiques pour les variables de type entier

```
var = var + 1; ⇔ var += 1; ⇔ var ++;
```

```
var = var - 1; ⇔ var -= 1; ⇔ var --;
```

- ◆ Instructions considérées comme des expressions

→ Peuvent être utilisés dans des expressions englobantes

- ◆ Lecture de la valeur de la variable

`var ++` → incrémentation après lecture

`++ var` → incrémentation avant lecture

```
i = 1;
j = i ++;          /* j vaut 1 et i vaut 2 */
k = ++ j;         /* k vaut 2 et j vaut 2 */
```

Opérateur ternaire

- ◆ Évaluation conditionnelle de deux expressions en fonction de la véracité d'une expression
- ◆ Forme générale
 - `expression ? expr1 : expr2`
- ◆ Remarque
 - Seul une des expressions « `expr1` » ou « `expr2` » est évaluée
- ◆ Exemple

```
max = (a > b) ? a : b
```

Types énumérés

- ◆ Type énuméré ↔ sous ensemble du type `int`
+ valeurs symboliques

- ◆ Exemple

```
enum Forme_ {  
    FORME_RECTANGLE,          /* majuscules par convention */  
    FORME_CERCLE,  
    FORME_CARRE  
};  
  
enum Form_ f1, f2;           /* définition des variables */  
  
enum Forme_ {               /* Définition combinée */  
    FORME_RECTANGLE,  
    FORME_CERCLE,  
    FORME_CARRE  
} f1, f2;
```

Type énumérés (suite)

- ◆ Utilisation des valeurs symboliques comme des constantes numériques

```
switch (f1) {  
    case FORME_CARRE:  
    case FORME_RECTANGLE:  
        afficher_rectangle ();  
        break;  
    case FORME_CERCLE:  
        afficher_cercle ();  
        break;  
    default:  
        erreur ();  
}
```

Type énumérés (fin)

- ◆ Par défaut, constantes numérotés consécutivement à partir de 0
- ◆ Possibilité de modifier explicitement la valeur à l'aide de l'opérateur « = »

```
enum nombre_premier_ {  
    UN = 1,  
    DEUX,  
    TROIS,  
    CINQ = 5,  
    SEPT = 7  
};
```

Variables références

◆ Cas général

- On identifie une variable par son nom, codé en dur dans le programme

◆ Problème

- Parfois nécessaire de faire varier dynamiquement le nom de la variable sur laquelle porte un traitement
 - » Impossible de changer dynamiquement le programme car déjà compilé
 - » Utilisation de variables spécifiques servant à *référencer* (identifier) d'autres variables

Déclaration de variable référence

◆ Utilisation de l'opérateur « * »

```
int x;  
int *y;           /* Référence sur une variable de type int */  
int *a, *b;      /* Autres références à des variables de entières */
```

◆ Lecture

- `var` référence une variable de type `type`
- `*var` est de type `type`

Valeurs des variables références

◆ Constante

□ NULL désigne une référence sur rien

◆ Variable

□ Opérateur « & » pour obtenir la référence sur une variable

◆ Utilisation

```
int v = 22;
int *x, *y, *z;
x = NULL;           /* Référence nulle */
y = &v;             /* Référence sur v */
z = y;              /* Autre référence sur v */
```

Manipulation de variables références

- ◆ Accéder au contenu d'une variable référence
 - Utilisation de l'opérateur d'indirection « * »

```
int a, b;
int *maxptr;

a = 12;
b = 7;
maxptr = (a > b) ? &a : &b;

printf (« Le maximum vaut : %d\n », *maxptr);
```

Références de références

```
int    a;  
int *  x;  
int ** y;  
  
a = 4;  
x = &a;  
y = &x;  
**y = 7;           /* Maintenant a vaut 7 */  
**y = *x - 3 ;    /* Maintenant a vaut 4 */
```

Occupation mémoire

- ◆ Chaque variable est stocké dans une zone mémoire
- ◆ Taille de la zone dépend du type de la variable de l'architecture
 - ❑ char 1 octet
 - ❑ int 2, 4 ou 8 octets
 - ❑ float 4 octets
 - ❑ double 8 octets
- ◆ Opérateur `sizeof` donne la taille en octets
 - ❑ du type passé en argument
 - ❑ de la variable passé en argument
- ◆ Fonction spécifique
 - ❑ Évaluation pendant la compilation → constante entière

Occupation mémoire

◆ Utilisation de `sizeof`

```
int x;  
printf (« La taille d'un int est %d octets\n », sizeof(int));  
printf (« La taille de x est %d octets\n », sizeof(x));
```

◆ Identification de la zone mémoire servant à stocker la valeur d'une variable

- Adresse de début
- Taille en nombre d'octets

Les références en C

◆ Références en C → pointeurs

- La référence est un pointeur sur l'adresse mémoire de début de la variable

◆ Pointeurs C permissifs

- Possibilité de positionner des pointeurs à des adresses invalides
- Arithmétique des pointeurs

Allocation dynamique

- ◆ Besoin de mémoire pendant l'exécution du programme
 - ❑ On ne connaît pas la taille nécessaire pendant la compilation
 - ❑ Exemples: chaîne de caractère, tableaux, ...
- ◆ Allocation de mémoire
 - ❑ Utilisation de la fonction `malloc`
 - » Passage en paramètre du nombre d'octets souhaité (calculé souvent à partir de `sizeof`)
 - » Retourne un pointeur sur le début de la zone allouée, ou `NULL` si plus de mémoire

```
char *s = (char *) malloc (NBCHAR * sizeof(char) + 1);  
/* allocation d'une chaîne de taille NBCHAR */
```

Libération mémoire

- ◆ Utilisation de la fonction `free`
- ◆ Permet de libérer la mémoire allouée par `malloc`
 - ❑ Ne pas utiliser avec les zones non allouées par `malloc`
 - ❑ Ne pas appeler plusieurs fois sur la même zone

◆ Exemple

```
free (s);      /* Nous n'avons plus besoin de la chaine par la suite */
```

◆ Remarque

- ❑ Pour éviter les fuites mémoires, il est préférable d'appeler `free` dès que la zone mémoire n'est plus utilisé

Liste simplement chaînée

```
struct list_t_ {
    /* données d'un élément de la liste */
    char name[8];
    int  score;

    /* Gestion du chaînage */
    struct list_t_ *next;
};
typedef struct list_t_ * list_t;

/* Fonctions de manipulation */
list_t list_new ();
list_t list_insert (list_t l, char *name, int score);
void   list_print  (list_t l);
list_t list_remove (list_t l, char *name);
void   list_delete (list_t l);

int list_search (list_t l, char *name);
list_t list_filter_greater (list_t l, int score);
```

Liste simplement chaînées - suite

- ◆ Nous souhaitons avoir des fonctions de modification avec effet de bord. Modifier le code précédent en conséquence.

```
/* Fonctions de manipulation */
list_t list_new ();
void list_insert (list_t l, char *name, int score);
void list_print (list_t l);
void list_remove (list_t l, char *name);
void list_delete (list_t l);

int list_search (list_t l, char *name);
list_t list_filter_greater (list_t l, int score);
```

Tableaux et pointeurs

- ◆ Collection de variables de même type
- ◆ Accès possible à un élément en utilisant son indice dans le tableau
 - ❑ Les indices commencent à 0
 - ❑ Aucun contrôle de débordement sur les indices!
- ◆ Données du tableau stockées dans une zone mémoire contiguë
 - ❑ Tableau d'éléments de type T \Leftrightarrow pointeur de T

```
int t[4];           // Tableau de 4 entiers
int  *r;           // Pointeur sur un entier

r = t;             // t est en fait de type (int *)
a = t[0];          // lecture du premier élément de t
b = *r;            // on lit aussi t[0]
c = *t;            // ceci marche aussi
t = r;             // Pas possible car t est une constante
```

Tableaux et pointeurs - suite

- ◆ Utilisation de la notation `[]` à partir de variables de types pointeurs

```
int t[4];           // Tableau de 4 entiers
int  *r;           // Pointeur sur un entier

r = &t[2];         // Référence sur la 2ème case du tableau t
r[1] = 1;         // r[1] ⇔ t[3]
```

- ◆ Arithmétique des pointeurs

□ $p + i \Leftrightarrow *p[i]$

□ $p - i \Leftrightarrow \&p[-i]$

```
int t[4];           // Tableau de 4 entiers
int  *r;           // Pointeur sur un entier

r = t + 2;         // Identique à &t[2]
r[1] = 1;
```

Opérateurs sur les pointeurs

◆ Différence entre deux pointeurs

□ $p - q \Leftrightarrow (\text{adresse de } p - \text{adresse de } q) / \text{taille du type pointé}$

◆ Autres opérateurs classiques

□ « += », « -= », « ++ », « -- »

◆ Exemple

```
int t[4];           // Tableau de 4 entiers
int *r, end;       // Pointeur sur un entier
int i;

for (i = 0; i < 4; i++)
    printf ("%d", t[i]);

end = t + 4;        // Pointeur sur la première case après la fin de t
for (r = t; r < end; r++) // équivalent à la boucle précédente
    printf ("%d", *r);
```

Tableaux multi-dimensionnels

◆ Utilisation de plusieurs paires de []

```
int t[4][5];          // Tableaux de 4 lignes de 5 colonnes
```

◆ Stockage des données par ligne majeure

□ `t[i+1][0]` stocké juste après `t[i][m-1]`

```
int t[2][3] = { {1,2,3}, {11,12,13} };
int *l0;      // Pointeur sur la première ligne
int *l1;      // Pointeur sur la deuxième ligne

l0 = t[0];
l3 = t[1];
l1[1]++;      // Incrémente la deuxième case de la seconde ligne
t[0][4]--;    // Pas de vérification de débordement
```

Tableaux et pointeurs

◆ Tableaux et pointeurs de pointeurs

```
int  *t[2];      /* Tableau de pointeur sur int, t est de type (int **) */
int  t1[3] = {1,2,3};
int  t2[3] = {11,12,13};

t[0] = t1;
t[1] = t2;      // On obtient le même tableau que précédemment
t[0][1] = t[1][2]; // Valide
t[1][3] = 0;    // Débordement des limites mais pas de vérifications
```

◆ Tableau dynamique

□ Allocation de la taille du tableau à l'exécution

```
int  len;      /* Nombre de cellules du tableau */
int  *tab;     /* Tableau d'éléments de type int */

printf ("Nombre d'éléments: "); scanf("%d", &len);
tab = (int *) malloc (len * sizeof(int));
assert (tab != NULL); // Termine le programme si pas vrai

for (i = 0; i < len; i++)
    t[0] = i;      // Remplissage du tableau
```

Pointeurs de fonctions

- ◆ Comme tout objet, une fonction possède une adresse qui l'identifie de manière unique
 - Adresse de début de la première instruction de la fonction
- ◆ Pointeur de fonctions
 - `int (* funptr) (char *, int)`
 - » `funptr` Pointeur sur une fonction
 - » La fonction pointé par `funptr` prend une chaîne de caractères et un `int` en argument et renvoie un `int`

Pointeurs de fonction

```
typedef struct list_ {
    void *elt;      // data
    list next;     // Pointeur sur élément suivant
} * list;

// Fonction générique de filtrage sur les listes génériques
list
list_filter (list l, int (*predicate) (void *elt)) {
    list tmp = list_new();
    while (l) {
        if (predicate(l->elt))
            list_insert(tmp, l->elt);
        l = l->next;
    }
    return tmp;
}
```