COURS 5 - PORTÉE DES VARIABLES ET DÉFINITION DE FONCTIONS

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

PORTÉE D'UNE VARIABLE

NAISSANCE ET VIE D'UNE VARIABLE

▷ Un bloc est délimité par des accolades

▷ Toute variable a une durée de vie bornée au bloc où elle est déclarée

▷ Ce bloc définit la portée de la variable

▷ L'espace mémoire pour stocker la variable est alloué lors de sa déclaration et libéré à la fin du bloc où elle a été allouée

VARIABLE GLOBALE

▷ Les variables déclarées hors de tout bloc sont appelées globales et utilisables dans tout le fichier (à éviter)

▷ Une variable locale masque obligatoirement une variable de même nom située dans une portée plus englobante

```
int a=5, b=12;
int main(void)
{
    int a=3, i=0;
    printf("%d",a);//display 3
    for(i=0; i<10; i=i+1){
        int a=4;//A EVITER
        printf("%d",a);//display 4
    }
    printf("%d",b);//display 12
}
```

LES FONCTIONS

DÉFINITION

▷ La fonction est l'unité modulaire fondamentale en langage C

▷ Une fonction est généralement conçue pour effectuer une tâche spécifique et son nom reflète souvent cette tâche

▷ Une fonction permet de factoriser du code

▷ Une fonction contient des déclarations et des instructions

CARACTÉRISTIQUES

- Une fonction est caractérisée par son prototype
  - Un nom
  - Un liste (possiblement vide) d'arguments (correspondant à des variables locales de la fonction)
  - Un type de retour

```
type_retour nom(arguments);
int max(int a, int b);
```

VOID

▷ `void` est un type spécial

▷ En type de retour, il indique l'absence de valeur de retour

```
void print_sum(int a, int b){
    ...
}
```

▷ En paramètre, il indique l'absence de paramètre

```
int get_day(void){
...
}
```

DÉFINITION VS DÉCLARATION

▷ Définition = code de la fonction

▷ Déclaration = juste son prototype suivi de ';' (indispensable avant toute utilisation)

▷ Une définition vaut pour déclaration

```
void foo(void);

void bar(void){
    ...
    foo();
    ...
}

void foo(void){
    ...
    bar();
    ...
}
```

RETOURNER UNE VALEUR

▷ L'instruction `return` permet de quitter la fonction - quelque soit l'endroit où elle est appelée

▷ Renvoie la valeur produite par une expression si le type de retour est différent de `void`

▷ Inutile seulement en fin de fonction avec retour de type `void`

```
void print_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
}
...
float print_and_return_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
    return a+b;
}
```

VALEUR DE RETOUR

▷ On peut ignorer la valeur de retour d'une fonction

▷ On ne peut pas utiliser une fonction ne renvoyant rien dans une expression

```
float print_and_return_sum(float a, float b){
    printf("%f+%f=%f\n",a,b,a+b);
    return a+b;
}
int main(void){
    float sum=0;
    print_and_return_sum(3.5, 1.1);
    sum=print_and_return_sum(3.5, 1.1);
    printf("%f",sum);
    return EXIT_SUCCESS;
}
```

LE CAS DE MAIN

▷ `main` est une fonction particulière où `return` quitte le programme et renvoie le code de retour du programme

▷ Si `main` est appelée explicitement par une fonction, `return` fonctionne normalement

```
int i=0;
void foo();
int main(void){
    printf("Main function was called\n");
    if(i==0){
        foo();
    }
    return EXIT_SUCCESS;
}
void foo(){
    i=i+1;
    printf("Main function returned %d\n",main());
}
```

APPEL D'UNE FONCTION

▷ Un appel de fonction est une expression qui passe le contrôle et des arguments (le cas échéant) à une fonction et qui se présente sous la forme

```
expression (liste_d_expressions)
```

où `expression` est un nom de fonction et où `liste_d_expressions` est une liste d'expressions (séparées par des virgules)

```
max(5,6)
```

▷ Les valeurs des expressions de `liste_d_expressions` sont les arguments passés à la fonction

APPEL D'UNE FONCTION

▷ Tous les arguments sont passés par valeur

▷ Cela signifie que seule la valeur de l'expression correspondant à l'argument est assignée au paramètre

▷ La fonction ne connaît pas l'origine (e.g. emplacement mémoire) de la valeur de l'argument passé

▷ La fonction utilise cette valeur sans affecter les éventuelles variables de l'expression de laquelle elle était initialement dérivée

APPEL D'UNE FONCTION

```
int max(int a, int b){
//1er appel a=3 et b=4
//2nd appel a=1 et b=4
    if(a<b){
        return b;
    }
    return a;
}
int main(void){
    int a=1, x=3, y=4, m=0;
    m=max(x, y);
    printf("max(%d,%d)=%d", x, y, m);
    printf("max(%d,%d)=%d", a, y, max(a,y));
}
```

APPEL D'UNE FONCTION

▷ Les pointeurs fournissent un moyen à une fonction d'accéder en écriture à la valeur d'une variable non locale

▷ Comme un pointeur contient l'adresse d'une variable, la fonction peut utiliser cette adresse pour accéder à la valeur de la variable

```
void swap(int * num1, int * num2){
    int tmp = *(num1);
    *(num1) = *(num2);
    *(num2) = tmp;
}
int main(void)
{
    int x=3, y=5;
    printf("%d %d",x,y); // display 3 5
    swap(&x, &y);
    printf("%d %d",x,y); // display 5 3
}
```

LE CAS PARTICULIER DES TABLEAUX

▷ Sachant qu'une variable de type tableau contient l'adresse de son premier élément, lorsqu'un tableau est passé en paramètre, c'est en fait cette adresse qui est passée en paramètre

▷ Il serait donc naturel de déclarer ce dernier comme étant de type pointeur

```
void print_tab(int * t, int nb_elem)
```

▷ Inconvénient: Comment faire la différence entre un pointeur vers un seul élément ou un tableau ?

▷ Le langage C admet que l'on puisse déclarer un paramètre de type tableau (e.g., `int []`) – qui sera considéré comme un pointeur par le compilateur (e.g., `int*`)

```
void print_tab(int t[], int nb_elem)
```

LE CAS PARTICULIER DES TABLEAUX

▷ Rappelons que `return` renvoie la valeur produite par une expression

▷ Il ne faut JAMAIS renvoyer l'adresse d'une variable locale car l'espace mémoire allouée pour cette dernière sera libéré à la fin du bloc où elle est définie

▷ L'adresse devient donc invalide car elle peut être réutilisée pour une autre variable ultérieurement

▷ Il est INUTILE ou DANGEREUX de renvoyer un tableau

  ▷ Est-ce un problème ?

LE CAS PARTICULIER DES TABLEAUX

▷ Rappelons que `return` renvoie la valeur produite par une expression

▷ Il ne faut JAMAIS renvoyer l'adresse d'une variable locale car l'espace mémoire allouée pour cette dernière sera libéré à la fin du bloc où elle est définie

▷ L'adresse devient donc invalide car elle peut être réutilisée pour une autre variable ultérieurement

▷ Il est INUTILE ou DANGEREUX de renvoyer un tableau

▷ Est-ce un problème ?

▷ Non puisque le contenu du tableau est accessible dans la fonction et donc toujours modifiable par défaut

LE CAS PARTICULIER DES TABLEAUX

▷ Rappelons que `return` renvoie la valeur produite par une expression

▷ Il ne faut JAMAIS renvoyer l'adresse d'une variable locale car l'espace mémoire allouée pour cette dernière sera libéré à la fin du bloc où elle est définie

▷ L'adresse devient donc invalide car elle peut être réutilisée pour une autre variable ultérieurement

▷ Il est INUTILE ou DANGEREUX de renvoyer un tableau

  ▷ Est-ce un problème ?

  ▷ Non puisque le contenu du tableau est accessible dans la fonction et donc toujours modifiable par défaut

  ▷ `const int t[]` permet d'interdire la modification des éléments du tableau passé en paramètre

DÉFINIR UNE FONCTION

ECRIRE UNE FONCTION

- Il faut réfléchir à l'utilité de la fonction
- Une fonction doit correspondre à une tâche
  - Par exemple, on ne mélange pas calcul et affichage

```
int minimum(int a, int b){
    int min=b;
    if(a<b){
        min=a;
    }
    printf("minimum=%d\n",min); //A EVITER !
    return min;
}
```

  - Pourquoi ?

Définir le prototype

- De quoi la fonction a-t-elle besoin ?
  - Paramètres
- Retourne-t-elle quelque chose ?
- Y a-t-il des cas d'erreurs ?
- Si oui, il y a 3 solutions
  - Mettre un commentaire
  - Renvoyer un code d'erreur
  - Afficher un message et quitter le programme

LE COMMENTAIRE

▷ Le commentaire est utile quand la fonction n'est pas censée être appelée dans certains cas

```
/**
* Copies the array 'src' into the 'dst' one.
* 'dst' is supposed to be large enough.
*/
void copy(int src[], int dst[]);

/**
* Returns 1 if 'w' is an English word;
* 0 otherwise. 'w' is not supposed to be
* NULL;
*/
int is_english_word(char* w);
```

▷ Mais ne permet pas de prévenir les erreurs du programmeur liées au non respect de ce dernier

Le code d'erreur

▷ Il est possible de renvoyer un code d'erreur si la fonction ne devait rien renvoyer

```
#define ERROR_CODE 0
#define SUCCESS_CODE 1

int init(int t[], int size){
    int i=0;
    if(size<=0){
        return ERROR_CODE;
    }
    for(i=0; i<size; i++){
        t[i]=0;
    }
    return SUCCESS_CODE;
}
```

▷ ⚠Sinon, on doit toujours pouvoir distinguer un cas d'erreur d'un cas normal (attention à la valeur choisie)

LE CODE D'ERREUR

```c
#define ERROR_CODE -1
int minimum(int t[], int size){
    if(size<=0){
        return ERROR_CODE;
    }
    int min=t[0];
    int i;
    for(i=1; i<size; i=i+1){
        if(min<t[i]){
            min=t[i];
        }
    }
    return min;
}
```

```c
#define ERROR_CODE -1
/**
* Returns the length of the given
* string or ERROR_CODE if NULL.
*/
int length(char* s){
    if(s==NULL){
        return ERROR_CODE;
    }
    int i;
    for(i=0; s[i]!='\0'; i=i+1);
    return i;
}
```

- ⚠ On ne peut pas savoir si on a une erreur ou si le minimum est -1
- La taille d'une chaîne ne peut être négative

Le code d'erreur

▷ Si toutes les valeurs possibles sont prises, il faut utiliser un pointeur pour le résultat ou pour le code d'erreur

```
#define ERROR_CODE 0
#define SUCCESS_CODE 1
int quotient(int a, int b, int * res){
    if(b==0){
        return ERROR_CODE;
    }
    *res=a/b;
    return SUCCESS_CODE;
}
```

LES ERREURS GRAVES

▷ En cas d'erreur insurmontable, il est possible d'interrompre le programme en appelant l'instruction **exit** définie dans **stdlib.h** qui prend en paramètre EXIT_FAILURE ou EXIT_SUCCESS

```
int foo(){
    ...
    if(erreur_fatale){
        printf("A fatal error occured"); //A EVITER (c.f. CM 9)
        exit(EXIT_FAILURE);
    }
    ...
}
```

▷ Il est important de quitter dès que possible une fonction en traitant les cas d'erreur le plus tôt possible

DOGGY BAG

TO TAKE AWAY ...

▷ Toute variable a une durée de vie bornée au bloc où elle est déclarée

▷ Eviter les variables globales

▷ Toute fonction doit être déclarée avant utilisation

▷ Tous les arguments et la valeur de retour d'une fonction sont le résultat d'évaluation d'expressions

▷ Il ne faut jamais renvoyer l'adresse d'une variable locale

▷ On doit toujours pouvoir distinguer un cas d'erreur d'un cas normal

Questions?