

Présentation du langage C



Diapositives de H. BETTAHAR, SR01

1 – Introduction



- Historique
- Intérêts du langage
- Qualités attendues d'un programme

Historique

- ☑ Langage de programmation développé en 1970 par Dennie Ritchie aux Laboratoires Bell d'AT&T. Il est l'aboutissement de deux langages :
 - ☞ BPCL développé en 1967 par Martin Richards.
 - ☞ B développé en 1970 chez AT&T par Ken Thompson.
- ☑ Il fut limité à l'usage interne de Bell jusqu'en 1978 date à laquelle Brian Kernighan et Dennie Ritchie publièrent les spécifications définitives du langage :
 - ☞ << The C programming Language >>.

Historique

Au milieu des années 1980 la popularité du langage était établie.

De nombreux compilateurs ont été écrits, mais comportant quelques incompatibilités portant atteinte à l'objectif de portabilité.

Un travail de normalisation a été effectué par le comité de normalisation X3J11 de l'ANSI (American National Standards Institute) qui a abouti en 1988 avec la parution par la suite du manuel :

<< The C programming Language - 2ème édition >>.

Intérêt du langage

- ☑ Langage polyvalent permettant le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- ☑ Langage structuré.
- ☑ Langage évolué qui permet néanmoins d'effectuer des opérations de bas niveau (<< assembleur >>).
- ☑ Portabilité (en respectant la norme !) due à l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine.
- ☑ Grande efficacité et puissance.

Qualités attendus d'un programme

- ☑ Clarté
- ☑ Simplicité
- ☑ Modularité
- ☑ Extensibilité

2 - Généralités



Structure d'un programme C

Préprocesseur

Déclarations

Operations E/S

structure générale d'un programme C

```
/*c1_max.c*/  
#include <stdio.h>  
  
/*fonction: max de deux entiers*/  
int max(int a, int b)  
{  
    if (a > b)  
        return (a);  
    else  
        return (b);  
}  
  
main() {  
  
    int x, y, z;  
  
    printf("\n Donner x:  ");  
    scanf ("%d",&x);  
    printf("\n Donner y:  ");  
    scanf ("%d",&y);  
  
    z = max(x,y);  
    printf("le maximum de %d et %d est %d",x,y,z);  
}
```

instruction du préprocesseur

commentaires

définition du fonction max()

fonction main()

déclaration des variables

fonctions d'Entrée/Sortie

appel du fonction

préprocesseur

- ✓ Le pré-processeur est un utilitaire qui traite le fichier source avant le compilateur. C'est un manipulateur de chaînes de caractères. Il retire les parties de commentaires (`/*...*/`). Il prend aussi en compte les lignes du texte source qui commencent par un `#`. Le préprocesseur permet:

☞ d'inclure des fichiers spécifiés dans le programme:

```
#include <nom_fichier.h> /* pour fichier dans les repertoires standard*/
```

```
#include "nom_fichier.h" /*pour les autres fichier*/
```

- des sources en langage C qui pourront contenir des modules d'applications ou la définition de fonctions
- fichiers d'entêtes (.h) qui contiennent:
 - des constantes symboliques et des macro-instructions standard
 - la déclaration de fonctions incluses dans une bibliothèque
 - des définitions de types

```
#include <stdio.h> /* pour les E S standard*/
```

```
#include <math.h> /*pour les fonctions mathématique*/
```

préprocesseur

☞ de définir des constantes symboliques:

#define identificateur chaine_de_caracteres

- le préprocesseur remplace chaque occurrence de l'identificateur par la chaîne de caractères qui lui est associée

#define MAX 100

#define MIN 0

#define EOF (-1)

- pour supprimer la définition d'une constante symbolique

#undef identificateur

- ☑ On préférera n'utiliser que des majuscules pour écrire ces identificateurs afin de les différencier des autres (variables, vecteurs, fonctions)

préprocesseur

☞ de faire des compilations conditionnelles, ce qui permet de :

- d'écrire des programmes portables en testant des constantes symboliques représentant le type et les caractéristiques de la machine utilisée
- de pouvoir optimiser un code en utilisant les avantages spécifiques de chaque machine.

☞ Ce sont les directives **#ifdef** et **#ifndef** qui permettent de tester l'existence d'une pseudo-constante.

```
#if SYS == VAX
    #include "vax.h"
#elif SYS == SUN
    #include "sun.h"
#else
    #include "autres.h"
#endif
```

```
#ifndef EOF
    #define EOF (-1)
#endif
```

préprocesseur

- ☞ de définir des macros ou pseudo-fonctions qui sont des substitutions paramétrables:

```
#define ABS(x) x>0 ? x : -x
#define NB_ELEMENTS(t) sizeof t / sizeof t[0]
#include <stdio.h>

main()
{
    int    tab[][2] = { 1,  2,  3,  9,
                       10, 11, 13, 16};

    int r ;
    int  i, j;

    for(i=0; i < NB_ELEMENTS(tab); i++)
        for(j=0; j < 2; j++)
            tab[i][j] = i + j;

    r= 5 - NB_ELEMENTS(tab);
    printf("%f\n", ABS(r));
}
```

Forme générale d'une déclaration

```
type nom;  
type nom = valeur;
```

- ✓ `type` : types de base, ou types définis par l'utilisateur
- ✓ `nom` : un identificateur (plus des constructeur homogènes) .

☞ `identificateur`:

- ☞ le 1er caractère doit être alphabétique ou `_` (le caractère souligné)
- ☞ les autres doivent être alphanumérique (lettres et chiffres) ou `_`
- ☞ les mots clés sont réservés

☞ `constructeurs homogènes`:

- ☞ pointeurs : `*`
- ☞ tableaux (vecteurs): `[]`
- ☞ fonctions: `()`

Les types de base

<code>void</code>	type «vide» sur 0 octet!
<code>char</code>	caractère: entier sur 1 octet (-128 → 127) ou (0 → 255) suivant le compilateur
<code>short int</code>	entier court sur 2 octets (-32768 → 32767)
<code>int</code>	entier par défaut (en général sur 4 octets)
<code>long int</code>	entier long
<code>float</code>	réel simple précision
<code>double</code>	réel double précision
<code>long double</code>	réel double précision étendue

- ☑ `short int` et `long int` peuvent être abrégés en `short` et `long`
- ☑ `char` est considéré comme un entier, il peut être utilisé dans des expression arithmétique.
- ☑ `unsigned` / `signed` : s'applique sur les types entiers.
 - ☞ `int`: entier signé (par défaut) sur 4 octet ($-2^{31}-1 \rightarrow +2^{31}-1$)
 - ☞ `unsigned int`: entier non signé sur 4 octet ($0 \rightarrow 2^{32}-1$)

Exemples

☑ exemples

```
int i, j, k=0;           /*initialisation de k à 0*/
float tab[10];           /*tableau de 10 réels*/
char nom[10] = "toto";   /*tableau de 10 carac.*
int *ptr;                /*pointeur sur un entier*/
int max(int,int);        /*fonction max*/
```

☑ identificateurs acceptés:

```
somme
sous_total1
_TOTAL
```

☑ identificateur erronés

```
somme$           /* $ non alphanumérique*/
1sous_somme      /* commence par un chiffre*/
sous somme        /* contient un blanc*/
case             /* mots clé*/
SOUS-TOTAL       /* contient -*/
```

Déclaration des constantes

```
const type nom=valeur;
```

- ☑ permet de définir une variable dont on s'interdit de modifier le contenu.

```
const double e = 2.71828182845905;
```

```
const float pi = 3.14159;
```

```
const char message[] = "erreur";
```


les structures

```
struct nom {  
    déclarations;  
};
```

- ✓ *nom* : facultatif, permet de déclarer des variables de ce type ultérieurement.
- ✓ *déclarations* : chaque déclaration correspond à un champ de la structure (un membre)
- ✓ les champs d'une même structure peuvent être de types différents.
- ✓ la taille en mémoire d'une structure est au moins égale à la somme de ses membres.

les structures: exemples

```
struct {  
    int x;  
    int y;  
} point1, point2;  
  
struct date{  
    short jour;  
    short mois;  
    short annee;  
};  
  
struct personne{  
    char nom[80];  
    char prenom[80];  
    struct date naissance;  
};  
  
struct personne untel, deuxitel,*inconnu;  
  
struct personne liste_de_personne[100];
```

les structures: accès et initialisation

```
struct personne untel, deuxt看, *inconnu;
```

```
/*initialisation du champ annee d' untel*/
```

```
struct personne untel = {"dupond", "jean", 14, 2, 1984};
```

```
/*initialisation du champ annee d' untel*/
```

```
inconnu = & untel;
```

```
inconnu->naissance.annee = 1978;
```

```
untel.naissance.annee = 1978;
```

```
(*inconnu).naissance.annee = 1978;
```

} ≡

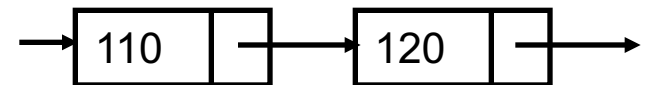
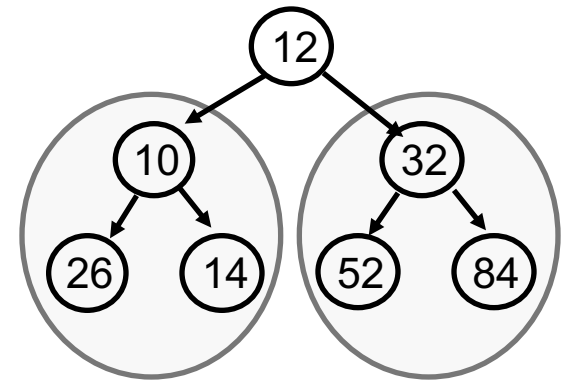
```
deuxt看.naissance.annee = inconnu->naissance.annee;
```

Structures auto-référencées

- ☑ un ou plusieurs membres de la structures ont le même type que la structure elle même
- ☑ utile pour définir des structures chaînées: listes chaînées , arbres,

```
struct arbre_bin{  
    int val;  
    struct arbre_bin *droite;  
    struct arbre_bin *gauche;  
};
```

```
struct cellule{  
    int val;  
    struct cellule *suivant;  
};
```

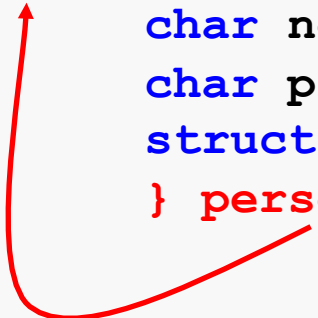


Définition de types: *typedef*

```
typedef type nouveau_nom;
```

- ☑ permet de renommer des types (par exemple construit par le programmeur avec struct , union)

```
typedef struct pers{  
    char nom[80];  
    char prenom[80];  
    struct date naissance;  
} personne;  
  
struct pers untel, deuxitel,*inconnu;  
  
personne untel, deuxitel,*inconnu;
```



Pointeurs

```
type *identificateur;
```

- ☑ Définition: un pointeur est une variable (ou une constante) dont la valeur est une **adresse** (d'une autre variable).
- ☑ les pointeurs sont typés, un pointeur ne peut être utilisé que pour contenir des adresses de variables d'un seul type: un pointeur sur un entier , un pointeur sur caractère,...

Pointeurs

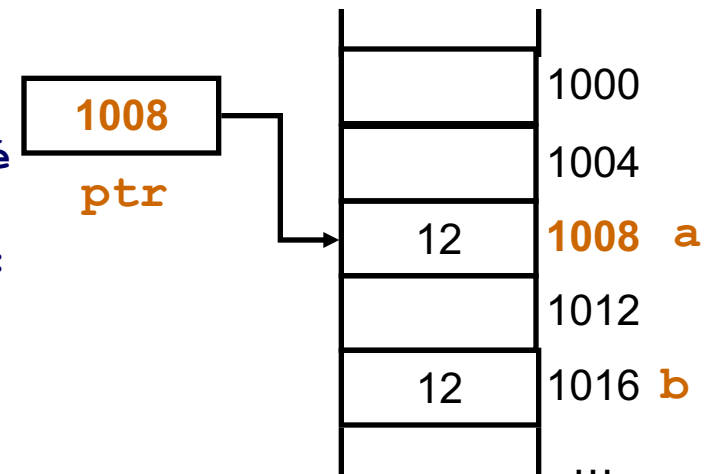
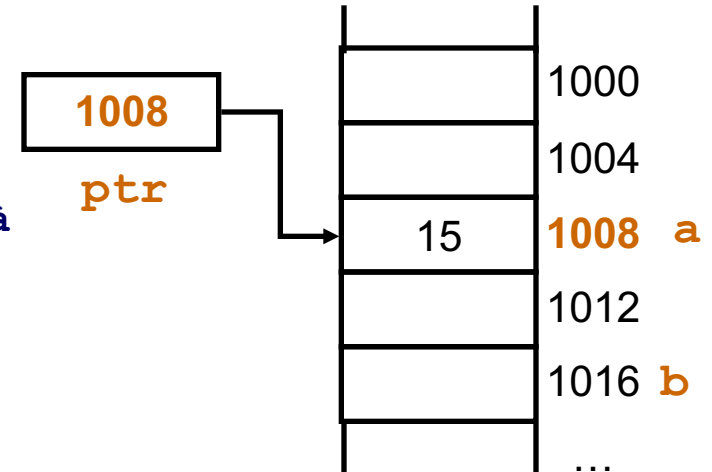
```
int *ptr;  
int a=15, b;
```

- ☑ l'opérateur "adresse de" **&** , appliqué à une variable, permet d'obtenir l'adresse de celle-ci.

```
ptr = &a;
```

- ☑ l'opérateur d'indirection (contenu de) ***** permet de référencer un pointeur, il fournit la valeur de la variable pointé par celui-ci. Il permet aussi d'affecter une valeur à cette variable:

```
*ptr = 12; ≡ a = 12;  
b = *ptr; ≡ b = a;
```



exemples

ptr_ptr1.c

ptr_ptr2.c

Tableau à une dimension

☑ un tableau est une collection de variables du même type, toutes accessibles individuellement.

☑ déclaration

```
type nom[taille];
```

☑ le type de base peut être quelconque:

```
int tab[10];
```

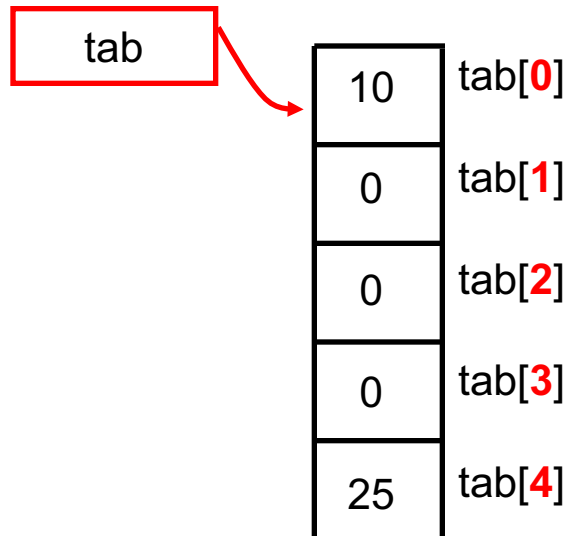
☑ déclaration avec initialisation :

```
☞ int tab[5] = {4, 10, 12, 30, 25};
```


Tableau à une dimension

- ✓ les éléments d'un tableau sont rangés à des adresses consécutives dans la mémoire
- ✓ les éléments d'un tableau sont accessibles par leur indices, l'indice d'un élément correspond au déplacement relatif à l'adresse du tableau (offset)
- ✓ **Attention:** il n'y a pas de vérification des limites d'un tableau lors de l'accès à un élément. pas de contrôle de débordement sur les indices.

```
int tab[5], i;  
  
for (i=0; i<5; i++){  
    tab[i] = 0;  
}  
  
tab[0] = 10;  
tab[4] = 25;
```



exemples

```
ptr_tab1.c  
ptr_tab2.c  
ptr_tab3.c
```

Tableau à une dimension

- ✓ En C, le **nom** d'un tableau peut être utilisé en tant que **pointeur constant** dont la valeur est égale à l'adresse du début du tableau:

☞ `int tab[5]; tab ≡ &tab[0]`

- ✓ **Attention:** les instructions suivantes ne sont pas valides:

☞ `tab = ptr;`

☞ `tab++;`

```
int tab[5];
int *ptr, i;

ptr = tab;
for (i=0; i<5; i++){
    ptr[i] = 0;
}

tab[0] = 10;
ptr[4] = 25;
```

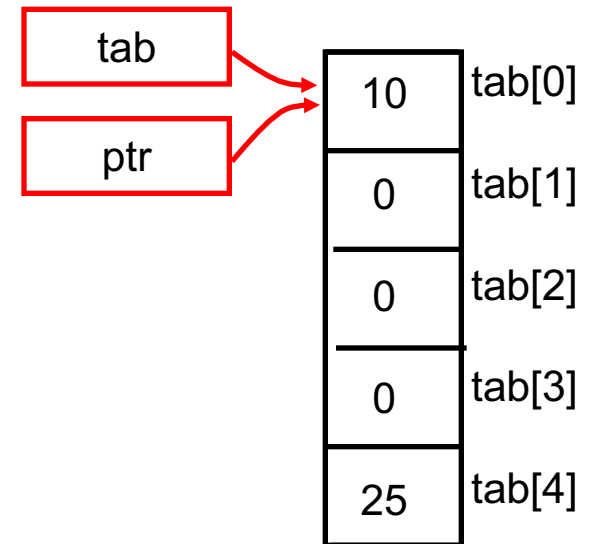


Tableau à une dimension

tableaux: indices et pointeurs

- ☑ pour les opération arithmétiques avec les pointeurs, l'unité n'est pas l'octet, mais la taille de l'objet

```
int tab[5];
int *ptr, i;

ptr = tab;
for (i=0; i<5; i++){
    tab[i] = 10;
}
```

```
int tab[5];
int *ptr, i;

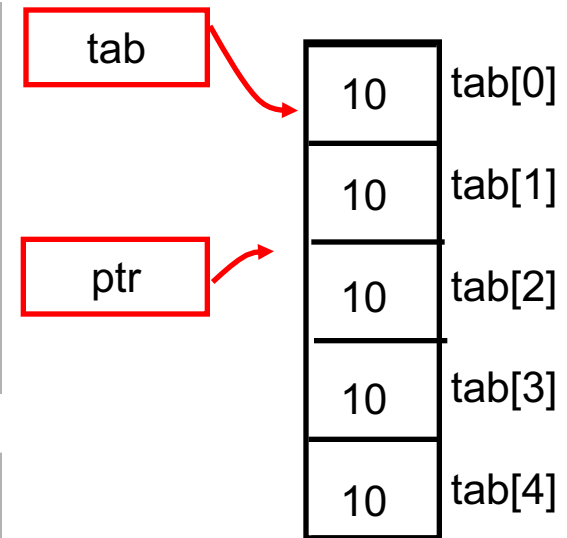
ptr = tab;
for (i=0; i<5; i++){
    *(tab+i) = 10;
}
```

```
int tab[5];
int *ptr, i;

ptr = &tab[0];
for (i=0; i<5; i++){
    *ptr = 10;
    ptr++;
}
```

```
int tab[5];
int *ptr, i;

ptr = &tab[4];
for (i=0; i<5; i++){
    *ptr = 10;
    ptr--;
}
```

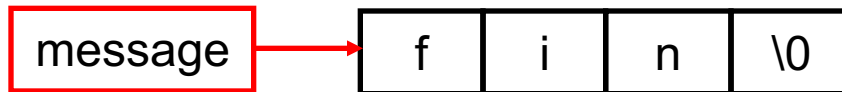


exemples

`ptr_tab_ptr.c`

chaînes de caractères

- ✓ en C, il n'existe pas de type de base pour les chaînes de caractères, on utilise des tableaux de caractères. par convention, le dernier caractère est le caractère NULL ('**0**').



```
char message[4]= {'f', 'i', 'n', '\0'};  
char message[4] = "fin";  
char message[] = "fin";
```

```
scanf("%s", message);  
printf("%s", message);
```

```
scanf("%s", &message[0]);  
printf("%s", &message[0]);
```

exemples

```
ptr_chaine1.c  
ptr_chaine2.c
```

Allocation mémoire et pointeurs

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr)
```

- ☑ les fonction de gestion de mémoire `malloc()` et `free()` permettent d'allouer et de restituer dynamiquement de l'espace mémoire.
- ☑ `size` est en nombre d'octets: on utilise en générale la fonction `sizeof()` pour déterminer la taille de données à stockées.
- ☑ la fonction `malloc()` rend un pointeur générique `void *` sur l'espace réservé. il faut forcer le type de retour de `malloc` (faire un `cast`) pour récupérer un pointeur de type voulu.

Allocation mémoire et pointeurs

```
typedef struct pers{
    char nom[80];
    char prenom[80];
    struct date naissance;
} personne;

int *ptr;
personne *liste_personne;

ptr = (int *)malloc(sizeof(int));

liste_personne = (personne *)malloc(3*sizeof(personne));

if (liste_personne == NULL){
    printf("Allocation mémoire impossible !\n");
    ...
}
```

printf()

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

- ☑ la fonction printf() réalise la conversion d'un ensemble d'arguments dans un format donné en vue de leur affichage sur la sortie standard *stdout* (ecran)
- ☑ format: %[flag][n][.p][l][t]
 - ☞ flag :précise l'alignement souhaité:
 - - alignement à gauche (droite par défaut),
 - + les valeurs numériques sont préfixées par + ou –
 - ☞ n : précise la largeur minimum du champ correspondant
 - ☞ p : indique la précision retenue pour l'affichage
 - nombre min des chiffres pour les types d, o, u, x, ou X,
 - nombre des chiffres après la virgule pour les types e et f
 - nombre max de caractères pour le type s, ...
 - ☞ l : indique une variable entière au format long pour les types d, o, u, x, ou X

printf()

☞ **t** : indique le type d'affichage:

- **d** décimal
- **o** octal
- **x** hexadécimal (abcdef)
- **X** hexadécimal (ABCDEF)
- **f** flottant ([-]ddd.ddd) 6 chiffres par défaut,
- **e** flottant ([-]d.ddde[+/-]dd) 6 chiffres par défaut,
- **E** flottant ([-]d.dddE[+/-]dd) 6 chiffres par défaut,
- **g** le plus court des types f ou e
- **G** le plus court des types f ou E
- **c** caractère
- **s** chaîne de caractères
-

exemples

```
exemple_printf_1.c  
exemple_printf_2.c  
exemple_printf_3.c
```


printf(): exemples

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

```
int main() {
```

```
int a=22;
```

```
double p=3.14159;
```

```
/* Variations sur la taille */
```

```
printf("a=%d a=%4d\n", a, a);
```

```
/* Variations sur 'p' : pour un entier écriture de zéro en tête */
```

```
printf("a=%4.4d a=%6.6d \n", a, a);
```

```
/* Variations sur 'p' : pour un double, nombre de chiffres après  
le point */
```

```
printf("pi=%.2lf pi=%.4lf \n", p, p);
```

```
}
```

exécution:

a=22 a= 22

a=0022 a=000022

pi=3.14 pi=3.1416

printf(): exemples

```
#include <stdio.h>
int main() {
int i=22; long int li=22;
float fp=3.14159; double dp=3.14159;

/* Variations sur la longueur par défaut de l'argument */
printf("fp=%f dp=%lf \n", fp, dp);
printf("i=%d li=%ld \n", i, li);
}
```

exécution:

fp=3.141590 dp=3.141590

i=22 li=22

printf(): exemples

```
#include <stdio.h>
int main() {
char * mess1="pi :", * mess2=" e :";
double dp=3.14159, de=2.718281828;

/* Variations sur la taille de la zone écrite */
printf("%*s %*s \n", 9, mess1, 15, mess2);
printf("%*lf %*.9lf \n", 9, dp, 15, de);
}
```

exécution:

```
    pi :                e :
3.141590 2.718281828
```

scanf()

```
int scanf(const char* format [, adr1, adr2 ...]);
```

- ☑ Lit une suite de caractères depuis le flux d'entrée standard *stdin*. Ces caractères sont formatés en champs, suivant les spécifications de format indiquées par la chaîne 'format'.
- ☑ format: *%[*] [taille][h|l|L] type*
- ☑ Une spécification de format contient au minimum le caractère '%' et le champ type.
 - ☞ % : marque le début de la zone
 - ☞ * : supprime l'affectation du prochain champ d'entrée (facultatif).
 - ☞ taille : nombre maximal de caractères a lire(facultatif).
 - ☞ [h|l|L] : précise la taille de l'objet pointé par l'argument (facultatif).

scanf()

👉 **type**: un ou plusieurs caractères, donnant le type de l'objets pointé par l'argument, pouvant être:

- **c** lit un caractère
- **s** lit une suite de caractères(terminée par un espace, une tabulation ou un retour-chariot)
- **p** lit une valeur hexadécimale non signée
- **d** lit un entier sous forme décimale
- **u** lit un entier non signé sous forme décimale
- **o** lit un entier non signé sous forme octale
- **x** lit un entier non signé sous forme hexadécimale
- **e f g** lit une valeur flottante
- **n** fournit le nombre de caractères déjà lus
- **[...]** lit les caractères de l'ensemble
- **[^...]** lit les caractères n'appartenant pas à l'ensemble.

☑ **Important**: Le programmeur doit s'assurer, lors de lecture de chaînes en particulier, que la zone mémoire dont il fournit l'adresse est assez grande pour recevoir les caractères lus, La fonction `scanf()` ne fait aucune vérification sur la taille des zones mémoires qu'elle modifie.

exemples

`exemple_scanf_1.c`

`exemple_scanf_2.c`

scanf(): exemples

```
#include <stdio.h>
int main() {
char c; char tc[80];
float x; double dx;

scanf("%c",&c); /*on tape A */
scanf("%s",tc); /*on tape bonjour */
scanf("%f%lf",&x, &dx); /*on tape 3.14 et -2.718*/

printf("Les données lus sont: %c, %s, %d, %x, %f, %lf\n",
c,tc,d,h,x, dx);
}
```

exécution:

Les données sont: A, bonjour, 3.140000, -2.718000

4 - Expressions et opérateurs



- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateur de taille

Opérateurs arithmétiques

- ☑ Une **expression** est constituée de variables et constantes (littérales et/ou symboliques) reliées par des **opérateurs**.

Il existe 5 opérateurs arithmétiques :

- ☞ l'addition (+),
- ☞ la soustraction (-),
- ☞ la multiplication (*),
- ☞ la division (/),
- ☞ le reste de la division entière (%).

- ☑ Leurs opérandes peuvent être des **entiers** ou des **réels** hormis ceux du dernier qui agit uniquement sur des entiers.
- ☑ Lorsque les types des deux **opérandes** sont différents, il y a **conversion implicite** dans le type le plus fort suivant certaines règles.

Opérateurs logiques

- ☑ Le type **booléen** n'existe pas. Le résultat d'une expression logique vaut **1** si elle est vraie et **0** sinon.
- ☑ Réciproquement toute valeur **non nulle** est considérée comme **vraie** et la valeur nulle comme **fausse**.
- ☑ Les opérateurs logiques comprennent :

☞ 4 opérateurs relationnels :

- inférieur à (**<**),
- inférieur ou égal à (**<=**),
- supérieur à (**>**),
- supérieur ou égal à (**>=**).
- l'opérateur de négation (**!**).

☞ 2 opérateurs de comparaison :

- identique à (**==**),
- différent de (**!=**).

Opérateurs logiques

👉 2 opérateurs de conjonction :

- le et logique (&&),
- le ou logique (||).

☑ Le résultat de l'expression :

👉 **!expr1** est vrai si *expr1* est fausse et faux si *expr1* est vraie ;

👉 **expr1&&expr2** est vrai si les deux expressions *expr1* et *expr2* sont vraies et faux sinon. L'expression *expr2* n'est évaluée que dans le cas où l'expression *expr1* est vraie ;

👉 **expr1||expr2** est vrai si l'une au moins des expressions *expr1*, *expr2* est vraie et faux sinon. L'expression *expr2* n'est évaluée que dans le cas où l'expression *expr1* est fausse.

Opérateur de taille

- ✓ L'opérateur `sizeof` renvoie la taille en octets de son opérande, l'opérande est soit une expression soit une expression de type.

- ✓ Syntaxe

`sizeof expression`
`sizeof (expression-de-type)`

- ✓ L'opérateur `sizeof` appliqué à une constante chaîne de caractères renvoie le nombre de caractères de la chaîne y compris le caractère NULL de fin de chaîne.
- ✓ Si `p` est un pointeur sur un type `t` et `i` un entier :

l'expression `p + i`
a pour valeur `p + i*sizeof(t)`

Opérateur de taille

Exemple

exemples

test_sizeof.c

```
int menu[1000];
```

```
typedef struct cel {  
    int      valeur;  
    struct cel *ptr;  
} Cel;
```

`sizeof menu/sizeof menu[0]` ==> nombre d'éléments du vecteur menu.

`sizeof(long)` ==> taille d'un entier long.

`sizeof(float)` ==> taille d'un flottant simple précision.

`sizeof(struct cel)` ==> taille d'un objet du type struct cel.

`sizeof(Cel)` ==> taille d'un objet du type Cel.

5 - Instructions



- Instructions élémentaires
- Structures de contrôle
- Instructions d'échappement

Instructions élémentaires

- ☑ Une **instruction élémentaire** est une expression terminée par un **;**
- ☑ Contrairement aux expressions, les instructions n'ont ni type ni valeur. Lorsqu'on forme une instruction à partir d'une expression la valeur de cette dernière est perdue.
- ☑ N'importe quelle expression peut former une instruction.
- ☑ Une **instruction composée** ou **bloc** est un ensemble d'instructions élémentaires et/ou composées, précédées éventuellement de déclarations, délimité par des **accolades**.

Instructions élémentaires

Exemples

```
#include <stdio.h>
#include <math.h>
main()
{
    int    i = 10;
    double r = acos(-1.) ;

    i *= 2;
    {
        double cosh_pi;

        cosh_pi = (exp(r) + exp(-r)) / 2;
        printf("cosh_pi : %f\n", cosh_pi);
    }
}
```

Structures de contrôle

- ☑ Les structures de contrôle sont les **tests**, les **boucles** et l'**aiguillage**.

Les tests : syntaxe

```
if (expression)  
    partie-alors  
    [else  
        partie-sinon]
```

- ☑ La partie-alors et la partie-sinon peuvent être indifféremment une instruction élémentaire ou composée.
- ☑ La partie-alors sera exécutée si la valeur de l'expression entre parenthèses est non nulle. Sinon, si le test comporte une partie-sinon c'est celle-ci qui sera exécutée.

Exemples

```
if ((c>'a') && (c<='z'))  
    printf(" %c est minuscule \n ",c);  
else  
    if ((c>'A') && (c<='Z'))  
        printf(" %c est majuscule \n ",c);
```

`if (x)` est équivalent à `if (x != 0)`

`if (!x)` est équivalent à `if (x == 0)`

Structures de contrôle

- ✓ Si plusieurs tests sont imbriqués, chaque partie-sinon est reliée au if le plus proche qui n'est pas déjà associé à une partie-sinon.

Exemples

```
/** exemple_if_1.c **/
```

```
if (x > 0)
    ecrire("positif");
else if (x < 0)
    ecrire("négatif");
else
    ecrire("nul");
```

<==>

```
if (x > 0)
    ecrire("positif");
else
{
    if (x < 0)
        ecrire("négatif");
    else
        ecrire("nul");
}
```

exemples

`exemple_if_1.c`

Structures de contrôle

Les boucles "tant-que" : syntaxe

```
while (expression)  
    corps-de-boucle
```

```
do  
    corps-de-boucle  
while (expression);
```

- ☑ La partie *corps-de-boucle* peut être soit une instruction élémentaire soit une instruction composée.
- ☑ Dans la boucle *while* le test de continuation s'effectue *avant* d'entamer le corps-de-boucle qui, de ce fait, peut ne jamais s'exécuter.
- ☑ Par contre dans la boucle *do-while* ce test est effectué *après* le corps-de-boucle, lequel sera alors exécuté au moins une fois.

Structures de contrôle

```
/*exemple_while.c*/
#include <stdio.h>
main()
{
    int chiffre = 0;

    printf("Boucle \"while\"\n\n");
    while (chiffre) {
        printf(" %d", chiffre++);
        if (!(chiffre%5))
            printf("\n");
    }
    printf("Boucle \"do-while\"\n\n");
    do {
        printf(" %3d", ++chiffre);
        if (!(chiffre%5))
            printf("\n");
        if (chiffre == 100)
            chiffre = 0;
    }
    while (chiffre);
}
```

exemples

exemple_while.c

Structures de contrôle

La boucle "pour" : syntaxe

```
for ([expr1]; [expr2]; [expr3])  
    corps-de-boucle
```

- ☑ L'expression **expr1** est évaluée une seule fois, au début de l'exécution de la boucle.
- ☑ L'expression **expr2** est évaluée et testée avant chaque passage dans la boucle.
- ☑ L'expression **expr3** est évaluée après chaque passage.
- ☑ Ces 3 expressions jouent respectivement le rôle :
 - ☞ d'expression d'initialisation,
 - ☞ de test d'arrêt,
 - ☞ d'incrémentatation.

Structures de contrôle

```
/**exemple_for.c**/  
main()  
{  
    int    tab[] = {1, 2, 9, 10, 7, 8, 11};  
    int    i, j;  
    char   buffer[] = "Voici une chane qui se termine par un blanc ";  
    char   *p;  
    int    t[4][3];  
    for (i=0; i < sizeof tab / sizeof tab[0]; i++)  
        printf("tab[%d] = %d\n", i, tab[i]);  
    for (p=buffer; *p; p++);  
    *--p = '\\0';  
    printf("buffer : %s\n", buffer);  
    for (i=0; i < 4; i++)  
        for (j=0; j < 3; j++)  
            t[i][j] = i + j;  
    for (i=0; i < 4; i++)  
    {  
        for (j=0; j < 3; j++)  
            printf("%d ", t[i][j]);  
            printf("\\n");  
    }  
    exit(0);  
}
```

exemples

exemple_for.c

Structures de contrôle

L'aiguillage

- ☑ L'instruction **switch** définit un aiguillage qui permet d'effectuer un branchement à une étiquette de cas en fonction de la valeur d'une expression.

Syntaxe

```
switch (expression)
{
    case etiq1 :
        [ liste d'instructions ]
    case etiq2 :
        [ liste d'instructions ]
        ...
    case etiqn :
        [ liste d'instructions ]
    [ default:
        [ liste d'instructions ] ]
}
```

Structures de contrôle

- ☑ Les étiquettes de cas (etiq1, etiq2, ..., etiqn) doivent être des **expressions constantes**.
- ☑ Une fois le branchement à l'étiquette de cas correspondante effectué, l'exécution se poursuit, par défaut, jusqu'à la fin du bloc **switch**. L'instruction d'échappement **break**; permet de forcer la sortie du bloc.
- ☑ L'expression indiquée au niveau du switch doit être de type **entier**.

Structures de contrôle

Exemples : calculer le nombre des caractères, espaces et lignes

```
/*exemple_switch.c*/
#include <stdio.h>
main()
{
    char *buffer = "\nCeci est une chaîne\n"
                  "de caractères\tsur\n\n"
                  "plusieurs      lignes.\n";
    int    NbCar  = 0, NbEsp = 0, NbLignes = 0;

    for (; *buffer; buffer++, NbCar++)
        switch (*buffer) {
            case '\n': NbLignes++;
                       break;

            case '\t':
            case ' ': NbEsp++;
            default  : break;
        }
    printf("NbCar = %d, NbEsp = %d, NbLignes = %d\n",
           NbCar, NbEsp, NbLignes);
}
```

exemples

exemple_switch.c

Instructions d'échappement

Instructions d'échappement

Ces instructions permettent de rompre le déroulement séquentiel d'une suite d'instructions.

Instruction `continue;`

- ☑ Le rôle de l'instruction `continue;` est de forcer le passage à l'itération suivante de la boucle la plus proche.

Instructions d'échappement

```
/** exemple_continue.c */
#include <stdio.h>
main()
{
    char *buffer = "\nCeci est une chaîne\n"
                  "de caractères\tsur\n\n"
                  "plusieurs      lignes.\n";
    int  NbCar   = 0, NbEsp = 0, NbLignes = 0;

    for (; *buffer; buffer++) {
        switch (*buffer) {
            case '\n': NbLignes++;
                       break;
            case '\t': continue;
            case ' ':  NbEsp++;
            default :  break;
        }
        NbCar++;
    }
    printf("NbCar = %d, NbEsp = %d, NbLignes = %d\n",
          NbCar, NbEsp, NbLignes);
}
```

exemples

exemple_continue.c

Instructions d'échappement

Instruction break;

L'instruction **break**; permet de quitter la **boucle** ou l'**aiguillage** le plus proche.

```
/** exemple_break.c */
#include <stdio.h>
main()
{
    char  buffer[] = "Wolfgang Amadeus Mozart\n"
                    " est un musicien divin.\n";
    char *p;

    for (p=buffer; *p; p++)
        if (*p == '\n')
        {
            *p = '\0';
            break;
        }
    printf("Nom : %s\n", buffer);
}
```

exemples

exemple_break.c

Instructions d'échappement

Instruction `return`;

Syntaxe

```
return [expression];
```

☑ Cette instruction permet de sortir de la fonction qui la contient :

☞ si elle est suivie d'une expression, la valeur de celle-ci est transmise à la fonction appelante après avoir été convertie, si nécessaire et si possible, dans le type de celui de la fonction,

☞ sinon la valeur retournée est indéterminée.

Instructions d'échappement

Exemples

```
#include <stdio.h>
main()
{
    char c;
    char majus(char c);

    ...

    printf("%c\n", majus(c));
    return;
}
char majus(char c)
{
    return c >= 'a' && c <= 'z' ?
        c + 'A' - 'a' : c;
}
```

6 – Les fonctions



- Définition d'une fonction
 - Retour de l'appel
 - Appel d'une fonction
 - Déclaration d'une fonction
- Passage d'arguments

Définition d'une fonction

```
type nom (type_1 arg_1, ..., type_n arg_n )  
    {  
        déclarations des var. locales;  
        instructions;  
    }
```

- ☑ Le *type* indique le type de la valeur renvoyée par la fonction.
- ☑ Le *nom* de la fonction doit respecter les mêmes règles que celles concernant les noms des variables.
- ☑ Les *arguments* de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée
- ⊙ Le type d'une fonction par défaut est *int* et non pas void

Retour de l'appel

- ✓ Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par *l'instruction de retour à la fonction appelante*, **return**, dont la syntaxe est:

return (expression);

- ✓ La valeur de **expression** est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type **void**), sa définition s'achève par:

return;

- ✓ Plusieurs instructions return peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le **premier return rencontré** lors de l'exécution. Voici quelques exemples de définitions de fonctions :

Retour de l'appel

Exemples

```
int produit (int a, int b)
{
    return(a*b);
}
```

```
int puissance (int a, int n)
{
    if (n == 0)
        return(1);
    return(a * puissance(a, n-1));
}
```

```
void imprime_tab (int *tab, int nb_elements)
{
    int i;
    for (i = 0; i < nb_elements; i++)
        printf("%d \t", tab[i]);
    printf("\n");
    return;
}
```

Appel d'une fonction

- ✓ L'appel d'une fonction se fait par l'expression :

nom_fonction(param_1,param_2,...,param_n)

- ✓ L'ordre et le type des **paramètres effectifs** de la fonction doivent concorder avec ceux donnés dans l'entête de la fonction
- ✓ Les paramètres effectifs peuvent être des expressions

Déclaration d'une fonction

- ☑ La définition d'une fonction doit être placée soit avant, soit après la fonction principale *main()*.
- ☑ Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée: de même que pour une variable, une fonction doit être, donc, déclarée avant d'être utilisée.
- ☑ la **déclaration** d'une fonction se fait à travers son **prototype**:
- ☑ un **prototype**, donne le type de la fonction et celui de ses paramètres, sous la forme :
type nom_fonction(type_1,...,type_n);

Déclaration d'une fonction

Déclaration
Prototype



```
#include <stdio.h>
/*Prototypes des fonctions*/
int  somme(int *, int );
```

Appel



```
main()
{
    int  tab[5] = { 1, 9, 10, 14, 18};
    int s;

    s = somme(tab, 5);
    printf("la somme des elements = %d \n", s);
}
```

Définition



```
int  somme(int *t, int n)
{
    int i, som=0;
    for (i=0; i < n; i++) som = som + t[i];
    return som;
}
```

Passage des arguments

- ✓ Dans les langages de programmation il existe deux techniques de passage d'arguments:

- ☞ par adresse

- ☞ par valeur

- ✓ Des langages comme *Fortran* utilise la 1^{ère} solution, tandis qu'un langage comme *Pascal* offre les deux possibilités au programmeur.
- ✓ Le langage C a choisi la 2^e solution: Par Valeur
- ✓ Si un argument doit être passé par adresse, c'est le programmeur qui en prend l'initiative et ceci grâce à l'opérateur d'adressage (&).

Passage des arguments

Exemple 1.1 : le but est d'échanger les valeurs de deux variables

```
/** exemple_fonction_2a.c **/  
#include <stdio.h>  
void echangel (int a, int b)  
{  
    int tmp;  
  
    tmp = a;  
    a = b;  
    b = tmp;  
}  
  
main()  
{  
    int x = 1, y = 2;  
  
    echangel(x,y);  
    printf("x = % d , y = %d", x, y);  
}
```

exemples

exemple_fonction_2a.c

Passage des arguments

- ☑ La fonction `echangel()` ne donne pas ici le résultat désiré: la permutation des valeurs de variables passées en paramètres.
- solution: il faut passer les **adresses** de deux variables à modifier.
- ceci est possible en passant ces adresses à travers des variables de type **pointeur**.

Passage des arguments

Exemple 1.2

```
/** exemple_fonction_2b.c */
#include <stdio.h>
void exchange2 (int *a, int *b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

main()
{
    int x = 1, y = 2;

    exchange2 (&x , &y) ;
    printf("x = % d , y = %d" , x, y) ;
}
```

exemples

exemple_fonction_2b.c

Passage des arguments

- ☑ Remarque: le mode de passage des arguments de la fonction `echange2()` est toujours un mode de **passage par valeur**; la seule différence est que les valeurs passées en paramètres sont des **adresses** et qu'il est donc possible de modifier les valeurs référencées.

Passage des arguments

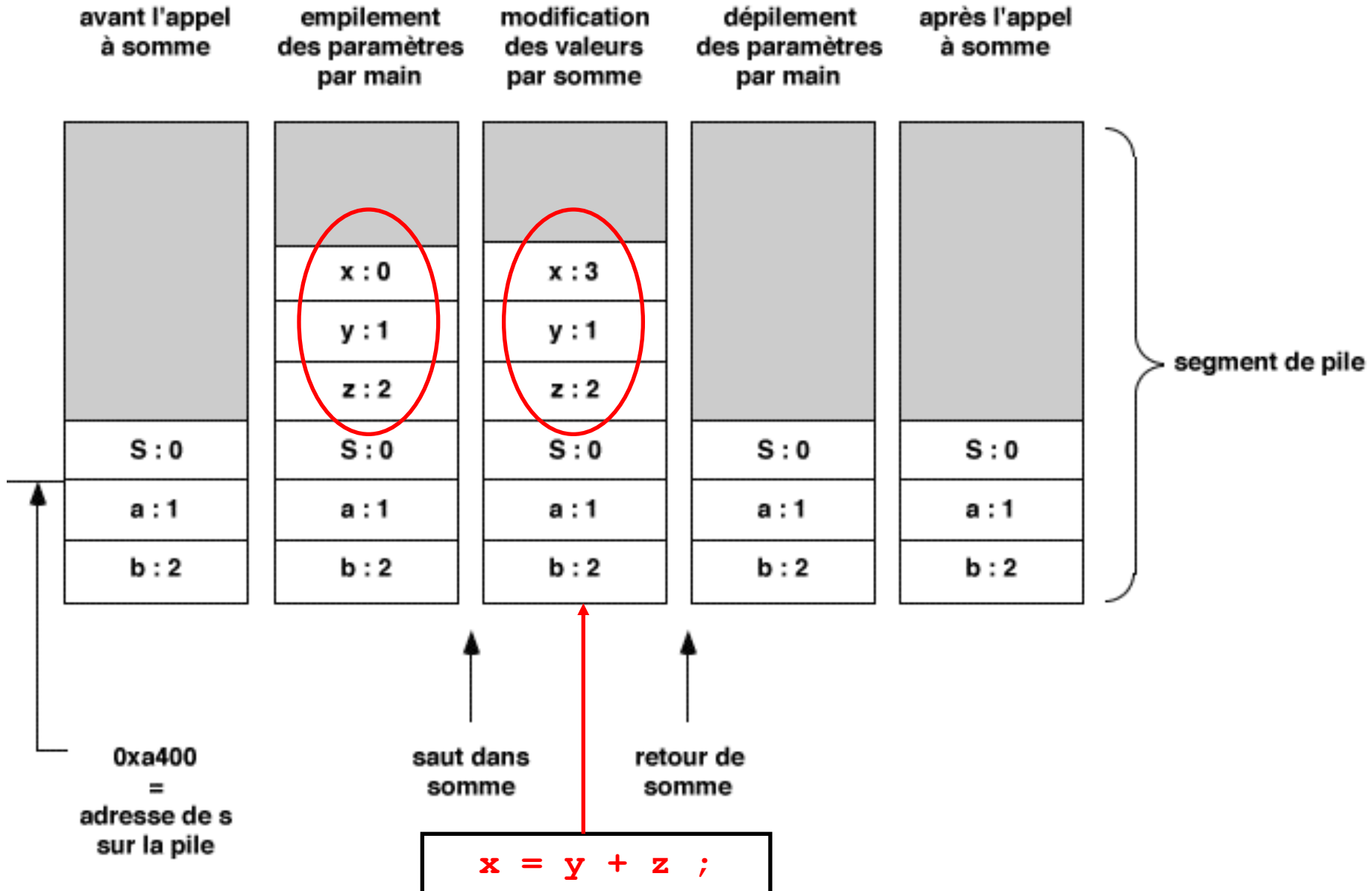
Exemple 2.1

```
/** exemple_fonction_3a.c */  
  
#include <stdio.h>  
  
void somme(int x, int y, int z)  
{  
    x = y + z ;  
}  
  
main()  
{  
    int s , a , b;  
    s = 0;  
    a = 1;  
    b = 2;  
    somme(s, a, b);  
    printf("%d + %d = % d ", a, b , s);  
}
```

exemples

exemple_fonction_3a.c


Passage des arguments



Passage des arguments

Exemple 2.2

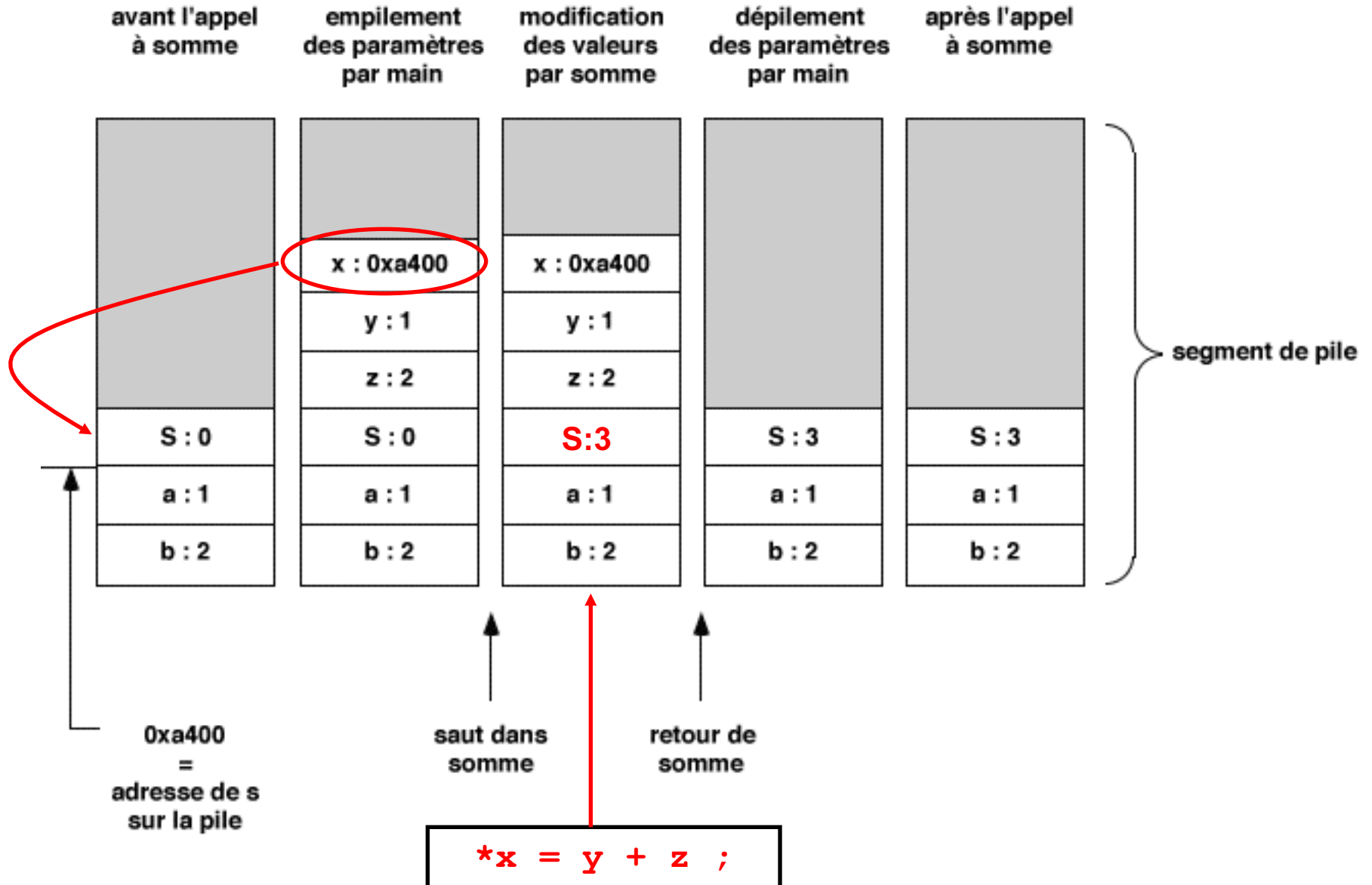
```
/** exemple_fonction_3b.c */  
  
#include <stdio.h>  
  
void somme(int *x, int y, int z)  
{  
    *x = y + z ;  
}  
  
main()  
{  
    int s , a , b;  
    s = 0;  
    a = 1;  
    b = 2;  
    somme(&s, a, b);  
    printf("%d + %d = % d ", a, b , s);  
}
```



exemples

exemple_fonction_3b.c

Passage des arguments



Passage des arguments

Passage d'un tableau comme argument

- ☑ Un tableau est une constante symbolique dont la valeur est l'adresse de son 1^{er} élément. Lorsqu'un tableau est passé en argument, c'est donc l'**adresse** de son 1^{er} élément qui est transmise par valeur.

Passage des arguments


Exemple 1: sans modification du tableau

```
/** exemple_fonction_4a.c */
#include <stdio.h>
/*Prototypes des fonctions*/
int  somme(int *, int );

main()
{
    int  tab[5] = { 1, 9, 10, 14, 18};
    int  s;

    s = somme(tab, 5);
    printf("la somme des elements = %d \n", s);
}

int  somme(int *t, int n)
{
    int  i, som=0;
    for (i=0; i < n; i++) som = som + t[i];
    return som;
}
```



exemples

exemple_fonction_4a.c

Passage des arguments

Exemple 2: avec modification du tableau

```
/** exemple_fonction_4b.c **/  
#include <stdio.h>  
/*Prototypes des fonctions*/  
int  somme(int *, int );  
void initialisation(int *, int );  
main()  
{  
    int s, tab[5] ;  
  
    initialisation(tab, 5);  
    s = somme(tab, 5);  
    printf("la somme des elements = %d \n", s);  
}  
int  somme(int *t, int n)  
{...}  
void initialisation(int *t, int n)  
{  
    int i=0;  
    for (i=0; i < n; i++){  
        printf("t[%d] = ? \n", i);  
        scanf("%d", t);  
        t++;  
    }  
}
```

exemples

exemple_fonction_4b.c

exemple_fonction_4c.c

Passage des arguments

Remarque Importante

☑ Ces déclarations sont toutes équivalentes:

☞ `void initialisation(int *t, int n)`

☞ `void initialisation(int t[], int n)`

☞ `void initialisation(int t[5], int n)`

☑ La fonction `initialisation()` récupère toujours le paramètre formel `t` comme **un pointeur (variable, locale) vers int** qui peut représenter (ou non) un tableau.

passage d'une structure comme argument

- ☑ une structure peut être passée en argument par sa valeur ou à travers son adresse. En général on préfère passer une structure à travers un **pointeur** afin de limiter le nombre des valeurs à mettre dans la pile d'appel lors de l'exécution du programme.

Passage des arguments

Exemple 3.2: passage en paramètre d'une structur

```
/** exemple_fonction_6.c **/
```

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

```
typedef struct {
```

```
    char nom[80];
```

```
    char prenom[80];
```

```
    int cle;
```

```
}personne;
```

```
/*Prototypes des fonctions*/
```

```
void saisir(personne *);
```

```
void afficher(personne *);
```

```
main()
```

```
{
```

```
    personne p;
```

```
    saisir(&p);
```

```
    afficehr(&p);
```

```
}
```

```
void saisir(personne *ptr)
```

```
{
```

```
    printf("Nom prenom cle \n");
```

```
    scanf("%s%s%d", ptr->nom, ptr->prenom, &(ptr->cle));
```

```
}
```

```
void afficher(personne *ptr)
```

```
{
```

```
    printf("Nom: %s Prenom: %s Cle: %d \n", ptr->nom, ptr->prenom,  
ptr->cle);
```

```
}
```

```
personne *p;
```

```
p= (personne *)malloc(sizeof(personne));
```

```
saisir(p);
```

```
affichehr(p);
```

exemples

exemple_fonction_6.c

Passage des arguments

A RETENIR

- ☑ Dés lors que le contenu d'une variable doit être modifié par une fonction, le paramètre correspondant doit être du type pointeur.
- ☑ un tableau est passé en argument à une fonction à travers un pointeur.
- ☑ Les arguments de type pointeur seront généralement utilisés pour passer en paramètre une variable de type structure.