Fonctions, pointeurs et structures

Définition d'une fonction

```
type identificateur(liste_déclaration_paramètres){
    liste_declarations
    liste_instructions
}
```

- `type` est le type de valeur retournée par la fonction,

Définition d'une fonction

```
type identificateur(liste_déclaration_paramètres){
    liste_declarations
    liste_instructions
}
```

- `type` est le type de valeur retournée par la fonction,
- `identificateur` est le nom de la fonction,

Définition d'une fonction

```
type identificateur(liste_déclaration_paramètres){
    liste_declarations
    liste_instructions
}
```

- `type` est le type de valeur retournée par la fonction,
- `identificateur` est le nom de la fonction,
- `liste_déclaration_paramètres` est la liste de déclaration des paramètres formels,

Définition d'une fonction

```
type identificateur(liste_déclaration_paramètres){
    liste_declarations
    liste_instructions
}
```

- `type` est le type de valeur retournée par la fonction,
- `identificateur` est le nom de la fonction,
- `liste_déclaration_paramètres` est la liste de déclaration des paramètres formels,
- `liste_declarations` permet de déclarer des variables locales à la fonction,

Définition d'une fonction

```
type identificateur(liste_déclaration_paramètres){
    liste_declarations
    liste_instructions
}
```

- `type` est le type de valeur retournée par la fonction,
- `identificateur` est le nom de la fonction,
- `liste_déclaration_paramètres` est la liste de déclaration des paramètres formels,
- `liste_declarations` permet de déclarer des variables locales à la fonction,
- `liste_instructions` est l'ensemble des instructions exécutées à l'appel de la fonction.
  - En dehors des fonctions de type `void`, il doit au moins y avoir une instruction du type

    ```
    return expression;
    ```

Prototype de fonction

Le **prototype** de la fonction est

```
type identificateur(liste_déclaration_paramètres)
```

Valeur de retour

Si la fonction n'est pas de type `void`, il doit au moins y avoir une instruction du type

```
return expression;
```

- `expression` est évaluée,
- la valeur rendue par la fonction est celle de l'expression évaluée,
- cela interrompt l'exécution de la fonction.

Valeur de retour

Si la fonction n'est pas de type `void`, il doit au moins y avoir une instruction du type

```
return expression;
```

- `expression` est évaluée,
- la valeur rendue par la fonction est celle de l'expression évaluée,
- cela interrompt l'exécution de la fonction.

Le type retourné ne peut être un tableau ou une fonction.

`expression` est convertie au type de la fonction

Appel d'une fonction

L'appel d'une fonction est une **expression**

```
nom_fonction( liste_expressions )
```

Les expressions de `liste_expressions` sont évaluées puis passées en paramètre de la fonction.

La valeur de l'expression est celle renvoyée par la fonction.

Appel d'une fonction

L'appel d'une fonction est une **expression**

```
nom_fonction( liste_expressions )
```

Les expressions de `liste_expressions` sont évaluées puis passées en paramètre de la fonction.

La valeur de l'expression est celle renvoyée par la fonction.

Attention : l'ordre d'évaluation des expressions de la liste n'est pas spécifié :

```
fonction1( n=2, fonction2(n) );
```

résultat aléatoire...

Généralités

Pas de fonctions imbriquées en C : une fonction de ne peut être définie à l'intérieur d'une autre fonction.

Généralités

Pas de fonctions imbriquées en C : une fonction de ne peut être définie à l'intérieur d'une autre fonction.

Rien de spécial pour appeler la fonction elle-même dans sa définition (récursivité)

```
int facto(int n){
    if (n == 1) return(1);
    else return(n * facto(n-1));
}
```

Paramètres formels d'une fonction

- ont la même durée de vie et visibilité que les variables locales à la fonction,

Paramètres formels d'une fonction

- ont la même durée de vie et visibilité que les variables locales à la fonction,
- la liste des paramètres peut être vide

```
type fonction(void){
    ...
}
fonction();
```

Paramètres formels d'une fonction

- ont la même durée de vie et visibilité que les variables locales à la fonction,
- la liste des paramètres peut être vide

```
type fonction(void){
    ...
}
fonction();
```

- lors de l'appel de la fonction :
  - les expressions doivent être du même type et dans le même ordre que les paramètres formels,
  - les paramètres formels prennent alors la valeur des expressions.

Paramètres formels d'une fonction

```
int max(int a, int b){
    a = (a > b ? a : b);
    return a;
}
int main(){
    int a = 2;
    int b = 3;
    b = max(a,b) + 1;
    return 0;
}
```

Equivalent à

```
int main(){
    int a = 2;
    int b = 3;
    {
        int x = 2;
        int y = 3;
        x = (x > y ? x : y);
        b = x + 1;
    }
}
```

Problème

Les paramètres d'une fonction sont une copie de la valeur des variables à l'appel de la fonction.

Donc si le paramètre est modifié dans le corps de la fonction, la valeur de la variable ne change pas.

```
void echange(int x, int y){
    int z = y;
    y = x;
    x = z;
}
int main(){
    int a = 2;
    int b = 3;
    echange(a,b);
    printf("%d   %d\n", a, b);
    return 0;
}
```

Solution 1 : utiliser une fonction qui renvoie plusieurs valeurs

```
/*
couple_entier echange(int x, int y){
    int z = y;
    y = x;
    x = z;
    return (x,y);
}
int main(){
    a = 2;
    b = 3;
    (a,b) = echange(a,b);
    printf("%d   %d\n", a, b);
    return 0;
}
*/
```

Solution 2 : variables globales

Les **variables globales** sont définies en dehors de toute fonction, et sont accessibles partout.

```
int a,b;
int echange(void){
    int c = b;
    b = a;
    a = c;
}
int main(){
    a = 2;
    b = 3;
    echange();
    printf("%d   %d\n", a, b);
    return 0;
}
```

Solution 2 : variables globales

Les **variables globales** sont définies en dehors de toute fonction, et sont accessibles partout.

```
int a,b;
int echange(void){
    int c = b;
    b = a;
    a = c;
}
int main(){
    a = 2;
    b = 3;
    echange();
    printf("%d   %d\n", a, b);
    return 0;
}
```

Dangereux en général, mais utile si la variable globale est l'objet principal du programme.

Solution 3 : modification du paramètre passé

Idée : passer en paramètre l'adresse de la variable dont on veut modifier la valeur.

Solution 3 : modification du paramètre passé

Idée : passer en paramètre l'adresse de la variable dont on veut modifier la valeur.

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Solution 3 : modification du paramètre passé

Idée : passer en paramètre l'adresse de la variable dont on veut modifier la valeur.

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Le type d'un pointeur est fonction du type de la variable pointée.
Déclaration d'un pointeur sur un entier

```
int* pt;
```

Solution 3 : modification du paramètre passé

Idée : passer en paramètre l'adresse de la variable dont on veut modifier la valeur.

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Le type d'un pointeur est fonction du type de la variable pointée.
Déclaration d'un pointeur sur un entier

```
int* pt;
```

Attention : Les adresses ont des valeurs entières (sauf adresse `NULL`) mais ne sont pas de type `int`.

2 opérateurs fondamentaux :

- opérateur `&` qui délivre l'adresse d'une **lvalue**
- opérateur `*` qui délivre la **lvalue** pointée par un pointeur

```
int b = 5;
int* pt = &b;
printf("b vaut %d et est à l'adresse %p\n", b, &b);
printf("b vaut %d et est à l'adresse %p\n", *pt, pt);
```

2 opérateurs fondamentaux :

- opérateur `&` qui délivre l'adresse d'une **lvalue**
- opérateur `*` qui délivre la **lvalue** pointée par un pointeur

```
int b = 5;
int* pt = &b;
printf("b vaut %d et est à l'adresse %p\n", b, &b);
printf("b vaut %d et est à l'adresse %p\n", *pt, pt);
```

Si *i* et *j* sont 2 pointeurs, que valent :

```
*i**j
*i/*j
```

Solution 3

```
int echange(int* x, int* y){
    int z = *y;
    *y = *x;
    *x = z;
}
int main(){
    a = 2;
    b = 3;
    echange(&a, &b);
    printf("%d   %d\n", a, b);
    return 0;
}
```

Piège

```
void reinitPointeur(int* p){
    p = NULL;
}

int main(){
    int a = 1;
    int* p = &a;
    reinitPointeur(p);
    printf("l'adresse de p est %p\n", p);
    return 0;
}
```

Solution 1

Changer le type retourné par la fonction

```
int* reinitPointeur(){
    return NULL;
}

int main(){
    int a = 1;
    int* p = &a;
    p = reinitPointeur();
    printf("l'adresse de p est %p\n", p);
    return 0;
}
```

Solution 2

Passer l'adresse du pointeur dans la fonction

```
void reinitPointeur(int** p){
    *p = NULL;
}

int main(){
    int a = 1;
    int* p = &a;
    reinitPointeur(&p);
    printf("l'adresse de p est %p\n", p);
    return 0;
}
```

Structures

Déclaration d'une structure

```
struct article {
    int numero;
    double prix;
    int quantite;
};
```

Structures

Déclaration d'une structure

```
struct article {
    int numero;
    double prix;
    int quantite;
};
```

Déclaration et initialisation d'une variable du type structuré

```
struct article art1 = {357, 9.99, 85};
```

Opérateurs sur les structures

Accès aux champs d'une structure : opérateur `.` (point)

```
art1.numero
art1.prix
art1.quantite
```

Chaque champ se comporte comme une variable.

Opérateurs sur les structures

Accès aux champs d'une structure : opérateur . (point)

```
art1.numero
art1.prix
art1.quantite
```

Chaque champ se comporte comme une variable.

Affectation de deux variables de la même structure

```
struct article art1, art2;
...
art1 = art2;
```

Opérateurs sur les structures

Accès aux champs d'une structure : opérateur `.` (point)

```
art1.numero
art1.prix
art1.quantite
```

Chaque champ se comporte comme une variable.

Affectation de deux variables de la même structure

```
struct article art1, art2;
...
art1 = art2;
```

Pas de comparaisons possibles sur les structures.

Opérateurs sur les structures

Accès aux champs d'une structure : opérateur . (point)

```
art1.numero
art1.prix
art1.quantite
```

Chaque champ se comporte comme une variable.

Affectation de deux variables de la même structure

```
struct article art1, art2;
...
art1 = art2;
```

Pas de comparaisons possibles sur les structures.

La taille d'une structure est la somme de la taille de ses champs.

Exemple 1

```
struct tableau{
  int tab[3];
};
int main(void)
{
  int i;
  struct tableau t1 = {{5,9,3}};
  struct tableau t2 = t1;

  for(i=0; i<3; i++)
    printf("%d ", t2.tab[i]);
  printf("\n");

  return 0;
}
```

Exemple 2

```
struct entier{
  int e;
};

int main(void)
{
  int i;
  struct entier e1 = {2};
  struct entier e2 = {3};
  /*
  e1 < e2;
  ERREUR
  */

  return 0;
}
```

Définition de types

Utile pour faciliter l'écriture des programmes et améliorer la lisibilité :

```
typedef int entier;

typedef struct article article;

typedef struct{
  int numero;
  double prix;
  int quantite;
} article;
```

Tableaux de structures

Un tableau de structure se déclare par

```
struct article art[100];
```

Pour accéder aux champs de l'article `i`

```
art[i].numero
art[i].prix
art[i].quantite
```

Pointeur vers une structure

```
struct article art = {357, 9.99, 85}
struct article* ptr = &art;
```

Pointeur vers une structure

```
struct article art = {357, 9.99, 85}
struct article* ptr = &art;
```

Accès aux champs d'une structure pointée

```
// *ptr.prix
```

Incorrect car l'opérateur `.` est prioritaire sur `*`

Pointeur vers une structure

```
struct article art = {357, 9.99, 85}
struct article* ptr = &art;
```

Accès aux champs d'une structure pointée

```
// *ptr.prix
```

Incorrect car l'opérateur `.` est prioritaire sur `*`

```
(*ptr).prix
```

ou

```
ptr->prix
```

Exemple

```
struct liste{
  int val;
  struct liste* suiv;
};

int main(void)
{
  struct liste elt1 = {3, NULL};
  struct liste elt2 = {7, &elt1};
  struct liste elt3 = {1, &elt2};
  struct liste elt4 = {5, &elt3};

  printf("%d %d %d %d\n", elt4.val, elt4.suiv->val,
       elt4.suiv->suiv->val, elt4.suiv->suiv->suiv->val);

  return 0;
}
```

Qu'affiche le programme suivant ?

```
struct tableau{
  int tab[3];
};

void modif(int T[3], struct tableau t){
  T[0] = 1;
  t.tab[0] = 1;
}

int main(void)
{
  int T[3] = {0};
  struct tableau t = {{0}} ;
  modif(T,t);
  printf("%d %d\n", T[0], t.tab[0]);
  return 0;
}
```

Enumérations

```
enum {LUNDI, MARDI, MERCREDI, JEUDI,
                    VENDREDI, SAMEDI, DIMANCHE};
```

déclare les identificateurs `LUNDI`, `MARDI`, etc., comme des constantes entières de valeur 0, 1, etc.

Enumérations

```
enum {LUNDI, MARDI, MERCREDI, JEUDI,
                    VENDREDI, SAMEDI, DIMANCHE};
```

déclare les identificateurs `LUNDI`, `MARDI`, etc., comme des constantes entières de valeur 0, 1, etc.

Exemple d'utilisation :

```
switch(jour){
   case LUNDI : ...; break;
   case MARDI : ...; break;
   ...
}
```

Enumérations

```
enum {LUNDI, MARDI, MERCREDI, JEUDI,
                    VENDREDI, SAMEDI, DIMANCHE};
```

déclare les identificateurs `LUNDI`, `MARDI`, etc., comme des constantes entières de valeur 0, 1, etc.

Exemple d'utilisation :

```
switch(jour){
   case LUNDI : ...; break;
   case MARDI : ...; break;
   ...
}
```

Possibilité de définir un nouveau type

```
enum jour {LUNDI, MARDI, MERCREDI, JEUDI,
                         VENDREDI, SAMEDI, DIMANCHE};
enum jour j1, j2;
```

Unions

```
union nombre{
    int i;
    float f;
}
```

Comme une structure sauf qu'à chaque instant un seul champ a une valeur.

Unions

```
union nombre{
   int i;
   float f;
}
```

Comme une structure sauf qu'à chaque instant un seul champ a une valeur.

```
union nombre n;
```

La valeur de la variable `n` aura soit le type `int` soit le type `float`.

```
n.i = 10;
n.f = 3.14;
```