Langage C avancé : Séance 3

Pointeurs

Fonctions

Variables locales/globales

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Pointeurs

Définitions : zone mémoire, adresse

La mémoire de travail de l'ordinateur est constituée d'une suite d'unités de mémoire (octets ou bytes en anglais) numérotées (ou indicées) de 0 à N (où N est peut être très grand, actuellement souvent > 10^{10})

Une zone mémoire est un sous-ensemble d'unités de mémoire consécutives.

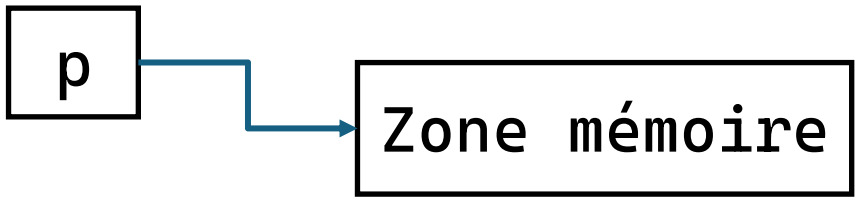
L'adresse de la zone mémoire est l'indice de la $1^{ère}$ unité mémoire de la zone mémoire, c'est un nombre entier.

C définit un type d'entier spécifique, size_t, capable de représenter correctement une adresse.

Pointeurs

- On a déjà vu la notion de pointeurs pour manipuler des vecteurs et des chaines de caractères
- Ce ne sont pas les seules utilisations possibles des pointeurs

- Définition générale : un pointeur est une variable, cette variable occupe une zone mémoire et sa valeur est l'adresse d'une autre zone mémoire
- On peut représenter un pointeur p comme ci-dessous:



Pointeurs générique et typé

- Un pointeur peut contenir l'adresse d'une zone mémoire dont on ne connait pas la structure : on parle d'un pointeur générique

```
void * v;
```

- Un pointeur peut contenir l'adresse d'une zone mémoire qui ne peut contenir que des valeurs d'un **type** choisi par le programmeur (int, double, char, struct, pointeur, ..)

```
int * p;
double * q;
```

Copies entre pointeurs générique et typé

- Il est possible de copier la valeur d'un pointeur typé ou d'un pointeur générique dans un autre pointeur générique:

```
int * p = …;
void * q = p;
```

- Par contre, on ne peut pas copier la valeur d'un pointeur générique dans un pointeur typé, sauf si on change le type du pointeur générique (et le compilateur ne fait aucune vérification):

```
void * q = …;
double * p = q; /* erreur */
double * r = (double *) q;
```

Copies entre pointeurs générique et typé

- De même, on ne peut pas copier la valeur d'un pointeur typé dans un pointeur d'un type différent:

```
int * q = …;
double * p = q; /* erreur */
```

Il est toujours possible de forcer la copie en changeant le type d'un des pointeurs, mais c'est dangereux (et rarement une bonne idée)

```
int * q = …;
double * p = (double *) q;
```

Exemples d'initialisation d'un pointeur générique

1. Un pointeur générique peut être initialisé avec une adresse spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse utilisable:

```
void * v;
v = NULL;
```

ou

```
void * v = NULL;
```

On pourra écrire, pour tester si un pointeur est utilisable:

```
if (v == NULL)
  printf("erreur v n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur générique (2)

2. Un pointeur générique peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc

```
void * v;
v = malloc(n);
```

ou

```
void * v = malloc(n);
```

Ici n représente la taille de la zone mémoire en octets.

La fonction malloc donne comme résultat un pointeur générique (void *) qui est copié directement dans v.

Exemples d’initialisation d’un pointeur générique (3)

Le résultat de malloc est NULL si la zone mémoire disponible n’est pas suffisante (n est trop grand) ou si n ≤ 0.

Une règle de bonne programmation est de vérifier que ce n’est pas le cas:

```
if (v == NULL)
  printf("erreur la mémoire demandée n’a pas pu"
         "être réservée\n");
```

Exemples d'initialisation d'un pointeur générique (4)

3. Un pointeur générique peut être initialisé avec l'adresse d'une variable existante (de type quelconque):

```
int k;
void * v;
v = &k;
```

ou

```
int k;
void * v = &k;
```

&k représente l'adresse de la variable k.

On ne doit pas tester si le pointeur est NULL puisque toute variable existante a forcément une adresse valable.

Exemples d'initialisation d'un pointeur typé

1. Un pointeur typé peut être initialisé avec une valeur spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse:

```
double * d;
d = NULL;
```

ou

```
double * d = NULL;
```

On pourra donc tester si un pointeur typé est utilisable ou non dans la suite du code:

```
if (d == NULL)
  printf("erreur d n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur typé (2)

2. Un pointeur typé peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc:

```
int * i;
i = (int *) malloc(n * sizeof(int));
```

ou

```
int * i = (int *) malloc(n * sizeof(int));
```

Dans ce cas, le pointeur i contient l'adresse d'une suite de n éléments du type choisi (ici int).

Il faut donc fournir à malloc la taille cette zone mémoire: n x la taille d'une valeur du type choisi (en bleu dans les exemples ci-dessus).

Exemples d’initialisation d’un pointeur typé (3)

La fonction malloc a pour résultat un pointeur générique void * et il faut changer son type pour initialiser le pointeur typé (ici int *):

```
int * i;
i = (int *) malloc(n * sizeof(int));
```

C’est un des seuls changements de type de pointeurs qu’il est « raisonnable » de s’autoriser.

Ne pas oublier de tester si le pointeur initialisé par malloc n’est pas NULL.

Exemples d'initialisation d'un pointeur typé (4)

3. Un pointeur typé peut être initialisé avec l'adresse d'une variable existante du même type :

```
int k;
int * v;
v = &k;
```

ou

```
int k;
int * v = &k;
```

&k représente l'adresse de la variable k.

4. On peut aussi définir et initialiser une zone mémoire de plusieurs valeurs du même type, et en même temps, définir un pointeur vers le début de cette zone:

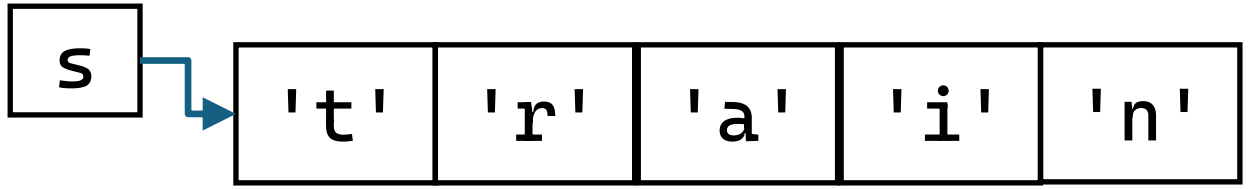
```
double * v = {1.2, 3.4, 4.5};
```

Remarque: traitement particulier des caractères

Notez la différence entre :

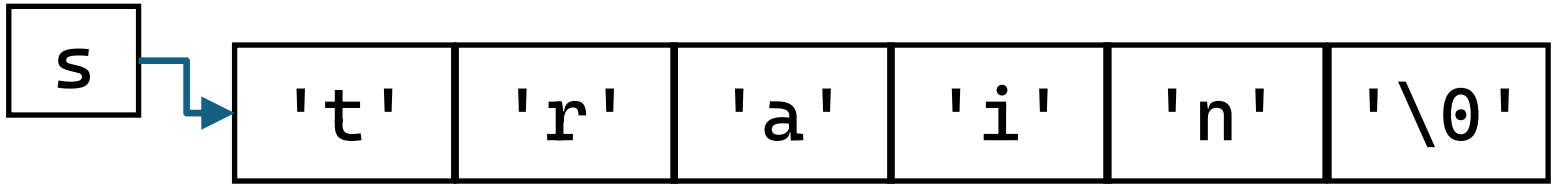
- Un vecteur de caractères (une suite de caractères quelconques)

```
char * s = {'t', 'r', 'a', 'i', 'n'};
```



- Une chaine de caractères (une suite de caractères dont le dernier est le caractère spécial '\0')

```
char * s = "train";
```



Accès à la valeur pointée par le pointeur

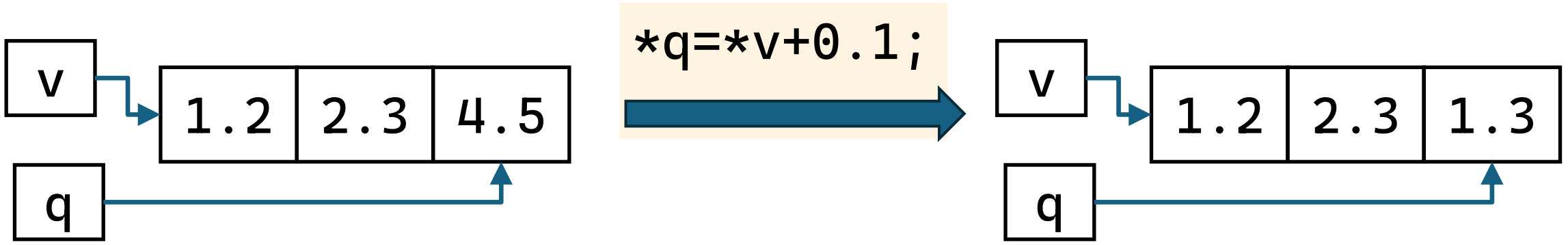
- Pour utiliser ou modifier la valeur contenue dans la mémoire à l'adresse contenue dans le pointeur, on utilise l'opérateur d'indirection *.

Par exemple

```
double v[] = {1.2, 2.3, 4.5};
double *q = v + 2;
*q = *v + 0.1;
```

L'effet est le même que si on avait écrit

```
v[2] = v[0] + 0.1
```



Caractère * dans le code source

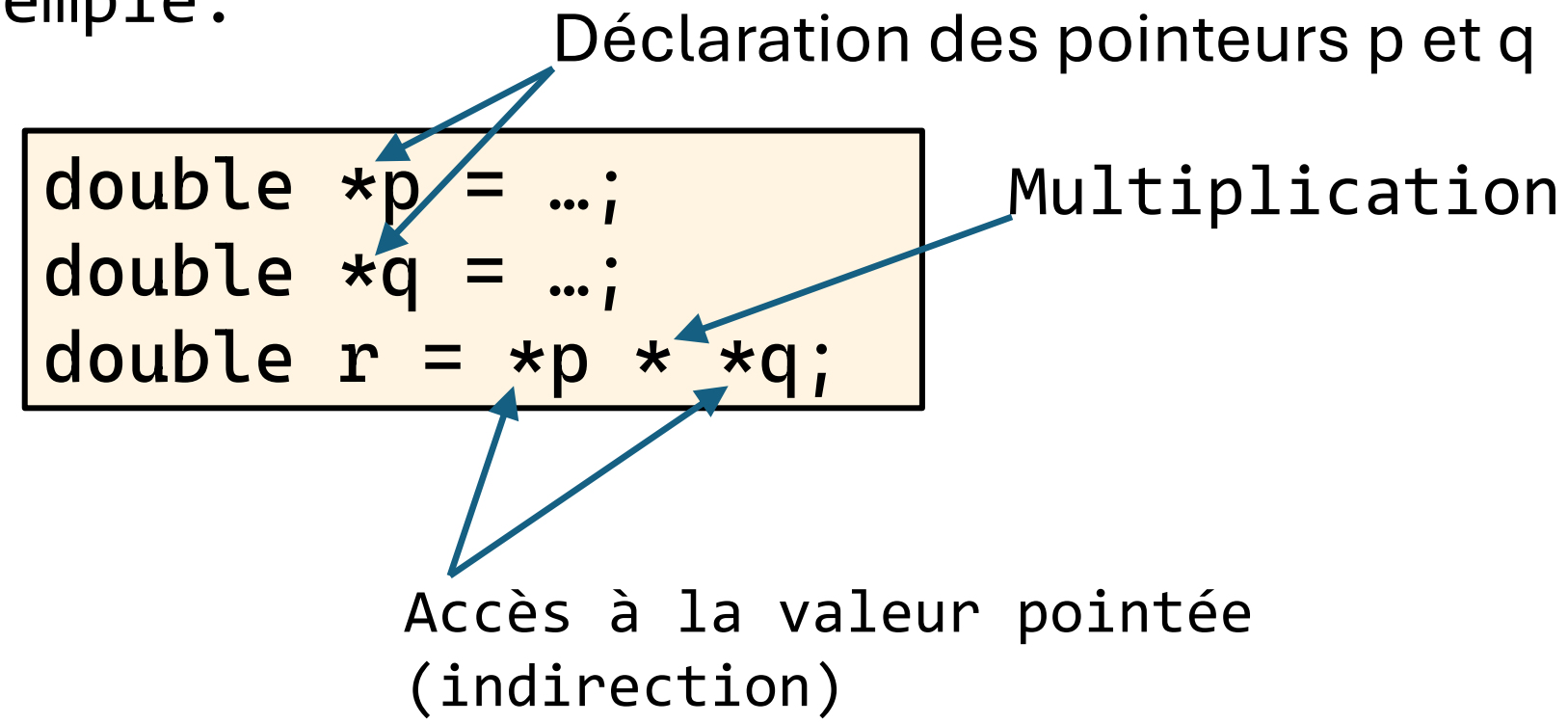
Noter que, en C, le caractère * a plusieurs significations, notamment:

- Opérateur de multiplication entre des nombres
- Bornes d’un commentaire /* … */
- Définition d’un pointeur (int *p = …)
- Accès à la valeur pointée par un pointeur ou indirection (*p = 3)

Un caractère * dans une instruction à l’une des significations ci-dessus en fonction du contexte.

Caractère * dans le code source (2)

Par exemple:

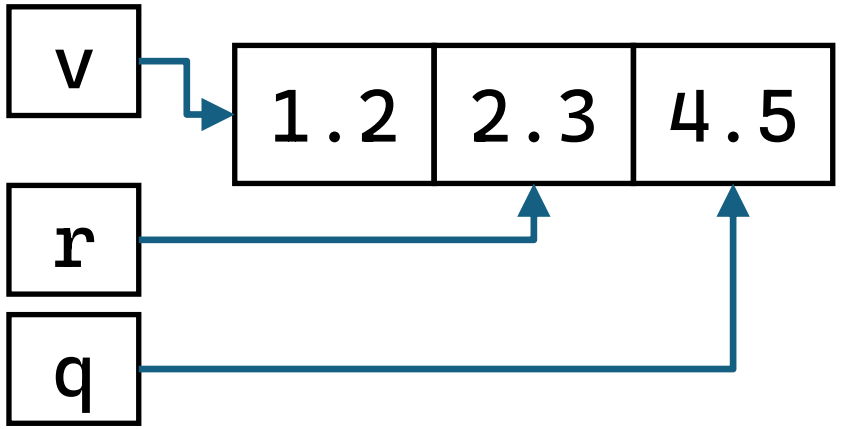


En général, le compilateur déduit la signification du contexte. Mais il peut arriver, c'est rare, que l'instruction soit ambiguë. Dans ce cas, il faut l'aider, en particulier en ajoutant des (): `double r = (*p)*(*q);`

Operations avec des pointeurs

- Un pointeur contient une adresse qui est un type particulier d'entier. On peut donc imaginer de faire des opérations arithmétiques ou autres sur les pointeurs.
- Seules certaines opérations ont un sens:
  - Addition d'un entier positif à un pointeur, le résultat est un pointeur, on dit qu'on décale le pointeur vers une adresse supérieure.
  - Soustraction d'un entier positif à une adresse, on dit qu'on décale un pointeur vers une adresse inférieure

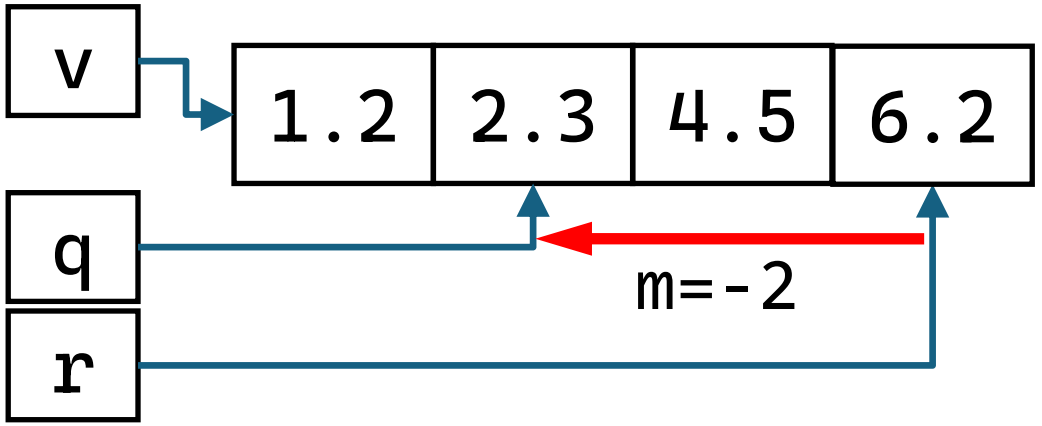
```
double v[] = {1.2, 2.3, 4.5};
double *q = v + 2;
Double *r = q – 1;
```



Operations avec des pointeurs (2)

- Différence entre 2 pointeurs

```
double v[] = {1.2, 2.3, 4.5, 6.2};
double *q = &v[1]; /* ou q = v+1 */
double *r = &v[3]; /* ou r = v+3 */
int m = r - q;
```



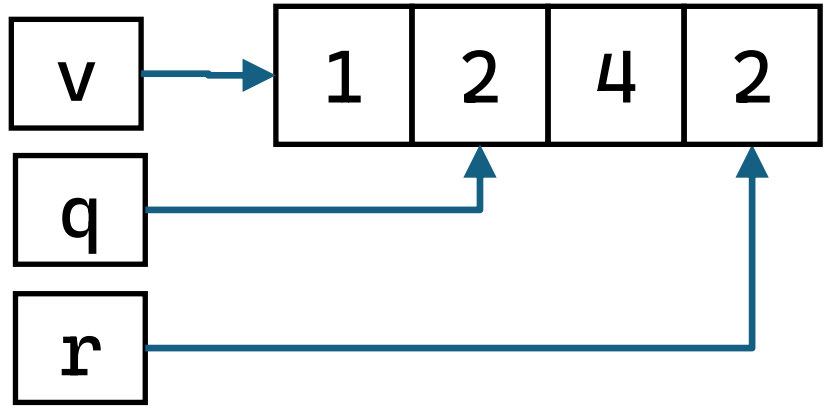
- Par contre, les opérations d'addition de 2 pointeurs, de multiplication et de division entre des pointeurs et des entiers n'ont pas de sens

Opérations avec des pointeurs (3)

- Comparaison de deux pointeurs

```
int v[] = {1, 2, 4, 2};
int *q = &v[1];
int *r = &v[3];
r == q; /* est faux */
*r == *q ; /* est vrai */
r > q;  /* est vrai */
```

La comparaison de pointeurs consiste à comparer les adresses contenues dans les pointeurs



Affichage des pointeurs / adresses

Un pointeur contient une adresse (un entier de type size_t).
La fonction printf peut l'afficher comme un entier, mais utilise aussi un format spécifique (%p) :

```
double x ;
double *p = &x ; /* p contient l'adresse de x */
printf("adresse de x = %ld\n", p) ;
printf("adresse de x = %p\n", p) ;
```

Ces lignes compilent mais le compilateur affiche des messages d'avertissement.
Ajouter ce qui manque pour ne plus avoir de messages du compilateur

Opérations avec des pointeurs (4)

Il vaut mieux ne pas comparer des pointeurs sur des types différents

Certains compilateurs l’acceptent (et affichent peut-être un message d’avertissement), d’autres non

Exercice: somme des composantes d'un vecteur

Le code ci-dessous initialise un vecteur de n doubles et calcule la somme des ses composantes.

Modifiez-le en utilisant uniquement des pointeurs pour accéder aux composantes du vecteur.

```
for (i=0; i<n; i++)
  v[i] = sin(i*1.0);

somme = v[0];

for (i=1; i<n; i++)
  somme += v[i];
```

Fonctions
Variables locales et globales

Fonction

Une fonction possède :

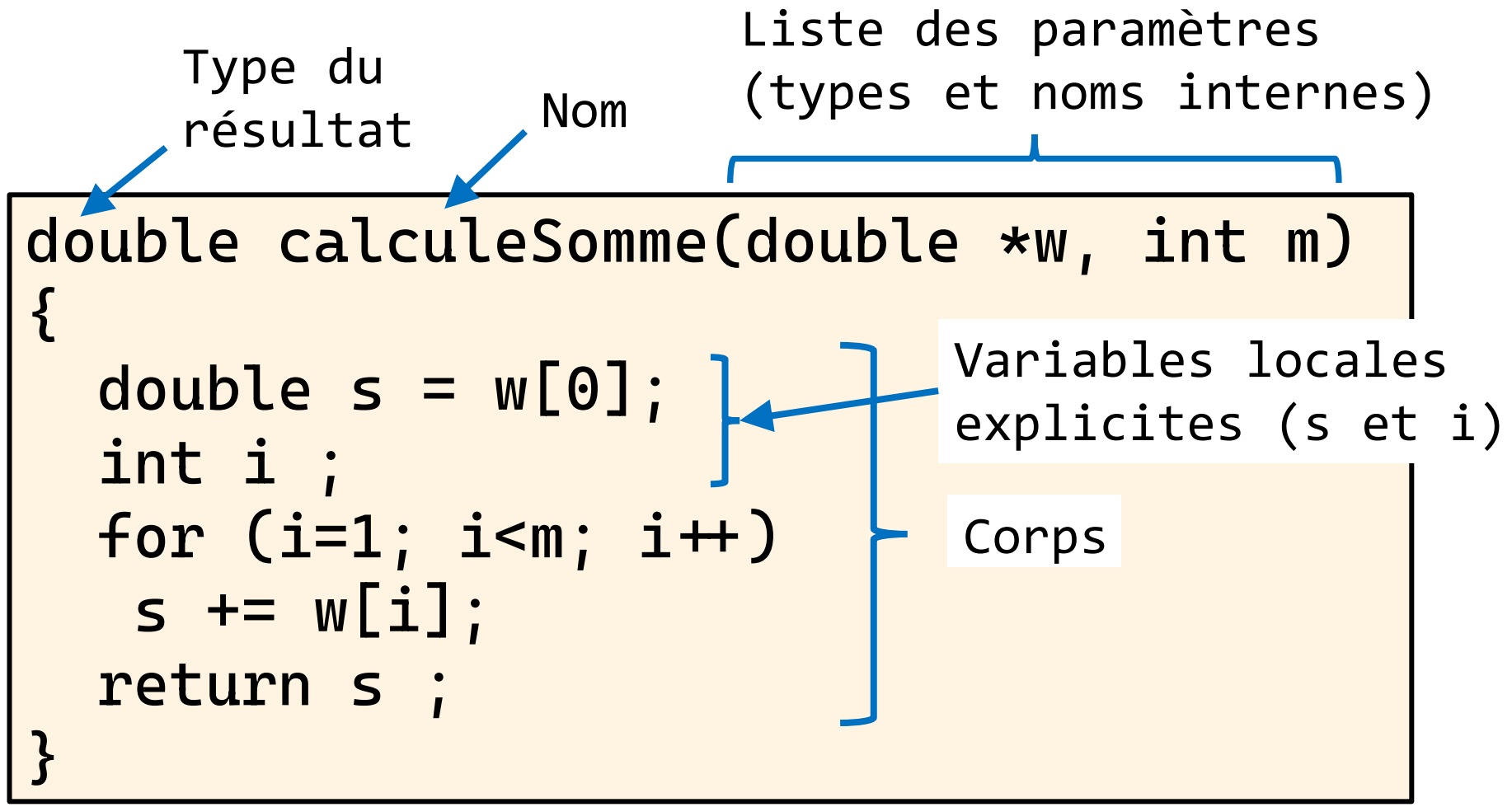
- un nom,
- un ensemble de paramètres (valeurs reçues du contexte extérieur de la fonction, en spécifiant leur type)
- zéro ou un seul résultat (une unique valeur),
- un corps (un ensemble d'instructions à exécuter quand on utilise la fonction)

Une fonction peut définir des variables supplémentaires (variables locales explicites) qui n'existeront que pendant l'exécution de cette fonction.

La fonction définit pour chaque paramètre, une copie locale (variable local implicite) et travaille avec la copie locale (détruite à la fin de la fonction).

Exemple de fonction

On peut mettre les instructions qui calculent la somme des composantes d'un vecteur (exercice précédent) dans une fonction:




```
double calculeSomme(double *w, int m)
{
  double s = w[0];
  int i ;
  for (i=1; i<m; i++)
   s += w[i];
  return s ;
}
```

Déclaration et définition d'une fonction

- La définition d'une fonction dans le code source est l'écriture complète de la fonction : nom, paramètres, type du résultat et corps
- La déclaration d'une fonction est l'ensemble des information sans le corps de la fonction : nom paramètres, type de retour

La déclaration d'une fonction est souvent appelée le prototype de la fonction ou la signature de la fonction.

Exemple de fonction (2)

L'utilisation de cette fonction, par exemple dans la fonction main (programme principal), s'écrit

```
int main()
{
  int n =  ...  ;
  double * v =  ... ;
  double somme ;
  somme = calculSomme(v, n) ;
}
```

Le nombre et les types de paramètres doit correspondre à la définition de la fonction page précédente.
La type de résultat de la fonction doit être compatible (identique ou convertible) avec la variable où on range le résultat

Exemple de fonction (3)

Si on ne s'intéresse pas au résultat de la fonction, on ne le range pas dans une variable:

```
calculSomme(v, n) ;
```

Certains compilateur affichent un avertissement, dans ce cas, on peut écrire:

```
(void) calculSomme(v, n) ;
```

Si une fonction ne renvoie pas de résultat, on indique dans la définition que le « résultat » est de type void:

```
void calculSomme(double *v, int n)
{  …
}
```

Opérations effectuées à l'exécution de la fonction

1. Un premier ensemble de variables locales sont automatiquement créées qui contiennent une copie des valeurs passées à la fonction (paramètres de la fonction).
2. Les autres variables locales (explicitement définies dans la fonction) sont créées.
3. Le corps de la fonction est exécuté.
4. Si la fonction a un résultat, la dernière instruction exécutée doit être « return valeur » qui range cette valeur dans le résultat de la fonction.
5. Les variables locales sont détruites

Fonctions ayant une forme particulière

Certaines fonctions ont un mode d’exécution particulier :

- Fonctions qui ne prennent pas de paramètres (qui n’ont pas besoin de données) pour s’exécuter
- Fonctions qui ne produisent pas de résultat
- Fonctions qui modifient des variables globales ou des valeurs indirectement à travers les paramètres

Fonctions ayant une forme particulière (2)

Fonctions qui ne prennent pas de paramètres (qui n'ont pas besoin de données du contexte) pour s'exécuter.

La fonction « Question », ci-dessous, demande à l'utilisateur de rentrer un nombre entre 1 et 10 :

```
int Question()
{
  int r;
  do {
    printf("Entrer un entier entre 1 et 10 ");
    scanf("%d", &r);
  }
  while ((r > 0) && (r < 11));
  return r:
}
```

Fonctions ayant une forme particulière (3)

Fonctions qui ne produisent pas de résultat.

Par exemple, la fonction ci-dessous affiche un vecteur:

```
void AfficheVecteur(double *v, int n, char *nom)
{
  int i;
  printf("%s\n", nom);
  for (i=0; i<n; i++)
    printf("%3d: %g\n", i, v[i]);
  printf ("\n");
}
```

Dans ce cas, il n’y a pas d’instruction « return valeur » (ou alors « return; »)

Fonctions ayant une forme particulière (4)

Fonctions qui produisent, en plus de la valeur de retour, d’autres résultats.

Les cas les plus courants sont:

- Une fonction modifie une variable globale
- Une fonction reçoit un pointeur en paramètre et modifie la valeur pointée par le pointeur

Ce sont des cas particuliers de ce qu’on appelle en informatique un « effet de bord » (side effect en anglais)

Fonctions ayant une forme particulière (5)

Exemple d’une fonction qui modifie une variable globale:

```
int nAppels; /* variable globale */

void f(double x) {
  nAppels += 1;
  printf("x = %g\n" , x);
}

int main() {
  nAppels = 0;
  f(1.0); f(2.0);
  printf("La fonction f a été appelée %d fois\n", nAppels);
  return 0;
}
```

Fonctions ayant une forme particulière (6)

La fonction ci-dessous modifie les composantes d'un vecteur passé en paramètre:

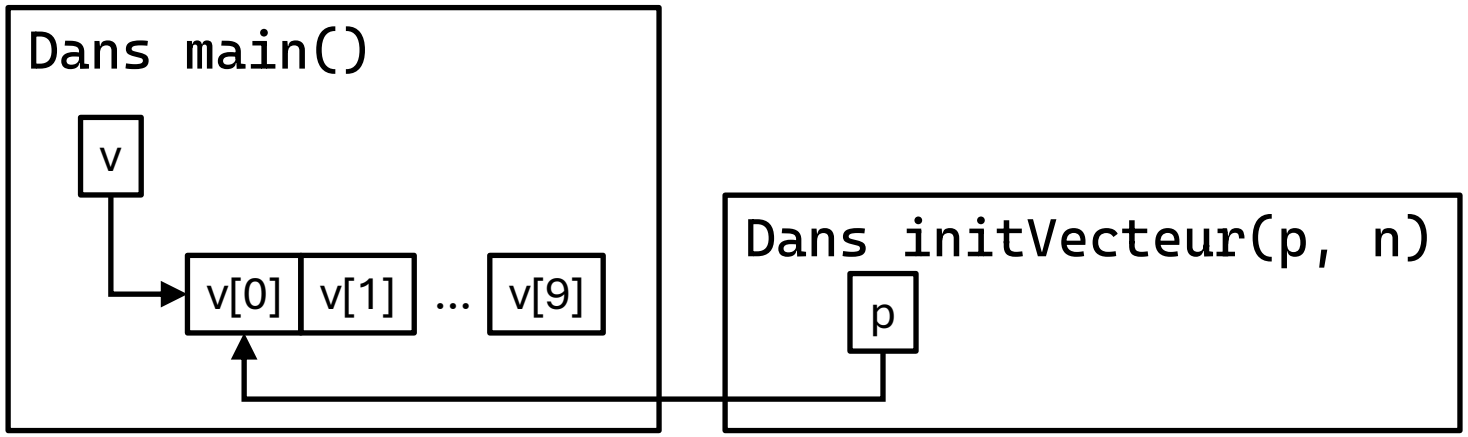
```
void initVecteur(double * p, int n) {
  int i;
  for (i=0; i<n; i++) p[i] = 0.0;
}

int main() {
  double *v = (double *) malloc(10*sizeof(10));
  initVecteur(v, 10);
  free(v);
  return 0;
}
```

Fonctions ayant une forme particulière (7)

Quand on rentre dans la fonction initVecteur:

- La fonction crée un pointeur local p et y copie l'adresse du vecteur v passé en paramètre dans le programme principal
- La fonction utilise le pointeur p pour utiliser ou modifier les composantes du vecteur
- Le pointeur local p est détruit automatiquement à la fin de la fonction



Code C

Un code C est constitué :

- d’un ensemble de fonctions, chaque fonction définit des variables qui n’existent que dans cette fonction, appelées variables locales

- de variables définies en dehors des fonctions, appelées variables globales ; ces variables existent pendant toute la durée de l’exécution

Code C (2)

Quand le compilateur compile un code C :

- Le compilateur lit le code source de la première ligne à la dernière, une et une seule fois
- Quand une variable ou une fonction sont utilisées, if faut que cette variable soit déclarées/définies avant leur utilisation

Exercice :

```c
#include
<stdio.h>

void g(int b) {
  a = a * b;
}

int a;

void f(int c) {
  a = a + c;
}

int main() {
  a = 0;
  f(2); g(5);
  return 0;
}
```

Le code contient 3 fonctions (main, f, g), une variable globale (a) et deux variables locales implicites (b et c, paramètres dans f et g).

Il y a une erreur dans ce code, corrigez cette erreur.