

Utiliser, écrire des sous programmes

M1102 – Algorithmie et Programmation en C

N. Gruson

Utiliser des fonctions

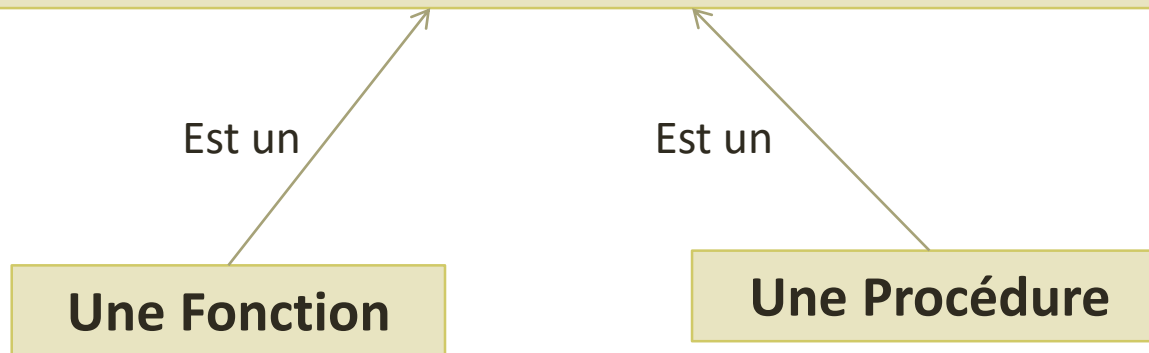
M1102 – Algorithmie et Programmation en C

N. Gruson

Définition – Sous programme

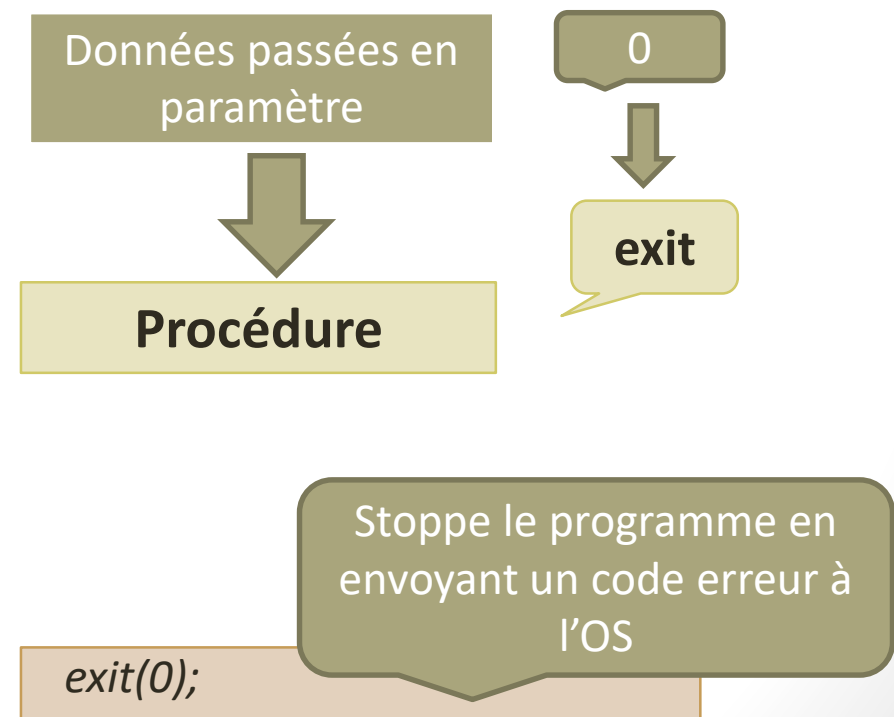
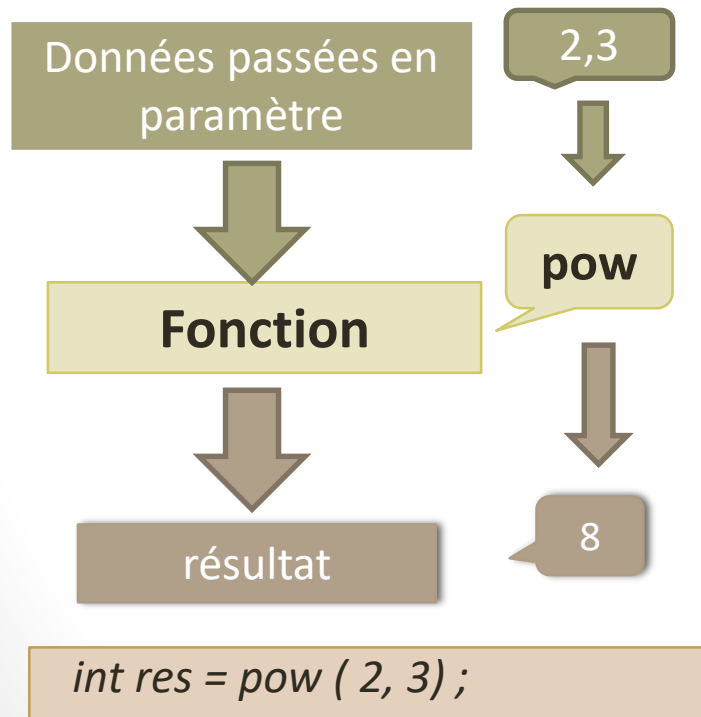
Un sous-programme :

- réalise un traitement particulier
- est appelé par un autre programme (programme appelant)
- porte un nom (Ex : pow, printf...)
- peut exploiter des données passées en paramètre



Définitions – Fonction/Procédure

La fonction retourne un résultat. La procédure elle ne retourne rien, généralement, elle sert à afficher ou à faire des traitements sans résultat.



Signature

Procédures et fonctions ont une signature : ligne d'entête, ligne de définition :

- **typeRetour nomFonction (type param1, type param2,.....)**
- **void nomProcédure (type param1, type param2,.....)**

retour

paramètres

double pow (double base, double exponent);

Raise to power

Returns *base* raised to the power *exponent*: $\text{base}^{\text{exponent}}$

Programme appelant/appelé

Un programme appelant peut faire appel à un ou plusieurs sous-programmes.

Programme Appelant

```
int main( )  
{  
  
    double nb = 2 , puissance = 3;  
    double res = pow ( nb, puissance) ;  
    printf( « %lf », res );  
  
    return 0 ;  
}
```

Sous programmes Ou programmes appelés

pow

printf

Communication appellant/appelé

Paramètre : donnée transmise par le programme appellant à un sous-programme.

Programme Appellant

```
int main( )  
{  
  
    double nb = 2 , puissance = 3;  
    double res = pow ( nb, puissance );  
    printf( « %lf », res );  
  
    return 0 ;  
}
```

Sous programmes

pow

2,3

« %i »,8

printf

Paramètres effectifs /formels

Les variables utilisées comme paramètres effectifs n'ont pas à porter le même nom que les paramètres définis formellement par le créateur de la fonction.

Programme Appelant

```
int main( )  
{  
  
    double nb = 2 , puissance = 3;  
    double res = pow ( nb, puissance) ;  
    printf( « %lf », res );  
  
    return 0 ;  
}
```

Paramètres effectifs :

- nb : 2
- puissance : 3

Sous programme

```
double pow (double base, double exponent )  
{ .... }
```

Paramètres formels:

- base : 2
- exponent : 3

2,3

Communication appelé/appelant

Retour : donnée transmise par le sous-programme au programme appelant

Programme Appelant

```
int main( )  
{  
  
    double nb = 2 , puissance = 3;  
    double res = pow ( nb, puissance) ;  
    printf( « %lf », res );  
  
    return 0 ;  
}
```

Sous programme

pow

8

A red arrow points from the 'pow' box to the '8' text, which is positioned above the arrow's path. The arrow then points to the 'double res = pow (nb, puissance) ;' line in the 'Programme Appelant' box.

Notion de librairie

- Le langage C comprend un certain nombre de fonctions déjà définies, prêtes à l'emploi. Elles sont regroupées dans la librairie standard.
- Il existe des fichiers d'entêtes (de signatures) de toutes ces fonctions . *Ex : math.h, ctype.h, stdio.h, stdlib.h, string.h, time.h ...*

Notion de librairie

- Pour connaître ces fonctions. Consultez une doc :
- Extrait du site : www.cplusplus.com/reference

Reference

C library:

<cassert> (assert.h)

<cctype> (ctype.h)

<cerrno> (errno.h)

<cfenv> (fenv.h)

<cfloat> (float.h)

< cinttypes> (inttypes.h)

<ciso646> (iso646.h)

<climits> (limits.h)

<locale> (locale.h)

<cmath> (math.h)

<csetjmp> (setjmp.h)

<csignal> (signal.h)

<csdarg> (stdarg.h)

<csdbool> (stdbool.h)

<csddef> (stddef.h)

<csdint> (stdint.h)

<csdio> (stdio.h)

<csdlib> (stdlib.h)

<cstring> (string.h)

<ctgmth> (tgmath.h)

<ctime> (time.h)

<cuchar> (uchar.h)

<wchar> (wchar.h)

header

<cmath> (math.h)

C numerics library

Header <cmath> declares a set of functions to compute common mathematical operat

fx Functions

Trigonometric functions

cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)

Hyperbolic functions

cosh	Compute hyperbolic cosine (function)
sinh	Compute hvnerbolic sine (function)

Utiliser une fonction

Pour utiliser une fonction, il faut :

- Mentionner le lien vers son fichier d'entête : `#include <...h>`
- Respecter sa signature :
 - le nombre et type de paramètres
 - le type de retour

retour

paramètres

`double pow (double base, double exponent);`

Raise to power

Returns *base* raised to the power *exponent*: $\text{base}^{\text{exponent}}$

Utiliser une fonction

Attention : la signature est une définition !

Ex : **double pow (double base, double exponent);**

Il ne faut pas la recopier telle quelle pour l'utiliser :

- **Les paramètres formels doivent être remplacés par des paramètres effectifs :**
 - Soit des valeurs. Ex : **pow(2,3);**
 - Soit des variables déjà définies. Ex : **pow(nb, puissance);**
 - Soit des expressions. Ex : **pow (nb+1 , nb*2);**
- **Le retour doit être :**
 - récupéré au sein d'une variable. Ex : **res= pow(nb, puissance);**
 - utilisé au sein d'une expression. Ex : **printf (« %lf » , pow(2,3));**

%i et %f possible aussi !

Utiliser une fonction

Attention :

- À l'ordre des paramètres
- Au type des paramètres

`pow(2,3) != pow(3,2)`

Le type double étant le plus grand des numériques, il accepte tous les autres numériques (float, int ...)

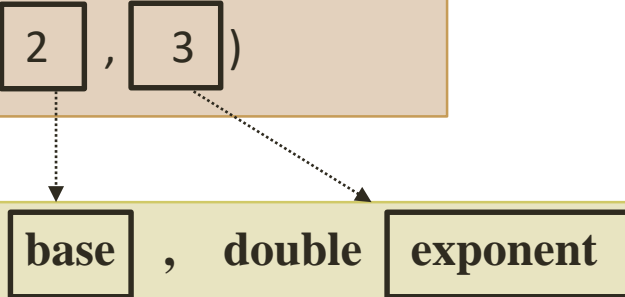
```
#include <math.h>
int main( )
{
    double res1 = pow(2,3);
    int res2, nb = 2 , puissance = 3 ;
    res2 = pow( nb, puissance );
}
```

Paramètres

Les paramètres sont traités dans l'ordre.

```
double res = pow ( 2 , 3 )
```

```
double pow (double base , double exponent )  
{  
    // ....  
}
```



Appel de fonction

On peut utiliser une fonction au sein d'une expression.

Attention, cela n'est pas possible pour une procédure.

```
printf (« %i » , pow(2,3 ));
```

```
res = pow(2,3);  
printf (« %i » , res );
```


Particularité

Certaines fonctions en C ont un nombre de paramètre(s) variable(s).

```
int printf ( const char * format, ... );
```

Print formatted data to stdout

Writes the C string pointed by *format* to the standard output (stdout). If *format* includes *format specifiers* (subsequences beginning with %), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

Char * = Chaîne de caractère

2 paramètres

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

```
int main( )
```

```
{ int a = 3, b = 4;
```

```
printf(" test ");
```

```
printf("valeur de a : %i\n" , a);
```

```
printf(" %i + %i\n" , a , b);
```

1 paramètre

3 paramètres

Particularité

On doit souvent exploiter le résultat d'une fonction, ce n'est pas pour autant obligatoire ...

int printf (const char * format, ...);

Return Value : On success, the total number of characters written is returned. If a writing error occurs, the *error indicator* (error) is set and a negative number is returned.

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

```
int main( )
```

```
{
```

```
    int r =printf("test");
```

```
    printf("\nresultat de printf : %i\n",r);
```

```
}
```

```
test
resultat de printf : 4
```

Retour du scanf...

```
int scanf ( const char * format, ... );
```

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error.

```
printf("Entrez un entier");
res = scanf("%i",&chiffre);
while( res!=1)
{
    printf("Erreur. Entrez un entier :\n");
    fflush(stdin);
    res = scanf("%i",&chiffre);
}
printf("entier : %i \n",chiffre);
```

```
printf("caractère compris entre a et z") ;
res=scanf("%1[a-z]c", &rep);
while (res !=1)
{
    printf("caractère compris entre a et z") ;
    fflush(stdin);
    res=scanf("%1[a-z]c", &rep);
}
```

Que des fonctions

Ou presque !

Il existe dans la librairie standard peu de procédures.

La plupart des sous programmes existants sont des fonctions !

Utiliser des méthodes - SFML

M1102 – Algorithmie et Programmation en C

N. Gruson

Définition méthode

Une méthode = fonction ou procédure en programmation objet.

Elle a aussi une signature :

typeRetour nomMethode (typeParam1 param1,...)

Ex: void setPosition (float x, float y)

Tout comme en programmation procédurale, il faut respecter ses paramètres, son retour

Différences méthodes/fonctions

La méthode s'appuie sur un objet !

```
CircleShape disque(100);
```

```
disque.setPosition(200,200);
```

Ici, setPosition s'appuie sur l'objet disque.

Conséquences :

- **Moins de paramètres** à passer car un grand nombre des données nécessaires est contenu au sein de l'objet
- **Plus de procédures**, car comme pour setPosition, beaucoup de sous programmes **auront pour but de faire évoluer l'objet** sans pour autant avoir à retourner un résultat

Accesseurs : getter & setter

La plupart des méthodes (dans SFML) commencent par :

- **set** : méthode pour modifier des données contenues dans l'objet à l'aide de paramètres.
Ex: setPosition, setFillColor, setOrigin, ...
- **get** : méthode pour récupérer des données contenues dans l'objet. Il n'y a donc pas de paramètres. Tout est déjà dans l'objet.
Ex: getPosition, getFillColor,....

```
CircleShape disque(100);  
disque.setPosition (Vector2f (200,200));
```

```
Vector2f pos =disque.getPosition();  
printf ("(%.0f,%.0f)", pos.x, pos.y);
```

Set : paramètres
Pas de retour

get : pas de paramètres
Un retour

Surcharge

Certaines méthodes sont surchargées. Elles proposent des alternatives d'utilisation. Ex: setPosition

```
void setPosition (float x, float y)  
    set the position of the object
```

```
disque.setPosition (200, 200);
```

```
void setPosition (const Vector2f &position)  
    set the position of the object
```

```
disque.setPosition (Vector2f(200, 200));
```

```
Vector2f positionDisque (200, 200) ;  
disque.setPosition (positionDisque );
```

Ecrire des sous programmes: Fonctions ou procédures

M1102 – Algorithmie et Programmation en C

N. Gruson

Pourquoi faire des sous-programmes ?

- Pour alléger le code du programme principal : moins de lignes dans le main => programme plus lisible !
- Pour factoriser le code et le réutiliser.

Algo non modulaire	Algo modulaire
<pre>scanf(« %i », &a); scanf(« %i », &b); if (a < b) min = a ; else min = b ; printf (« min : %i \n», min);</pre>	<pre>scanf(« %i », &a); scanf(« %i », &b); min = minimum(a,b); printf (« min : %i \n», min);</pre>

La fonction minimum allège le code et peut être réutilisée

Définitions – Fonction/Procédure

La fonction retourne un résultat.

La procédure elle ne retourne rien, elle sert à :

- Afficher
- A faire les traitements sans résultat.

Données passées en paramètre



Fonction



résultat

hormis des fonctions dédiées à l'affichage et saisies ne doivent pas contenir d'instructions printf/scanf !

Données passées en paramètre



Procédure

Exemple simple de fonction

Le programme **appelant** **récupère** le résultat de la fonction.

```
int main ( )
{ int a, b, mini ;
  scanf(« %i », &a);
  scanf(« %i », &b);
  mini = minimum(a,b);
  printf ( « min : %i \n», mini);
  return 0;
}
```

mini sert à stocker le résultat
retourné par la fonction

La **fonction retourne un résultat** , elle a :

- un type de retour
- une instruction de return

```
int minimum ( int val1 , int val2)
{
  int min = val1 ;
  if (val2 < val1)
    min = val2 ;
  return min ;
}
```

Exemple simple de procédure

Le programme appelant appelle la procédure sans rien récupérer.

```
int main ( )
{
    int a, b ;
    scanf(« %i », &a);
    scanf(« %i », &b);
    afficheMinimum(a,b);
    return 0;
}
```

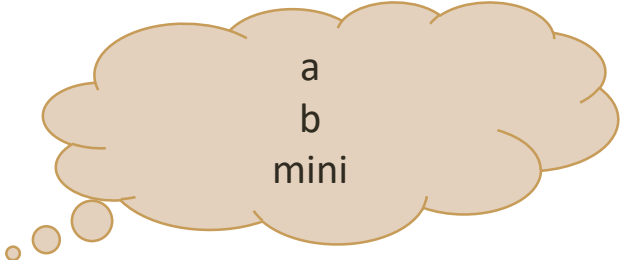
La procédure a pour type de retour : **void**
Et **ne fait pas de return !**

```
void afficheMinimum( int val1 , int val2)
{
    int min = val1 ;
    if (val2 < val1)
        min = val2 ;
    printf ( « min : %i \n», min);
}
```

Fortement déconseillé ! Mieux vaut préférer la fonction, car ici le code n'est pas réutilisable dans un programme en mode graphique ! On mélange logique et affichage

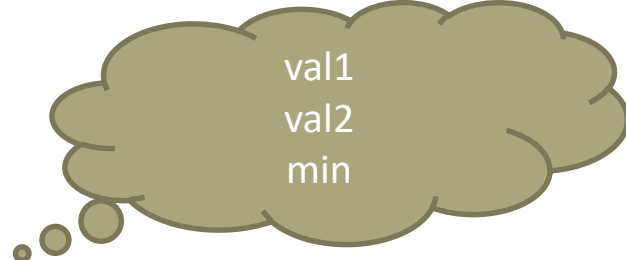
Notion de variable locale

Chaque programme a son espace mémoire à lui !



a
b
mini

```
int main ( )  
{ int a, b, mini ;  
  scanf(« %i », &a);  
  scanf(« %i », &b);  
  mini = minimum(a,b);  
  printf ( « min : %i \n», mini);  
  return 0;  
}
```

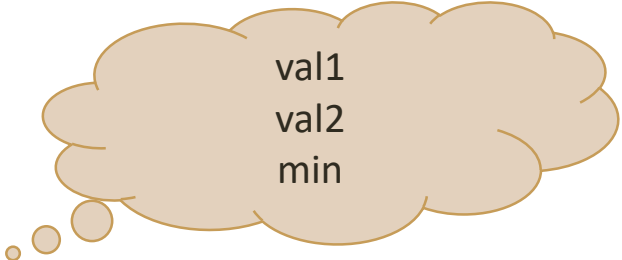


val1
val2
min

```
int  minimum ( int val1 , int val2)  
{  
  int min = val1 ;  
  if (val2 < val1)  
    min = val2 ;  
  return min ;  
}
```

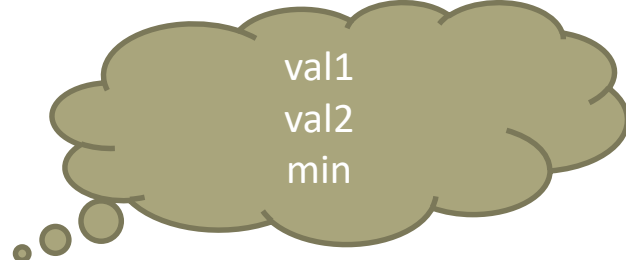
Notion de variable locale

On peut donner le même nom aux variables du sous-programme et programme appelant : il ne s'agira pas des mêmes !



val1
val2
min

```
int main ( )  
{  
    int val1, val2, min ;  
    scanf(« %i », &val1);  
    scanf(« %i », &val2);  
    min = minimum(val1,val2);  
    printf ( « min : %i \n», min);  
    return 0;  
}
```



val1
val2
min

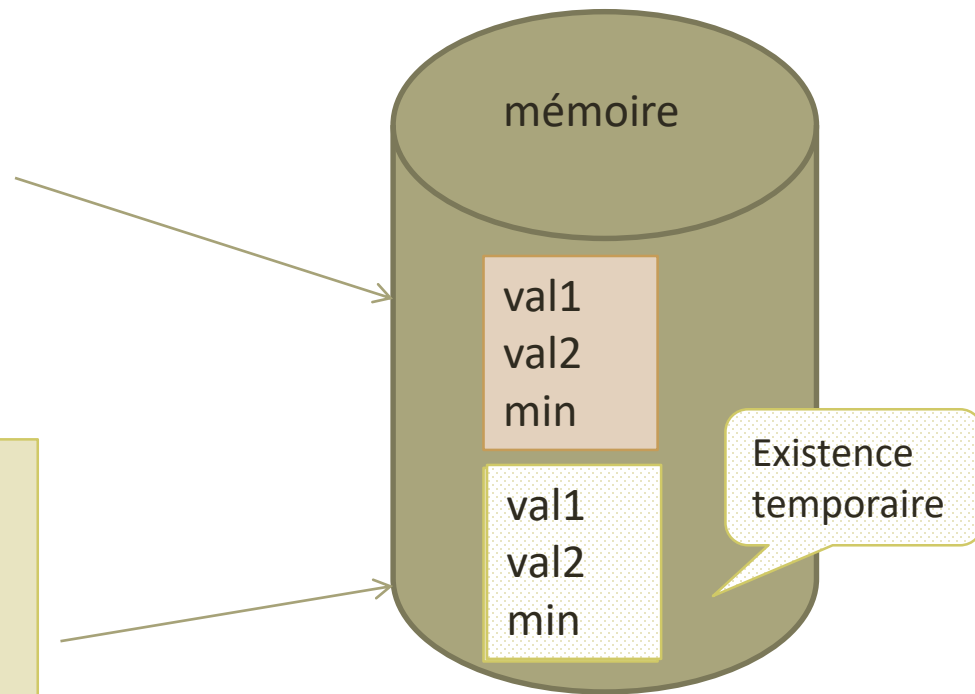
```
int  minimum ( int val1 , int val2)  
{  
    int min = val1 ;  
    if (val2 < val1)  
        min = val2 ;  
    return min ;  
}
```


Durée de vie des variables

Les variables liées aux fonctions n'existent que le temps d'exécution de la fonction.

```
int main ( )  
{  
    int val1, val2, min ;  
    scanf(« %i », &val1);  
    scanf(« %i », &val2);  
    min = minimum(val1,val2);  
    printf(« %i », min) ;  
}
```

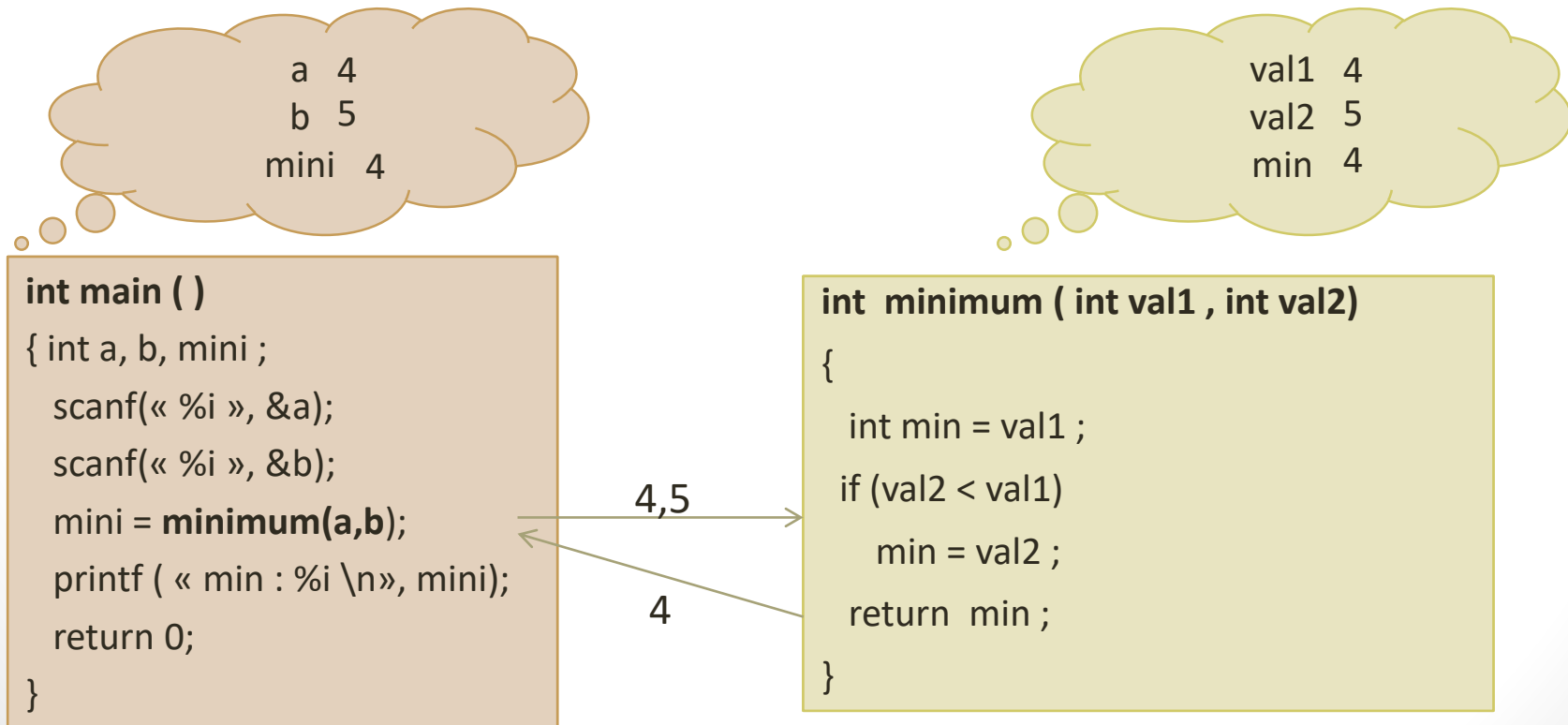
```
int minimum ( int val1 , int val2 )  
{  
    int min = val1 ;  
    if (val2 < val1)  
        min = val2 ;  
    return min ;  
}
```



Partage des données

Les données sont transférées pour être recopiées dans chaque espace mémoire :

- Lors du passage de paramètre : recopie dans l'espace mémoire de la fonction
- Lors du retour : recopie dans l'espace mémoire du programme appelant



Notion de retour

Une fonction :

- ne renvoie qu'une seule valeur à la fois ! Elle doit avoir un seul objectif !
- doit toujours retourner une valeur.

Conseil : sortez l'instruction de return et placez la une fois pour toute à la fin de votre fonction.

```
int minimum ( int val1 , int val2)
{
    if (val2 < val1)
        return val2 ;
    else
        return val1 ;
}
```



```
int minimum ( int val1 , int val2)
{
    int min ;
    if (val2 < val1)
        min = val2 ;
    else
        min = val1 ;
    return min ;
}
```



Notion de paramètre

Les paramètres effectifs (programme appelant) portent souvent le même nom que les paramètres formels, mais ce n'est pas une obligation !

Attention : un paramètre est une variable locale un peu spécifique :

- Elle n'est pas initialisée au sein du sous-programme, puisqu'elle est initialisée par défaut lors de l'appel du sous-programme.
- Elle n'est pas déclarée dans la zone de déclaration des variables locales, puisqu'elle est déclarée au sein de la signature

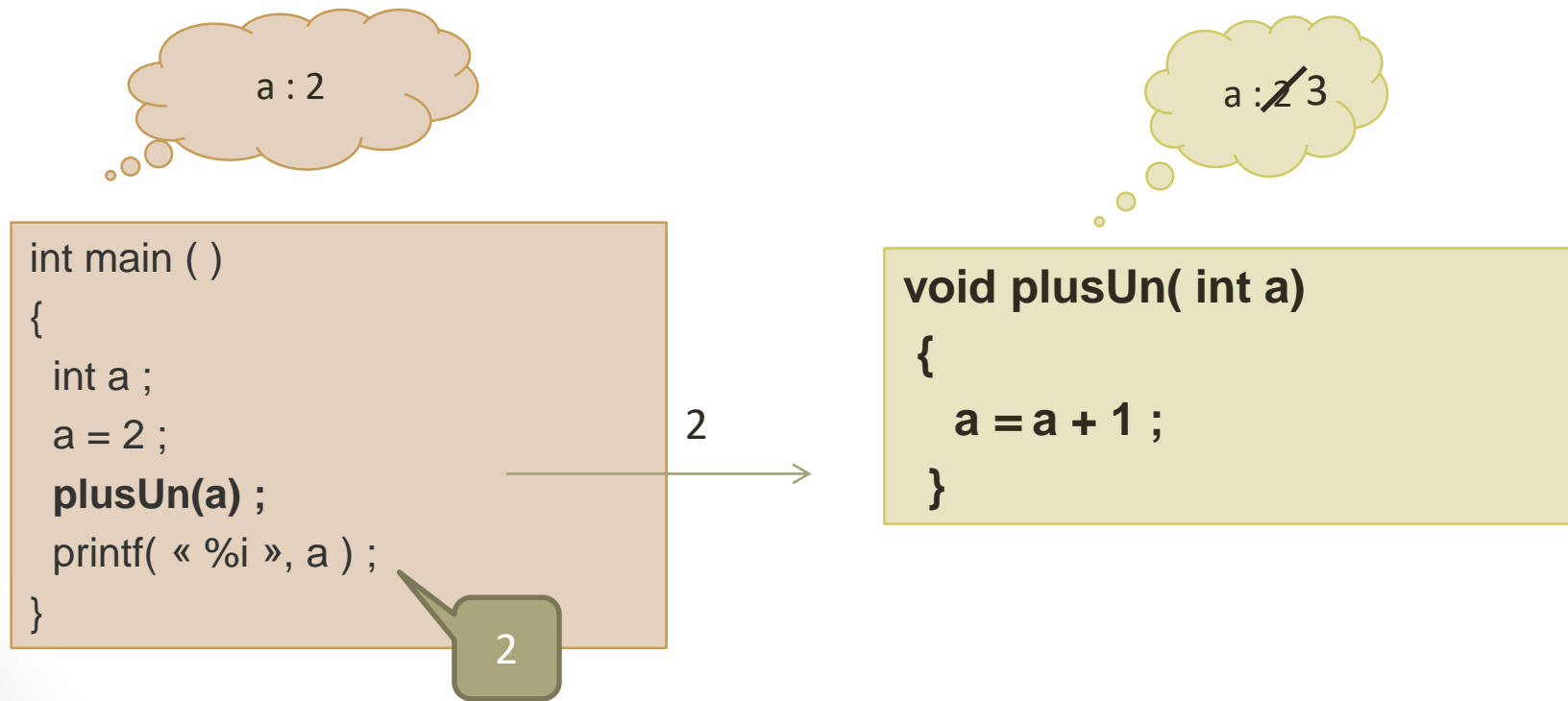
min n'est pas un paramètre

```
int minimum ( int val1 , int val2)
{
    int min ;
    if (val2 < val1)
        min = val2 ;
    else
        min = val1 ;
    return min ;
}
```

val1 et val2 initialisées
lors de l'appel: on les
manipule directement

Passage par valeur

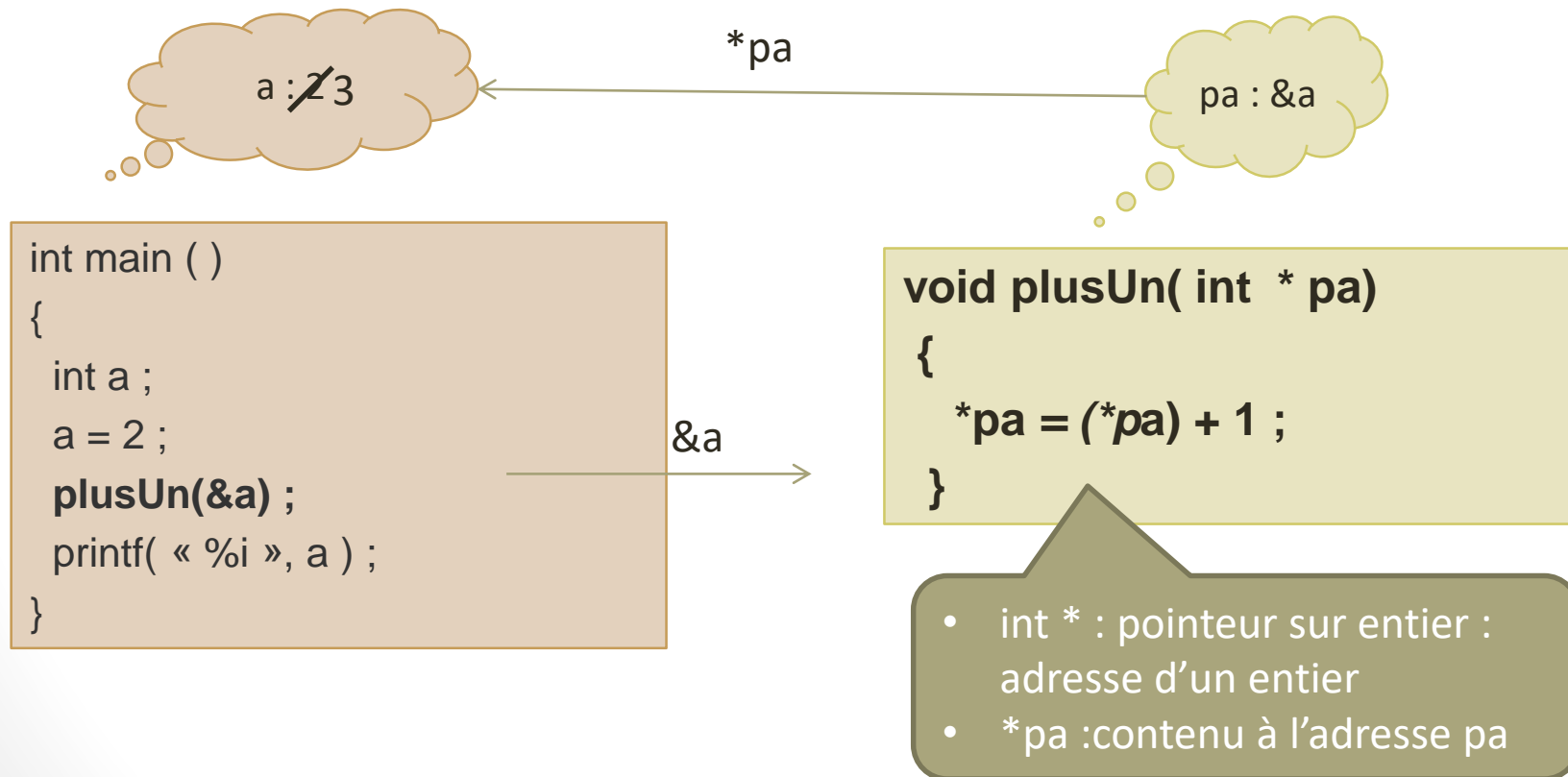
Modifier un paramètre formel ne sert à rien car on modifie alors une recopie de donnée qui n'existe qu'un court instant !



Passage par adresse

Ou comment modifier un paramètre durablement !

Un sous-programme peut modifier les variables du programme appelant si elles sont passées par adresse



Passage par adresse

Adresse ?

Toutes les variables ont une adresse en mémoire

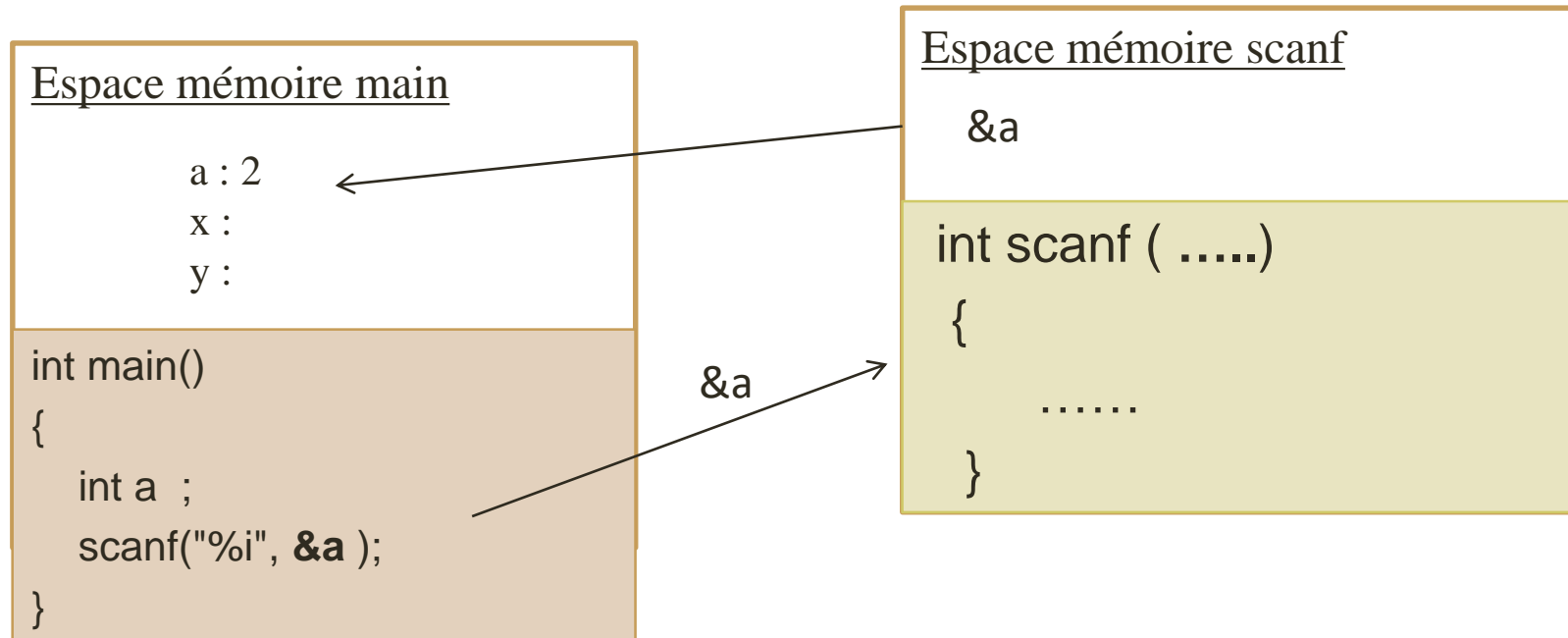
Autre exemple :

```
int main ()
{
    int a,b,c;
    printf("taille: %i \n",sizeof(a));
    printf("Adresse de a : %i \n",&a);
    printf("Adresse de b : %i \n",&b);
    printf("Adresse de c : %i \n",&c);
}
```

```
taille: 4
Adresse de a : 2293620
Adresse de b : 2293616
Adresse de c : 2293612
```

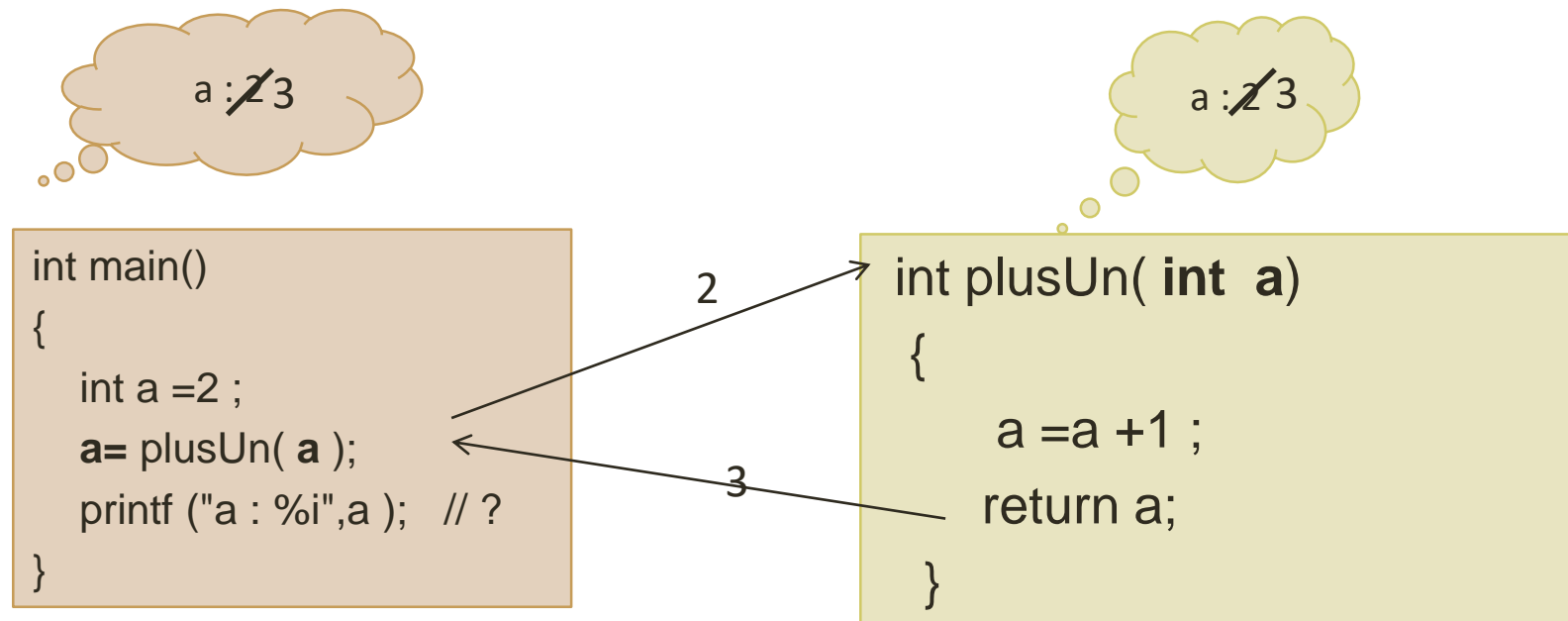
Passage par adresse

Un exemple bien connu : scanf !



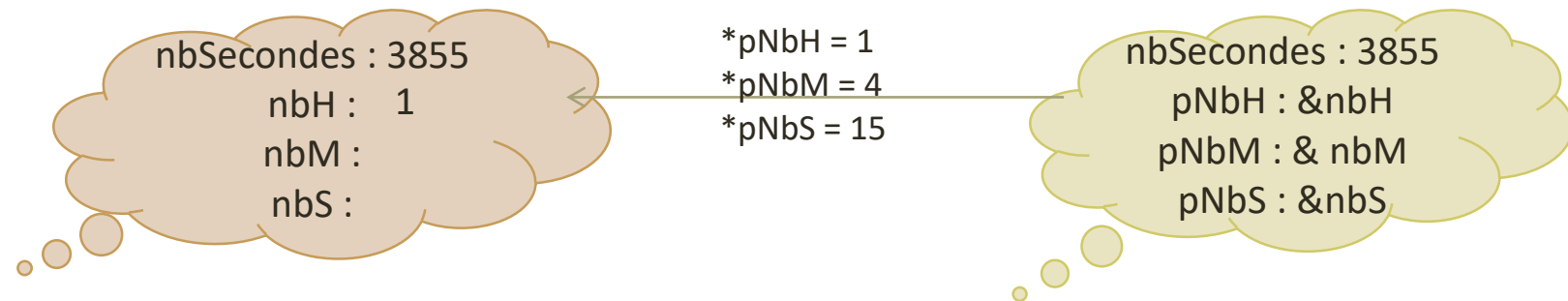
Passage par adresse

On peut souvent s'en passer



Passage par adresse

Pour retourner plusieurs valeurs alors que c'est impossible !



```
int main()
{
    int nbSecondes = 3855;
    int nbH, nbM, nbS ;
    convertSecondesEnHMS (
        nbSecondes, &nbH, &nbM, &nbS);
}
```

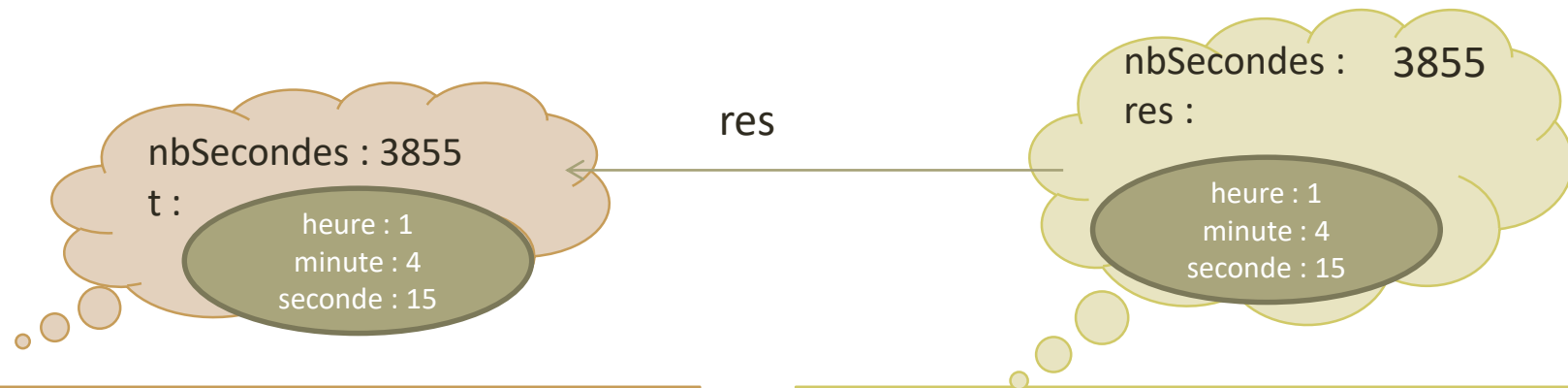
3855, &nbH,
&nbM, &nbS

```
void convertSecondesEnHMS
( int nbSecondes, int * pNbH, int * pNbM, int
  * pNbS)
{
    *pNbH = nbSecondes / 3600;
    int reste = nbSecondes % 3600;
    *pNbM = reste / 60 ;
    *pNbS = reste % 60 ;
}
```

Sous programmes et variables structurees

Retourner une variable structurée

Avec une structure, c'est possible de retourner plusieurs résultats !



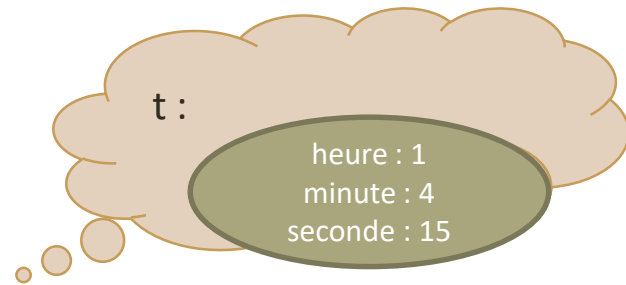
```
int main()
{
    int nbSecondes = 3855;
    int nbH, nbM, nbS ;
    Temps t = convertSec( nbSecondes);
}
```

3855

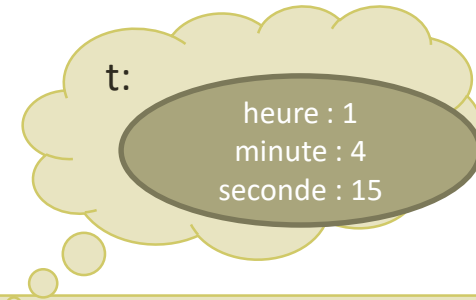
```
Temps convertSec ( int nbSecondes)
{
    Temps res ;
    res.heure = nbSecondes / 3600;
    int reste = nbSecondes % 3600;
    res.minute = reste / 60 ;
    res.seconde = reste % 60 ;
    return res ;
}
```

Variable structurée en paramètre

Passage par valeur par défaut



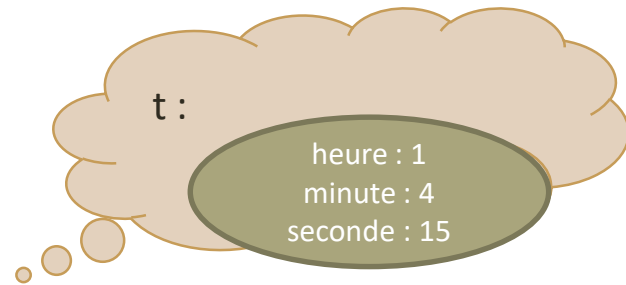
```
int main()
{
    Temps t = {1,4,15} ;
    afficheTemps( t );
}
```



```
void afficheTemps( Temps t )
{
    if ( t.heure <=9 )
        printf ( «0» ) ;
    printf ( «%i : »,t.heure ) ;
    if ( t.minute <=9 )
        printf ( «0» ) ;
    printf ( « %i:», t.minute ) ;
    if ( t.seconde <=9 )
        printf ( «0» ) ;
    printf ( « %i:», t.seconde ) ;
}
```

Variable structurée en paramètre

Passage par adresse possible



```
int main()  
{  
    Temps t = {1,4,15} ;  
    PlusUneSeconde( &t );  
}
```

&t

ptps->seconde ++;
(*ptps).seconde ++;

ptps: &t

```
void PlusUneSeconde( Temps * ptps )  
{  
    ptps -> seconde ++ ;  
    if ( ptps -> seconde == 60 )  
    {  
        ptps -> seconde =0;  
        ptps -> minute ++ ;  
        if ( ptps -> minute == 60 )  
        {  
            ptps -> minute = 0;  
            ptps -> heure ++ ;  
            if ( ptps -> heure == 24 )  
                ptps -> heure = 0;  
        }  
    }  
}
```

Structures et fonctions

Pour toute nouvelle structure, il est bien définir les opérations élémentaires pour pouvoir la manipuler.

Il faut donc définir des sous-programmes pour:

- **Afficher une variable structurée .**
 - Ex: void afficheTemps (Temps t)
- **Saisir une variable structurée .**
 - Ex: Temps saisieTemps ()
- **Comparer 2 variables structurées.**
 - Ex: int TempsCmp (Temps t1, Temps t2) : elle renvoie:
 - 0 si les 2 temps sont identiques
 - > 0 si le 1^{er} est > au 2eme
 - <0 si le 1^{er} est < au 2eme

À la manière de
strCmp !

Sous programmes et tableaux

Fonctions et tableaux

Un tableau est en fait un pointeur : contient l'adresse de la zone allouée pour stocker toutes les données !

```
int main()
{
    int tab[4] = {1,2,3,4};
    printf("%i\n", tab) ;
    printf("%i\n", &tab[0]) ;
    printf("%i\n", &tab[1]) ;
    printf("%i\n", &tab[2]) ;
    printf("%i\n", tab[0]) ;
    return 0;
}
```

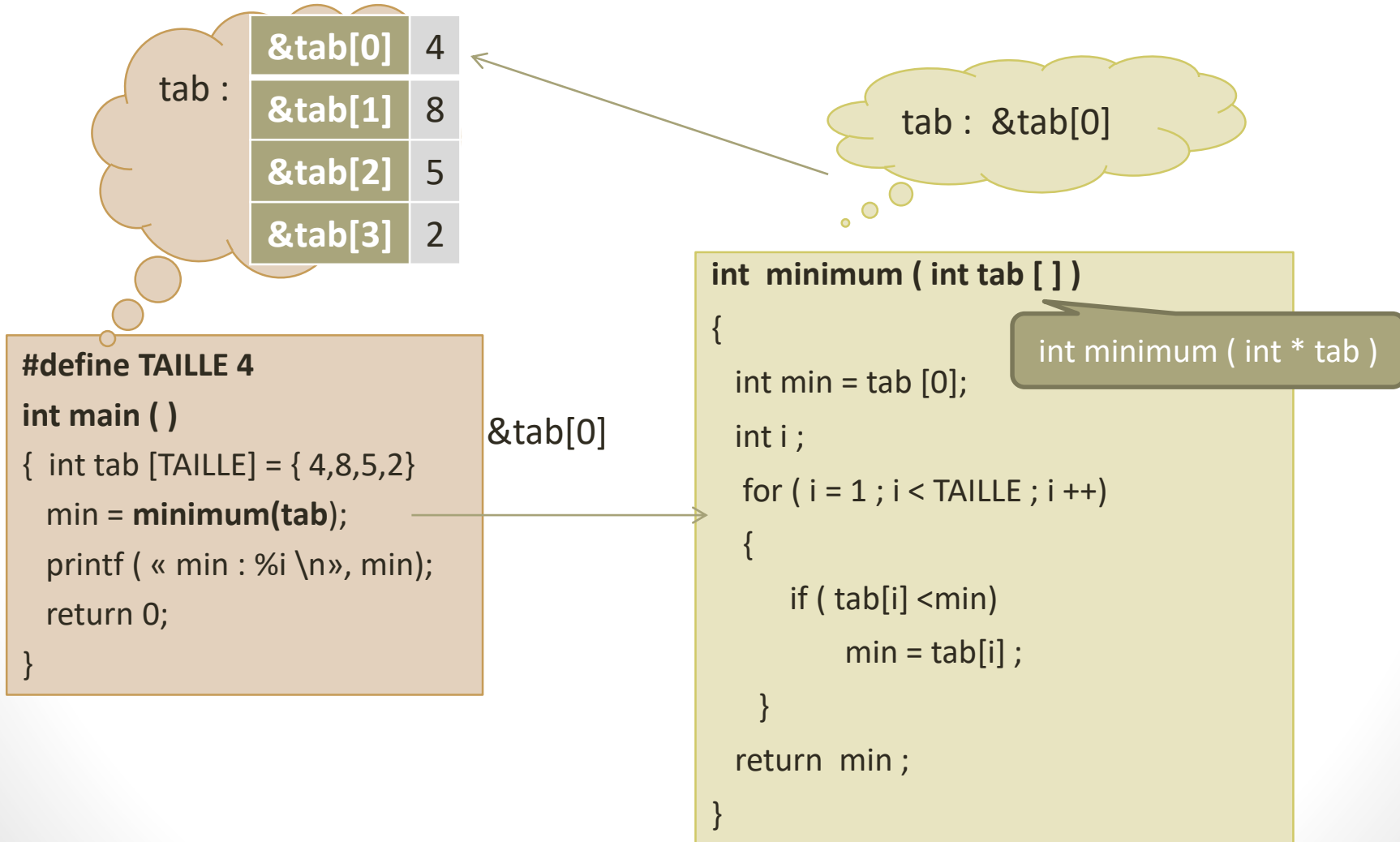
6356736
6356736
6356740
6356744
1

tab ⇔ &tab[0]

&tab[0]	1
&tab[1]	2
&tab[2]	3
...	

Fonctions et tableaux

Passer un tableau en paramètre => passer un pointeur = adresse.
Il n'y a pas de copie.



Fonctions et tableaux

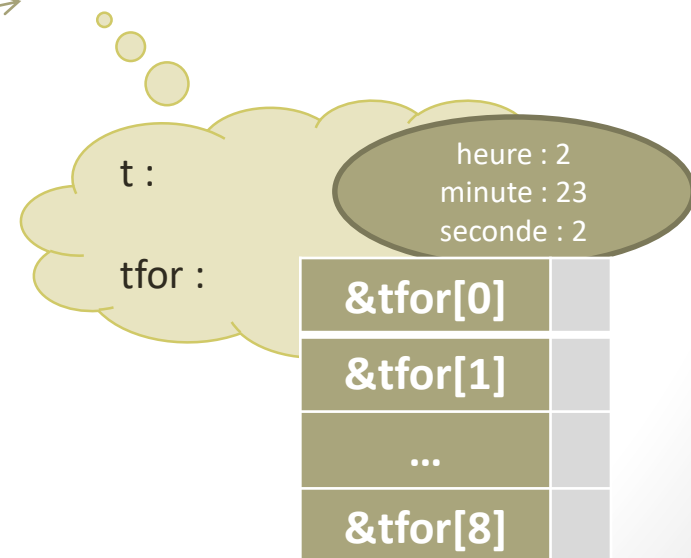
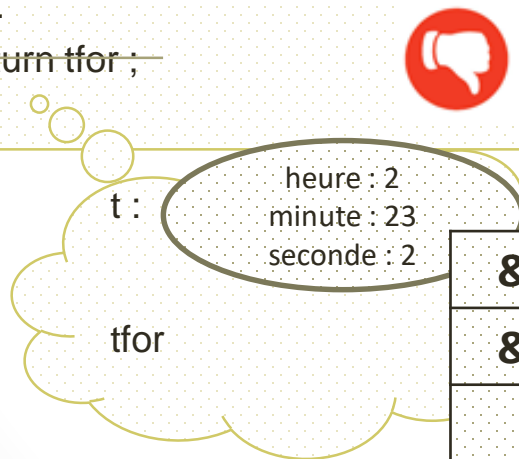
Une fonction ne doit pas retourner un tableau déclaré localement à la fonction !

Ex ici : on veut faire une fonction qui retourne un temps au format »01:09:22 «

```
char [ ] formateTemps( Temps t )
{
char tfor [9];
  if ( t.heure <=9 )
    sprintf (tfor , «0%i»,t.heure) ;
  else
    sprintf (tfor , «%i»,t.heure) ;
  ....
return tfor ;
}
```

```
int main ()
{
  Temps t = { 2,23,2};
  char tfor [ 9 ];
  tfor= formateTemps(t);
}
```

&tfor[0] →



Fonctions et tableaux

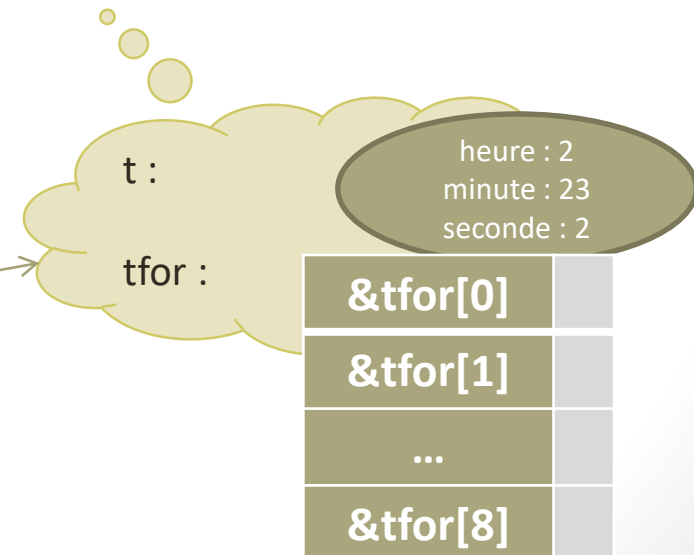
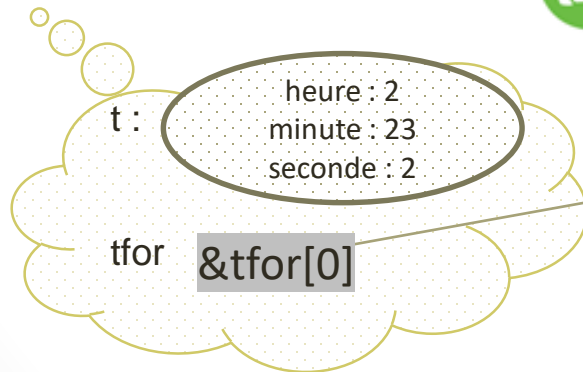
Passer un tableau en paramètre pour stocker le résultat : étrange !

Une fonction ne doit pas retourner un tableau local à la fonction !

```
void formateTemps( Temps t , char tfor [ ] )
{
    if ( t.heure <=9 )
        sprintf (tfor , «0%i»,t.heure) ;
    else
        sprintf (tfor , «%i»,t.heure) ;
    ....
}
```



```
int main ()
{
    Temps t = { 2,23,2};
    char tfor[ 9 ];
    formateTemps(t, tfor);
}
```



Fonctions et chaînes

Les chaînes étant avant tout des tableaux, passez une chaîne, c'est passer l'adresse de la 1ere case du tableau dédié à la chaîne

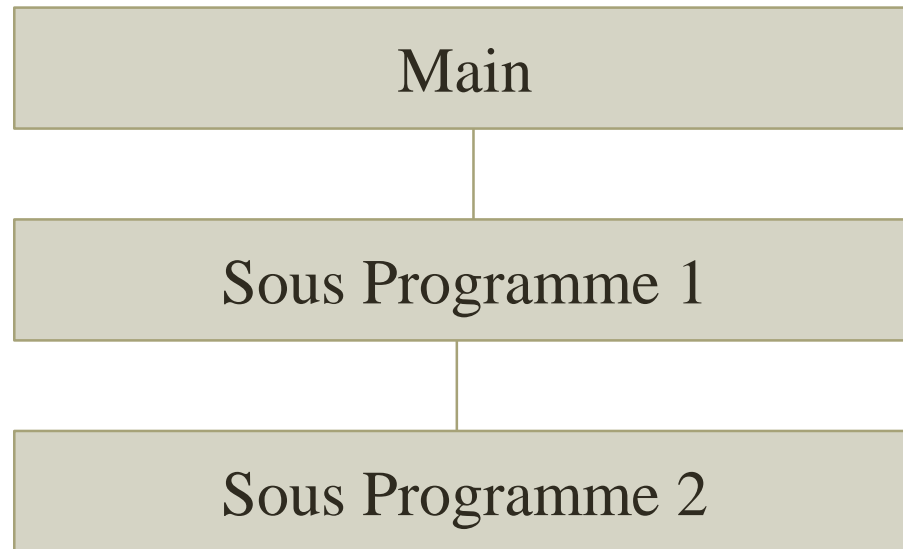
```
char chaine[50] ;  
scanf(« %s », chaine) ;
```

chaine ⇔ &chaine[0]

Ce qui explique que l'on omette le &
pour une chaîne de caractère

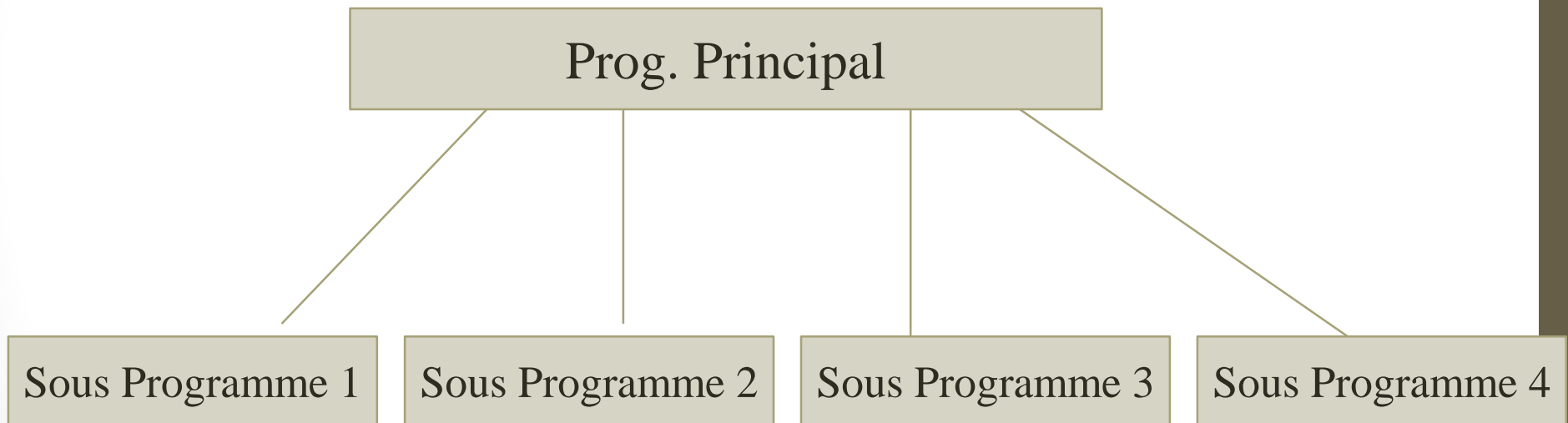
Imbrication de sous programmes

On peut définir un sous programme qui s'appuie sur un autre sous programme. Mais pour un bon code, il faut éviter plus de 2 niveaux d'imbrication.



Appels successifs de sous programmes

Pour plus de lisibilité et de simplicité, mieux vaut préférer des solutions avec des appels successifs !



Règles de bon sens

- Le sous-programme fonction ou procédure n'a pas à vérifier les valeurs passées en paramètre. C'est le travail du programme appelant.
- Bien différencier les sous-programmes dédiés :
 - à la vue : pouvant contenir des printf/scanf
 - à la logique : contenant uniquement des règles, des calculs, des traitements

Où définir les sous-programmes

Un sous-programme doit être déclaré avant d'être utilisé.

```
int addition( int a , int b)
{
    int res = a + b ;
    return res;
}
```

```
int main( )
{
    int nb1=12,nb2=23;
    int r = addition(nb1,nb2);
    printf("%i\n",r);
    system("pause");
    return 0;
}
```

Où définir les sous-programmes

On préférera ne mettre que les entêtes (signatures) des fonctions au début pour atteindre le cœur du programme plus vite

```
int addition( int a , int b ) ;  
int main( )  
{  
    int nb1=12,nb2=23;  
    int r = addition(nb1,nb2);  
    ...  
}  
int addition( int a , int b)  
{ int res = a + b ;  
    return res;  
}
```

Où définir les sous-programmes

Remarque : il est possible de découper notre code physiquement

calcul.h

```
int addition( int a , int b );
```

Calcul.c

```
#include "calcul.h"
int addition( int a , int b )
{ int res = a + b ;
  return res;
}
```

main.c

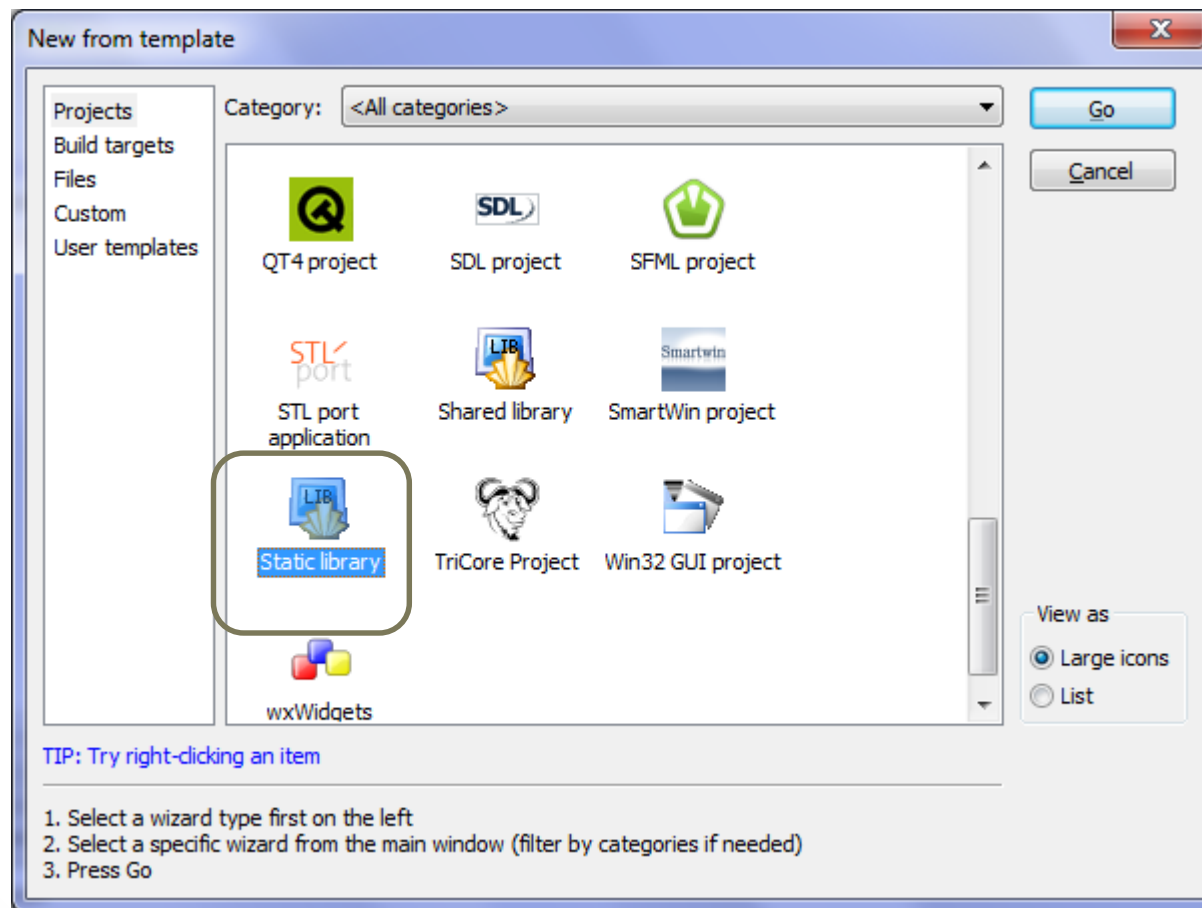
```
#include "calcul.h"
int main(int argc, char *argv[])
{
    int nb1=12,nb2=23;
    int r = addition(nb1,nb2);
    printf("%i\n",r);
    system("pause");

    return 0;
}
```



Où définir les sous-programmes

- Vous pouvez aussi faire une librairie statique ou dynamique pour réutiliser vos fonctions ! (tout comme SFML)



Sous programmes et sfml (C++)

Passage d'objet en paramètre

Par valeur

Les objets (RectangleShape, RenderWindow,...) sont passés par défaut par valeur. Ils sont recopiés à l'aide d'un mécanisme :

- **De construction**
- **De destruction...**

Les modifier ne sert à rien, car la modif a lieu sur une copie temporaire.

Passage d'objet en paramètre

Par référence

Du coup, vous avez intérêt à passer les objets par référence (⇔ par adresse) :

- pour un gain de temps (pas de recopie)
- Pour pouvoir modifier les objets.

+ simple qu'en C !!
Un simple & à mettre
dans la signature

// Ici passage par & en C ++

```
void changeCouleur ( RectangleShape &rect)
{ rect.setFillColors( rgb(random()%255, random()%255, random()%255);
```

```
int main( )
{
    RectangleShape r(Vector2f(200,50));
    changeCouleur ( r );
}
```

// ici passage par & en C

```
void sousprog ( int *var)
{ *var = 2 ; }
```

```
int main()
{ int v;
  sousprog(&v);}
```