UNIVERSITÉ PARIS DESCARTES

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

Cours n° 7

Les fondamentaux de la programmation impérative

Révisions

1. PROJET

Rappel des deadlines du projet

Date de remise des dossiers de développement
Le lundi 6 novembre au secrétariat (retard pénalisé)

Date des recettes
La semaine du 13 novembre
1ère semaine du module SDA-Structures de Données et Algorithmes en séance de SDA3 (2ème TP de SDA)

1. REVISIONS

Le domaine des révisions

Rappel sur le type pointeur

Les fonctions

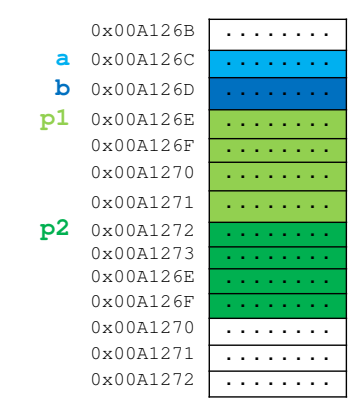
- Interface et déclaration/prototypage
  Les différents modes de passage de paramètre
- Le domaine de définition de la fonction
- Le corps d'une fonction
- L'appel des fonctions
- La spécificité du passage d'un paramètre tableau

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L10 Réservation mémoire
des paramètres formels de la fonction (`main`)
et des variables locales à la fonction

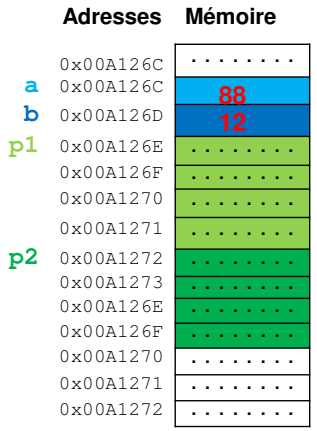
```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
                  *p1, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
              *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



Les pointeurs – Déclaration, initialisation, indirection

L13 Initialisation des variables `a` et `b`

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1)
L17   printf("p2=%x *p2=%d\n ", p2, *p2)
                  *p1, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
              *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



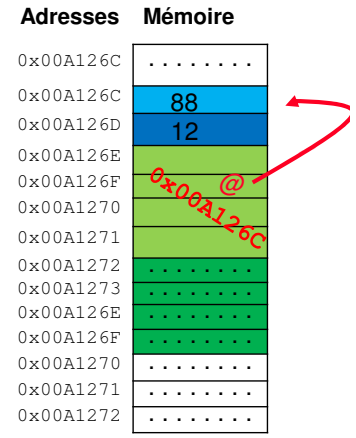
Les pointeurs – Déclaration, initialisation, indirection

L14 Initialisation du pointeur `p1`

Déréférencement
2 noms pour une même zone mémoire

Référencement

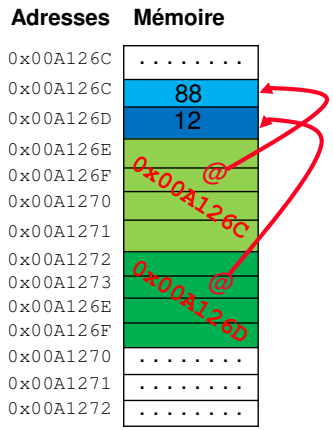
```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
              *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables `a` et `b`

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1=*p2+10;
L19   *p2+=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
              *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```

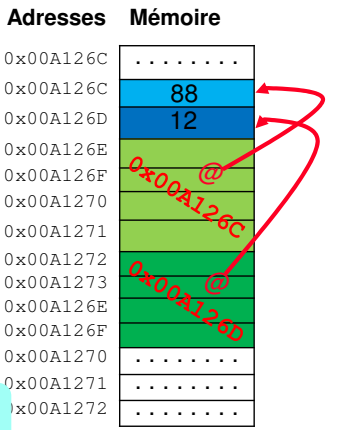


Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables `a` et `b`

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
              *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```

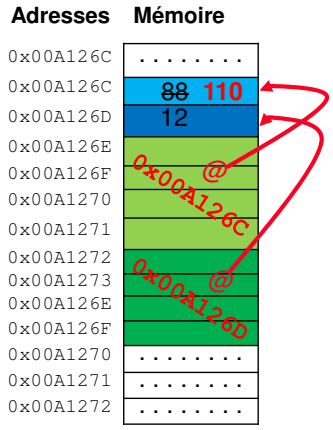
L16 **p1=0x00A126C *p1=88**
L17 **p2=0x00A126D *p2=12**



Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

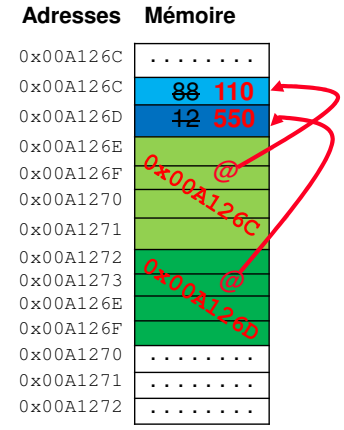
```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
                *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

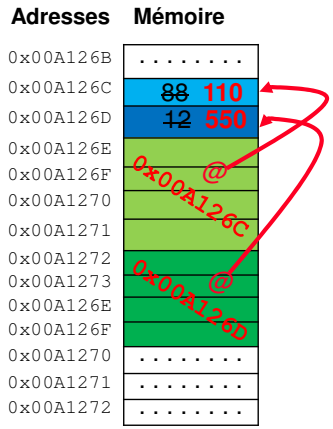
```
L10 int main() {
L11   int a, b;
L12   int *p1, *p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
                *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

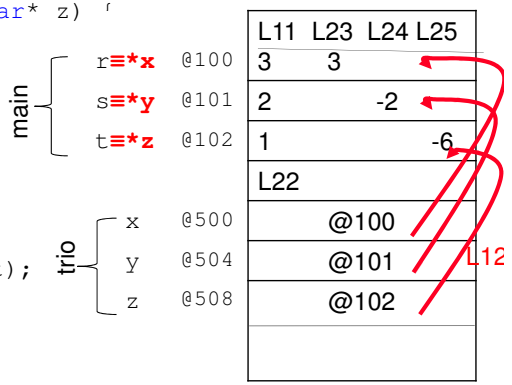
```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
                *p1, *p2);
L21   system("pause");
L22   return 0;
      }
```



L20 ***p1=110 *p2=550**

Les pointeurs – paramètres de fonctions

```
L22  void trio(char* x, char* y, char* z) {
L23     *x = *y + *z;
L24     *y = *z - *x;
L25     *z = *x * *y;
     }

L10  int main() {
L11     char r=3, s=2, t=1;
L12     trio(&r, &s, &t);
L13     printf("%d %d %d\n", r, s, t);
L14     system("pause"); return 0;
     }
```



L12 du main : la copie des adresses respectives de r, s et t dans x, y et z.
permet la modification des paramètres d'appel (effectifs) r, s et t.
En effet, **r≡*x, s≡*y et t≡*z**

Affichage : **3 -2 -6**

Prototypage – Les bonnes questions à se poser (1/2)

(1) Quel nom pour la fonction ?

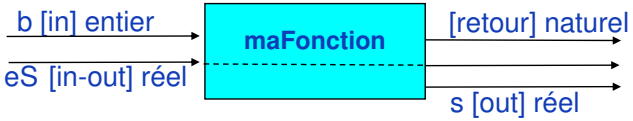
(2) De quelles données a-t-on besoin pour résoudre le problème ?
Paramètres d'entrée [in]
- utilisés pour le traitement,
- caractéristique : dans le corps de la fonction, en partie droite d'affectation et/ou condition

(3) Quels sont les modes de transmission des résultats calculés par la fonction
(a) transmis par retour [return]
- caractéristique : unique résultat, renvoyé par un return

(b) transmis par paramètre de sortie [out]
- caractéristique : mode de passage par pointeur, utilisé dans le corps de la fonction en partie gauche d'affectation

(c) transmis par paramètre d'entrée-sortie [in-out]
- caractéristique : utilisés pour le traitement et modifié dans le corps en partie droite et droite d'affectation et/ou condition)

Prototypage – L'algorithme du prototypage (2/2)



(1) Y a-t-il un retour de la fonction ?
Oui : écrire le type du retour
Non : écrire void

(2) Ecrire le nom de la fonction

(3) Ecrire tous les autres paramètres formels entre parenthèses ([in], [in-out] et [out])

Pour tout paramètre [in] : le paramètre est passé par valeur, écrire le type et le nom du paramètre

Pour tout paramètre [out] ou [in-out] : le paramètre est passé par pointeur, écrire le type suivi de * et le nom du paramètre

Résultat de l'algorithme du prototypage :

```
unsigned int maFonction(int b, float* eS, float* s);
```


Prototypage – avec paramètre de retour

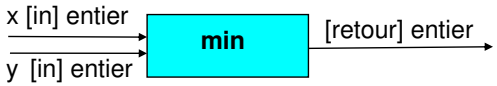
Le minimum de deux entiers


```
int min(int x, int y){
  if (x<y) return x;
  else return y;
}

int main() {
  int x=7, y=3, leMin;
  leMin=min(x, y);
(1) printf("min(%d, %d)=%d\n", x, y, LeMin);
(2) printf("min(%d, %d)=%d\n", x, y, min(x, y));
  system("pause"); return 0;
}
```

```
min(7, 3)=3
min(7, 3)=3
```


Prototypage – avec paramètre de retour et documentation

Le minimum de deux entiers

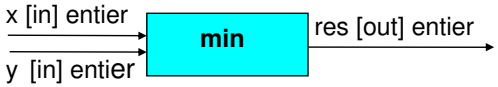



```
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * [retour] le minimum des deux entiers
 */
int min(int x, int y){
  if (x<y) return x;
  else return y;
}

int main() {
  int x=7, y=3, leMin;
  leMin=min(x, y);
  printf("min(%d, %d)=%d\n", x, y, LeMin);
  printf("min(%d, %d)=%d\n", x, y, min(x, y));
  system("pause"); return 0;
}
```

```
min(7, 3)=3
min(7, 3)=3
```

Prototypage – avec paramètre de sortie (1/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

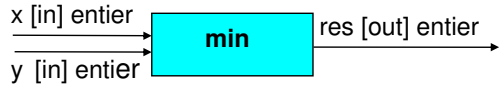
```
void min(int x, int y, int* res){
  if (x<y)
      *res=x;
  else
      *res=y;
}

int main() {
  int x=7, y=3, r;
  min(x, y, &r);
  printf("min(%d, %d)=%d\n", x, y, r);
  system("pause"); return 0;
}
```

`min(7, 3)=3`

4. REVISIONS

Prototypage – avec paramètre de sortie (1/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

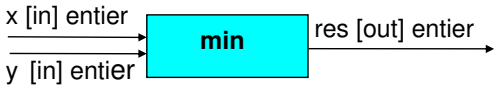
```
void min(int x, int y, int* res){
  if (x<y)
      *res=x;
  else
      *res=y;
}

int main() {
  int x=7, y=3, r;
  min(x, y, &r);
  printf("min(%d, %d)=%d\n", x, y, r);
  system("pause"); return 0;
}
```

`min(7, 3)=3`

4. REVISIONS

Prototypage – avec paramètre de sortie et documentation (2/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

```
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers
 */
void min(int x, int y, int* res){
  if (x<y)
      *res=x;
  else
      *res=y;
}
int main() {
  int x=7, y=3, res;
  min(x, y, &res);
  printf("min(%d, %d)=%d\n", x, y, res);
  system("pause"); return 0;
}
```

`min(7, 3)=3`

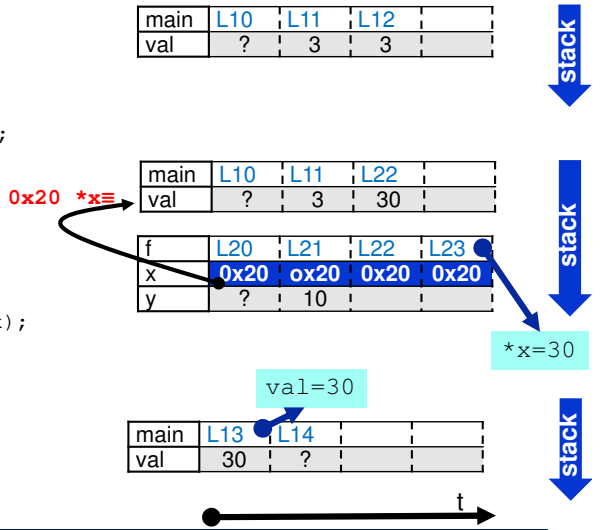
? état de la mémoire

4. REVISIONS

Techniquement – Fonction avec paramètre de sortie

(1) Appel du `main()`

```
L10  int main() {
L11   int val=3;
L12   f(&val);
L13   printf("val=%d\n", val);
L14   system("pause"); return 0;
     }

L20  void f(int* x){
L21         int y=10;
L22         *x=(*x)* y;
L23         printf("*x=%d\n",*x);
     }
```



(2) L12 : appel de `f()`

***x** est l'autre nom logique (dans la fonction `f`) de la variable de l'appelant (`val`) implantée en `0x20`

Documentation de source

Le cartouche du programme

```
/* main.c
 * Auteur : Marie-José Caraty
 * Date de création : 22/10/2017
 */
void min(int x, int y, int* res); // void min(int, int, int*);

int main() {
  int x=7, y=3, res;
  min(x, y, &res);
  printf("min(%d, %d)=%d\n", x, y, res);
  system("pause"); return 0;
}

/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers
 */
void min(int x, int y, int* res){
  if (x<y)
     *res=x;
  else
     *res=y;
}
```

Documentation des fonctions
- Rôle
- Paramètre(s) d'entrée
- Paramètre(s) de sortie
- Paramètre()s d'entrée-sortie
- Paramètre de retour (en dernier s'il existe)

`min(7, 3)=3`

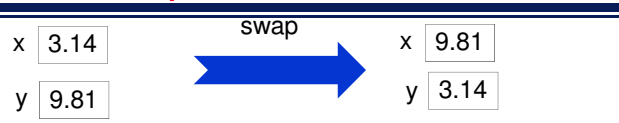
Précondition des fonctions

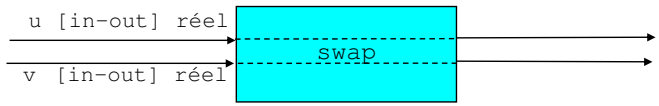
Déclarez une fonction (`f`) qui au nombre réel `x` donne pour résultat `1/(2-x)`
Documentez et codez la fonction f
Vous veillerez aux éventuelles préconditions de la fonction

```
/* Calcul de la fonction 1/(2-x)
 * x [in] l'argument
 * [retour] la valeur de la fonction
 * precondition : x différent de 2
 */
double f(double x) {
  assert(x!=2);
  return 1./(2-x);
}
```

Tout paramètre d'entrée peut faire l'objet d'une précondition

Cas du `swap` – Echange du contenu de deux variables paramètres d'entrée-sortie






```
/* Echange le contenu de deux variables
 * u [in-out] le premier argument
 * v [in-out] le deuxième argument
 */
void swap(float* u, float* v) {
    float aux = *u;
    *u = *v;
    *v =  aux;
}

int main() {
   float x = 3.14, y = 9.81;
   printf("x=%.2f  y=%.2f\n", x, y);
   swap(&x, &y);
   printf("x=%.2f  y=%.2f\n", x, y);
}
```

```
x=3.14 y=9.81
x=9.81 y=3.14
```


Tableaux statiques – Déclaration de tableau (1/2)

Un tableau est une collection d'objets tous du même type

Déclaration de tableau statique

`<type> <identificateur> [constante_entière]*`

autant de […] que de dimensions

```
char tab[5]; // un tableau de 5 entiers codés sur 1 octet
             // Attention : utilisation d'un nombre magique
```

Bonne Pratique **Utiliser une constante pour définir la taille du tableau**

```
const int N=5; // Constantes non autorisées en C90
```

On utilisera une constante d'énumération de valeur 5

```
enum {N=5};
```

ou encore

```
#define N 5
short tab[N]={3, 5, 7, 2, 6}; //Initialisation par liste
```

3	5	7	2	6
0	1	2	3	4

Index du tableau : {0, .., 4}

accès au 4ème élément par `tab[3]`

Tableaux statiques – Déclaration et initialisation explicite (2/2)

On peut également initialiser un tableau par une liste explicite sans préciser sa taille

```
int tab[]={3, 5, 7, 2, 6};
```

Dans ce cas, on peut calculer sa taille (nombre d'éléments) par :

```
unsigned int taille = sizeof(tab) /sizeof(int);
```


Tableaux statiques – Déclaration

Un tableau est une collection d'objets tous du même type

Déclaration de tableau statique

```
<type> <identificateur> [constante_entière]*
```

autant de […] que de dimensions

```
char tab[5]; // un tableau de 5 entiers codés sur 1 octet
             // Attention : utilisation d'un nombre magique
```

Bonne Pratique **Utiliser une constante pour définir la taille du tableau**

```
const N=5; // Constantes non autorisées en C90
```

On utilisera une variable d'énumération de valeur 5

```
enum {N=5}; // ou #define N 5
short tab[N]={3, 5, 7, 2, 6}; //Initialisation explicite
```

tab	3	5	7	2	6
	0	1	2	3	4

Index du tableau : {0, .., 4}

accès au 4ème élément par `tab[3]`

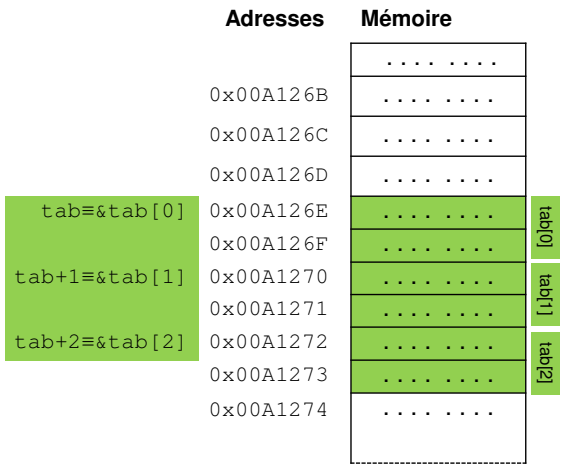
Schématiquement en mémoire

A la déclaration d'un tableau de 3 entiers de type `short`

```
short tab[3];
```

L'identificateur du tableau (`tab`) est l'adresse du premier élément du tableau alloué en mémoire
`tab` vaut `0x00A126E`

Zone contigüe en mémoire des 3 éléments du tableau (arithmétique des pointeurs)



.... état (aléatoire) de la mémoire

Passage de tableau dans une fonction

Rappel

`tab` est l'adresse du premier élément du tableau

```
int* tab; // n'intègre aucune information sur la taille de tab
```

Pour traiter un tableau dans une fonction

(1) passer la variable tableau

(2) passer sa taille

Ou encore,

encapsuler le tableau et sa taille dans une structure `Tableau`

```
enum {N=20};

typedef struct {
   int tab[N];
   unsigned int taille;
} Tableau;
```

Passage de tableau dans une fonction – Passage du tableau et de sa taille

Fonction d'initialisation d'un tableau de 5 entiers (de 1 à 5)
Fonction d'affichage du tableau

```
#include <stdio.h>
#include <stdlib.h>

void initialiser(int* tab, unsigned int size) {
    int i;
    for (i=0; i<size; ++i)
        tab[i]=i+1;
}
void afficher(const int tab[], unsigned int size) {
                // ou (const int* tab, unsigned int size)
    int i;
    for (i=0; i<size; ++i)
        printf("tab[%d]=%d ", i, tab[i]);
    printf("\n");
}
int main() {
    enum {TAILLE=5};
    int t[TAILLE];
    initialiser(t, TAILLE);
    afficher(t, TAILLE);
    system("pause"); return 0;
}
```

Dans le cas d'un paramètre d'entrée (type tableau ou structuré), on utilisera le modifieur `const` pour prévenir, par erreur du compilateur, toute modification dans le corps de la fonction

```
tab[0]=1 tab[1]=2 tab[2]=3 tab[3]=4 tab[4]=5
```


Passage de tableau dans une fonction – Encapsulation de la taille dans un type structuré

Taille en octet de t : 84
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

```
#include <stdio.h>
#include <stdlib.h>

enum {N=20};

typedef struct {
   int tab[N];
   unsigned int taille;
} Tableau;

afficher(const Tableau* t) {
   unsigned int i;
   for (i=0; i<t->taille; ++i)
      printf(" %d", t->tab[i]);
}

initialiser(Tableau* t) {
   unsigned int i;
   for (i=0; i<t->taille; ++i)
      t->tab[i]=i+1;
}

int main () {
   Tableau t;
   printf("Taille en octet de t : %d\n", sizeof(t));
   t.taille=N;
   initialiser(&t);
   afficher (&t);
   system("pause"); return 0;
}
```


Passage de variable de type structuré dans une fonction

Optimisation

Pour tout paramètre formel d'entrée (mode `[in]`) de type structuré de taille supérieure à 4 octets,
on **évitera** à l'appel de la fonction, la copie du paramètre effectif dans le paramètre formel en passant le paramètre par **pointeur**.

Sécurisation du paramètre d'entrée

On sécurisera le tableau passé en mode `[in]` (paramètre d'entrée) par le modifieur **`const`** pour empêcher toute modification par le corps de la fonction. Tout accès en écriture dans la fonction provoquera alors une erreur à la compilation.

```
void afficher (const Tableau* t) ;
void afficher(const int t[], unsigned int size);
void afficher(const int* t, unsigned int size);
```


Etude de la fonction minOf [6 points] (1/5)

La fonction minOf a pour rôle de trouver dans un tableau d'entiers `tab` de taille `size` :

(i) la valeur minimum des éléments dans l'intervalle des index compris entre les indices i et j (inclus) avec i<j,

(ii) et l'indice de ce minimum dans le tableau.

(a) Pour les valeurs de i et j respectivement égales à 2 et 7, donnez la valeur du minimum et son indice dans le tableau suivant :

tab = [12, 2, 3, 1, 7, 24, -1, 94]

12	2	3	1	7	24	-1	94
0	1	2	3	4	5	6	7

`min et indice : -1 et 6`

Etude de la fonction minOf (2/5)

(b) Déclarez la fonction `minOf` en utilisant une variable résultat de type structuré nommé `Solution`

```
typedef struct {
     int min;
     unsigned int index;
} Solution;


Solution minOf(const int* t, unsigned int size,
unsigned int i, unsigned int j);
```


Etude de la fonction minOf

(c) Donnez un extrait de code permettant de vérifier les assertions du test unitaire vérifiant le résultat de la fonction minOf pour l'exemple donné à la question (5.a).

```
int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
unsigned int size=sizeof(tab)/sizeof(int); //L2
Solution s; //L3
s=minOf(tab, size, 2, 7); //L4
assert((s.min==-1) && (s.index=6));
```

(d) Donnez l'assertion qui vérifie toutes les préconditions de la fonction minOf.

```
assert((size>0) && (i>=0) && (i<j) && (j<size));
```


Etude de la fonction minOf

(c) Donnez un extrait de code permettant de vérifier les assertions du test unitaire vérifiant le résultat de la fonction minOf pour l'exemple donné à la question (5.a).

```
int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
unsigned int size=sizeof(tab)/sizeof(int); //L2
Solution s; //L3
s=minOf(tab, size, 2, 7); //L4
assert((s.min==-1) && (s.index=6));
```

(d) Donnez l'assertion qui vérifie toutes les préconditions de la fonction minOf.

```
assert((size>0) && (i>=0) && (i<j) && (j<size));
```


Etude de la fonction minOf

(d) Donnez l'assertion qui vérifie toutes les préconditions de la fonction `minOf`.

```
assert((size>0) && (i>=0) && (i<j) && (j<size));
```

(e) Définissez la fonction minOf.

```
Solution minOf(const int* t, unsigned int size,
                unsigned int i, unsigned int j) {
   assert((size>0) && (i>=0) && (i<j) && (j<size));
   Solution s;
   s.min= t[i];
   s.index=i;
   for (unsigned int k=i+1; k<=j; ++k) {
        if (t[k]<s.min) {
             s.min=t[k];
             s.index=k ;
        }
   }
   return s;
}
```

Etude de la fonction minOf

(f) Codez le programme principal qui affiche les résultats (minimum et index du minimum) de la fonction `minOf` appliquée pour les valeurs de `i=2` et `j=7` au tableau défini en (5.a).

Spécifications : le tableau sera initialisé par une liste explicite de ses éléments et sa taille calculée à partir de sa taille-mémoire.

```
int main() {
   int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
   unsigned int size= sizeof(tab)/sizeof(int); //L2
   Solution s ;                                //L3
   s=minOf(tab, size, 2, 7); //L4
   printf("%d %d\n", s.min, s.index);
   system("pause"); return 0;
}
```


Spécification – Ordonnancement (1/3)

Ordonnancement du source

(1) **Cartouche du fichier source** (documentation du fichier physique)

(2) **Les inclusions** des entêtes de bibliothèque (".h")
(guidées par l'utilisation d'une au moins de leurs fonctions dans le source)

(3) **Déclaration des constantes** par directives du préprocesseur (`#define`)

(4) Définition des **types structurés** avec **documentation** du type et des ses champs
(si leurs identificateurs ne sont pas suffisamment explicites)

(5) Tous les **déclarations/prototypes** des fonctions utilisées et codées dans le source

(6) Le `main()`

(7) La **définition documentée** de toutes les fonctions prototypées avant le `main()`

Spécification – Documentation (2/3)

Cartouche de programme

Informations à donner sur le fichier source

(1) le nom du fichier (fichier physique du système de fichier)
information nécessaire dans un contexte de compilation séparée où plusieurs sources participent à une application

(2) les auteurs du source

(3) a minima, la date de création

Documentation des fonctions

(1) rôle de la fonction

(2) Liste de tous les paramètres formels
leur identificateur avec leur type [`in`], [`out`] et [`in-out`] et rôle
et en dernier, en cas de retour [`retour`] avec son rôle

Spécification – Exemple (3/3)

```
/* main.c
 * Auteur : Marie-José Caraty
 * Date de création : 07/10/2016
 */
// Toutes les déclarations de fonction (prototypes)
void min(int x, int y, int* res); // void min(int, int, int*);

int main() {
   int x=7, y=3, res;
   min(x, y, &res);
   printf("min(%d, %d)=%d\n", x, y, res);
   system("pause"); return 0;
}
// Toutes les définitions de fonction
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers //[retour] le minimum…
 */
void min(int x, int y, int* res){
   if (x<y)
      *res=x;
   else
      *res=y;
}
```

min(7, 3)=3

Eléments de qualité d'un projet

Structuration des données

Définition de tous les types structurés (`Equipe`, `Championnat`, `Rencontre`, `Calendrier`) pour mémoriser toutes les informations nécessaires aux traitements de l'application

Architecture logicielle

Analyse des fonctionnalités/fonctions

Conception des fonctions utiles (rôle et prototypage) de l'application

Documentation des sources

Cartouche, constantes, structures de données et leur champs, fonctions

Code

Notion différée : Architecture logicielle

Dossier de développement logiciel

- Respect des spécifications

Respect des spécifications données pour le dossier de développement logiciel

Respect du cahier des charges de présentation : présence dans le dossier **a)** une page de garde indiquant le nom et le **groupe** des membres du binôme, l'objet du dossier, **b)** une table des matières de l'ensemble du dossier (incluant les annexes) avec la **pagination** de toutes les rubriques, la pagination est continue du début à la fin du dossier, **c)** présentation de l'application, **d)** organisation des tests, **e)** bilan de validation des tests, **f)** bilan de projet.
Brochez vos dossiers.

- Analyse

Présentation de l'application : l'art de synthétiser le projet (taille donnée)
Rôle fonctionnel de l'application (ce que fait l'application), les entrées et sorties (résultats attendus) (spécification : en 1 page)

✓ Structurations des données en mémoire
✓ Qualité des fonctions (pertinence des primitives)

Qualité de code (1/2)

- Lisibilité du code

Code parfaitement indenté (tabulation)
Ligne de code limitée à 80 colonnes

- Documentation

Du fichier source (son nom, son/ses auteurs et a minima sa date de création) des types, des champs des types structurés, des variables représentant les données

Des fonctions :
rôle de la fonction,
rôle des paramètres formels, leur mode ([in], [out] et [in-out])
rôle du paramètre de retour éventuel

Qualité de code (2/2)

- Absence de nombres magiques

  Absence de littéraux, utilisation de constantes (d'énumération ou de macros) motivée par des changements possibles des constantes considérées.

- Longueur des fonctions

  Le `main` doit être court (à très court, quelques lignes).
  Un `main` est à un niveau macroscopique (constitué d'appels de fonction). Un main trop long traduit un manque au niveau de l'analyse, des fonctions aurait dû être conçues.
  Par exemple on n'y développe pas une interface (e.g., un menu).
  Une fonction ne doit pas dépasser une vingtaine de lignes. Une fonction trop longue indique souvent un manque d'analyse fonctionnelle et s'accompagne de redondance : dans ce cas, on relit et on analyse le code en visant un niveau mascroscopique (de l'algorithme) pour introduire les fonctions (à coder) et qui seront à appeler (la longueur du code diminuera).

- Code redondant

  Introduire la/les fonction(s) qui généralisent les traitements.

Les tests

- Tests

  D'une manière générale, vous devez préciser votre stratégie de test : comment sont organisés vos tests.
  Dans notre cas, les tests sont organisés en 5 Sprints
  Un test comprend : un objectif (ce que l'on veut tester), un JDT (jeu de données de test), un résultat attendu (résultat de référence), un résultat (**trace d'exécution** de votre programme votre `run.txt`) et un **bilan de validation** (ce que valide le test).
  Il faut rédiger cette partie.

  Dans le rapport, on doit trouver une partie de « Bilan de validation des tests » où on résume tous les tests et ce qu'ils valident.

  Des JDT « personnels » doivent servir à améliorer la couverture de test

- Bilan de projet

  Retour d'expérience. Les difficultés rencontrées, ce qui est réussi, ce qui peut être amélioré

Recette de l'application en séance de TP (1/2)

Objectif

En présence de votre enseignant, tester votre programme du Sprint de plus haut niveau (`h`, $1 \leq h \leq 5$) que vous avez atteint et validé sur un nouveau jeu de données conçu pour votre groupe `n` (`in`**`h`**`G`**`n`**`.txt`)

Le test se fera sur l'invite de commande par redirection des entrées et sorties de l'application.
La validation de votre application se fera par la comparaison des résultats de votre application (`run.txt`) avec le résultat de référence (`out`**`h`**`G`**`n`**`.txt`) que vous communiquera votre enseignant.
Vous utiliserez l'archive `jar` (`diff`) pour cette comparaison.
Si aucune différence n'est constatée
  votre application est validée pour le sprint `#h` testé
  sinon il n'est pas validé et vous devez valider le sprint `#h-1`

Recette de l'application en séance de TP (2/2)

Mise en œuvre de la recette

- Préparez un projet Visual de votre application sur une machine de votre choix (machine de l'IUT ou portable personnel).
  Chargez le source du sprint `#h` de plus haut niveau que vous avez validé
  Votre enseignant vous communiquera le fichier `inSph.txt`
  Exécutez votre programme avec la redirection des entrées-sorties de votre application (`inSph.txt` et `run.txt`)
- Vérification par votre enseignant de TP de la validité de votre solution

  si votre fichier `run.txt` est identique à la solution de référence `outSph.txt`
    alors votre logiciel est accepté, le Sprint est validé
    sinon le logiciel est refusé,
      des corrections sont à envisager (5 minutes vous sont accordées)
      Le délai passé, vous devez vous préparer à passer la recette du Sprint `#h-1`