Rappels sur les sous-programmes et les passages de paramètres

Entrées/sorties des sous-programmes

Entrées/sorties des sous-progs.

- *Un sous-programme **appelé** peut utiliser des données qui sont fournies par l'**appelant** : **ses entrées***
- *Un sous-programme **appelé** peut renvoyer des données à l'**appelant** : **ses sorties***

```
int triple(int x) // Appelé
{
    return 3*x;
}

int main()  // Appelant (pas forcément le main)
{
    int b, a=4;
    b = triple(a);  // b prend la valeur 12
    ...
```

Entrées/sorties des sous-progs.

- *Dans le cas d'une fonction : la valeur retournée est toujours une **sortie***

```
int triple(int x)
{
    return 3*x;
}

int main()
{
    int b, a=4;

    b = triple(a);   // b prend la valeur 12
```

Entrées/sorties des sous-progs.

- *Dans le cas d'une fonction, la valeur retournée est toujours une **sortie**, utilisée ou pas...*

```
int triple(int x)
{
    return 3*x;
}

int main()
{
    int b, a=4;
    triple(a);      // 12 retourné : ignoré
                    // On a le droit de ne pas
                    // utiliser une sortie
                    // mais il faut le savoir
```

?

12

a vaut toujours 4

Entrées/sorties des sous-progs.

- *Dans le cas d'une fonction, la valeur retournée est toujours une **sortie**, appel utilisable dans tout contexte compatible avec le type de retour*

```
int triple(int x)
{
    return 3*x;
}

int main()
{
    int b, a=4;

    printf("%d\n", triple(a)); // affiche 12
```

Entrées/sorties des sous-progs.

- *Dans le cas d'une fonction, la valeur retournée est toujours une **sortie**, 1) fonction évaluée 2) utilisation*

```
int triple(int x)
{
    return 3*x;
}

int main()
{
    int a=4;

    a = triple(a);   // a prend la valeur 12

    équivaut à  a=3*a;  ou encore  a*=3;
```

valeur de a **avant** l'appel et l'affectation : 4

12

Entrées/sorties des sous-progs.

- *Les données en **entrées** peuvent venir de toute expression compatible avec le type attendu*

```
int triple(int x)
{
    return 3*x;
}

int main()
{
    int b, c, a=4;

    b = triple(4);  // b prend la valeur 12
    printf("%d\n", triple(4)); // affiche 12
    a = triple(2*a-1); // a prend la valeur 21
    c = triple( triple(2) ); // c vaudra 18
```

Entrées/sorties des sous-progs.

- *Cas d'un sous-programme avec paramètre passé par adresse, utilisé en tant que **sortie***

```
void triple(int x, int *py)
{
    *py = 3*x;
}

int main()
{
    int b, a=4;

    triple(a, &b);
```

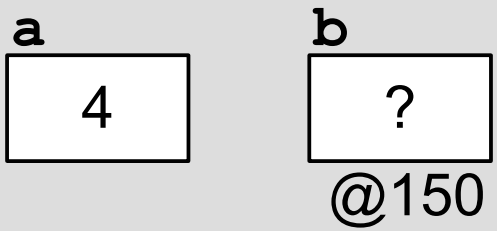
Entrées/sorties des sous-progs.

- ***Avant** l'appel*

```
void triple(int x, int *py)
{
     *py = 3*x;
}

int main()
{
     int b, a=4;

     triple(a, &b);
```



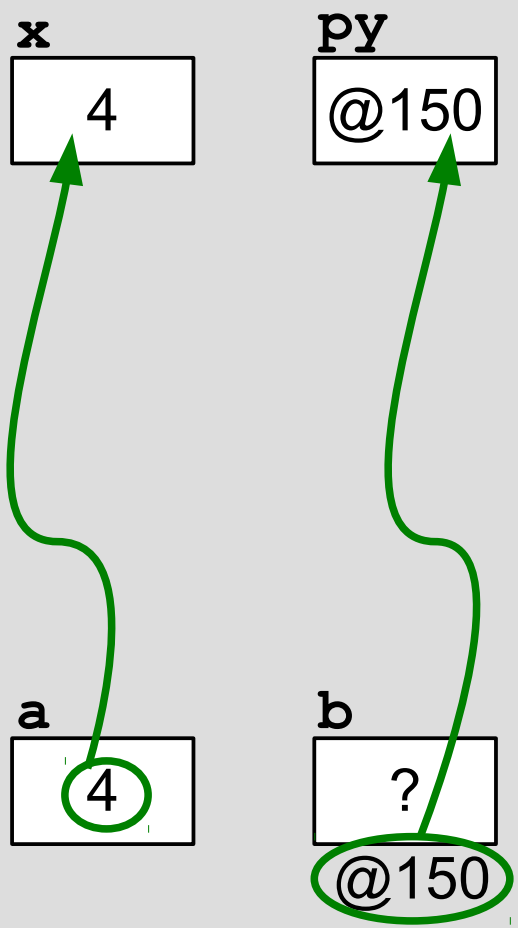
Entrées/sorties des sous-progs.

- ***Au moment** de l'appel*

```
void triple(int x, int *py)
{
     *py = 3*x;
}

int main()
{
     int b, a=4;

     triple(a, &b);
```



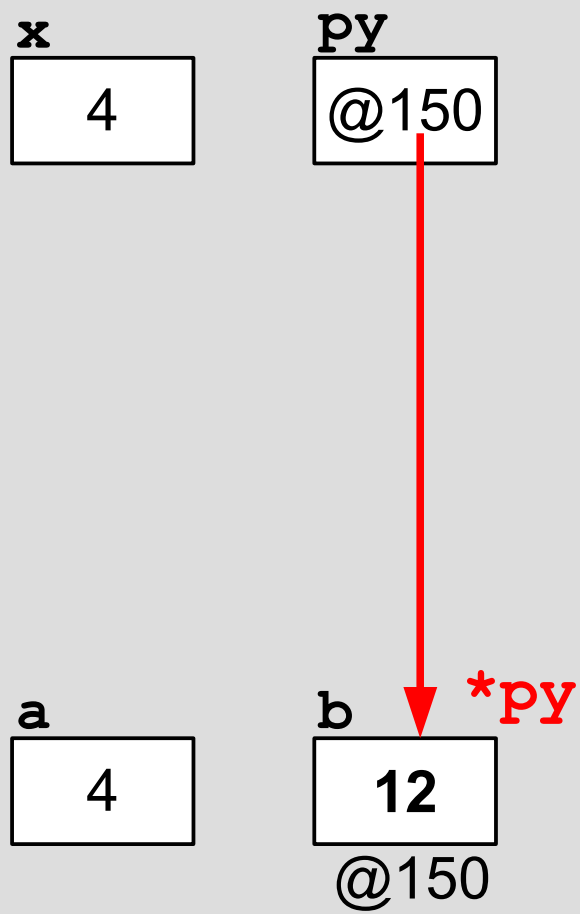
Entrées/sorties des sous-progs.

- ***Au moment*** *de l'affectation dans le sous-programme*
  **py est un "alias" vers la variable de l'appelant*

```
void triple(int x, int *py)
{
     *py = 3*x;
}

int main()
{
     int b, a=4;

     triple(a, &b);
```



Entrées/sorties des sous-progs.

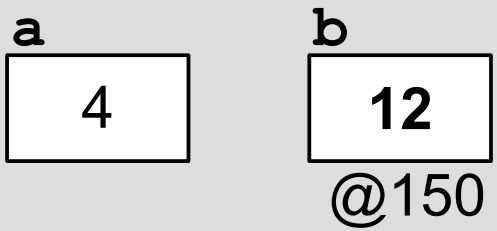
- ***Après** l'appel*

```
void triple(int x, int *py)
{
     *py = 3*x;
}

int main()
{
     int b, a=4;

     triple(a, &b);

     // b vaut 12 …
```



Entrées/sorties des sous-progs.

- Dans cette situation :
  - *le 1er paramètre est une copie de a, (a non modifié)*
  - *la valeur initiale de b, pointée par le 2ème paramètre est toujours ignorée*

```
void triple(int x, int *py)
{
    *py = 3*x;
}

int main()
{
    int b, a=4;

    triple(a, &b);
```

Entrées/sorties des sous-progs.

- Définitions

  - *Un paramètre utilisé par l'appelé pour accéder à (**lire**) la valeur d'une donnée de l'appelant est une **entrée***

  - *Un paramètre utilisé pour modifier (**écrire**) la valeur d'une ressource de l'appelant correspond à une **sortie***

  - *Un paramètre désignant une donnée de l'appelant dont la valeur reçue est ignorée et qui ne sera jamais modifiée par l'appelé est **inutile***

  - *Un paramètre qui donne accès à la valeur initiale d'une ressource de l'appelant **et** qui la modifie est à la fois une **entrée et une sortie***

Entrées/sorties des sous-progs.

- *Un paramètre désignant une donnée de l'appelant dont la valeur reçue est ignorée et qui ne sera jamais modifiée par l'appelé est* ***inutile***

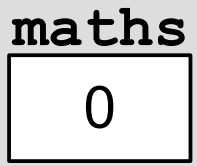
Exemple : passage par valeur d'une variable devant être modifiée (code non correct)

Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée (code non correct)

```
void saisie_note(int note)
{
     do
     {
        scanf("%d", &note);
     } while ( note<0 || note>20 );
}

int main()
{
     int maths=0;
     saisie_note(maths);
```

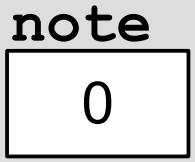


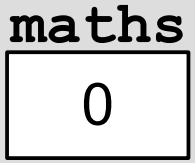
Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée (code non correct)

```
void saisie_note(int note)
{
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
}

int main()
{
    int maths=0;
    saisie_note(maths);
```





Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée (code non correct)

```
void saisie_note(int note)
{
     do
     {
     scanf("%d", &note);
     } while ( note<0 || note>20 );
}

int main()
{
     int maths=0;
     saisie_note(maths);
```

note

14

maths

0

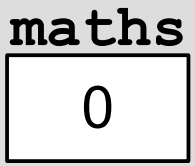
Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée (code non correct)

```
void saisie_note(int note)
{
     do
     {
        scanf("%d", &note);
     } while ( note<0 || note>20 );
}

int main()
{
     int maths=0;
     saisie_note(maths);
     ...
```

la donnée de l'appelant n'est pas modifiée !



Entrées/sorties des sous-progs.

- *Un paramètre désignant une donnée de l'appelant dont la valeur reçue est ignorée et qui ne sera jamais modifiée par l'appelé est* ***inutile***

Exemple : passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (code très maladroit)

Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (très maladroit)

```
int saisie_note(int note)
{
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note(maths);
    ...
```

maths

0

Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (très maladroit)

```
int saisie_note(int note)
{
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note(maths);
    ...
```

note

0

maths

0

Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (très maladroit)

```
int saisie_note(int note)
{
    do
    {
    → scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note(maths);
    ...
```

note

14

La valeur reçue ne servait à rien !

la donnée de l'appelant n'est pas modifiée !

maths

0

Entrées/sorties des sous-progs.

- passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (très maladroit)

```
int saisie_note(int note)
{
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note(maths);
    ...
```

note
14

14

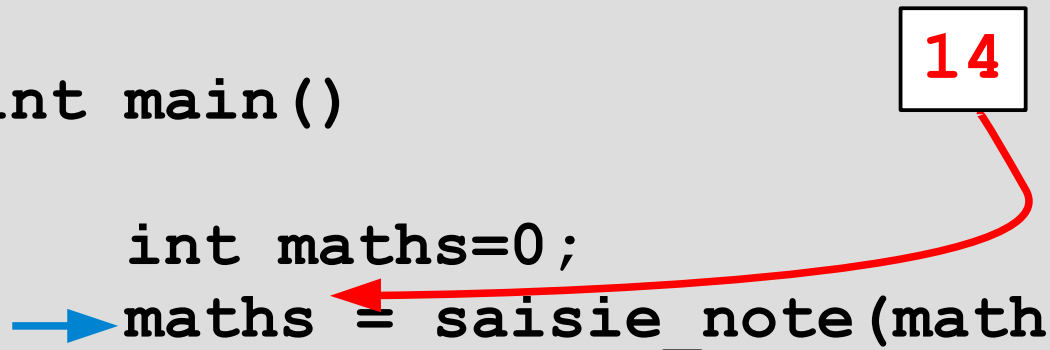
maths
0

Entrées/sorties des sous-progs.

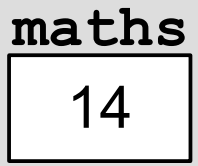
- passage par valeur d'une variable devant être modifiée en retour mais valeur de départ ignorée (très maladroit)

```
int saisie_note(int note)
{
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note(maths);
    ...
```



la donnée de l'appelant est bien modifiée, mais par le retour, pas par le paramètre entrant...



Entrées/sorties des sous-progs.

- ***Attention aux confusions paramètres / variables locales***

Exemple : variable de l'appelant devant être modifiée en retour mais sa valeur de départ ignorée

Entrées/sorties des sous-progs.

- Approche correcte

```
int saisie_note()
{
    int note;
    do
    {
       scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note();
    ...
```

maths

0

Entrées/sorties des sous-progs.

- Approche correcte

```
int saisie_note()
{
     int note;
     do
     {
        scanf("%d", &note);
     } while ( note<0 || note>20 );
     return note;
}

int main()
{
     int maths=0;
     maths = saisie_note();
     ...
```

variable locale

note

?

maths

0

Entrées/sorties des sous-progs.

- Approche correcte

```
int saisie_note()
{
    int note;
    do
    {
    scanf("%d", &note);
    } while ( note<0 || note>20 );
    return note;
}

int main()
{
    int maths=0;
    maths = saisie_note();
    ...
```

note

14

maths

0

Entrées/sorties des sous-progs.

- Approche correcte

```
int saisie_note()
{
     int note;
     do
     {
        scanf("%d", &note);
     } while ( note<0 || note>20 );
     return note;
}

int main()
{
     int maths=0;
     maths = saisie_note();
     ...
```

note
14

14

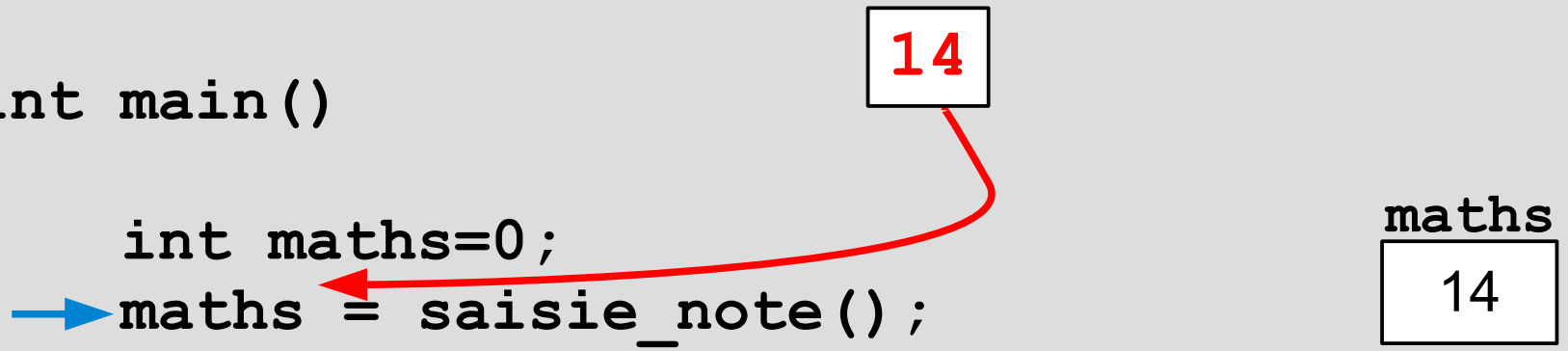
maths
0

Entrées/sorties des sous-progs.

- Approche correcte

```
int saisie_note()
{
     int note;
     do
     {
        scanf("%d", &note);
     } while ( note<0 || note>20 );
     return note;
}

int main()
{
     int maths=0;
     maths = saisie_note();
     ...
```



Entrées/sorties des sous-progs.

- Un paramètre est entrée et sortie quand il correspond à un passage par adresse et que
  - *la valeur initiale pointée par le paramètre est utilisée*
  - *la valeur pointée par le paramètre est modifiée*

```
void tripler(int *px)
{
    *px = 3 * *px;
}

int main()
{
    int a=4;

    tripler(&a);
```

Entrées/sorties des sous-progs.

- *la valeur initiale pointée par le paramètre est utilisée*
- *la valeur pointée par le paramètre est modifiée*

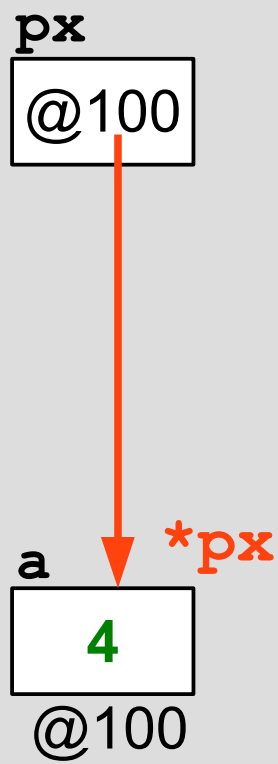
```
void tripler(int *px)
{
     *px = 3 * *px;
}

int main()
{
     int a=4;

     tripler(&a);
```

①

***px vaut 4**



Entrées/sorties des sous-progs.

- *la valeur initiale pointée par le paramètre est utilisée*
- *la valeur pointée par le paramètre est modifiée*

```
void tripler(int *px)
{
    *px = 3 * *px;
}
```

②

***px prend valeur 12**

```
int main()
{
    int a=4;

    tripler(&a);
```

Entrées/sorties des sous-progs.

Paramètres : le cas des tableaux

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*
  - *En entrées*
  - *En sorties*
  - *A la fois en entrées et en sorties*

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*

  - ***En entrée***

```
int tab_somme(int tab[10])
{
    int i, s;

    s = 0;
    for (i=0; i<10; i++)
        s = s + tab[i];

    return s;
}
```

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*

  - ***En entrée***

```
float tab_moyenne(int tab[10])
{
    return (float)tab_somme(tab)/10;
}
```

relayer le paramètre ...

version moins compacte

```
float tab_moyenne(int tab[10])
{
    int s;
    float moy;
    s = tab_somme(tab);
    moy = s/10.0;
    return moy;
}
```

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*

  - ***En entrée***

```
void tab_afficher(int tab[10])
{
    int i;

    for (i=0; i<10; i++)
        printf("%d\n", tab[i]);
}
```

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*

  - ***En sortie***

```
void tab_initialiser(int tab[10])
{
    int i;

    for (i=0; i<10; i++)
        tab[i] = 0;
}
```

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*

  - ***En sortie***

```
void tab_saisir(int tab[10])
{
    int i;

    for (i=0; i<10; i++)
        scanf("%d", &tab[i]);
}
```

Entrées/sorties des sous-progs.

- *Les paramètres **tableaux** sont **toujours passés par adresse**, ils peuvent être utilisés*
  - ***A la fois en entrée et en sortie***

```
void tab_trier(int tab[10])
{
    int i, j, tmp;

    for (i=0; i<9; i++)
        for (j=0; j<9-i; j++)
            if ( tab[j]>tab[j+1] )
            {
                tmp = tab[j];
                tab[j] = tab[j+1];
                tab[j+1] = tmp;
            }
}
```

Appelant

```
int notes[10];

tab_saisir(notes);

tab_trier(notes);

tab_afficher(notes);

printf("moy=%f\n",
 tab_moyenne(notes));
```

Entrées/sorties des sous-progs.

- *Un **tableau** n'est **jamais retourné** sauf si il n'existait pas avant l'appel et qu'il est **alloué** pendant l'appel*

```
int * tab_puissances(int b, int n)
{
    int e, p;
    int *tab;

    tab = (int *)malloc(n*sizeof(int));

    p = 1;
    for (e=0; e<n; e++)
    {
        tab[e] = p;
        p = b*p;
    }

    return tab;
}
```

Appelant

```
int * conv;

conv = tab_puissances(2,8);

// conv[0] vaut 1
// conv[1] vaut 2
// conv[2] vaut 4
…
// conv[7] vaut 128
```

Entrées/sorties des sous-progs.

- *Un **tableau** n'est **jamais retourné** sauf si il n'existait pas avant l'appel et qu'il est **alloué** pendant l'appel*

```
int * tab_puissances(int b, int n)
{
    int e, p;
    // CECI N'EST PAS CORRECT, PAS D'ALLOC DYNAMIQUE
    //      => espace de stockage non persistant
    int tab[n];

    p = 1;
    for (e=0; e<n; e++)
    {
        tab[e] = p;
        p = b*p;
    }

    return tab;
}
```

Appelant

```
int * conv;
```

l'appelant récupère l'adresse d'un tableau qui n'est plus valide après l'appel...

```
conv = tab_puissances(2,8);
```

Entrées/sorties des sous-progs.

- *De manière générale nous n'utiliserons **pas** la possibilité du C99 de dimensionner de façon variable des tableaux **automatiques***
- *Cette possibilité est offerte aux programmeurs expérimentés et comporte de nombreux pièges : peu utilisé en pratique*

```
int n  paramètre ou variable

...

int tab[n]; // NON, même si "ça compile"
```

Entrées/sorties des sous-progs.

- *Pour dimensionner de façon variable des tableaux on utilisera donc forcément l'**allocation dynamique***

```
int n  paramètre ou variable

int *tab;
...
tab = (int *)malloc(n*sizeof(int));

 tab est utilisable comme un tableau usuel
     tab[0]  tab[1]    tab[i] avec i dans [0 … n-1]

 on peut le retourner à un appelant

 quand on a fini de l'utiliser on doit le libérer :
free(tab);
```

n a une valeur connue
quand on arrive à l'alloc

Entrées/sorties des sous-progs.

- *Les tableaux **automatiques** seront toujours dimensionnés par une valeur constante*
- *Eventuellement il est possible d'utiliser un identifiant symbolique pour indiquer la constante*

```
const int taille=10;   // variable constante !

...
int tab[taille]; // OK, équivaut à int tab[10];

...
int autre_tab[taille]; // même taille que tab
```

Entrées/sorties des sous-progs.

- *Les tableaux **automatiques** seront toujours dimensionnés par une valeur constante*
- *Eventuellement il est possible d'utiliser un identifiant symbolique pour indiquer la constante*

```
#define TAILLE 10       // constante symbolique

...
int tab[TAILLE]; // OK, équivaut à int tab[10];

...
int autre_tab[TAILLE]; // même taille que tab
```

Entrées/sorties des sous-progs.

- *Les constantes symboliques permettent de "synchroniser" toutes les utilisation d'une même valeur*

```
#define NB_ENTIERS 10 // Avant les définitions

void tab_saisir(int tab[NB_ENTIERS])
{ … }
void tab_afficher(int tab[NB_ENTIERS])
{ … }
void tab_trier(int tab[NB_ENTIERS])
{ … }
int main()
{
    int notes[NB_ENTIERS];
    tab_saisir(notes);
    tab_trier(notes);
    tab_afficher(notes);
    ...
```

Entrées/sorties des sous-progs.

- *On peut utiliser les tableaux pour regrouper des informations jouant des rôles différents, et faciliter le passage par adresse*

```
#define POSLIG 0
#define POSCOL 1
#define DEPLIG 2
#define DEPCOL 3

void bouger_mobile(int mob[4])
{
    mob[POSLIG] = mob[POSLIG] + mob[DEPLIG];
    mob[POSCOL] = mob[POSCOL] + mob[DEPCOL];
    ...
}

    // dans la boucle de jeu...
    bouger_mobile(fantome);
    bouger_mobile(gorille);
```

Entrées/sorties des sous-progs.

- *Pratique grâce au passage par adresse des tableaux*
  mais c'est du bricolage

```
#define POSLIG 0
#define POSCOL 1
#define DEPLIG 2
#define DEPCOL 3

void bouger_mobile(int mob[4])
{
    mob[POSLIG] = mob[POSLIG] + mob[DEPLIG];
    mob[POSCOL] = mob[POSCOL] + mob[DEPCOL];
    ...
}

    // dans la boucle de jeu...
    bouger_mobile(fantome);
    bouger_mobile(gorille);
```

Entrées/sorties des sous-progs.

Paramètres : le cas des structs

Entrées/sorties des sous-progs.

- *Regrouper des informations jouant des rôles différents*
  struct
- ***Que les types soient distincts ou identiques***

```
typedef struct mobile
{
    int poslig;
    int poscol;
    int deplig;
    int depcol;
} t_mobile;
```

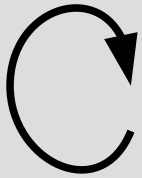
Entrées/sorties des sous-progs.

- *Les variables de types structs peuvent être passées par valeur et retournée : comme de simples scalaires*

```
t_mobile bouger_mobile(t_mobile mob)
{
    mob.poslig = mob.poslig + mob.deplig;
    mob.poscol = mob.poscol + mob.depcol;
    ...
    return mob;
}

int main()
{
    t_mobile fantome;
    t_mobile gorille;
    ...
    fantome = bouger_mobile(fantome);
    gorille = bouger_mobile(gorille);
    ...
```

Boucle de jeu

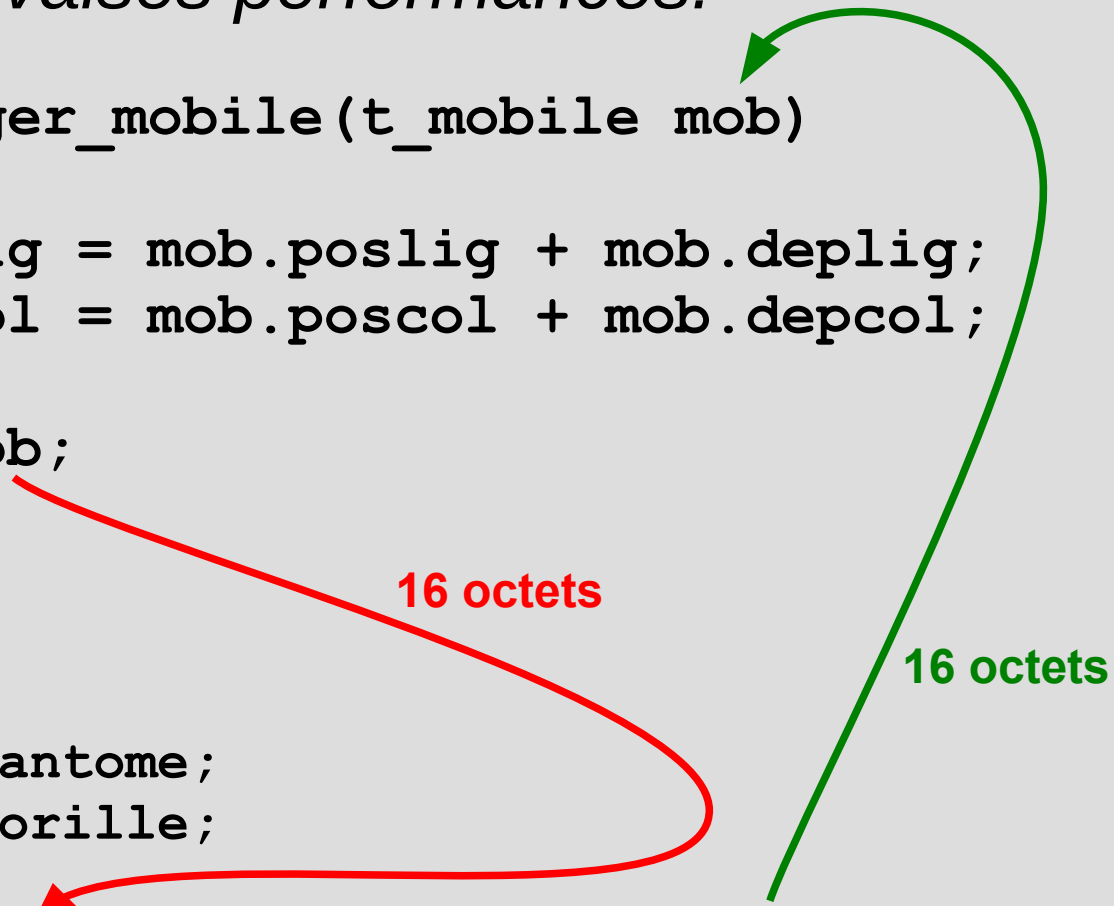


Entrées/sorties des sous-progs.

- *Le passage par valeur des structs est si possible à éviter : mauvaises performances.*

```
t_mobile bouger_mobile(t_mobile mob)
{
    mob.poslig = mob.poslig + mob.deplig;
    mob.poscol = mob.poscol + mob.depcol;
    ...
    return mob;
}

int main()
{
    t_mobile fantome;
    t_mobile gorille;
    ...
        fantome = bouger_mobile(fantome);
        gorille = bouger_mobile(gorille);
        ...
```

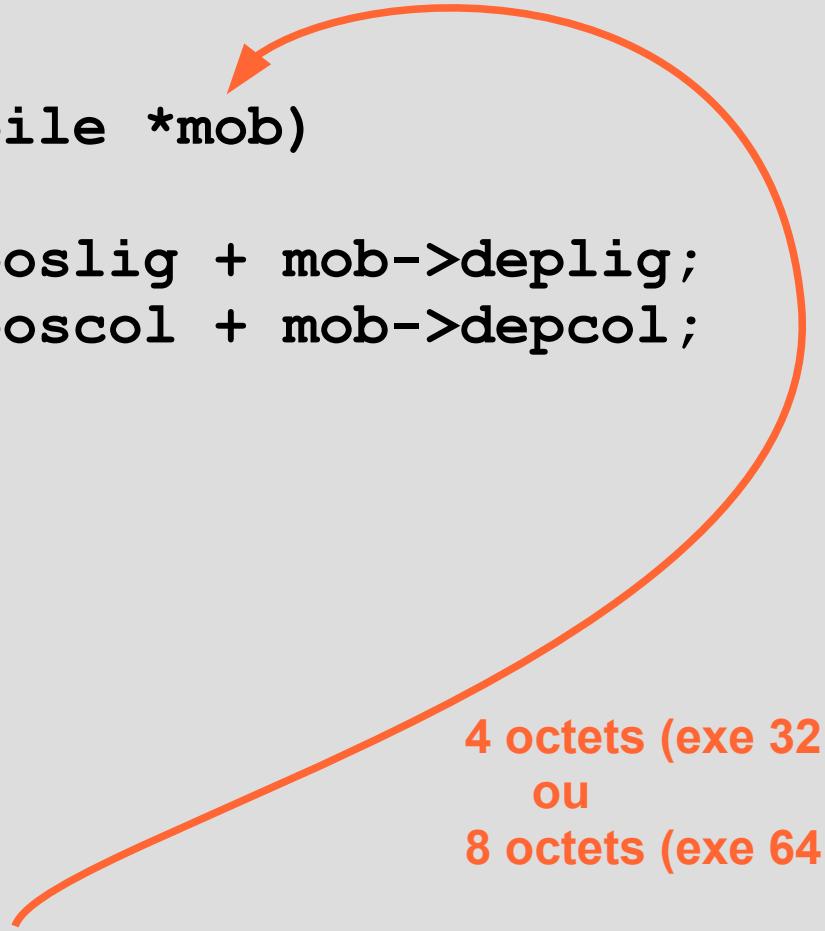


Entrées/sorties des sous-progs.

- ***Le passage par adresse des structs est à privilégier***

```
void bouger_mobile(t_mobile *mob)
{
    mob->poslig = mob->poslig + mob->deplig;
    mob->poscol = mob->poscol + mob->depcol;
    ...
}


int main()
{
    t_mobile fantome;
    t_mobile gorille;
    ...
        bouger_mobile(&fantome);
        bouger_mobile(&gorille);
        ...
```



4 octets (exe 32 bits)
ou
8 octets (exe 64 bits)

quelle que soit la taille de la struct

Entrées/sorties des sous-progs.

- *Donc autant déclarer des pointeurs directement... mais ça implique d'allouer dynamiquement.*

```
void bouger_mobile(t_mobile *mob)
{
     ...
}

int main()
{
    t_mobile *fantome;
    t_mobile *gorille;
    fantome = (t_mobile *)malloc(1*sizeof(t_mobile));
    gorille = (t_mobile *)malloc(1*sizeof(t_mobile));
    ...
        bouger_mobile(fantome);
        bouger_mobile(gorille);
        ...
```

Entrées/sorties des sous-progs.

- *Mais on n'aime pas beaucoup faire des mallocs dans le **code utilisateur** (appelant)*

```
t_mobile * creer_mobile(paramètres initiaux...)
{
    t_mobile *mob;
    mob = (t_mobile *)malloc(1*sizeof(t_mobile));
    ... initialiser mob->poslig mob->poscol ...
    return mob;
}

int main()
{
    t_mobile *fantome;
    t_mobile *gorille;
    fantome = creer_mobile(dans la cave);
    gorille = creer_mobile(sur l'échafaudage);
    ...
```

Entrées/sorties des sous-progs.

- *On arrive à une conception "**objets**"*

```
t_mobile * creer_mobile(paramètres initiaux...);

void bouger_mobile(t_mobile *mob, t_carte *terrain);

void dessiner_mobile(t_mobile *mob, t_carte *ecran);

void detruire_mobile(t_mobile *mob);

...
```

Entrées/sorties des sous-progs.

- *On arrive à une conception "**objets**"*

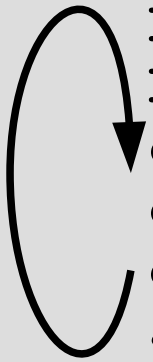
```
int main()
{   t_carte *terrain, *ecran;
    t_mobile *fantome, *gorille;

    terrain = charger_terrain("new_york.txt");
    ecran = creer_ecran(terrain->nblig, terrain->nbcol);
    fantome = creer_mobile(dans la cave);
    gorille = creer_mobile(sur l'échafaudage);

    bouger_mobile(fantome, terrain);
    bouger_mobile(gorille, terrain);
    dessiner_terrain(ecran);
    dessiner_mobile(fantome, ecran);
    dessiner_mobile(gorille, ecran);
    afficher_ecran(ecran);

    detruire_mobile(fantome); detruire_mobile(gorille);
    detruire_terrain(terrain); detruire_ecran(ecran);
```

Boucle de jeu



Contrats

Contrats

- *Conception → découpage en sous-programmes*
- *Sous-programme = sous-traitant spécialiste*
- *Prototype = nom + format d'appel du sous-programme*
  nom : résumé de la spécialité du sous-programme
  paramètres in : nécessaires au job du sous-programme
  paramètres out, retour : résultat(s) du job
- *Prototype + Commentaires/Documentation*

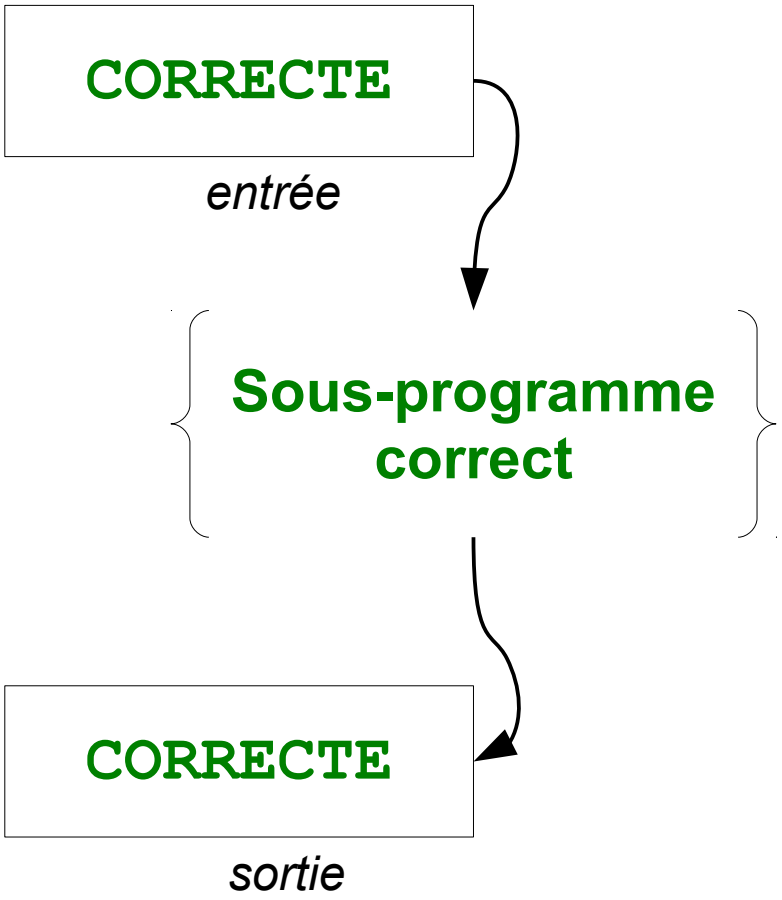
  → Définition du **CONTRAT** du sous-programme

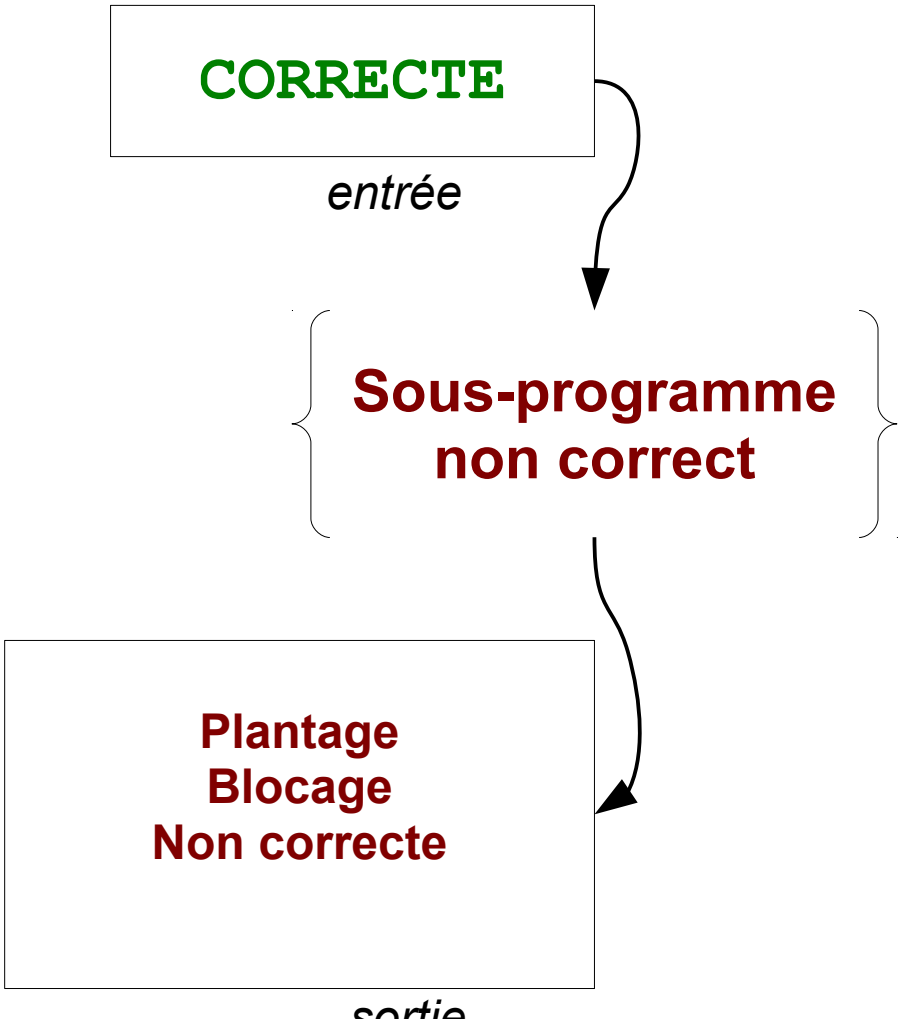
Contrats

- *Le contrat engage les 2 parties*
  - ***Appelant** (prog. utilisateur du sous-programme)*
  - ***Appelé** (le sous-programme)*
- *Il définit de manière explicite les **entrées correctes** sous forme de conditions à respecter*
- L'appelant s'engage à fournir à l'appelé
  des **entrées correctes**
- L'appelé s'engage à fournir en réponse à l'appelant
  des **sorties correctes**
  en **réponse à des entrées correctes**

Contrats

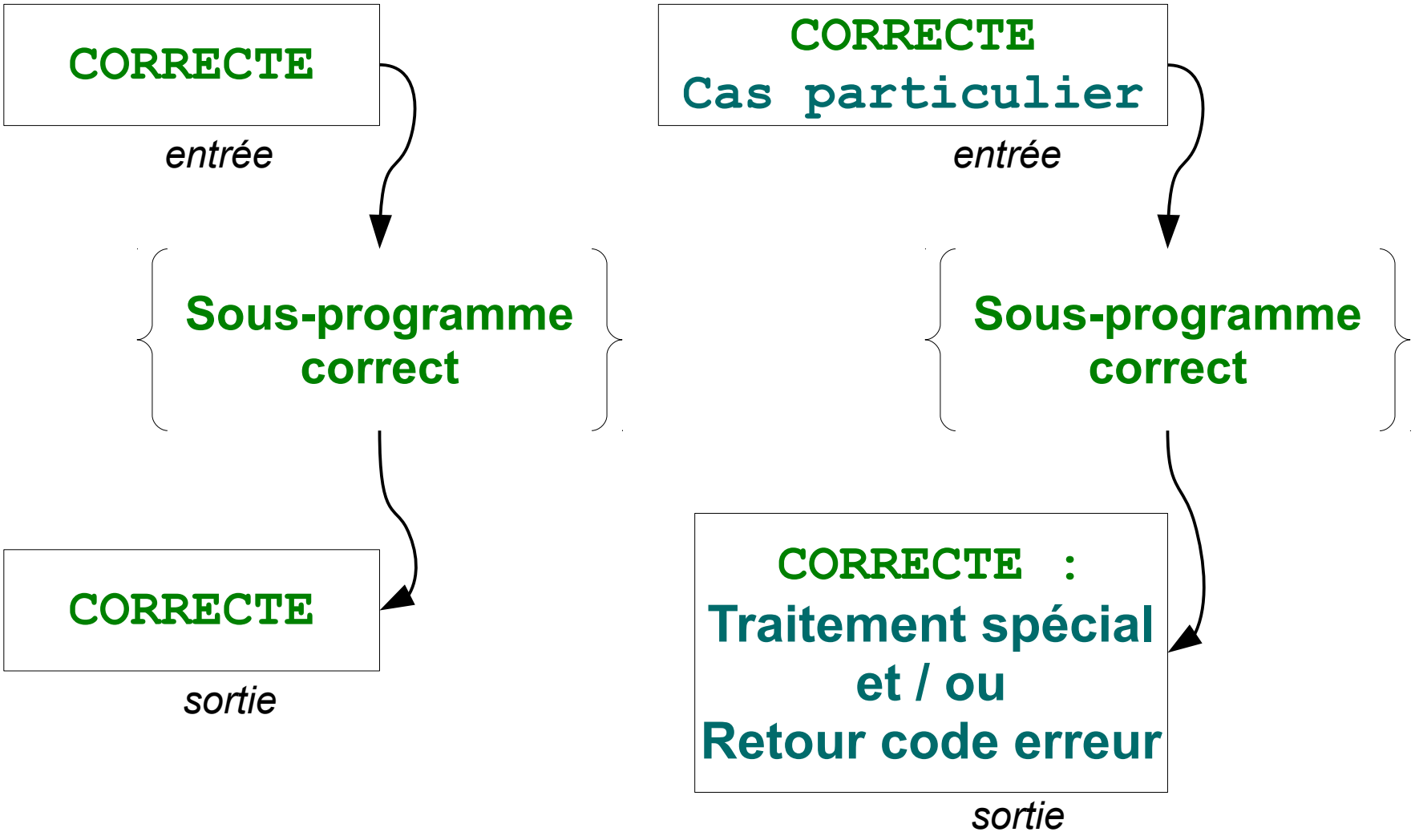
Le contrat définit les entrées correctes et les sorties correctes résultantes
Le sous-programme doit respecter le contrat pour être considéré correct
Le sous-programme est donc testé/validé sur des entrées correctes





Contrats

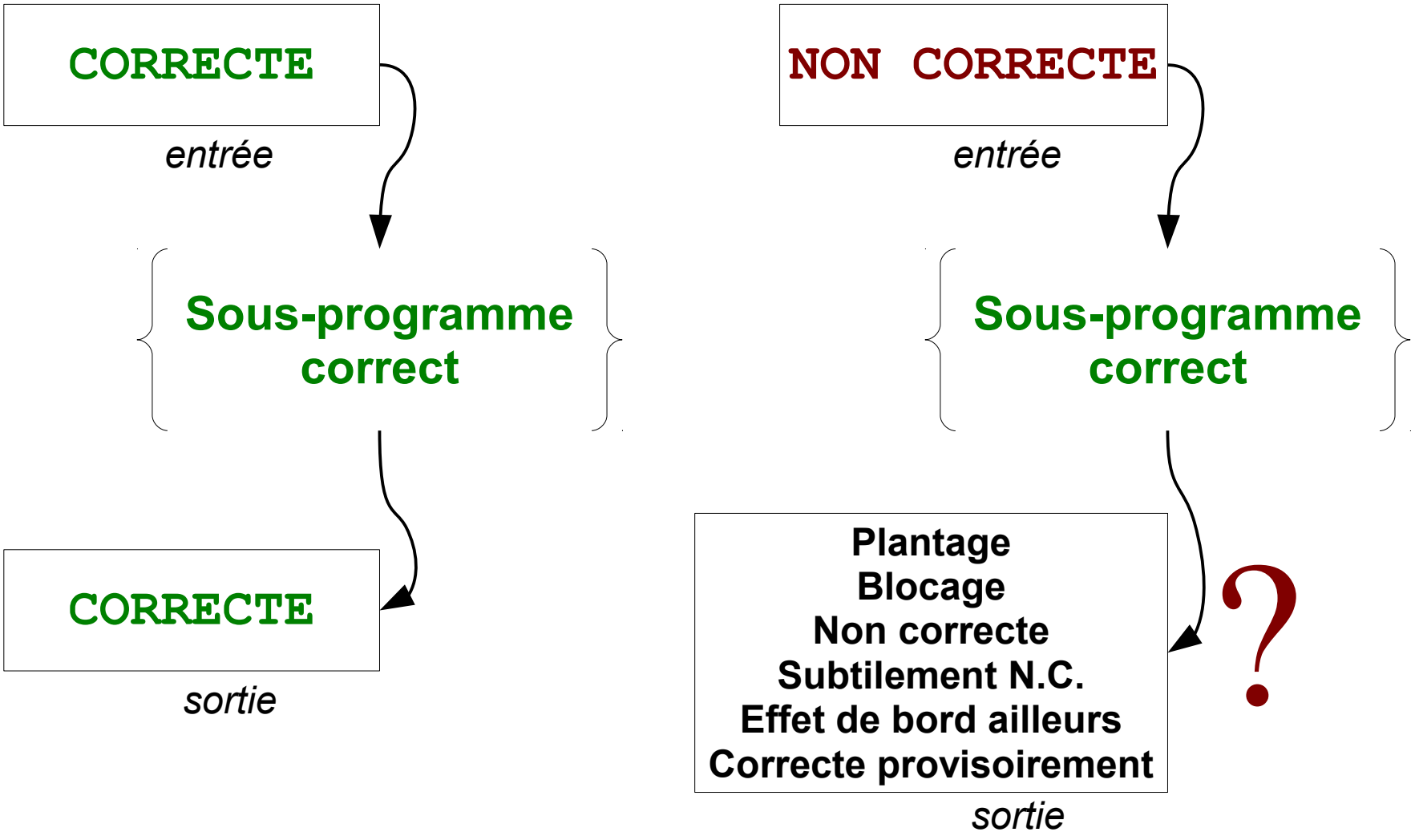
Le contrat définit les entrées correctes et les sorties correctes résultantes
On peut définir des cas particuliers "à problèmes" comme faisant partie des entrées "correctes" → **correctes car correctement gérées**



Contrats

Le contrat définit les entrées correctes et les sorties correctes résultantes

Le contrat ne dit rien sur l'issue pour mauvaises données venant de l'appelant



Contrats

- *La **rédaction des contrats** est un élément essentiel de la **conception***
  *Il faut **anticiper** les besoins futurs...*
- *La bonne compréhension et le respect du contrat est indispensable **côté appelé** pour implémenter un sous-programme correct.*
  - La gestion des cas particulier est indispensable
  - La gestion "correcte" des entrées non correctes n'est pas souhaitée et/ou pas possible
- *La bonne compréhension et le respect du contrat est indispensable **côté appelant** pour utiliser correctement un sous-programme correct.*

La lecture du code d'implémentation est inutile.

Contrats

Exemple : la fonction ***printf***

- ***Contrat*** *rédigé en 1978 (avec extensions ultérieures)*
  Depuis : `printf("%d\n", x);` *→ afficher l'entier x*
- ***Côté appelé*** *implémentations propres à chaque système/compilateur.*
  Plus de 2000 lignes de code directement impliquées...

- ***Côté appelant :*** *utilisez printf sur la base du contrat !*
  Conformité au contrat garantie :
  certification ANSI C89/C99

La lecture du code d'implémentation est inutile.