

# Opérateurs

- Opérateurs **binaires** de comparaison

Opérateur	Utilisation
> <	opérateur de test de supériorité, d'infériorité
>= <=	opérateur de test de supériorité ou égalité, d'infériorité ou égalité
= =	opérateur de test d'égalité
!=	opérateur de test de différence



A ne pas confondre avec = opérateur d'affectation

# Opérateurs

- Opérateurs **unaires** et **binaires** logiques

Opérateur	Utilisation
!	opérateur de négation
&&	opérateur de ET Logique
	opérateur de OU Logique

x	V	V	F	F
y	V	F	V	F
x&& y	V	F	F	F
x   y	V	V	V	F
!x	F	F	V	V

# Opérateurs

- Opérateurs **unaires** et **binaires** logiques

Opérateur	Utilisation
!	opérateur de négation
&&	opérateur de ET Logique
	opérateur de OU Logique

x	V	V	F	F
y	V	F	V	F
x&& y	V	F	F	F
x   y	V	V	V	F
!x	F	F	V	V

# Opérateurs

- Opérateurs **unaires** et **binaires** de traitement bas niveau
  - directement au niveau des bits
  - s'appliquent sur des **int** ou des **char**

Opérateur	Utilisation	
~	donne le complément à un d'une opérande inversion de tous les bits	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  28 \quad 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\  \sim 28 \quad 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \\  (227)  \end{array}  $
&	ET réalise un ET logique sur chacun des bits des 2 opérandes Sert à masquer certains bits	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  6 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \\  \& \quad 10 \quad \underline{0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0} \\  2 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0  \end{array}  $
	OU inclusif réalise un OU logique sur chacun des bits des 2 opérandes met à 1 tous les bits de op1 qui sont à 1 dans op2	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  1 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \\    \quad 121 \quad \underline{0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1} \\  121 \quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1  \end{array}  $

# Opérateurs

- Opérateurs **binaires** de traitement bas niveau (suite)

Opérateur	Utilisation	
<b>^</b>	OU exclusif met à 0 tous les bits de op1 qui sont identiques dans op2, et à 1 ceux qui diffèrent	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  1 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \\  \wedge \\  121 \quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \\  120 \quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0  \end{array}  $
<b>&lt;&lt;</b> op1<<op2	opérateur de décalage à gauche  décalage vers la gauche de op2 bits de op1, les bits de droite sont remplacés par des 0	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  1 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \\  1 \ll 2 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\  (4)  \end{array}  $ <p>Chaque décalage à gauche <math>\Leftrightarrow *2</math></p>
<b>&gt;&gt;</b> op1>>op2	opérateur de décalage à droite  décalage vers la droite de op2 bits de op1, les bits de droite sont remplacés par des 0 pour les entiers non signés, par des 0 ou bit de signe pour les entiers signés	$  \begin{array}{r}  2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \\  8 \quad 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \\  8 \gg 2 \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\  (2)  \end{array}  $ <p>Chaque décalage à droite <math>\Leftrightarrow /2</math></p>

# Opérateurs

- Opérateurs ternaire

Opérateur	Utilisation	
<code>? :</code> <code>op1? op2 : op3</code>	<p>Met en jeu 3 opérandes (ici des expressions) et construit une opération de test avec retour de valeur Opérande1 <math>\Leftrightarrow</math> condition du test</p> <p>Si la condition est vérifiée, l'opérande 2 est évaluée et le résultat de l'évaluation est retourné</p> <p>Sinon c'est l'opérande 3 qui est évaluée et le résultat de l'évaluation est retourné</p>	<p><code>int a=3, b=5, res;</code>  <code>res = (a&lt;=b) ? a : b;</code></p> <p>a <span style="border: 1px solid orange; padding: 2px;">3</span>    b <span style="border: 1px solid orange; padding: 2px;">5</span>    res <span style="border: 1px solid orange; padding: 2px;"><del>?3</del></span></p>

```
ValeurAbsolue.c
/*
 * Calcul et affichage de la valeur absolue
 */
#include <stdio.h>
#include <stdlib.h>

int main(){
    float nb;
    float val_abs;
    printf("Entrez une valeur numérique\n");
    fflush(stdout);
    scanf("%f",&nb);
    val_abs= (nb<0)? -nb : nb;
    printf("la valeur absolue de %.2f est %.2f\n",nb,val_abs);
    return EXIT_SUCCESS;
}
```

```
Problems Tasks Console Properties
<terminated> ProgrammeValeurAbsolue.exe [C/C++ Local App
Entrez une valeur numérique
-78.56
la valeur absolue de -78.56 est 78.56
```

```
Problems Tasks Console Properties
<terminated> ProgrammeValeurAbsolue.exe [C/C++ Local App
Entrez une valeur numérique
23.75
la valeur absolue de 23.75 est 23.75
```

# Opérateurs

- Priorité **+ prioritaire**

**- prioritaire**

( ), [ ], ->, .

~, ++, --, (signe +/-), \*(adresse), &, sizeof(var), !

\*, /, %

+, -

<<, >>

< <=, > >= (opérateurs de comparaison)

==, !=

&

^

|

&&

||

? :

tous les combinés

# Structures de contrôle

---

- Branchement conditionnel
  - Expression d'une prise de décision
    - Choix simple  $\Leftrightarrow$  si .....alors .... sinon
    - Choix multiple
- Bouclage
  - Répétition d'instructions



# Structures de contrôle

- Prise de décision par choix simple

```
if (condition){
```

```
....
```

```
}else{
```

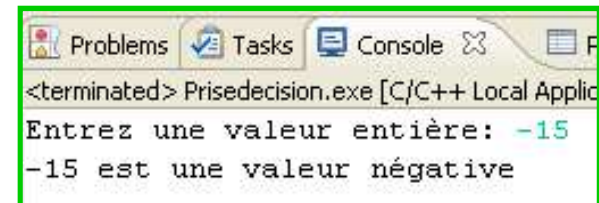
```
.....
```

```
}
```

```
#include <stdio.h>

int main() {
    int x;
    printf("Entrez une valeur entière: ");
    fflush(stdout);
    scanf("%d", &x);

    if (x < 0) {
        printf("%d est une valeur négative", x);
    } else {
        printf("%d est une valeur positive\n", x);
    }
    return 1;
}
```

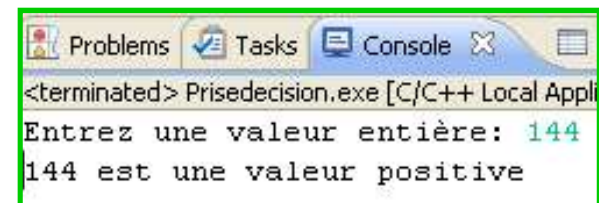


Problems Tasks Console

<terminated> Prisedecision.exe [C/C++ Local Appli

Entrez une valeur entière: -15

-15 est une valeur négative

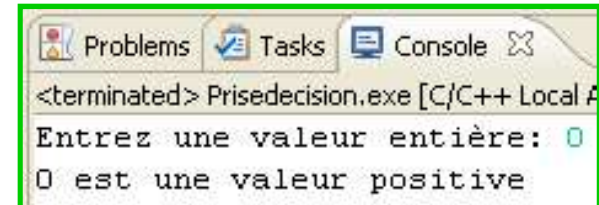


Problems Tasks Console

<terminated> Prisedecision.exe [C/C++ Local Appli

Entrez une valeur entière: 144

144 est une valeur positive



Problems Tasks Console

<terminated> Prisedecision.exe [C/C++ Local A

Entrez une valeur entière: 0

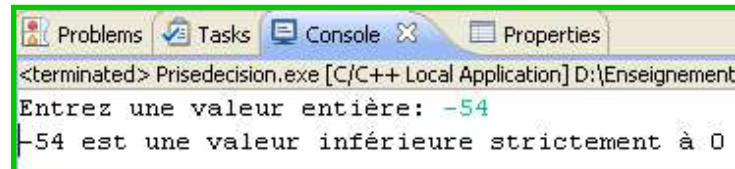
0 est une valeur positive

# Structures de contrôle

- Prise de décision par choix simple
  - Les *if* et *else* peuvent s'enchaîner

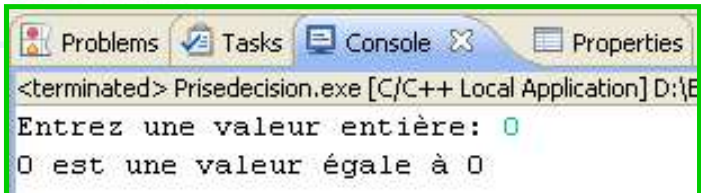
```
#include <stdio.h>
```

```
int main() {  
    int x;  
    printf("Entrez une valeur entière: ");  
    fflush(stdout);  
    scanf("%d", &x);  
  
    if(x<0) {  
        printf("%d est une valeur inférieure strictement à 0", x);  
    } else if (x>0) {  
        printf("%d est une valeur supérieure strictement à 0\n", x);  
    } else {  
        printf("%d est une valeur égale à 0\n", x);  
    }  
  
    return 1;  
}
```

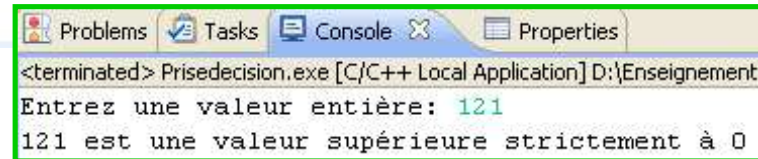


<terminated> Prisedecision.exe [C/C++ Local Application] D:\Enseignement  
Entrez une valeur entière: -54  
-54 est une valeur inférieure strictement à 0

En l'absence de { }, un *else* se raccroche toujours au *if* immédiatement supérieur en partant du bas



<terminated> Prisedecision.exe [C/C++ Local Application] D:\E  
Entrez une valeur entière: 0  
0 est une valeur égale à 0



<terminated> Prisedecision.exe [C/C++ Local Application] D:\Enseignement  
Entrez une valeur entière: 121  
121 est une valeur supérieure strictement à 0

# Structures de contrôle

- Prise de décision par choix multiple
  - Pour éviter les imbrications de *if* dans le cas de choix multiples sur des *valeurs constantes*
  - *switch* – syntaxe



Ne pas oublier l'instruction *break* à la fin de chaque *case*

## Syntaxe table de branchement

```
switch(expression){  
    case value1:  instruction1;  
                  instruction 2;  
                  .....  
                  break;  
  
    case value2 :  
  
    case value3 : instruction1;  
                  break;  
  
    default :     instruction1;  
}
```

## Règles

- L'expression est évaluée comme une valeur entière
- Les valeurs des case sont évaluées comme des constantes entières (int, short, char )
- L'exécution se fait à partir du *case* dont la valeur correspond à l'expression et se termine à la rencontre de l'instruction *break*
- Les instructions qui suivent la condition *default* sont exécutées quand aucune constante des *case* n'est égale à la valeur évaluée de l'expression

# Structures de contrôle

- switch
  - illustration

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

int main(void) {
    int x;
    printf("Entrez un nombre entier entre 0 et 9: \n");
    fflush(stdout);
    scanf("%d",&x);
    if(x==6 || x==8){
        printf("Le nombre est supérieur à 5\n");
        printf("Le nombre est pair\n");
    }else if(x==4 || x==2 || x==0){
        printf("Le nombre est pair\n");
    }else if(x==7 || x==9){
        printf("Le nombre est supérieur à 5\n");
        printf("Le nombre est impair\n");
    }else if (x==5 || x==3 || x==1)
        printf("Le nombre est impair\n");
    else
        printf("ce n'est pas un nombre compris entre 0 et 9");
    return EXIT_SUCCESS;
}
```

```
TestSwitch.c X
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x;
    printf("Entrez un nombre entier entre 0 et 9: \n");
    fflush(stdout);
    scanf("%d",&x);
    switch(x){
        case 0: case 2: case 4: printf("Le nombre est pair\n");
                        break;
        case 1: case 3: case 5 : printf("Le nombre est impair\n");
                        break;
        case 6: case 8: printf("Le nombre est supérieur à 5\n");
                        printf("Le nombre est pair\n");
                        break;
        case 7: case 9: printf("Le nombre est supérieur à 5\n");
                        printf("Le nombre est impair\n");
                        break;
        default:
            printf("ce n'est pas un nombre compris entre 0 et 9");
            break;
    }
    return EXIT_SUCCESS;
}
```

# Structures de contrôle

- Répétitions d'une suite d'instructions ⇔ bouclage

Répétition d'instructions	Règles
<pre>while(expression){     instructions; }</pre>	<ul style="list-style-type: none"><li>• Répétition des instructions tant que la valeur de l'expression s'interprète comme vraie (différente de 0)</li></ul> <pre>graph TD     Entry(( )) --&gt; Expression{expression}     Expression -- Faux --&gt; Exit(( ))     Expression -- Vrai --&gt; Instructions[Instruction(s)]     Instructions --&gt; Entry</pre>

```
int i = 0 ;           //initialisation
while(i < 10){        //condition
    printf("Bonjour N° %d\n ",i) ;
    printf("-----\n ") ;
    i++ ;             //incrémentatation
                      //ou modification de la valeur la variable impliquée dans la condition
}
```

# Structures de contrôle

- Répétitions d'une suite d'instructions  $\Leftrightarrow$  bouclage

Répétition d'instructions	Règles
<pre>for(initialisation; condition_vraie; instruction d'incrémentation){     instructions; }</pre> <pre>graph TD     E1[expression1] --&gt; E2{expression2}     E2 -- Faux --&gt; Exit(( ))     E2 -- Vrai --&gt; I[Instruction(s)]     I --&gt; E3[expression3]     E3 --&gt; E2</pre>	<ul style="list-style-type: none"><li>• Le for s'utilise avec 3 expressions séparées par des ; et qui peuvent être vides</li><li>expression 1 : instruction d'initialisation, exécutée une fois</li><li>expression 2 : condition d'exécution des instructions, testée à chaque tour de boucle</li><li>expression 3 : permet de calculer la prochaine valeur avec laquelle la condition va être testée</li></ul>
<pre>int i; for(i = 0 ; i &lt; 10 ; i++) {     printf("Bonjour N° %d\n ", i);     printf("-----\n "); }</pre>	<pre>int i, j; for(i = 0, j = 10 ; i &lt; j ; i++, j--) {     ..... ; }</pre>

# Structures de contrôle

- Répétitions d'une suite d'instructions  $\Leftrightarrow$  bouclage

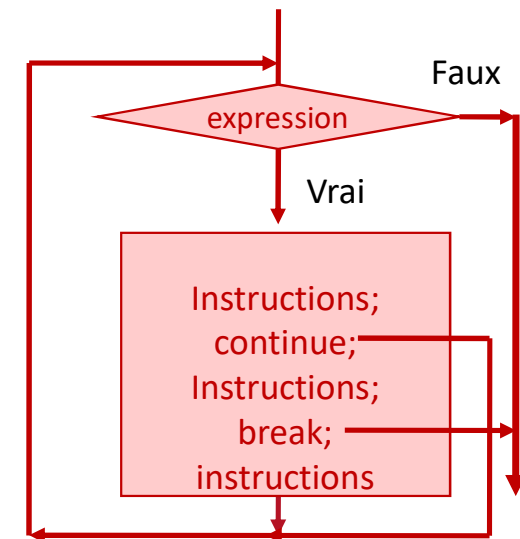
Répétition d'instructions	Règles
<pre>do{   instructions; } while(expression);</pre>	<ul style="list-style-type: none"><li>• test de continuation se fait en fin de boucle</li><li><math>\Leftrightarrow</math> on passe au moins une fois dans la boucle</li></ul> <pre>graph TD; Entry(( )) --&gt; Instructions[Instruction(s)]; Instructions --&gt; Expression{expression}; Expression -- Faux --&gt; Exit(( )); Expression -- Vrai --&gt; Entry;</pre>

```
int i = 0 ;           //initialisation  
do{  
  printf("Bonjour N° %d\n ",i) ;  
  printf("-----\n ") ;  
  i++ ;  
} while(i < 10) ;
```

# Structures de contrôle

- Structures itératives peuvent être imbriquées
  - Respecter les règles d'indentation pour la lisibilité du code
- Ruptures de séquences répétitives
  - Instructions correspondant aux mots clés

Mots clés	Utilisation
<del>goto +label</del>	<del>Permet de sortir d'une séquence et d'aller directement au niveau de l'étiquette précisée par label</del>
continue	Provoque le passage à l'itération suivante de la boucle, en sautant directement à la fin de la structure répétitive
break	Provoque la sortie de la structure répétitive



- Le recours à ces mots clés est souvent lié à une mauvaise réflexion au niveau des tests d'arrêt des structures répétitives



# COURS 3

---

Programmation impérative

Langage C – Éléments de base

- les pointeurs
- les tableaux

# SOMMAIRE

---

- Informations pratiques
- Introduction
- **Eléments de base**
  - Programmer en Langage C – Compilation
  - Structure d'un programme / Règles d'écritures
  - Types de base
  - Constantes/Variables
  - Opérateurs
  - Structures de contrôle
  - Pointeurs
  - Tableaux
- Fonctions
- Chaînes de caractères
- Pointeurs- Tableaux-Fonctions
- Types Construits
- Entrées – Sorties sur Fichiers
- Compilation séparée
- Implémentation de Types Abstraits de Données

# Types dérivés

---

- Types dérivés des types de base
  - Type créé à partir des types de base vus précédemment
    - Pointeurs
    - Tableaux
    - Structures (à voir plus tard dans la section types construits)

# Pointeur

---

- Variable destinée à contenir une adresse mémoire au lieu d'une valeur
- Taille mémoire nécessaire pour stocker un pointeur est connue du compilateur (2 ou 4 octets)
- Syntaxe de déclaration
  - Utilisation de l'opérateur \*
  - Association d'un type
    - Compilateur vérifie le type des adresses mises dans un pointeur
    - Le type du pointeur conditionne les opérations arithmétiques sur ce pointeur
    - l'objet atteint par l'intermédiaire du pointeur possède toutes les propriétés du type correspondant

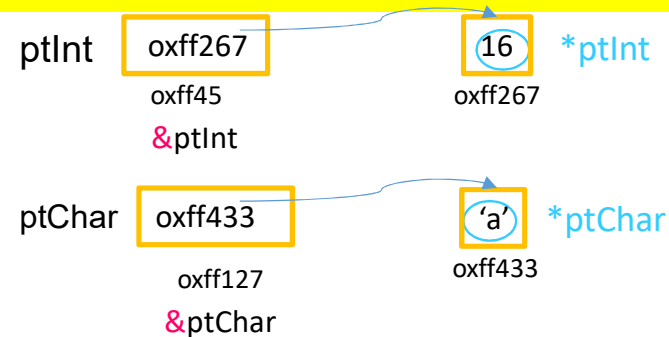
# Pointeur

- Définition de pointeur

Exemples	
<code>int *ptInt;</code>	ptInt est une variable de type pointeur sur un entier Peut contenir l'adresse d'une variable de type int
<code>char *ptChar;</code>	ptChar est une variable de type pointeur sur un caractère Peut contenir l'adresse d'une variable de type char



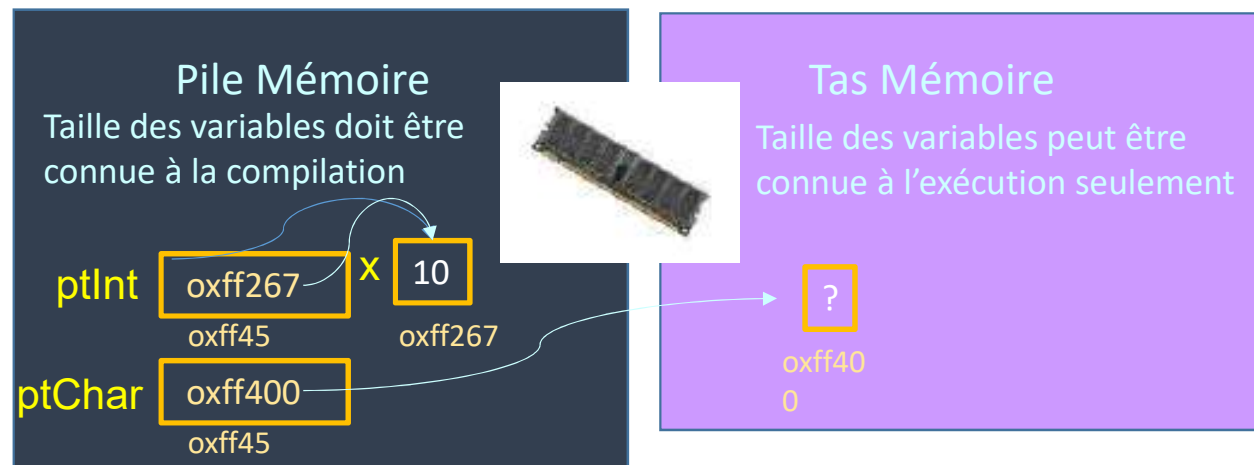
En supposant que les pointeurs et les cases mémoires sur lesquelles elles pointent ont été initialisés



# Pointeur

- Initialisation de pointeur

Exemples	
<pre>int *ptInt, x=10; ptInt = &amp;x;</pre>	Affectation d'une adresse existante dans la pile
<pre>char *ptChar, ptChar=(char *) malloc(sizeof(char));</pre>	Allocation dynamique d'une adresse sur le tas



# Pointeur

---

- Allocation dynamique dans le tas mémoire (heap)
  - Fonctions qui permettent de demander au système une zone mémoire d'une certaine taille, au cours de l'exécution du programme
  - Fonctions dans la librairie **stdlib.h**

Fonction	Rôle
<b>void * malloc(size_t taille)</b>	Allocation d'une zone mémoire dans le tas
<b>void * calloc(size_t nb_elt, size_t taille);</b>	Permet l'allocation dynamique d'une zone de cases contiguës en mémoire et initialise les cases allouées à 0
<b>void * realloc(void * oldBloc, size_t newSize);</b>	Permet de changer la taille d'une zone allouée par malloc ou calloc

# Pointeur

- Soit la définition des variables suivantes

**int \*p1,\*p2,\*p3,\*p4;**

Allocation	Résultat
<pre>p1 = (int *) malloc(sizeof(int)) ;  p2 = (int *) malloc(5 * sizeof(int)) ;</pre>	<p>p1 contient l'adresse d'un nouvel emplacement mémoire alloué dans le tas qui peut contenir un entier</p> <p>p2 contient l'adresse de la première case d'un nouvel emplacement mémoire alloué dans le tas qui peut contenir 5 entiers.</p>
<pre>p3 = (int *) calloc(5, sizeof(int)) ;</pre>	<p>p3 contient l'adresse de la première case d'un nouvel emplacement mémoire alloué dans le tas qui peut contenir 5 entiers ;</p> <p>La valeur de chacun des 5 entiers est mise à 0.</p>
<pre>(int *) realloc(p3, 10*sizeof(int)) ;</pre>	<p>p3 contient l'adresse de la première case d'un nouvel emplacement mémoire, dans le tas qui a été étendu par rapport à celui pointé par p3.</p> <p>Le contenu des cases pointées par p3 est conservé et l'espace mémoire supplémentaire n'est pas initialisé</p>
<pre>p4=(int *) realloc(p3, 10*sizeof(int)) ;</pre>	<p>Il vaudrait mieux récupérer le résultat du realloc dans un pointeur p4 dans le cas où la place mémoire à la suite n'est pas disponible</p> <p>⇔éviter les fuites mémoires</p>



# Pointeur

- Représentation mémoire

