Cours informatique : Maîtrise de Sciences Cognitives 2002-2003

Session 2 :

Structures de données : manipuler des structures hétérogènes

Déclaration, définition, exemple d'utilisation de structures (struct)

Illustration de l'organisation en mémoire (struct)

Comment utiliser la même zone mémoire selon deux points de vue différents (union)

Illustration de l'organisation en mémoire (union)

Fonctions : modulariser un programme

Déclaration, défintion, exemples de fonctions simples

Illustration des empilements et dépilements en mémoire au moment des appels et des retours de fonction

Fonctions récursives : définition, exemples

Illustration du mécanisme de gestion de pile pour gérer la récursivité

Discussion sur l'art et la manière d'utiliser les fonctions

Structures de données : manipuler des structures hétérogènes

Les structures de données `TABLEAU` et `CHAINE DE CARACTERES` sont des structures homogènes, c'est-à-dire que chaque élément de la structure est de même type. On peut avoir des tableaux d'entiers, de réels, de caractères mais pas de tableaux où le premier élément serai un entier, le second une chaîne de caractère, le troisième un réel et ainsi de suite. C'est pourtant bien utile de pouvoir manipuler une structure hétérogène de ce type pour désigner un ensemble de propriétés différentes d'un même objet. Par exemple, un étudiant impliqué dans une expérimentation pourrait être identifié par :

- son numéro dans le panel (un entier positif),
- son prénom (une chaîne de caractère),
- son âge (un entier positif),
- le score obtenu à l'expérimentation (un réel)

Le langage C permet de regrouper ce type de données avec le type `STRUCT`.

```
struct MON_PREMIER_TYPE_DE_STRUCTURE{int NUM_PANEL, char PRENOM[20], int AGE, float SCORE} MA_PREMIERE_STRUCTURE;
```

Le mot clé `struct` appartient au langage C ; `MON_PREMIER_TYPE_DE_STRUCTURE` est le nom que le programmeur donne à ce type de structure. Entre `{}`, on trouve les différents "champs" de la structure tels que le programmeur souhaite les déclarer (il peut bien sûr y avoir des champs structurés !) ; `MA_PREMIERE_STRUCTURE` est le nom de la variable correspondant à cette structure. L'accès à un champ se fait en suffixant le nom du champ au nom de la variable structurée (avec un "." entre les deux noms).

Par exemple `MA_PREMIERE_STRUCTURE.NUM_PANEL` désigne le premier champ de cette structure.

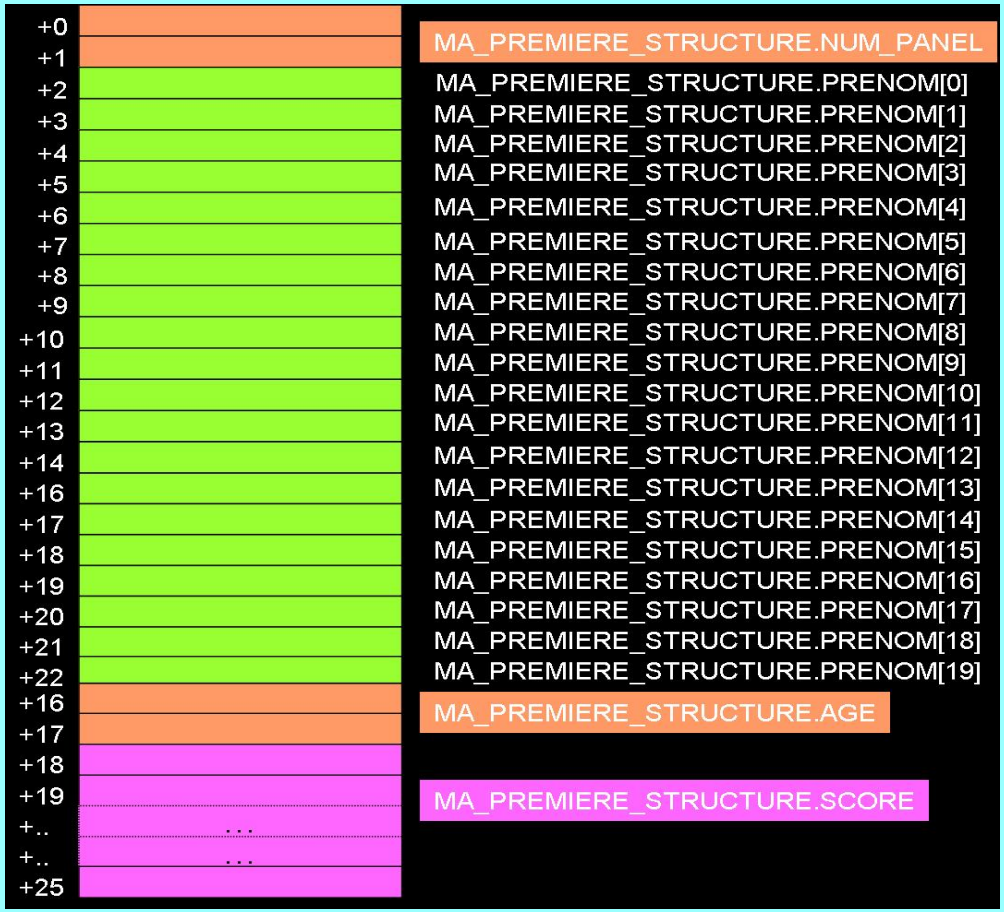
Illustration sur un code source :

```
#include
int main ()
```

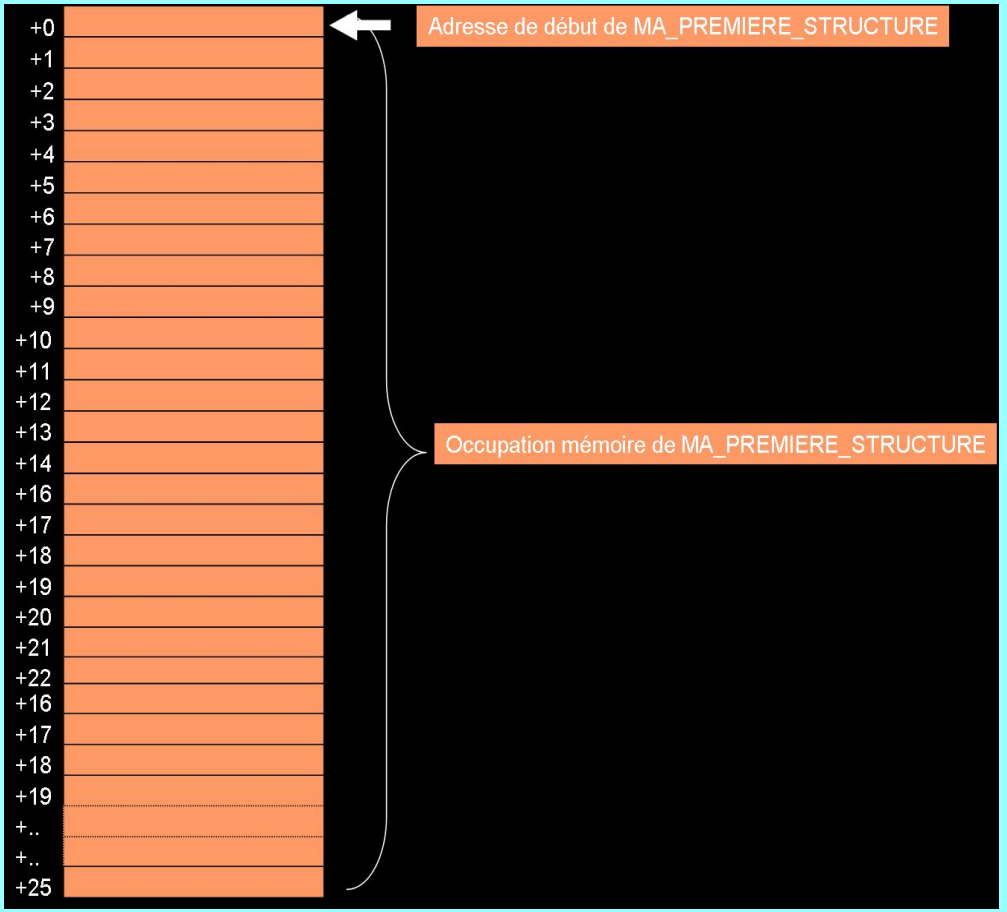
```
{
struct mon_premier_type_de_structure{int num_panel; char prenom[20]; int age; float score;} ma_premiere_structure ;
ma_premiere_structure.num_panel = 12;
strcpy(ma_premiere_structure.prenom,"alain");/* la fonction strcpy(arg1,arg2) copie la chaine contenue à l'adresse arg2 à l'adresse
arg1*/
ma_premiere_structure.age = 20; /* quand on aime...*/
ma_premiere_structure.score = 18.5;
printf(" %s qui est age de %i ans appartient au panel numero %i et a reussi le score de %f /n", ma_premiere_structure.prenom,
ma_premiere_structure.age, ma_premiere_structure.num_panel, ma_premiere_structure.score);
return 0;
}
```

Illustration de l'occupation mémoire de cet exemple :

Détail d'occupation des champs



Occupation globale de la structure



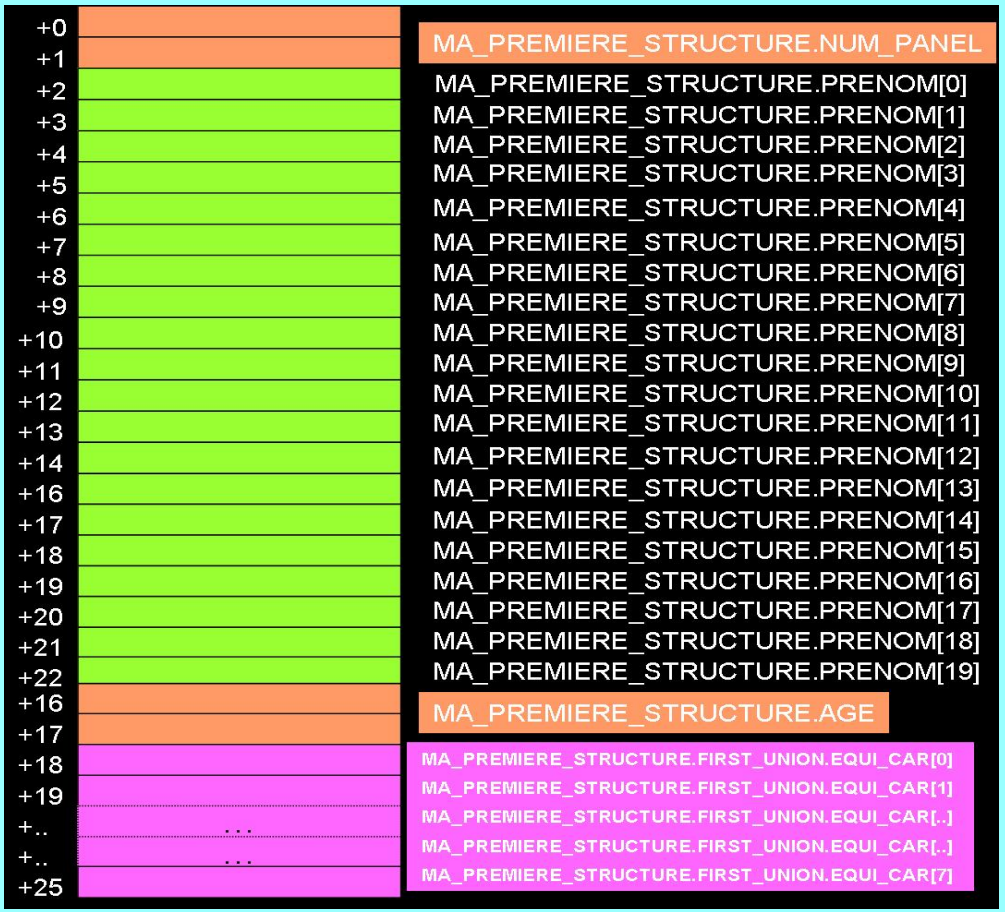

```
#include <stdio.h>

int main ()

{
struct mon_premier_type_de_structure{int num_panel; char prenom[20]; int age; union {float score;char equiv_car[8];} first_union;}
ma_premiere_structure ;
```

```
int i;

ma_premiere_structure.num_panel = 12;

strcpy(ma_premiere_structure.prenom,"alain");

ma_premiere_structure.age = 20; /* quand on aime...*/

ma_premiere_structure.first_union.score = 18.5;


printf(" %s qui est age de %i ans appartient au panel numero %i et a reussi le score de %f /n", ma_premiere_structure.prenom,
ma_premiere_structure.age, ma_premiere_structure.num_panel, ma_premiere_structure.first_union.score);

for (i=0;i<8;i++)
printf("equiv_car [ %i ] = %x ",i, ma_premiere_structure.first_union.equiv_car[i]);/* %x exprime une valeur en base 16
(hexadecimal)*/

return 0;
}
```

Le schéma de la figure suivant illustre ces deux "points de vue" sur la même zone mémoire.

Détail d'occupation des champs : point de vue "first_union.score"

Détail d'occupation des champs : point de vue "first_union.equiv_car"



Fonctions : modulariser un programme

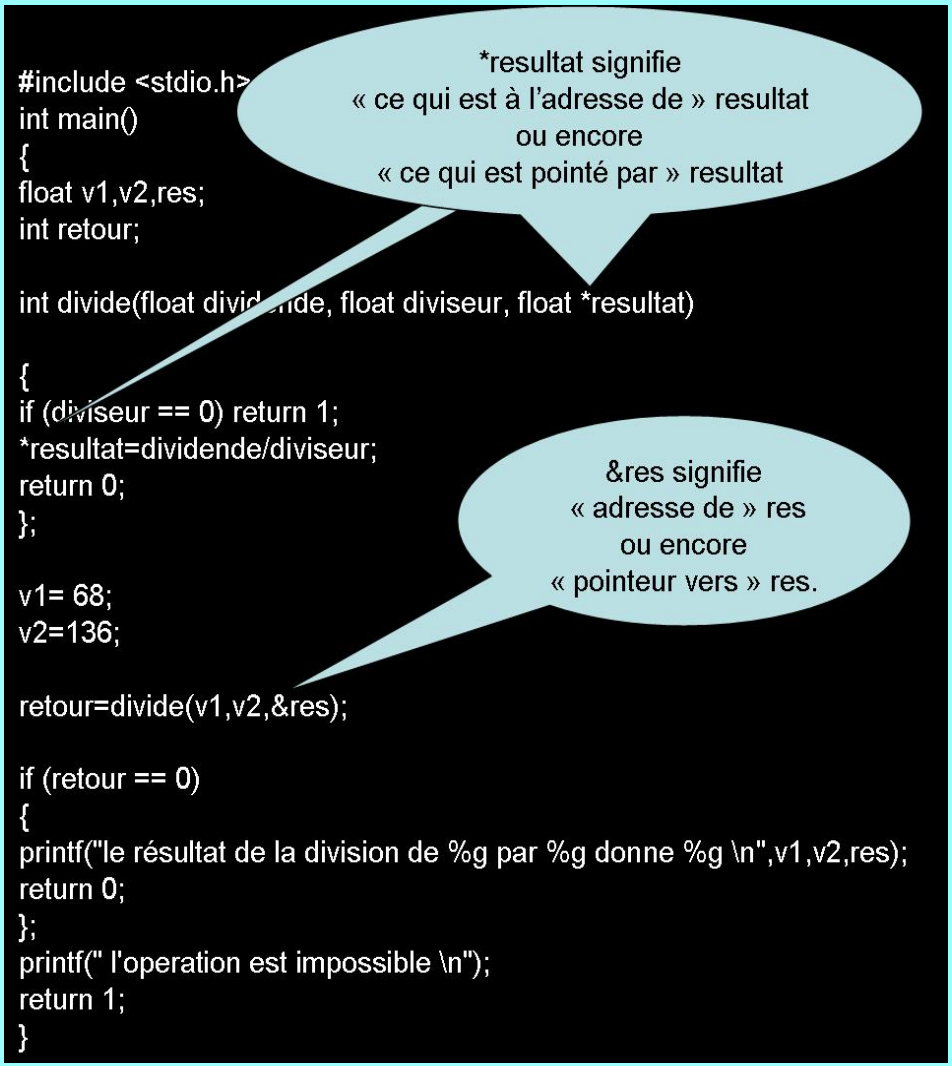
Un programme en C est constitué d'une fonction "principale". Quand un programme commence à devenir complexe, ou quand on veut pouvoir réutiliser des séquences de traitement d'un programme à un autre, on a tout intérêt à "encapsuler" les instructions concernées dans une "fonction" nouvelle qui pourra être utilisée en lui fournissant un certain nombre de paramètres pour son exécution et en récupérant le résultat pour continuer le traitement. Le langage C propose déjà tout un ensemble de fonctions "préconstruites" pour traiter "les entrées-sorties" (printf, getc, putc, etc.) ou les traitements de chaîne de caractères (strcpy, etc..).

Une fonction s'écrit à peu près de la même façon que la fonction principale (voir la définition d'une fonction divide dans l'exemple suivant);:

Un premier programme avec une fonction simple

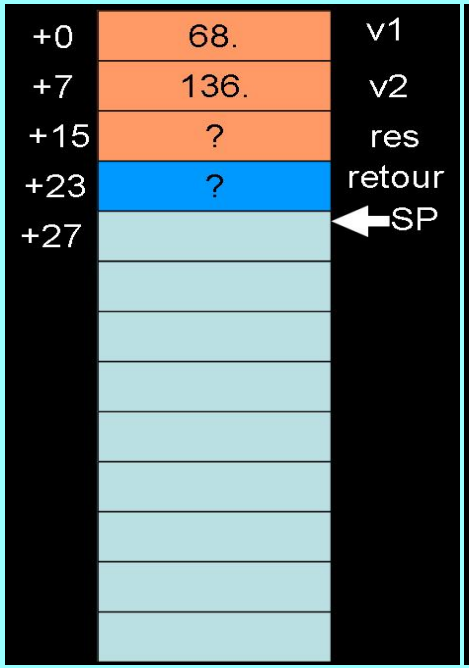
```
#include <stdio.h>

int main()

{
float v1,v2,res;
int retour;

int divide(float dividende, float diviseur, float *resultat)

/*déclaration et définition de la fonction divide */

{
if (diviseur == 0) return 1;
*resultat=dividende/diviseur;

/* le résultat de la division est copié à l'adresse de la variable
resultat*/
return 0;
};

v1= 68;
v2=136;


retour=divide(v1,v2,&res);/* appel de la fonction divide */

if (retour == 0)
{
printf("le résultat de la division de %g par %g donne %g
\n",v1,v2,res);
return 0;
};


printf(" l'operation est impossible \n");
return 1;
}
```

Notion de pointeur et d'adressage indirect

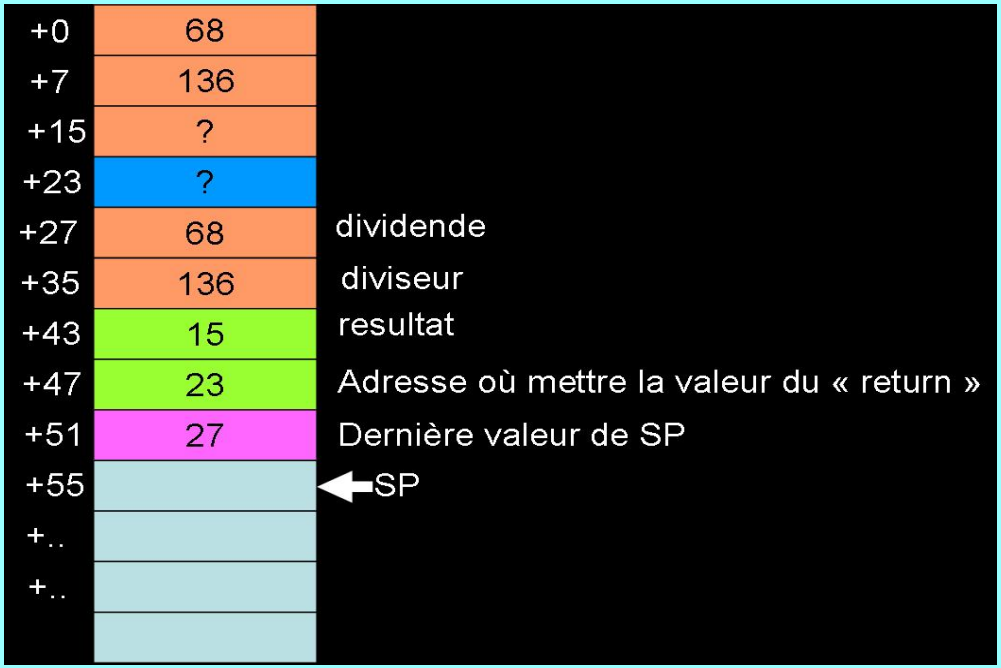


Une fonction est "étanche" aux variables de la fonction qui l'appelle. Pour que la fonction appelante ("main" dans notre exemple) communique avec la fonction appelée ("divide" dans notre exemple), il est nécessaire de passer par l'intermédiaire de "paramètres d'appels". L'appel de la fonction sera l'occasion de réaliser la copie des paramètres d'appels dans les zones mémoires allouées pour que la fonction puisse disposer d'un espace propre pour son exécution. Le schéma suivant résume les différentes étapes de l'appel, de l'exécution et de la fin de l'exécution d'une fonction (sur l'exemple de la fonction "divide").

L'état de la pile juste avant l'appel

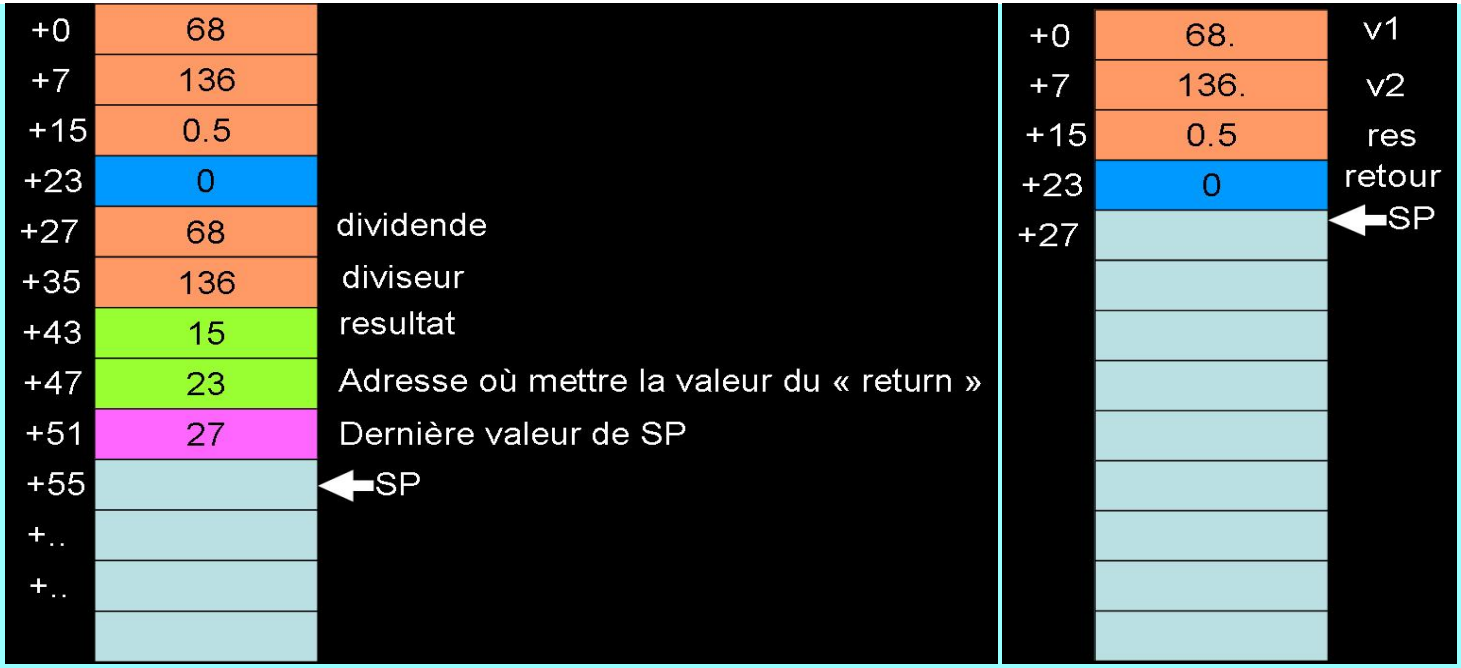


L'état de la pile au début de l'exécution de la fonction



L'état de la pile juste avant la fin de la fonction

L'état de la pile au retour dans la fonction principale



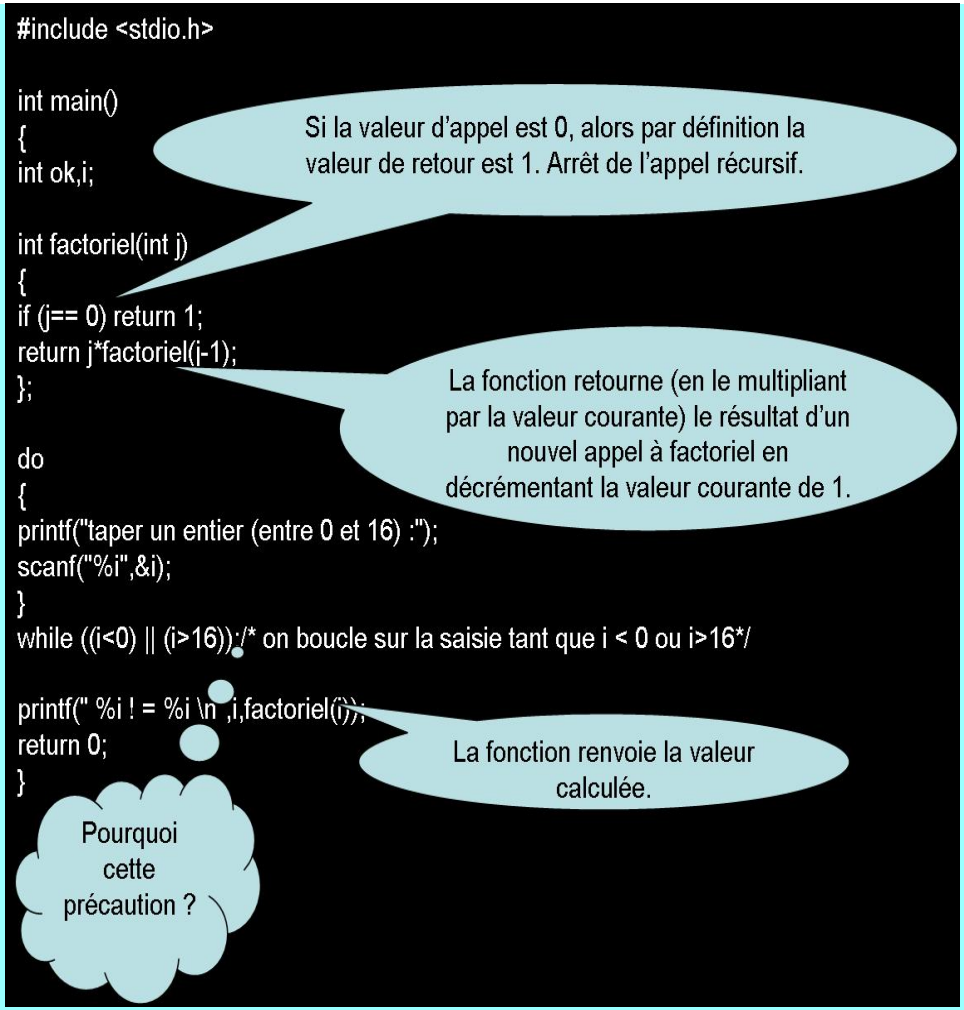
Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même. En pratique, la fonction porte bien le même nom et ce sera le même code qui sera exécuté, MAIS PAS SUR LES MEMES DONNEES. C'est le mécanisme de gestion de la pile tel qu'il vient d'être vu plus haut qui permet ce type de fonctionnement. Bien entendu, il convient que l'appel récursif s'arrête, permettant aux fonctions de revenir dans l'appelant et ceci jusqu'à revenir dans la fonction principale. Il est habituel de donner comme exemple de l'usage d'une fonction récursive le calcul d'une récurence. Par exemple, le calcul de "factoriel". On rappelle que n!=(n-1)!n, avec n>=0 et 0!=1 par convention (la notation n! se lit "factoriel n"). Le programme suivant est un programme permettant de calculer la factorielle d'un nombre entier entré au clavier.

Exemple de code source avec une fonction récursive **A observer !**

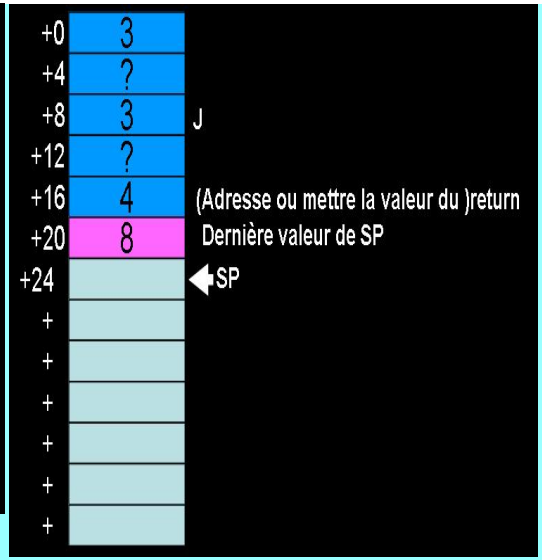
```
#include <stdio.h>

int main()

{
int _i;


int factoriel(int j)
{
if (j== 0) return 1;
return j*factoriel(j-1);
};
do
{
printf("taper un entier (entre 0 et 16) :");
scanf("%i",&i);
}
while ((i<0) || (i>16));/* on boucle sur la saisie tant que
```

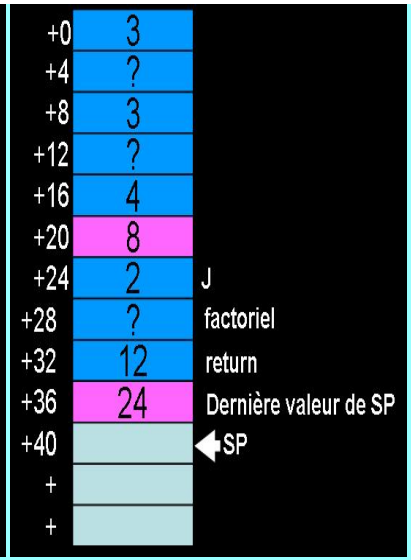
```
i < 0 ou i>16*/
printf(" %i ! = %i \n",i,factoriel(i));
return 0;
}
```

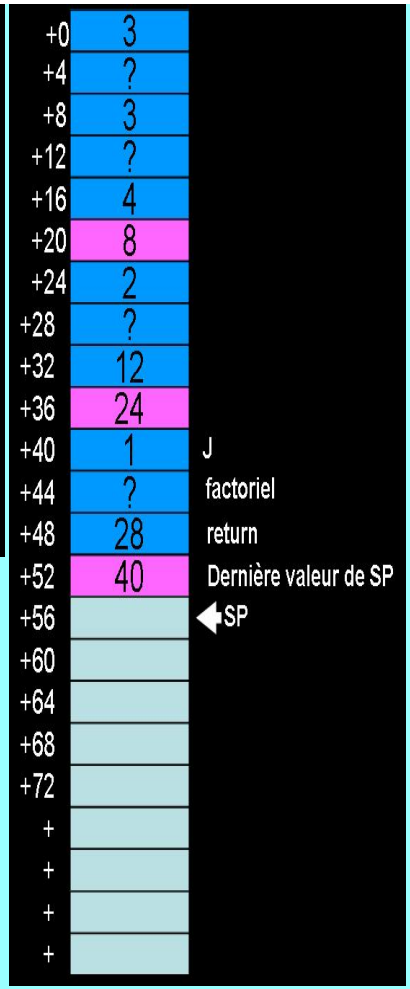


Les schémas suivants illustrent les différentes étapes d'empilement et de dépilement qui correspondent à l'exécution de ce programme pour une valeur de 3.

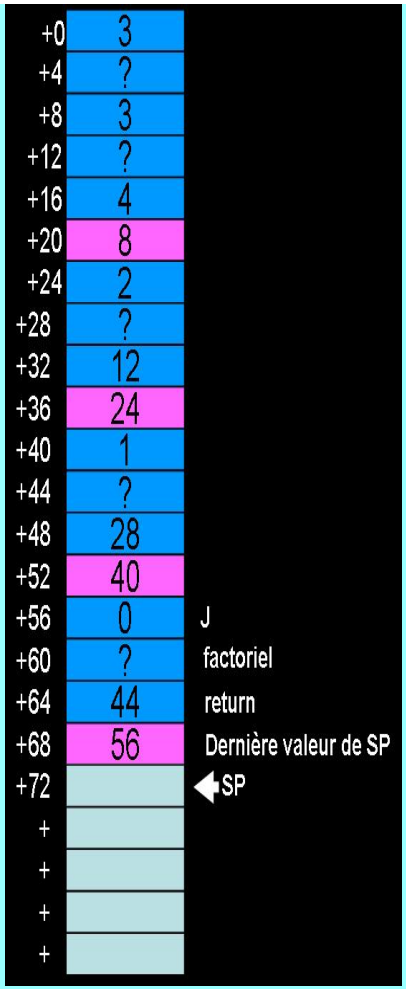
Appel factoriel(3) **Appel factoriel(2)** **Appel factoriel(1)** **Appel factoriel(0)**

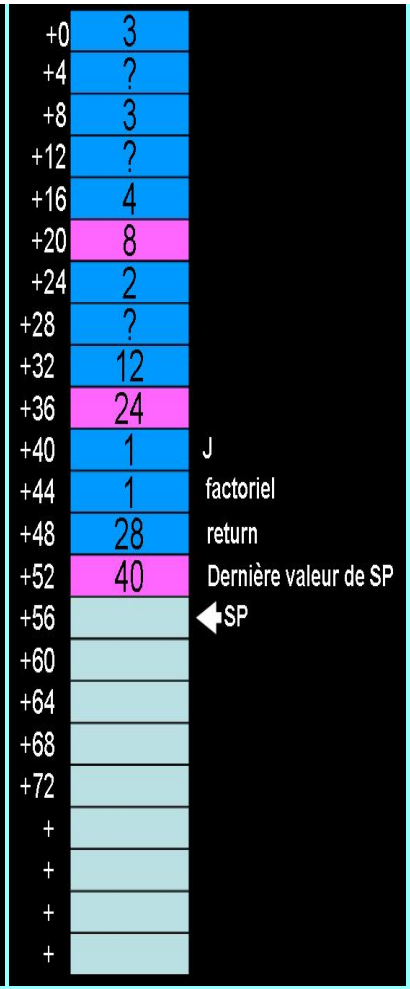


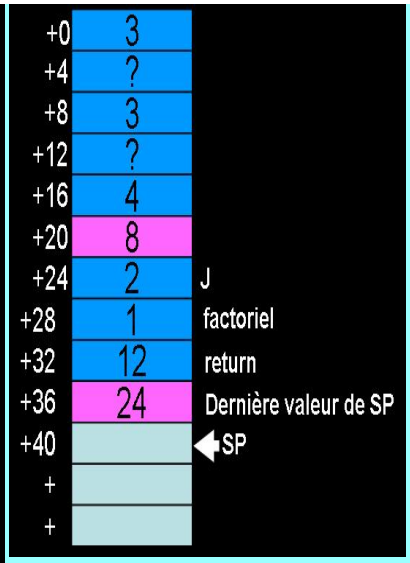


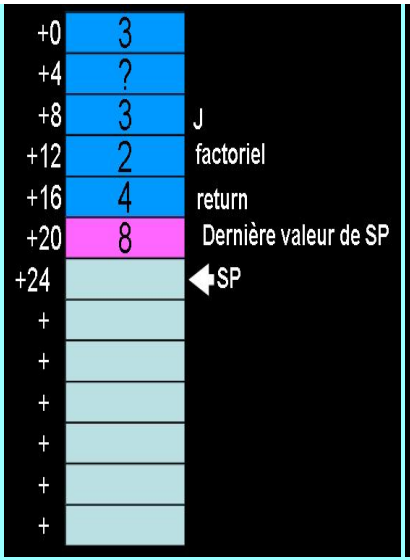


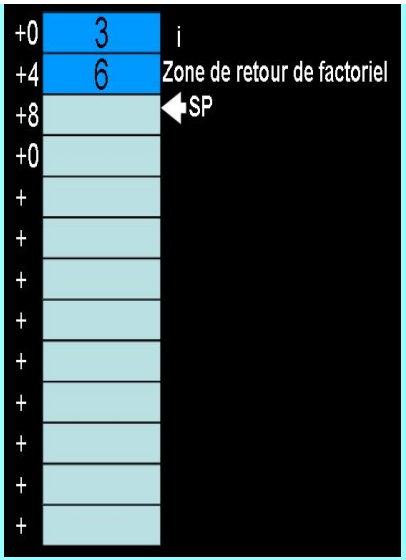
Affectation de 1 dans factoriel(0) **Retour dans factoriel(1)** **Retour dans factoriel(2)** **Retour dans factoriel(3)** **Retour (final) dans main()**


+0 3
+4 ?
+8 3
+12 ?
+16 4
+20 8
+24 2
+28 ?
+32 12
+36 24
+40 1
+44 ?
+48 28
+52 40
+56 0
J
+60 ?
factoriel
+64 44
return
+68 56
Dernière valeur de SP
+72
SP
+
+
+
+


+0 3
+4 ?
+8 3
+12 ?
+16 4
+20 8
+24 2
+28 ?
+32 12
+36 24
+40 1
J
+44 1
factoriel
+48 28
return
+52 40
Dernière valeur de SP
+56
SP
+60
+64
+68
+72
+
+
+
+


+0 3
+4 ?
+8 3
+12 ?
+16 4
+20 8
+24 2
J
+28 1
factoriel
+32 12
return
+36 24
Dernière valeur de SP
+40
SP
+
+


+0 3
+4 ?
+8 3
J
+12 2
factoriel
+16 4
return
+20 8
Dernière valeur de SP
+24
SP
+
+
+
+
+
+
+


+0 3
i
+4 6
Zone de retour de factoriel
+8
SP
+0
+
+
+
+
+
+
+
+

Fonctions de manipulation de chaînes de caractères

La bibliothèque standard fournit des fonctions de manipulation de chaînes de caractères. Il est nécessaire d'inclure le fichier $<string.h>$ pour avoir la définition des fonctions décrites dans la table 17.1.

Tableau 17.1: Fonctions de manipulation de chaines

Déclaration	Travail	Retour
char *s1, *s2;		
int n;		
char *strcat (s1, s2)	ajoute la chaîne s2 derrière la chaîne s1	pointeur sur s1
char *strncat (s1, s2, n)	ajoute au plus n caractères	pointeur sur s1
int strcmp (s1, s2)	compare s1 et s2	nombre positif, nul, négatif
int strncmp (s1, s2, n)	sur au plus n caractères	s1 $>$ s2 , s1==s2 , s1 $<$ s2
char *strcpy (s1, s2)	copie s2 dans s1	
char *strncpy (s1, s2, n)	au plus n caractères	
char *s;		
int strlen (s)		taille de s
int c;		
char *strchr (s, c)	cherche caractère c dans s	pointeur sur première occurrence
char *strrchr (s, c)		pointeur sur dernière occurrence
char c;		
char *index(s, c)	idem que strchr	
char *rindex(s, c)	idem que strrchr	
char *strpbrk (s1, s2)	cherche 1 car de s2 dans s1	pointeur sur première occurrence

int strspn (s1, s2)	1$^{}$ motif dans s1 ne contenant	longueur
	que des car de s2	
int strcspn (s1, s2)	1$^{}$ motif dans s1 ne contenant	longueur
	aucun car de s2	

next up previous contents index

Next: Types de caractères **Up:** Autres fonctions de la **Previous:** Autres fonctions de la
2002-06-21

Plus sur `printf()` et `scanf()`

Les fonctions `printf()` et `scanf()` transforment des objets d'une représentation à partir d'une chaîne de caractères (vision humaine) en une représentation manipulable par la machine (vision machine), et vice et versa. Pour réaliser ces transformations ces fonctions sont guidées par des formats qui décrivent le type des objets manipulés (vision interne) et la représentation en chaîne de caractères cible (vision externe). Par exemple, un format du type `%x` signifie d'une part que la variable est du type entier et d'autre part que la chaîne de caractère qui la représente est exprimée en base 16 (hexadécimal).

Pour printf un format est une chaîne de caractères dans laquelle sont
insérés les caractères représentant la ou les variables à écrire.
Pour `scanf()` un format est une chaîne de caractères qui décrit
la ou les variables à lire.
Pour chaque variable, un type de conversion est spécifié. Ce type de
conversion est décrit par les caractères qui suivent le caractère ``%''.

Les types de conversion les plus usuels sont donnés dans la table 4.2.

Tableau 4.2: Conversions usuelles de printf et scanf

`%d`	entier décimal
`%f`	flottant
`%c`	caractère (1 seul)
`%s`	chaîne de caractères

Dans une première approche de `scanf()`, nous considérerons qu'il ne faut mettre que des types de conversions dans le format de lecture. Le lecteur curieux peut se reporter à la section 16.5.

Le tableau 4.3 donne un résumé des déclarations de variables et des formats nécessaires à leurs

manipulations pour printf et scanf.

Tableau 4.3: Exemples de printf et scanf

déclaration	**lecture**	**écriture**	**format externe**
`int i;`	`scanf("%d",&i);`	`printf("%d",i);`	décimal
`int i;`	`scanf("%o",&i);`	`printf("%o",i);`	octal
`int i;`	`scanf("%x",&i);`	`printf("%x",i);`	hexadécimal
`unsigned int i;`	`scanf("%u",&i);`	`printf("%u",i);`	décimal
`short j;`	`scanf("%hd",&j);`	`printf("%d",j);`	décimal
`short j;`	`scanf("%ho",&j);`	`printf("%o",j);`	octal
`short j;`	`scanf("%hx",&j);`	`printf("%x",j);`	hexadécimal
`unsigned short j;`	`scanf("%hu",&j);`	`printf("%u",j);`	décimal
`long k;`	`scanf("%ld",&k);`	`printf("%d",k);`	décimal
`long k;`	`scanf("%lo",&k);`	`printf("%o",k);`	octal
`long k;`	`scanf("%lx",&k);`	`printf("%x",k);`	hexadécimal
`unsigned long k;`	`scanf("%lu",&k);`	`printf("%u",k);`	décimal
`float l;`	`scanf("%f",&l);`	`printf("%f",l);`	point décimal
`float l;`	`scanf("%e",&l);`	`printf("%e",l);`	exponentielle
`float l;`		`printf("%g",l);`	la plus courte
			des deux
`double m;`	`scanf("%lf",&m);`	`printf("%f",m);`	point décimal
`double m;`	`scanf("%le"&m);`	`printf("%e",m);`	exponentielle
`double m;`		`printf("%g",m);`	la plus courte
`long double n;`	`scanf("%Lf"&n);`	`printf("%Lf",n);`	point décimal
`long double n;`	`scanf("%Le"&n);`	`printf("%Le",n);`	exponentielle
`long double n;`		`printf("%Lg",n);`	la plus courte
`char o;`	`scanf("%c",&o);`	`printf("%c",o);`	caractère
`char p[10];`	`scanf("%s",p);`	`printf("%s",p);`	chaîne de caractères
	`scanf("%s",&p[0]);`		

Le & est un opérateur du langage C dont nous parlerons plus tard. Pour scanf, il faut le mettre devant le nom de la variable, sauf pour les variables du type tableau de caractères.

L'exemple 4.4, montre qu'il est possible de faire l'écriture ou la lecture de plusieurs variables en utilisant une seule chaîne de caractères contenant plusieurs descriptions de formats.

Tableau 4.4: Lectures multiples avec scanf()

\begin{table}\begin{small}
\begin{verbatim}1. main()
2. {
3. int i=10;
...
...',i,l,p);
9. }\end{verbatim}\end{small}\index{scanf}\index{lecture}\end{table}

---

next | up | previous | contents | index

Next: Opérateurs et expressions **Up:** Eléments de base **Previous:** Quelques opérations
2002-06-21