Cours 1

C’est chou comme écriture

C’est chou comme écriture, vraiment trop kiki hihihihihihihi

C’est chou comme écriture, vraiment trop kiki **hihihihihihihi** bonjour je

**C’est chou comme écriture**, vraiment trop kiki **hihihihihihihi** bonjour je

C’est chou comme écriture, vraiment trop kiki **hihihihihihihi** bonjour je m’appelle

| /\* fichiers en-tête qui contiennent les déclarations (prototypes)  \* de fonctions utilisées dans le programme \*/  #include <stdio.h>  int main(void){  int tab[]={-5, 8, 12, 9, -4, 21, -31};  double s = 0;  /\* calculer le nombre d'elements de tab \*/  int taille = sizeof(tab)/sizeof(tab[0]);  /\* calculer la somme de tous les éléments \*/  for(int i=0 ; i < taille ; i++){  s += tab[i];  }  s/=taille; /\* la moyenne s = s/taille \*/  printf("somme=%8.3f\n", s); /\*imprimer le resultat\*/  return 0; /\*main doit retourner un int\*/  } |
| --- |
| Explications   * L'exécution de programme commence toujours dans la fonction main:   + int main(void){ } * main() doit retourner un entier, la valeur 0 indique terminaison correcte du programme, une valeur > 0 un terminaison incorrecte * La fonction printf() est définie dans le fichier en-tête stdio.h sert à faire afficher les valeurs d'expressions |

# Compiler le programme C

compiler un programme qui se trouve dans fichier hello.c sur un terminal :

* gcc -Wall hello.c -o hello
* gcc - le compilateur C

-Wall une option de compilateur (obligatoire). Permet de voir tous les avertissements (warning). Compilation avec des avertissements signifie que le programme est presque sûrement incorrect (mais si la compilation réussie).

Exécuter le programme compilé depuis le terminal : ./hello

# Types entiers

Types entiers signés :

* signed char
* short (short int)
* int
* long (long int)
* long long (long long int)

Types entiers non-signés :

* unsigned char
* unsigned short
* unsigned int
* unsigned long
* unsigned long long

# Types pour les nombres réels

* float (à ne pas utiliser)
* double
* long double

# Comment C calcule la valeur d'une expression arithmétique ?

Si on mélange des réels et des entiers alors tous les arguments sont transformés en réels appropriés et le résultat est réel.

Avertissement : Les règles de calcule sont peu intuitives si une expression mélange les entiers signés et non-signés.

La règle de bon sens fortement recommandée:

* ne mélangez jamais les entiers signés et non-signés ni dans les expressions ni dans les relations (ou faites un cast sur les entiers non-signés)
* utilisez les non-signés uniquement pour les opérations bit à bit et pour rien d'autre

# Expressions arithmétiques comme conditions logiques

C traite une expression arithmétique dont la valeur est différente de 0 comme la valeur logique VRAI et une expression arithmétique dont la valeur est 0 comme FAUX.

Relations (renvoie 1 ou 0, pas un booléen)

* a < b
* a <= b
* a > b
* a >= b
* a == b
* a != b

# if/switch

| if( condition1 ){  instructions1  }  else if( condition2 ){  instructions2  }  ...  else{  instructions\_n  } |
| --- |

| switch( expression ) {  case expr-const : instructions ;  case expr-const : instructions ;  default : instructions ;  } |
| --- |

# Boucle

| while( condition ){  instructions  } |
| --- |

Remarque : si le corps de la boucle est vide alors on écrit

| while( tab[i++] ) ; /\* boucle vide, il faut un élément qui contient 0 pour l'arrêter \*/ |
| --- |

| do {  instructions  } while( condition ); |
| --- |

(1) on exécute les instructions

(2) on vérifie la condition, si satisfaite alors on revient à (1)

sinon on termine la boucle. La condition vérifiée après chaque exécution de la boucle.La boucle exécutée au moins une fois.

| for( initialisation ; condition ; incrémentation ){ } |
| --- |

* initialisation exécutée une fois, avant l'entrée dans la boucle
* condition est vérifiée au début de la boucle et si vraie alors la boucle est exécutée
* incrémentation évaluée à la fin de chaque boucle et on revient au début, à la vérification de la condition

break provoque la sortie de la boucle, le saut vers la première instruction après la boucle

continue termine l'itération courante de la boucle, on passe à l'itération suivante (on revient au début de la boucle et on refait le test de la condition de terminaison)

# Affection

* a \*= b -> a = a \* b;
* a /= b; -> a = a / b;
* a += b; -> a = a + b;
* a -= b; -> a = a - b;

a = expr;

est une expression dont la valeur est égale à la valeur de expr. Cette expression a un effet de bord qui consiste à affecter une nouvelle valeur à la variable a.

* a = b = c\*d;

# Tableau/vecteur

* il faut initialiser la taille des tableaux : int tab[5] (de 0 à 4)
* pour obtenir le nombre d’élément : sizeof(tab)/sizeof(tab[0])
  + sizeof() renvoie le nombre d’octet
  + ça ne marche pas quand le tableau est en paramètre d’une fonction, on doit ajouter la taille en paramètre sinon)

# Fonctions

* il faut déclarer les fonctions (structure la fonction)
  + fonction (type parametre, …)
    - il faut au moins le type, le nom de paramètre est facultatif

| double somme(int nb, double tab[]);  int main(void){  double tab[] = {-4.8, 6.1, 57.0, 23.99, -11.32, 4.5};  int n = sizeof(tab)/sizeof(tab[i]);  double s = somme(n, tab);  printf("somme = %f\n",s);  }  double somme(int nb\_elem, double t[]){  double s=0;  for(int i=0 ; i < nb\_elem; i++)  s += t[i]; return s;  } |
| --- |

Cours 2

# Eléments basiques

## Vecteur

Les changements de valeur d’un tableau pris en paramètre d’une fonction, s’applique dessus directement sans return (on ne pas return un tableau)

## Formatage du code

* Une déclaration par ligne
* Une instruction élémentaire par ligne
* les accolades ouvrante termine la ligne
* Une seule accolade fermante sur une ligne
* l’étiquette est seule sur une ligne
* if et else sur la même colonne
* Une “;” sur la ligne suivant une boucle vide

## Opération à effet de bord

avec int x = 7

* y = ++x
  + y = 8, x = 8
* y = x++
  + y = 7, x = 8

## Expression ternaire

condition ? val1 : val2 => si la condition est vraie alors val1 sinon val2

# Exécution du code :

Quand on lance gcc il y a en fait trois programmes différents qui s'exécutent à tour de rôle :

* Le préprocesseurs transforme le texte du programme suivant les directives commençant par #. Le résultat est un fichier texte.
* Le compilateur traduit le texte du programme vers un code binaire qui n'est pas encore exécutable.
* Le linker rassemble les différentes parties du code binaire, ajoute les références vers les fonctions des bibliothèques. Le résultat est un code binaire exécutable sur la machine.

# Structure (plus ou moins équivalent class java)

exemple

| struct personne {  char sex;  unsigned int année;  char nom[20];  char prenom[20]; }; // ne pas oublier le point-virgule |
| --- |

| struct personne bill {  .sex = 'm';  .année = 1996;  .nom = "clinton";  .prenom = "bill"; } |
| --- |

| struct personne beta; //déclare une variable de type personne /\* On initialise les champs un par un \* beta.sex = 'm'; \* beta.année = 1996; |
| --- |

* On compare des structures par leur champ et non directement (bill.année == beta.année)
* Pour éviter réécrire “struct” à chaque création d’instance on utilise :

| typedef struct personne personne  // ou |
| --- |

| typedef struct {  char sex;  unsigned int année;  char nom[20];  char prenom[20]; } personne; // structure anonyme, et alias à la fin |
| --- |

* vecteur (tableau) de struct : personnage tab[10]
* une structure peut avoir une autre structure en champ;
* Bill = beta est une affectation possible (contrairement aux vecteurs)
* contrairement aux vecteurs, les champs modifiés dans une fonctions ne changent pas celle de la structure initiale, sauf si c’est retourné dans la variable : p1 = mirror(p1);
* on peut retourner une structure qui contient un vecteur mais pas retourner un vecteur

# Enum

exemple

| enum color {BLUE, RED, GREEN}; //énumère des constantes numériques //typedef est aussi utilisé pour les alias // on peut préciser la valeur  typedef enum color {BLUE = 1, RED = 4} color; |
| --- |

# Goto (go to)

* sert à sortir d’une boucle intriquée

exemple

| for(...){  for(...){  if(...){  goto et;  }  } } et:  //suite code } |
| --- |

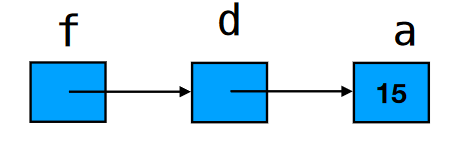
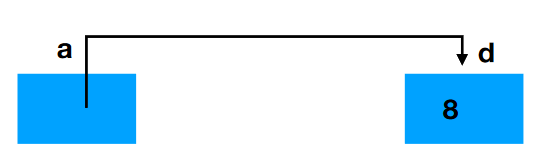
# digression : tableau à plusieurs dimensions

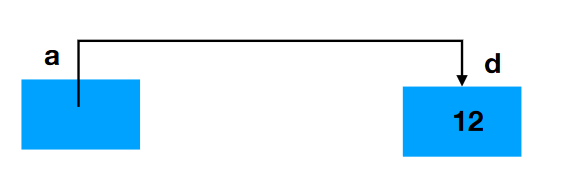
* int t[ ][4] = { {1, 2, 3, 4}, {5, 2, 3, 4}, {1, 1, 1, 8} };
* et non int t[ ] [ ];

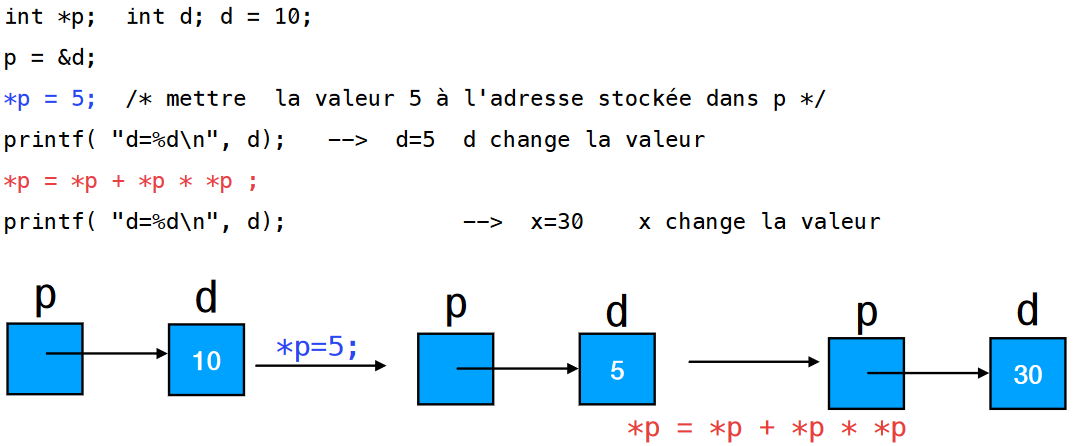
Cours 3

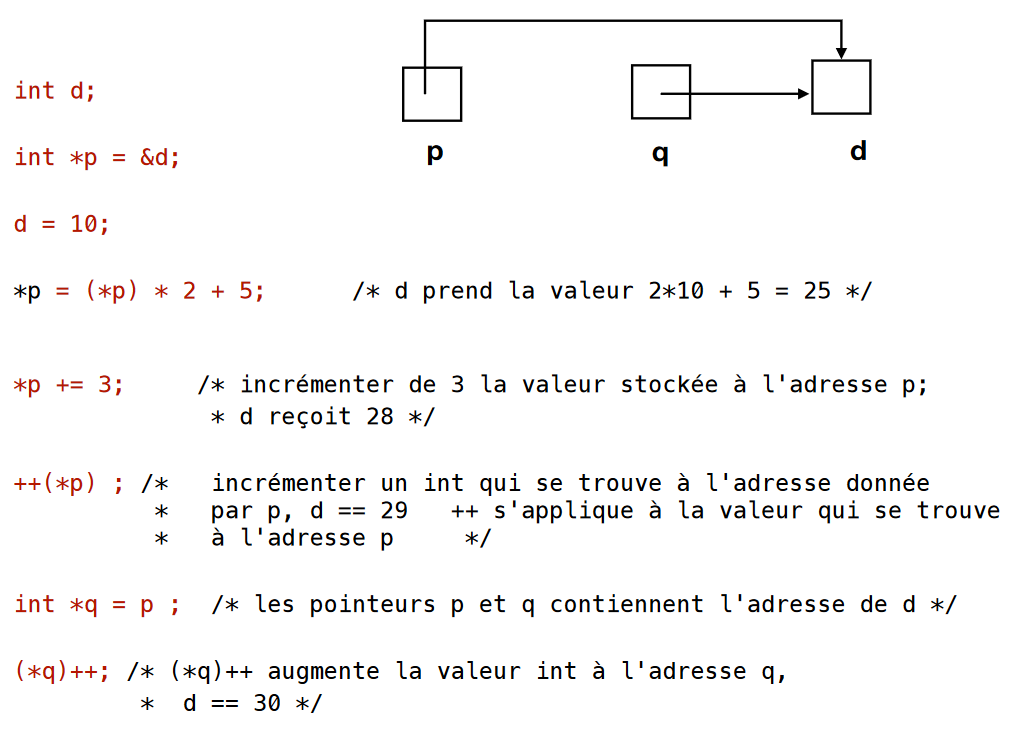
# Pointeurs - arithmétique de pointeurs

## Opérateur &

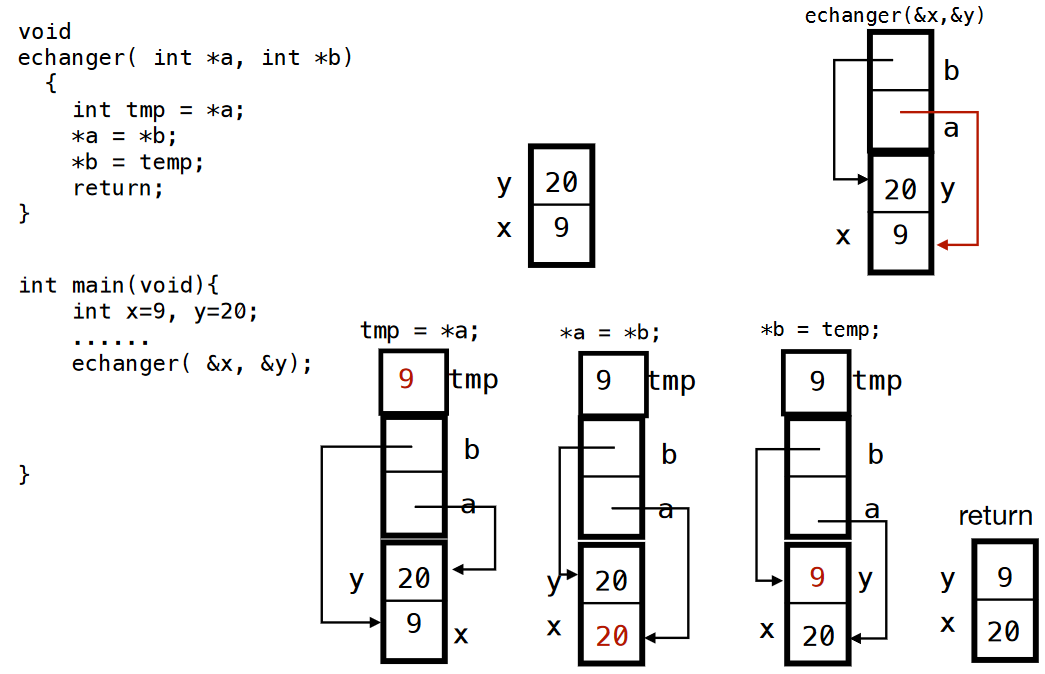
* variable de type pointeur pour mémoriser les adresses :
  + int b = 7;
  + int \*ps = &b; (pointeur de l’adresse de b)
* En C le nom de tableau dans une expression est évalué comme l'adresse du premier élément du tableau
  + int \*pt = tab[0];
  + int \*pq = tab;
  + Les variables pt et pq contiennent l'adresse du premier élément de tab
* int \*a, \*b =/= int \*c, d
* int a = 15; int \*d = &a; int \*\*f = &d;
* int d = 8; int \*a = &d; \*a = 12;

exemple





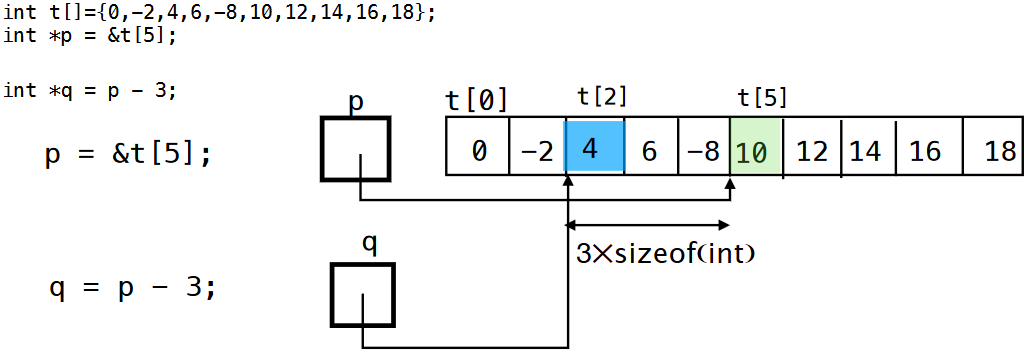
exemple dans une fonction

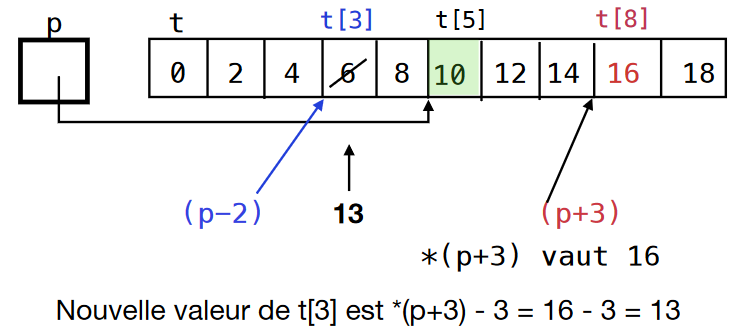


## valeur NULL

* NULL défini dans : stdio.h stddef.h
* NULL une valeur spéciale pour les pointeurs, différente de toutes les adresses réelles.
  + pd == NULL
  + \*pd = 5; provoque l'envoie d'un signal qui termine l'exécution de programme

## arithmétique de pointeurs

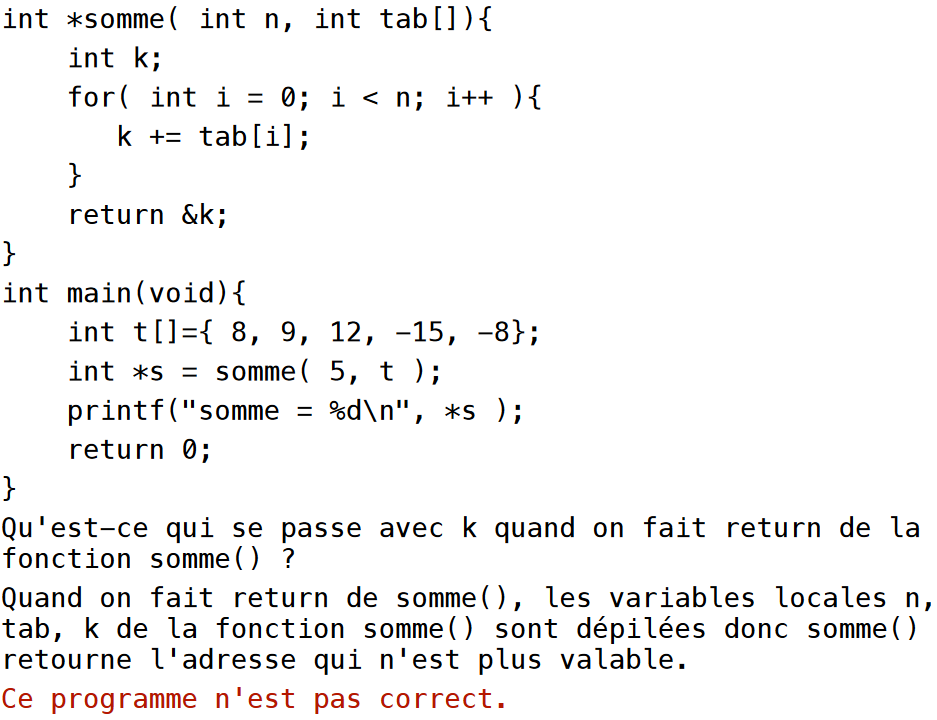
* Si p est un pointeur de type t : t \*p;
* n une expression de type int alors les valeurs des expressions
* p + n et p - n dépendent de type t du pointeur. 
* Le décalage de l'adresse calculé en nombre d'octets est de n \* sizeof( t )
* pour print un pointeur c’est le format : %p

exemple

| int t[]={0,2,4,6,8,10,12,14,16,18};  int \*p = &t[5];  \*(p - 2) = \*(p + 3) - 3; /\* p[-2]=p[3]-3;\*/ |
| --- |

* Le compilateur C traduit **p[k]** et **p[-k]** automatiquement en : **\*(p-k)** et **\*(p+k)**
* Le compilateur duC ne fait (presque) aucune vérification si les adresses calculées à l'aide de pointeurs sont "correctes".
* Le résultat de la différence de deux pointeurs (du même type) est de typeptrdiff\_t. Le type ptrdiff\_t défini dans stddef.h
* ptrdiff\_tun type entier signé qui dépend de l'implémentation.
* Lors de la traduction, le paramètre **int t[]** est traduit en **int \*t**, avec comme valeur &tab[0]
* a = moy(4, &tab[3]); **/\* moyenne sur les éléments tab[3]...tab[6] \*/**
* Il existe aussi les vrais pointeurs de tableau
  + int \*p\_t = &tab[0];
  + int \*p\_q = tab;
  + int (\*p\_tab)[4] = &tab;
  + Le type de la variable p\_tab est différent des types des variables p\_t et p\_q (mais les trois variables contiennent la même adresse après les trois affectations).

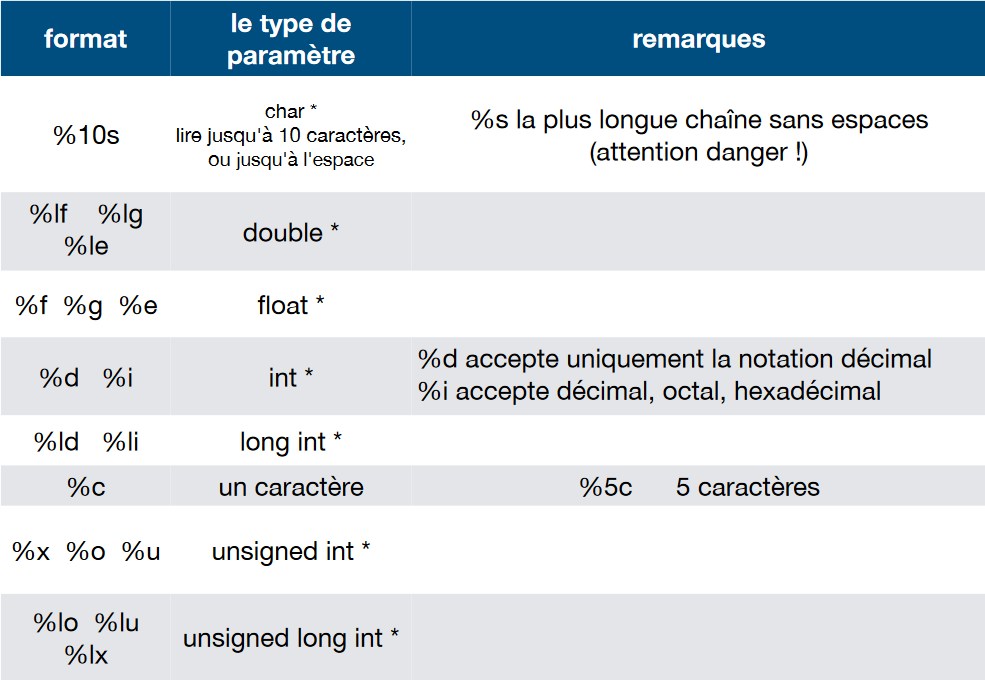
exemple



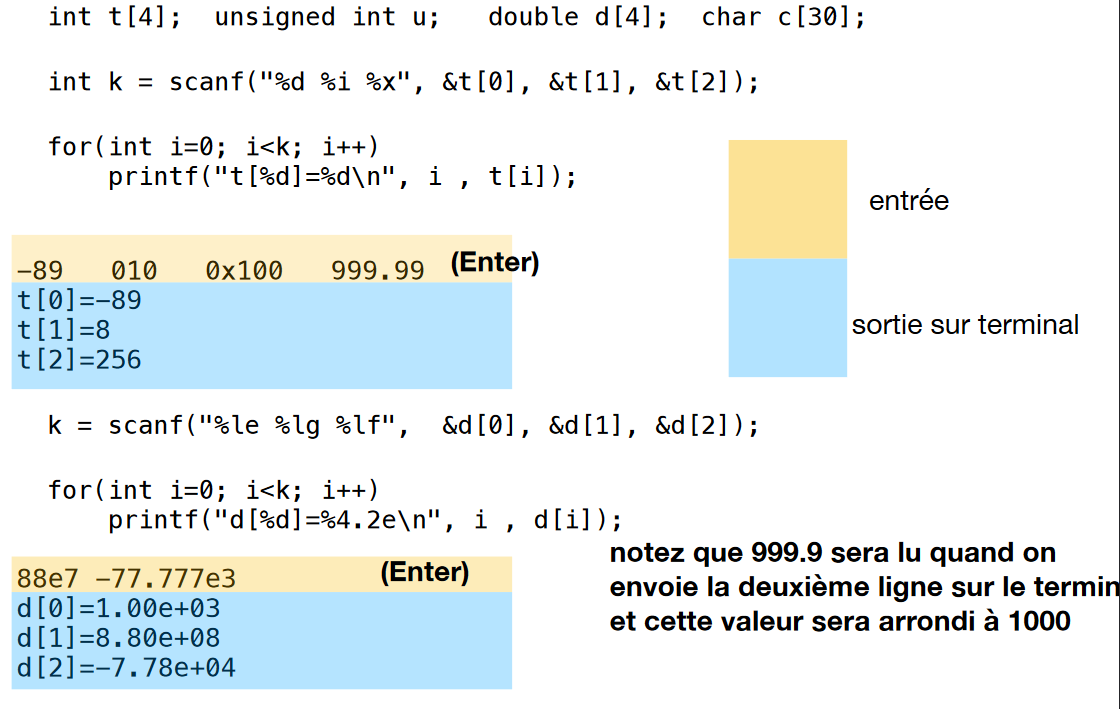
* A quoi sert le pointeur générique?
  + Nous pouvons faire une affectation entre un pointeur générique et un autre pointeur sans retypage "cast". C garantie que la valeur du pointeur est préservée.
  + Arithmétique de pointeurs ne s'applique pas aux pointeurs génériques
    - (t + 1) et (t - 1) => aucun sens
  + L'application de l'opérateur \* n'a pas de sens pour le pointeurs génériques
    - int k = \*t + 2; => aucun sens

# Scanf(), lecture sur le terminal

* scanf() lit depuis les terminal et met les valeurs lues dans des variables, le premier paramètre de scanf() le format, tous les paramètres suivant sont des pointeurs donnant les adresses de variables où scant place les valeurs lues.
* scanf() retourne les nombres d'éléments lus
  + int a,b; int l = scanf("%i %i\n", &a, &b); **/\* l sera 0 si la lecture échoue, 1 si la lecture de a réussie mais pas de b, 2 si la lecture de a et b réussie \*/**
* Un caractère blanc c'est un caractère espace ou le caractère de nouvelle ligne.
* Le caractère à l'entrée qui n'est pas "matché" par le format reste dans le tampon de l'entre, scanf() retourne sans lire la suite.





exemple

Cours 4

# allocation dynamique de la mémoire

allouer et libérer la mémoire

| #include <stdlib.h> void \*malloc(size\_t size) void \*calloc(size\_t count, size\_t size) void \*realloc(void \*ptr, size\_t size) void free(void \*ptr) |
| --- |

size\_t - type entier non-signé, utilisé souvent pour représenter la taille de données

void \*malloc(size\_t size)

malloc(size) alloue size d'octets de la mémoire et retourne l'adresse du premier octet de la mémoire allouée. En cas d'échec malloc() retourne NULL.

Allouer un tableau de 20 nombres doubles :

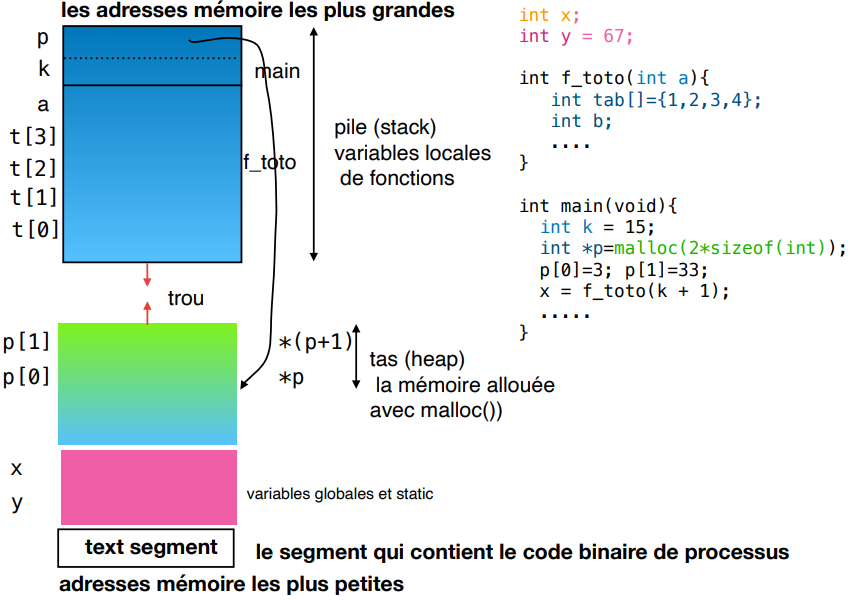
| double \*tab=malloc(20 \* sizeof(double)); if(tab == NULL){  perror("malloc"); exit(1);  } for(int i = 0; i < 20; i++){  tab[i] = i+5; /\* même chose que \*(tab+i)=i+5; \*/  } |
| --- |

void \*calloc(size\_t nb\_elem, size\_t elsize)

calloc() alloue un tableau de nb\_elem éléments, chaque élément de taille elsize d'octets. De plus, calloc() met à 0 tous les bits de la mémoire allouée. calloc() retourne l'adresse du premier octet de la mémoire allouée ou NULL en cas d'échec.

| long \*tab = calloc(100, sizeof(double)); /\* tab - tableau de 100 éléments double initialisés à 0 \*/ |
| --- |

void free(void \*ptr)

free() libère la mémoire allouée par malloc(), calloc() ou realloc(). Le paramètre de free() doit être le pointeur retournée par une de ces trois fonctions. Après l'appel à free(), les adresses dans le bloc de la mémoire libérée deviennent invalides.

mémoire d'un processus

Sur la pile pour chaque appel de fonction:

* les paramètres de la fonction
* les variables locales de la fonction (les variables static déclarées à l'intérieur de la fonction ne sont pas sur la pile)
* l'adresse de retour de la fonction (l'adresse de l'instruction qui sera exécutée juste après return de la fonction)
* peut-être d'autres éléments (par exemple les valeurs de différents registres sauvegardés à l'appel de la fonction, etc.)

dangers de malloc()

L'implémentation de malloc() utilise une liste chaînée de blocs alloués. La taille réelle d'un bloc peut être plus grande que la taille demandée.

Le bloc peut stocker en plus :

* le pointeur vers le block suivant,
* la taille du block (c'est grâce à cette information que free() "sait" combien de mémoire il doit libérer).

**Dangers :**

* **"memory corruption"** : l'écriture dans la mémoire dans le tas au-delà des adresses autorisées peut détruire les structures de données de malloc(). La conséquence : malloc(), free() suivants, et les accès à la mémoire, peuvent avoir le comportement imprévisible.
* **"memory leak" (fuite de mémoire)** : si on "oublie" l'adresse de la mémoire allouée par malloc() il n'est plus possible de libérer cette mémoire. La mémoire allouée s'accumule et nuit à l'exécution du programme. Particulièrement néfaste pour les serveurs qui tournent en permanence.

# pointeur de structure

| typedef struct {  double x; double y;  }point; point q = { .x = 3, .y =-7 }; point p\_q = &p; point \*p; p = malloc( sizeof(point) ); p->x = 2.8; /\* équivalent à (\*p).x = 2.8 \*/ p->y = -6.9; p\_q ->x = p->x + p->y; |
| --- |

# 

# 

# copier ou remplir une zone de mémoire

#include <string.h>

void \*memcpy(void \*dst, const void \*src, size\_t n)

* la fonction copie n octet de l'adresse src vers l'adresse dst. Si les deux zones chevauchent le résultat est indéfini. La fonction retourne dst.

void \*memmove(void \*dst, const void \*src, size\_t n)

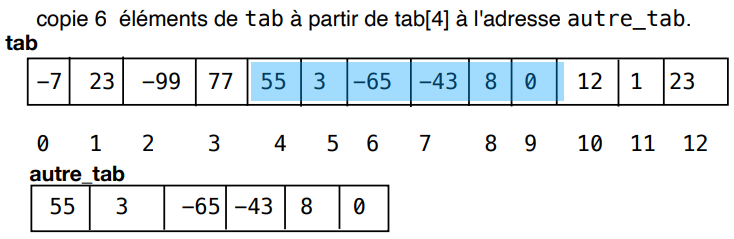
* même chose que la fonction précédente mais les deux zone peuvent chevaucher.

void \*memset(void \*s, int c, size\_t n)

* la fonction copie la valeur de c (transformée en unsigned char) sur n octets à partir de l'adresse s

exemple : copier une partie de tableau

| int tab[]={ -7, 23, -99, 77, 55, 3, -65, -43, 8, 0, 12, 1, -23} ; int \*autre\_tab = malloc( sizeof(int) \* 6); memmove(autre\_tab, &tab[4], 6 \* sizeof(int) ); |
| --- |



Cours 5

# (voir cours 4)

* void perror( const char \*s ) affiche un message d'erreur, à utiliser uniquement quand un appel fonction échoue, dans <stdio.h>
* void exit( int status ) termine l'exécution de programme avec le code status, dans <stdlib.h>

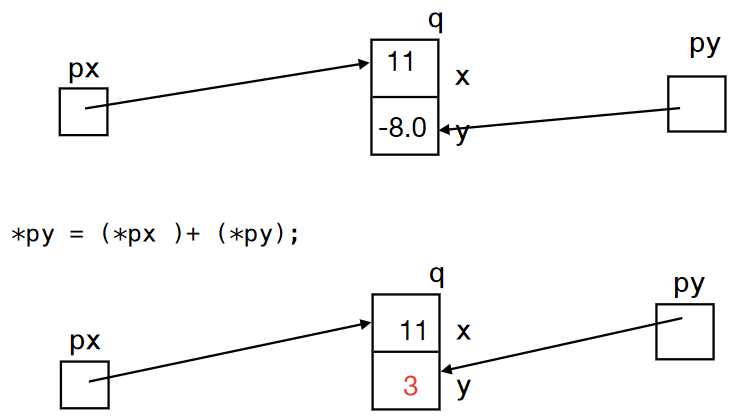
Une autre notation :

| double \*tab = malloc( sizeof( double [ n ] ) );  assert( tab != NULL ); |
| --- |

assert( condition ) si la condition est fausse **(dans le sens du C)** alors l'affichage d'un message qui donne le nom de fichier source et la ligne dans le code, et le programme termine

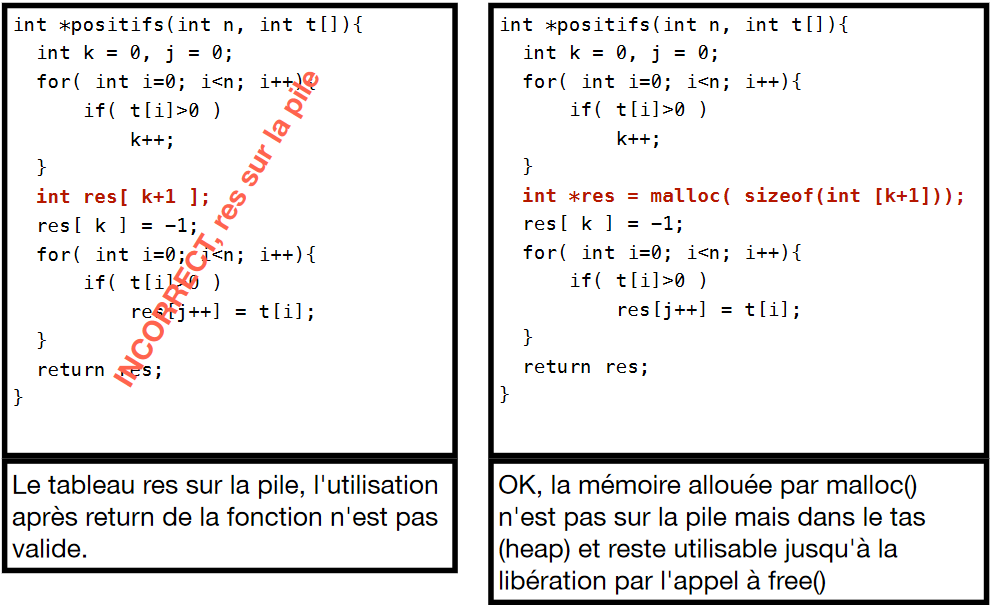
## Pointeurs de structure

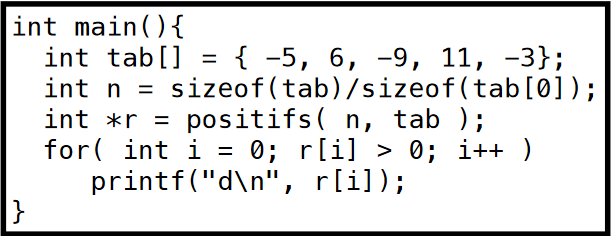
| struct point{  double x; double y;  };  typedef struct point point;  point q = { .y = -8.0, .x = 11.0 } ;  point \*p;  p = &q ;  int \*px = &q.x ;  int \*py = &q.y; |
| --- |



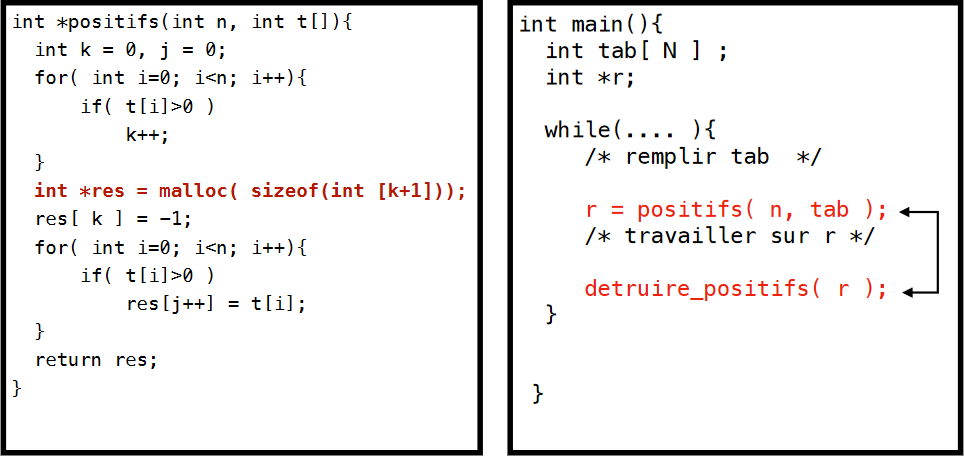
| struct point{  double x;  double y;  };  typedef struct point point;  point q = { .y = -8.0, .x = 11.0 } ;  int \*px = &q.x ;  int \*py = &q.y; |
| --- |

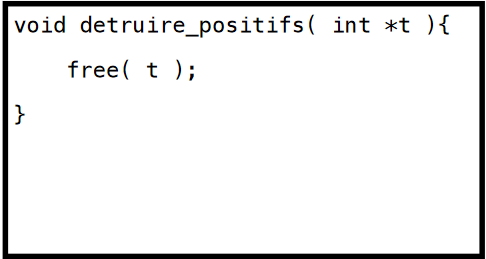
# à quoi sert malloc( ) ?





# fuites de mémoire





Pas de ramasse-miette en C, la mémoire allouée par malloc() reste allouée jusqu'à l'appel à free. Si on perd l'adresse de la mémoire allouée impossible de la libérer.

Deux bonnes pratiques :

* faire malloc() et free() dans la même fonction
* ou pour chaque fonction qui alloue la mémoire écrire une fonction correspondante qui libère la mémoire.

# void \*realloc(void \*ptr, size\_t size)

realloc() "modifie" la taille de la zone mémoire dont l'adresse est ptr

* ptr : l'adresse valide d'une zone de mémoire retourné auparavant par malloc() calloc() ou realloc() (zone mémoire dans le tas, pas sur la pile)
* size : la taille demandée en octets . Cette taille peut être plus grande ou plus petite que taille de la zone à l'adresse ptr. Les données qui se trouvent à l'adresse ptr sont recopiées dans la mémoire nouvellement allouée.

Si realloc()réussit à allouer la mémoire :

* l'adresse ptr devient invalide (en particulier ne faites plus free() sur ptr et n'utilisez plus la mémoire à l'adresse ptr, realloc() libère lui-même la mémoire à l'adresse ptr),
* realloc() retourne l'adresse du premier octet de la nouvelle zone mémoire.

Si realloc() échoue il retourne NULL et dans ce cas l'adresse ptr reste valide.

* realloc( NULL, n) est équivalent à malloc( n )

exemple

| int \*t = malloc( n \* sizeof(int));  ....  /\* doubler la taille du tableau \*/  n \*= 2;  int \*p = realloc(t, n \* sizeof(int) );  if(p == NULL){  /\* realloc() a échoué et t reste valide \*/  ...................  exit( 1 ) ; /\* exit() si on n'a rien à faire en cas de problèmes\*/  }  t = p ;/\* si p != NULL alors t n'est plus valide on peut réaffecter \*/ |
| --- |

# copier une zone de mémoire

copie 5 derniers éléments de tab dans t

| #include <string.h>  void \*memmove(void \*dst, const void \*src, size\_t n)  int tab[] = {4, 7, 9, -12, 7, 8, 22 }; int t = malloc( 5 \* sizeof(int) );  if( t == NULL )  exit(1);  memmove( t , &tab[2], 5\*sizeof( int ) ); |
| --- |

* memmove() copie n octet de l'adresse src vers l'adresse dst. La fonction retourne dst.
* Les deux zones de mémoire peuvent chevaucher.

# remplir une zone de mémoire

| #include <string.h>  void \*memset(void \*s, int c, size\_t n) |
| --- |

* la fonction copie la valeur de c (transformée en unsigned char) sur n octets à partir de l'adresse s

En pratique cela sert presque exclusivement pour mettre 0 dans une zone mémoire :

| #define N 1024 int tab[ N ]; memset( tab, 0, N \* sizeof(int) ); /\* à la place d'une boucle qui met 0 dans tous \* les élément de tab \*/ |
| --- |

Cours 6

Cours 7

Cours 8

Cours 9

Cours 10

Cours 11

Cours 12