Programmation de bas niveau en langage C

Aurélien BOSSARD (inspiré du cours de Chan LeDuc)

Organisation du cours

1. 8 cours (45'-60') + 8 TD (1h30) + 8 TP (1h30)

Organisation du cours

1. 8 cours (45'-60') + 8 TD (1h30) + 8 TP (1h30)
2. Objectifs :
   - maîtriser un langage de bas niveau
   - comprendre la gestion de la mémoire dynamique
   - développer des applications simples

Organisation du cours

1. 8 cours (45'-60') + 8 TD (1h30) + 8 TP (1h30)
2. Objectifs :
   - maîtriser un langage de bas niveau
   - comprendre la gestion de la mémoire dynamique
   - développer des applications simples
3. Pré-requis :
   - Logique élémentaire
   - Système (Représentation et Traitement des informations, Système d'Exploitation)
   - Algorithmique et Programmation

Organisation du cours

1. 8 cours (45'-60') + 8 TD (1h30) + 8 TP (1h30)
2. Objectifs :
   - maîtriser un langage de bas niveau
   - comprendre la gestion de la mémoire dynamique
   - développer des applications simples
3. Pré-requis :
   - Logique élémentaire
   - Système (Représentation et Traitement des informations, Système d'Exploitation)
   - Algorithmique et Programmation
4. Evaluations : tests sur Moodle, TD, TP

Organisation du cours

1. 8 cours (45'-60') + 8 TD (1h30) + 8 TP (1h30)
2. Objectifs :
   - maîtriser un langage de bas niveau
   - comprendre la gestion de la mémoire dynamique
   - développer des applications simples
3. Pré-requis :
   - Logique élémentaire
   - Système (Représentation et Traitement des informations, Système d'Exploitation)
   - Algorithmique et Programmation
4. Evaluations : tests sur Moodle, TD, TP
5. Références :
   C Programming Language, Brian Kernighan et Dennis Richie
   `http://zanasi.chem.unisa.it/download/C.pdf`

   `http://www.cs.cf.ac.uk/Dave/C`
   Cours C, Anne Canteaut à l'INRIA
   `https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf`

Sommaire

Le langage C - Historique

- conçu en 1972 par Ken Thompson et Dennis Ritchie.
- a accompagné le développement d'Unix
- première version stabilisée en 1978
- normalisé par l'ANSI en 1989, puis l'ISO en 1990
- révisé en 1999 et 2011

Le langage C - Spécificités

1. Langage de bas niveau
2. Langage entièrement compilé
3. A l'origine du développement d'Unix : proche du système
4. Fait usage des pointeurs et de gestion de mémoire

Le langage C - Avantages et Défauts

Avantages :

1. Rapide
2. Interface directe avec le SE
3. Fonctionnement optimisé selon la machine

Le langage C - Avantages et Défauts

Avantages :

1. Rapide
2. Interface directe avec le SE
3. Fonctionnement optimisé selon la machine

Défauts :

1. Pas de concepts de haut niveau (objets, ramasse-miette, etc.)
2. Portabilité limitée
3. Peu de vérifications à la compilation

Qualités attendues d'un programme C

- Clarté (lisible, naturel, etc.)
- Simplicité
- Modularité
- Extensibilité
- Documentation

Compiler un programme C

- 2 types de fichiers en C
  - Les fichiers d'en-tête : *nomfichier.h*
  - Les fichiers de code : *nomfichier.c*
- Compilateur sous Linux : `gcc`
  - `gcc nomfichier.c`
    - Génère le programme dans un fichier `a.out`
  - `gcc nomfichier.c -o nomprogramme`
    - Génère le programme dans un fichier `nomprogramme`

Structure d'un programme C

```
//Déclarations de bibliotheques
#include<stdio.h>

//Défintions de fonctions
UnType fonction1(...) {...}
UnType fonction2(...) {...}

//fonction main
void main(void){
     //Déclarations des variables

     //Code à exécuter
}
```

Types de variables

- Entiers : `short, int, long, unsigned int`
- Flottants : `float`, `double`
- Caractère :`char` (code d'ascii)
- Déclaration : `UnType nomvariable;`
- Affectation : `UneVariable = AutreVariable;`

Entrées/Sorties

- Afficher des données à la sortie standard

  Syntaxe :

  ```
  printf(const char* format [, param_1,...,param_n])
  ```

  Exemple :

  ```
  printf("Nom = %s, Age = %d\n", "Le Monde", 20);
  ```

Entrées/Sorties

- Afficher des données à la sortie standard

  Syntaxe :

  ```
  printf(const char* format [, param_1,...,param_n])
  ```

  Exemple :

  ```
  printf("Nom = %s, Age = %d\n", "Le Monde", 20);
  ```

- Saisir des données à l'entrée standard

  Syntaxe :

  ```
  scanf(const char* format [, param_1,...,param_n])
  ```

  Exemple :

  ```
  int a;
  scanf("%d", &a);
  ```

Opérateurs de bits

- le NON ($\sim a$),
- le ET ($a \& b$),
- le OU ($a | b$),
- le OU exclusif (a^b)

Opérateurs de bits : exemple

```
1     int a, b, c, flag;
2
3     a = 0x6db7;   |-0-|-0-|0110 1101|1011 0111|
4     b = 0xa726;   |-0-|-0-|1010 0111|0010 0110|
5     c = a&b ;     |-0-|-0-|0010 0101|0010 0110| (0x2526)
6     c = a|b ;     |-0-|-0-|1110 1111|1011 0111| (0xefb7)
7     c = a^b ;     |-0-|-0-|1100 1010|1001 0001| (0xca91)
```

Opérateurs de décalage

- Opérateurs de décalage :
  - décalage à gauche ($a << n$) : les bits de poids fort sont perdus, les bits rendus vacants sont remplis par des 0
  - décalage à droite ($a >> n$) : les bits de poids faible sont perdus, les bits rendus vacants sont remplis par le bit de signe pour les nombres signés

Opérateurs de décalage : exemple

```
1   int  etat;
2   int  oct;
3   int  ent;
4   int  a;
5
6   a = 0x6db7;          |-0-|-0-|0110 1101|1011 0111|
7   a = a << 6;          |-0-|0001 1011|0110 1101|1100 0000|
8
9   a = 0x6db7;
10  a = a >> 6;          |-0-|-0-|0000 0001|1011 0110|
11
12  ent = 0xf0000000;
13  ent = ent >> 10;  |1111 1111|1111 1100|-0-|-0-|
14
15  oct = (etat >> 8) & 0xff;
```

Opérateurs d'incrémentation et de décrémentation

- Ces opérations sont effectuées après ou avant l'évaluation de l'expression suivant que l'opérateur suit ou précède son opérande.
- Exemple :

```
int i = 1, a;
a = i++;
printf("i=%d, a=%d", i, a); -> i=2, a=1
a = ++i;
printf("i = %d, a= %d", i, a); -> i=3, a=3
```

Opérateurs d'incrémentation et de décrémentation : exemple 1

```
1  /* inc.c */
2  #include<stdio.h>
3  #define N 2
4  void main(void) {
5      int i=0;
6      int tab[N] = {1, 2};
7
8      while(i < N) {
9          printf("%d ", tab[i++]);
10     }
11 }
12
13 $ gcc inc.c -o inc
14 $ ./inc
15 1 2
```

Opérateurs d'incrémentation et de décrémentation : exemple 2

```
1  /* inc.c */
2  #include<stdio.h>
3  #define N 2
4  void main(void) {
5      int i=0;
6      int tab[N] = {1, 2};
7
8      while(i < N) {
9          printf("%d ", tab[++i]);
10     }
11 }
12
13 $ gcc inc.c -o inc
14 $ ./inc
15 2
```

Fonction

Définition d'une fonction avec des paramètres formels :

```
type_retour mafonc(type_1 param_1,...,type_n param_n) {
     declaration de variables locales;
     instructions;
     return exp;
}
```

Appel d'une fonction avec des paramètres effectifs :

```
type_retour  a;
type_1 var_1;
...
type_n var_n;
a = mafonc(val_1,...,val_n);
```

Caractérisques d'une variable

- Variable locale
  - Déclaration : à l'intérieur d'une fonction
  - Visibilité : de l'endroit où elle est déclarée jusqu'à la fin de la fonction
  - Mémoire : zone de pile
  - Durée de vie : de l'appel à la fonction jusqu'à la fin de la fonction

Caractérisques d'une variable

- Variable locale
  - Déclaration : à l'intérieur d'une fonction
  - Visibilité : de l'endroit où elle est déclarée jusqu'à la fin de la fonction
  - Mémoire : zone de pile
  - Durée de vie : de l'appel à la fonction jusqu'à la fin de la fonction
- Variable globale
  - Déclaration : à l'extérieur de toutes les fonctions
  - Visibilité : de l'endroit où elle est déclarée
  - Mémoire : zone de données
  - Durée de vie : du démarrage du programme à la fin

Effets de bord (l'exemple dans "C++ Tutorial")

L'interaction entre le code d'une fonction et le code extérieur

```
1  #include<stdio.h>
2   // declare global variable
3  int g_nMode = 1;
4
5  void doSomething() {
6      g_nMode = 2;
7  }
8  void main(void) {
9      g_nMode = 1;
10
11     doSomething();
12
13     // Programmer expects g_nMode to be 1
14     // But doSomething changed it to 2!
15
16     if (g_nMode == 1)
17         printf("No threat detected.");
18     else
19         printf("Launching nuclear missiles...");
20 }
```

Fonction - Exemple

Ecrire un programme en C qui affiche un rectangle en fonction d'une largeur et une longueur (saisies par l'utilisateur) comme présenté ci-dessous :

Tableau - Définition et Accès

Un tableau est une collection de données du même type.

- Déclarer un tableau
  `type_elements nom_tableau[nb_cases];`
  `Ex: int a[10];`
  définit 10 cases consécutives dont chacune contient un entier et est référencée par `a[0],...,a[9]`

Tableau - Définition et Accès

Un tableau est une collection de données du même type.

- Déclarer un tableau
  `type_elements nom_tableau[nb_cases];`
  `Ex: int a[10];`
  définit 10 cases consécutives dont chacune contient un entier et est référencée par `a[0],...,a[9]`
- Initialiser un tableau
  `type_elements nom_tableau[N] = {valeur_1,...,valeur_N};`

Tableau - Définition et Accès

Un tableau est une collection de données du même type.

- Déclarer un tableau
  `type_elements nom_tableau[nb_cases];`
  `Ex: int a[10];`
  définit 10 cases consécutives dont chacune contient un entier et est référencée par `a[0],...,a[9]`
- Initialiser un tableau
  `type_elements nom_tableau[N] = {valeur_1,...,valeur_N};`
- Affecter une valeur dans une case
  `nom_tableau[numero_case] = valeur;`

Tableau - Définition et Accès

Un tableau est une collection de données du même type.

- Déclarer un tableau
  `type_elements nom_tableau[nb_cases];`
  `Ex: int a[10];`
  définit 10 cases consécutives dont chacune contient un entier et est référencée par `a[0],...,a[9]`
- Initialiser un tableau
  `type_elements nom_tableau[N] = {valeur_1,...,valeur_N};`
- Affecter une valeur dans une case
  `nom_tableau[numero_case] = valeur;`
- Accéder à la valeur d'une case d'un tableau
  `nom_tableau[numero_case];`

Tableau de caractères

- `char a[8];`

Tableau de caractères

- `char a[8];`
- `char a[] = "Bonjour";`
  définit 7 cases consécutives dont chacune contient le code ASCII du caractère correspondant :
  `a[0]='B'=66,...,a[6]='r'=114,` et
  `a[7]='\0'=0`

Tableau de caractères

- `char a[8];`
- `char a[] = "Bonjour";`
  définit 7 cases consécutives dont chacune contient le code ASCII du caractère correspondant :
  `a[0]='B'=66,...,a[6]='r'=114,` et
  `a[7]='\0'=0`
- Attention : `'\0'` $\neq$ `""` $\neq$ `'0'`

Tableau - Caractéristiques

```
int a[10], b[10], i;
```

- Le nom d'un tableau est une constante :
  `a=b;` : illégal
  `for(i=0; i<10; i++) a[i]=b[i];` : OK

Tableau - Caractéristiques

```
int a[10], b[10], i;
```

- Le nom d'un tableau est une constante :
  `a=b;` : illégal
  `for(i=0; i<10; i++) a[i]=b[i];` : OK
- Mémoire fixe :
  `a[10]=1;` : illégal

Tableau - Caractéristiques

```
int a[10], b[10], i;
```

- Le nom d'un tableau est une constante :
  `a=b;` : illégal
  `for(i=0; i<10; i++) a[i]=b[i];` : OK
- Mémoire fixe :
  `a[10]=1;` : illégal
- Suppression d'un élément d'un tableau de caractères ?

Tableau - Caractéristiques

```
int a[10], b[10], i;
```

- Le nom d'un tableau est une constante :
  `a=b;` : illégal
  `for(i=0; i<10; i++) a[i]=b[i];` : OK
- Mémoire fixe :
  `a[10]=1;` : illégal
- Suppression d'un élément d'un tableau de caractères ?
- Insertion d'un élément dans un tableau de caractères ? Taille ?

Tableau à deux dimensions

```
type_elements
nom_tab[nb_cases_dim1][nb_cases_dim2];
```

```
int NOTE[10][20] = {{45, 34, ... , 50, 48},
                    {39, 24, ... , 49, 45},
                     ...    ...     ...
                    {40, 40, ... , 54, 44}};
```

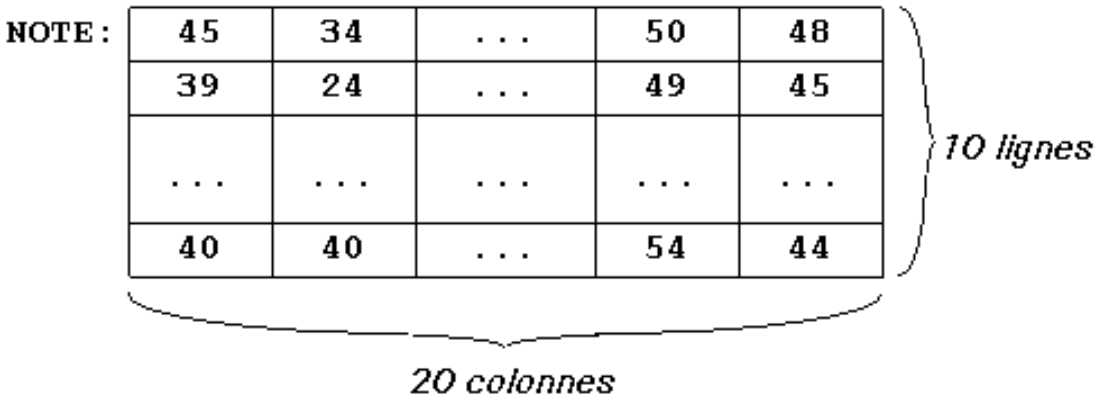


Tableau à deux dimensions - Accès

- Affecter une valeur dans une case
  `nom_tab[num_case_dim1][num_case_dim2] = valeur;`

Tableau à deux dimensions - Accès

- Affecter une valeur dans une case
  `nom_tab[num_case_dim1][num_case_dim2] = valeur;`
- Accéder à la valeur d'une case d'un tableau
  `nom_tab[num_case_dim1][num_case_dim2];`

Passage d'un tableau à une fonction

```
void maFonction(int t[]) {...}

void main(void)  {
   int a[2]={2, 5};
   maFonction(a);
}
```

- un passage par référence, i.e., l'adresse du premier élément du tableau "a" est copiée dans "t" (variable locale)
- si "t" est changé dans `maFonction` alors "a" est changé aussi
- Attention : la taille du tableau

Structure

- Utilité : les attributs d'une classe Java, un tuple d'une table, etc.

Structure

- Utilité : les attributs d'une classe Java, un tuple d'une table, etc.
- Déclaration :

```
struct s {
type_1 champ_1;
...
type_n champ_n;
};
```

Structure

- Utilité : les attributs d'une classe Java, un tuple d'une table, etc.
- Déclaration :

```
struct s {
type1 champ1;
...
typen champn;
};
```

- Définition de variables de type structure :

```
struct s var; // sans init.
struct s var={exp1,...,expn};//avec init.
```

Structure

- Utilité : les attributs d'une classe Java, un tuple d'une table, etc.
- Déclaration :

```
struct s {
type1 champ1;
...
typen champn;
};
```

- Définition de variables de type structure :

```
struct s var; // sans init.
struct s var={exp1,...,expn};//avec init.
```

- Sélection de la valeur d'un champ :

```
typei vari = var.champi;
```

Structure

- Utilité : les attributs d'une classe Java, un tuple d'une table, etc.
- Déclaration :
  `struct s {`
  $type_1$ $champ_1$`;`
  ...
  $type_n$ $champ_n$`;`
  `};`
- Définition de variables de type structure :
  `struct s` *var*`; // sans init.`
  `struct s` *var*`={`exp_1`,...,`exp_n`}; //avec init.`
- Sélection de la valeur d'un champ :
  $type_i$ var_i = *var*.$champ_i$`;`
- Affectation d'une valeur à un champ.
  var.$champ_i$ = var_i`;`

Structure - Exemple

Ecrire un programme qui

- utilise une structure pour stocker les coordonnées d'un point (et son nom).
- permettre de saisir les coordonnées d'un point, de calculer les coordonnées du milieu d'un segment et d'afficher les coordonnées d'un point.

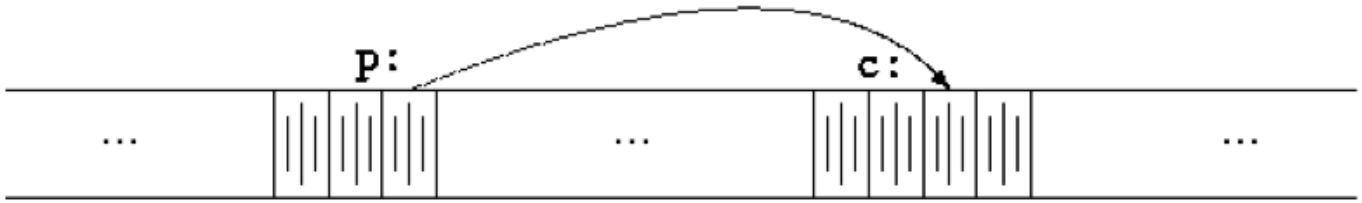
Pointeurs - Définition

Un pointeur est une variable qui contient l'adresse d'une variable (K&R).

Pointeurs - Définition

Un pointeur est une variable qui contient l'adresse d'une variable (K&R).

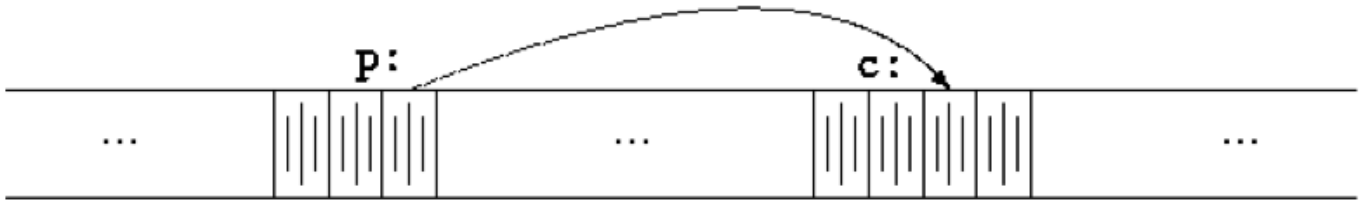
"c" est un `char` et "p" est un pointeur qui pointe sur "c"



Pointeurs - Définition

Un pointeur est une variable qui contient l'adresse d'une variable (K&R).

"c" est un `char` et "p" est un pointeur qui pointe sur "c"



Motivation :

- Accès à une variable locale
- Allocation dynamique de mémoire
- Gestion d'une structure de données non-contiguë
- Exécution d'un code via son adresse mémoire

Pointeurs en Assembleur

```
var1:       .word     10
var2:       .long      0
...
movw  var2, %ax // ax = 0
movl  $var1, var2 // var2 = l'adresse mémoire de "var1"
movw (var2), %ax // ax = 10
```

Pointeurs - Déclaration et Opérateurs

```
char c;
int i;
```

$\Rightarrow$ `int *p;`
déclaration d'un pointeur entier

Pointeurs - Déclaration et Opérateurs

```
char c;
int i;
```

⇒ `int *p;`
déclaration d'un pointeur entier

⇒ `p = &c;`
l'opérateur unaire & retourne l'adresse de la case mémoire contenant la valeur d'une variable, d'un élément de tableau, etc.

Pointeurs - Déclaration et Opérateurs

```
char c;
int i;
```

⇒ `int *p;`
déclaration d'un pointeur entier

⇒ `p = &c;`
l'opérateur unaire & retourne l'adresse de la case mémoire contenant la valeur d'une variable, d'un élément de tableau, etc.

⇒ `i = *p;`
l'opérateur unaire * s'applique à un pointeur et retourne la valeur contenue dans la case sur laquelle le pointeur pointe

Pointeurs - Exemple

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

Pointeurs - Passage de paramètres par référence

On considère :

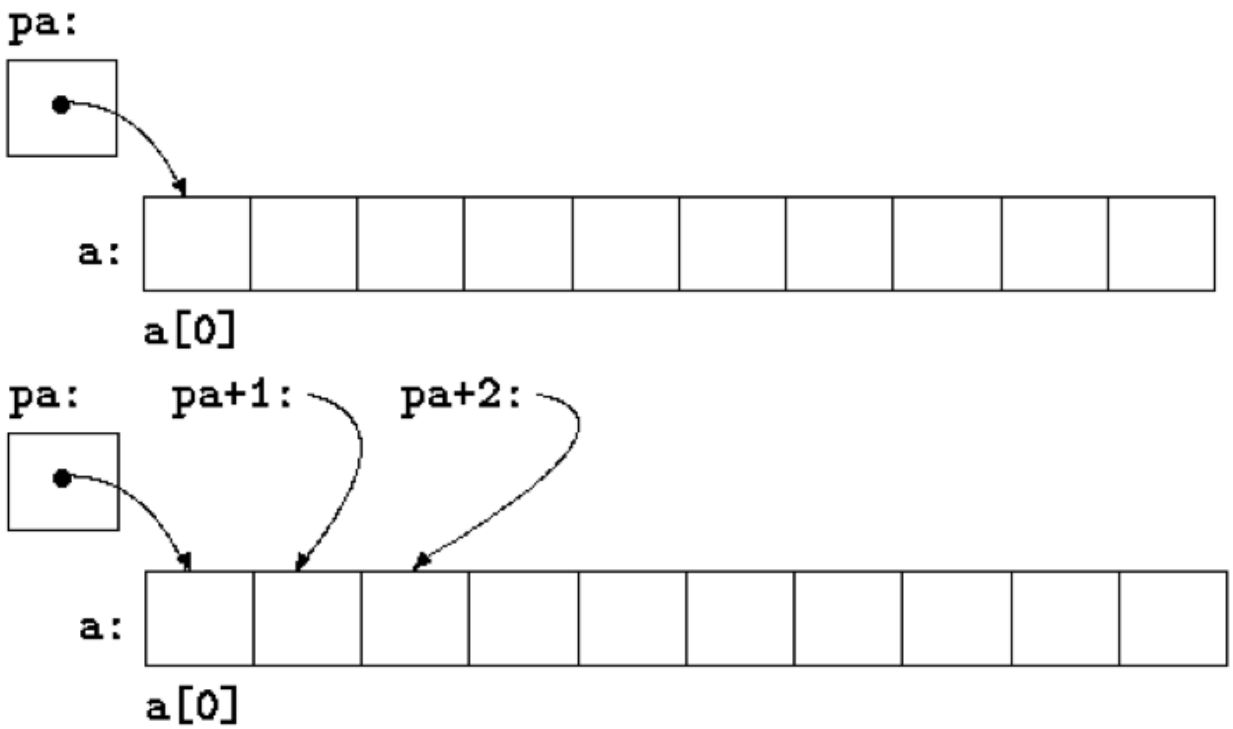
```
/*  version erronée */
void swap(int x, int y)  {
      int temp;
      temp = x;
      x = y;
      y = temp;
}
void main(void) {
     int a=2, b=5;
     swap(a,b);
}

/*  version corrigée */

?
```

Pointeurs - Tableaux

```
int  a[10];
int *pa;
pa = &a[0];
//"*pa" est équivalent à "a[0]"
//"*(pa+1)" est équivalent à "a[1]"
```



Pointeurs - Exemple

On peut utiliser un pointeur pour parcourir un tableau.

```
int strlen(char s[]) {
     int n;
     char *ps = s;
     for (n = 0; *ps != '\0'; ps++) n++;
     return n;
}

int strlen(char *s) {
     int n;
     for (n = 0; *s != '\0'; s++) n++;
     return n;
}
...
char *t="Bonjour";
int n = strlen(t);
```

Arithmétique des pointeurs

```
int a[10];
int *pa, *pb, n;
```

`pa + n`	pointe sur l'élément à l'indice `n`
`pa < pb`	est vraie si l'élément référencé par "`pa`" est placé avant celui référencé par "`pb`"
`pb - pa`	est le nombre d'éléments entre "`pa`" et "`pb`"

Arithmétique des pointeurs versus arithmétique standard

```
int a[2] = {1, 256};
int *pa, i;
char c;

//arithmétique des pointeurs
for (pa = a, i=0; i < 2; i++){
     printf("%d\n", *(pa+i) );
}
//arithmétique standard
for (pa = a; (long)pa < (long)a + sizeof(int)*2; ((long)pa)++){
     c = (char)*pa;
     printf("%d\n", c );
}
```

Qu'affiche le programme ?

Pointeurs de caractères

```
char tmessage[] = "Bonjour"; /* un tableau */
char *pmessage = "Bonjour"; /* un pointeur */
```

- Le tableau `tmessage` possède exactement 8 octets pour stocker “Bonjour\0”.
- Le contenu de `tmessage` peut être changé mais `tmessage` est non modifiable

Pointeurs de caractères

```
char tmessage[] = "Bonjour"; /* un tableau */
char *pmessage = "Bonjour"; /* un pointeur */
```

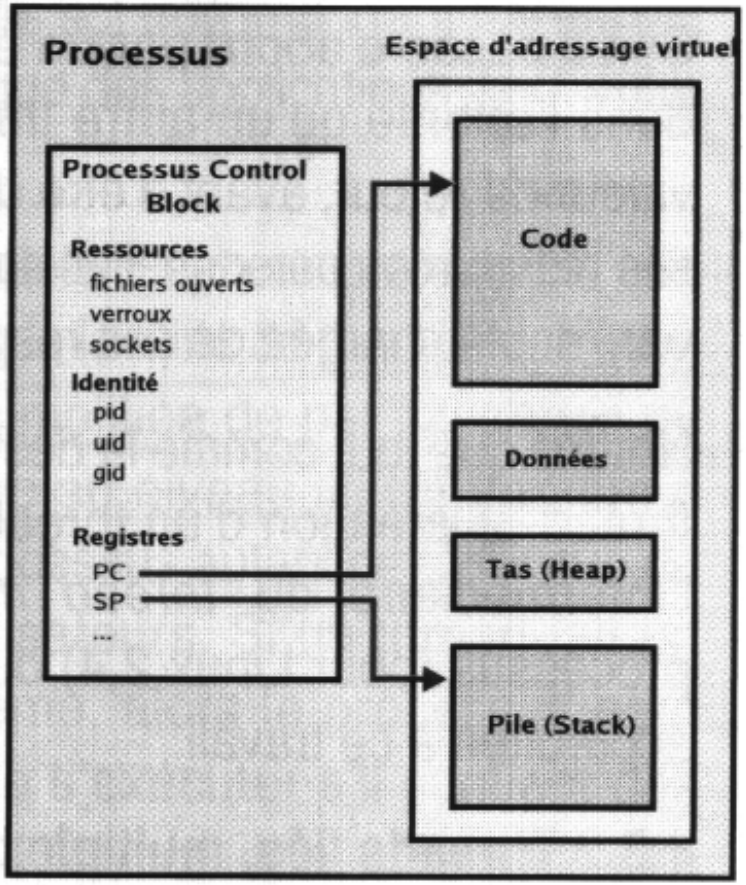
- Le tableau `tmessage` possède exactement 8 octets pour stocker “Bonjour\0”.
- Le contenu de `tmessage` peut être changé mais `tmessage` est non modifiable
- Le pointeur `pmessage` pointe sur la chaine de caractères constante “Bonjour\0”.
- La chaîne est non modifiable mais le pointeur `pmessage` peut pointer sur une autre case mémoire.

Pointeurs de caractères

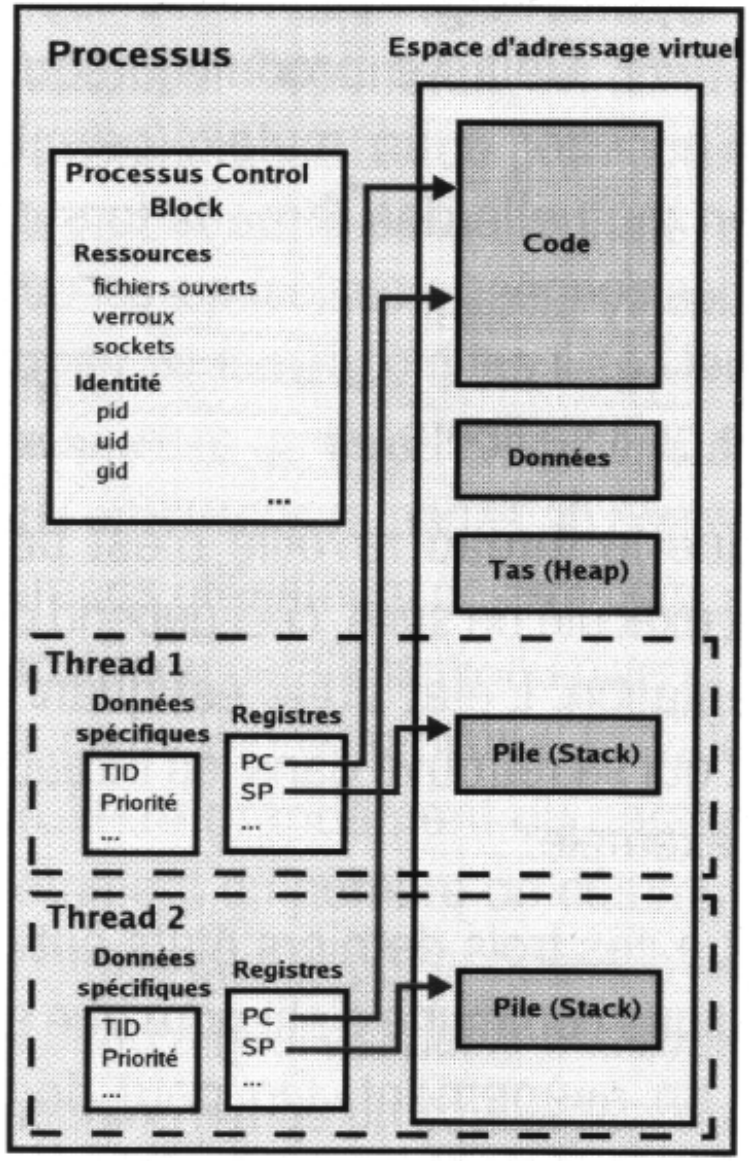
```
char tmessage[] = "Bonjour"; /* un tableau */
char *pmessage = "Bonjour"; /* un pointeur */
```

- Le tableau `tmessage` possède exactement 8 octets pour stocker “Bonjour\0”.
- Le contenu de `tmessage` peut être changé mais `tmessage` est non modifiable
- Le pointeur `pmessage` pointe sur la chaine de caractères constante “Bonjour\0”.
- La chaîne est non modifiable mais le pointeur `pmessage` peut pointer sur une autre case mémoire.
- Exemple : copier une chaîne de caractères.

Mémoire dynamique (GNU Linux Magazine)



Processus classiques



Processus multithreadé

Allocation dynamique de mémoire

La bibliothèque `#include <stdlib.h>`

Allouer avec
`void* malloc(size_t size)`

- `size` est un entier qui représente le nombre d'octets
- l'appel système `malloc` retourne un pointeur générique il faut préciser avant de l'utiliser (avec un cast)
- Exemple :

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

Allocation dynamique de mémoire

La bibliothèque `#include <stdlib.h>`

Allouer avec
`void* malloc(size_t size)`

- `size` est un entier qui représente le nombre d'octets
- l'appel système `malloc` retourne un pointeur générique il faut préciser avant de l'utiliser (avec un cast)
- Exemple :

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

Pour allouer la mémoire à une chaîne de caractères :

```
#define TAILLE_CHAINE 50
...
char* chaine;
chaine=(char*)malloc(TAILLE_CHAINE*sizeof(char));
```

Ajustement d'une mémoire allouée

Pour ajuster la taille de l'espace mémoire après allocation :

```
void *realloc (void *ptr, size_t size);
```

`ptr` est le pointeur désignant la mémoire à reconditionner
`size` est un entier qui représente le nouveau nombre d'octets à allouer

Exemple :

```
chaine=
(char*)realloc(chaine,TAILLE_CHAINE*sizeof(char));
```

Libération d'une mémoire allouée

Il faut libérer TOUTE la mémoire qui a été allouée dès que possible (sinon fuite mémoire).

`free(p);`
libère la mémoire allouée dynamiquement et pointée par `p`

Si vous arrêtez à un seul moment de pointer un espace de mémoire alloué dynamiquement, vous n'aurez plus moyen de le retrouver par la suite de votre programme.

Pointeurs - Structures

```
struct vecteur {
int x;
int y;
};
```

- Pour allouer la mémoire à une structure :

```
struct vecteur *s;
s = (struct vecteur*)malloc(sizeof(struct
vecteur));
```

- Pour accéder un champ d'une structure via un pointeur :

```
int n;
n = s->x;
```

Tableau de pointeurs ; Pointeurs sur pointeurs

- Déclarer d'un tableau de pointeurs

```
char *mots[3]; // tableau de chaînes de caractères
mots[0] = "defghi";
mots[1] = "jklmnopqrst";
mots[2] = "abc";
```

Tableau de pointeurs ; Pointeurs sur pointeurs

- Déclarer d'un tableau de pointeurs

```
char *mots[3]; // tableau de chaînes de caractères
mots[0] = "defghi";
mots[1] = "jklmnopqrst";
mots[2] = "abc";
```

- Trier le tableau

Implémenter "`int strcmp(char* s, char* t)`" et "`void trier (char *mots[])`"

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,
- `a[2][5]` est certainement alloué mais non pas `b[2][5]`

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,
- `a[2][5]` est certainement alloué mais non pas `b[2][5]`
- `b` comporte 5 lignes dont chacune peut avoir une longueur différente

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,
- `a[2][5]` est certainement alloué mais non pas `b[2][5]`
- `b` comporte 5 lignes dont chacune peut avoir une longueur différente
- `c = b` : OK

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,
- `a[2][5]` est certainement alloué mais non pas `b[2][5]`
- `b` comporte 5 lignes dont chacune peut avoir une longueur différente
- `c = b` : OK
- `b[0] = a[0]` : OK

Pointeurs - Tableaux à 2 dimensions

```
int a[5][10];
int *b[5];
int **c;
```

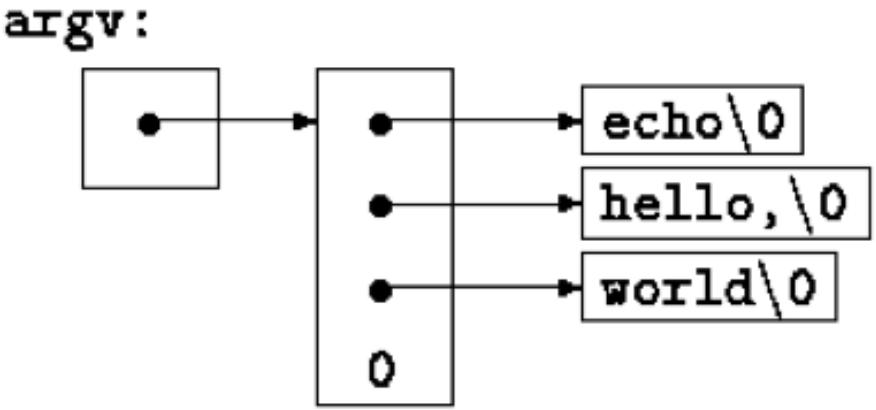
- `a[2][5]` et `b[2][5]` sont autorisés syntaxiquement,
- `a[2][5]` est certainement alloué mais non pas `b[2][5]`
- `b` comporte 5 lignes dont chacune peut avoir une longueur différente
- `c = b` : OK
- `b[0] = a[0]` : OK
- `c = a` : Non OK

Paramètres positionnels

`main(int argc, char *argv[]) {...}`

`int argc` : le nombre de paramètres

`char *argv[]` : le tableau de chaînes de caractères contenant les paramètres



Exemple : Afficher tous les paramètres positionnels (sauf le nom de programme) séparés par un espace.

Fichiers - Généralités

Un fichier : une suite d'octets terminée par EOF.
Un fichier : stocké dans une mémoire secondaire.

Fichiers - Généralités

Un fichier : une suite d'octets terminée par EOF.
Un fichier : stocké dans une mémoire secondaire.

- Un fichier texte est une suite de lignes dont chacune se termine par un saut ligne \n

Fichiers - Généralités

Un fichier : une suite d'octets terminée par EOF.
Un fichier : stocké dans une mémoire secondaire.

- Un fichier texte est une suite de lignes dont chacune se termine par un saut ligne \n
- La taille peut être très grande (par rapport à RAM)

Fichiers - Généralités

Un fichier : une suite d'octets terminée par EOF.
Un fichier : stocké dans une mémoire secondaire.

- Un fichier texte est une suite de lignes dont chacune se termine par un saut ligne \n
- La taille peut être très grande (par rapport à RAM)
- Le tri, la recherche, l'accès aléatoire sont coûteux

Fichiers - Généralités

Un fichier : une suite d'octets terminée par EOF.
Un fichier : stocké dans une mémoire secondaire.

- Un fichier texte est une suite de lignes dont chacune se termine par un saut ligne \n
- La taille peut être très grande (par rapport à RAM)
- Le tri, la recherche, l'accès aléatoire sont coûteux
- La lecture et l'écriture sont gérées avec un tampon et un curseur

Fichier - Création/Ouverture/Fermeture

Manipulations des fichiers de haut niveau : `fopen(...)`, `fread(...)`, `fwrite(...)`,`fclose(...)`, etc.

Fichier - Création/Ouverture/Fermeture

Manipulations des fichiers de haut niveau : `fopen(...)`, `fread(...)`, `fwrite(...)`, `fclose(...)`, etc.

- Tampon (buffer) : une zone de mémoire pour échanger les données entre la mémoire d'un programme (SE) et un fichier ;

Fichier - Création/Ouverture/Fermeture

Manipulations des fichiers de haut niveau : `fopen(...)`, `fread(...)`, `fwrite(...)`,`fclose(...)`, etc.

- Tampon (buffer) : une zone de mémoire pour échanger les données entre la mémoire d'un programme (SE) et un fichier ;
- Curseur (file pointer) : un pointeur pour indiquer la position courante dans un fichier ouvert.

Fichier - Création/Ouverture/Fermeture

Manipulations des fichiers de haut niveau : `fopen(...)`, `fread(...)`, `fwrite(...)`,`fclose(...)`, etc.

- Tampon (buffer) : une zone de mémoire pour échanger les données entre la mémoire d'un programme (SE) et un fichier ;
- Curseur (file pointer) : un pointeur pour indiquer la position courante dans un fichier ouvert.
- Création/Ouverture :
  `FILE* fopen(char* nom-fichier, char *mode)`
  où mode = "r", "w", "a", "r+", "w+", "a+". La position initiale du curseur dépend du mode.

Fichier - Création/Ouverture/Fermeture

Manipulations des fichiers de haut niveau : `fopen(...)`, `fread(...)`, `fwrite(...)`,`fclose(...)`, etc.

- Tampon (buffer) : une zone de mémoire pour échanger les données entre la mémoire d'un programme (SE) et un fichier ;
- Curseur (file pointer) : un pointeur pour indiquer la position courante dans un fichier ouvert.
- Création/Ouverture :
  `FILE* fopen(char* nom-fichier, char *mode)`
  où mode = "r", "w", "a", "r+", "w+", "a+". La position initiale du curseur dépend du mode.
- Fermeture :
  `int fclose(FILE* flux)` retourne 0 si OK, EOF sinon.

Fichier - Lecture

Avant de lire des données d'un fichier ouvert, il faut gérer la mémoire où les données lues seront mises et la position du curseur.

Fichier - Lecture

Avant de lire des données d'un fichier ouvert, il faut gérer la mémoire où les données lues seront mises et la position du curseur.

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *flux)`

Fichier - Lecture

Avant de lire des données d'un fichier ouvert, il faut gérer la mémoire où les données lues seront mises et la position du curseur.

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *flux)`
  - copie *size* × *nmemb* octets du flux fichier "`flux`" pointé par le curseur dans la zone de mémoire pointée par "`ptr`",
  - avance le curseur *size* × *nmemb* octets

Fichier - Lecture

Avant de lire des données d'un fichier ouvert, il faut gérer la mémoire où les données lues seront mises et la position du curseur.

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *flux)`
  - copie *size* × *nmemb* octets du flux fichier "`flux`" pointé par le curseur dans la zone de mémoire pointée par "`ptr`",
  - avance le curseur *size* × *nmemb* octets

  Exemple :

```
int n;
FILE* flux;
char* ptr=(char*)malloc(10*sizeof(char));
flux = fopen("fich.txt", "r");
n = fread(ptr, sizeof(char), 10, flux);
if(n > 0) ...
```

Fichier - Ecriture

Avant d'écrire des données dans un fichier ouvert, il faut gérer la mémoire où se trouvent les données et la position du curseur.

Fichier - Ecriture

Avant d'écrire des données dans un fichier ouvert, il faut gérer la mémoire où se trouvent les données et la position du curseur.

- `size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *flux)`

Fichier - Ecriture

Avant d'écrire des données dans un fichier ouvert, il faut gérer la mémoire où se trouvent les données et la position du curseur.

- `size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *flux)`
  - copie *size* × *nmemb* octets de la zone de mémoire pointée par "`ptr`" dans le flux fichier "`flux`" pointé par le curseur
  - avance le curseur *size* × *nmemb* octets

Fichier - Ecriture

Avant d'écrire des données dans un fichier ouvert, il faut gérer la mémoire où se trouvent les données et la position du curseur.

- `size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *flux)`
  - copie *size* × *nmemb* octets de la zone de mémoire pointée par "`ptr`" dans le flux fichier "`flux`" pointé par le curseur
  - avance le curseur *size* × *nmemb* octets

  Exemple :

```
int n;
FILE* flux;
char ptr[] = "Bonjour toto!\n";
flux = fopen("fich.txt", "w");
n = fwrite(ptr, sizeof(char), 14, flux);
fclose(flux);
```

Liste chaînée - Définition

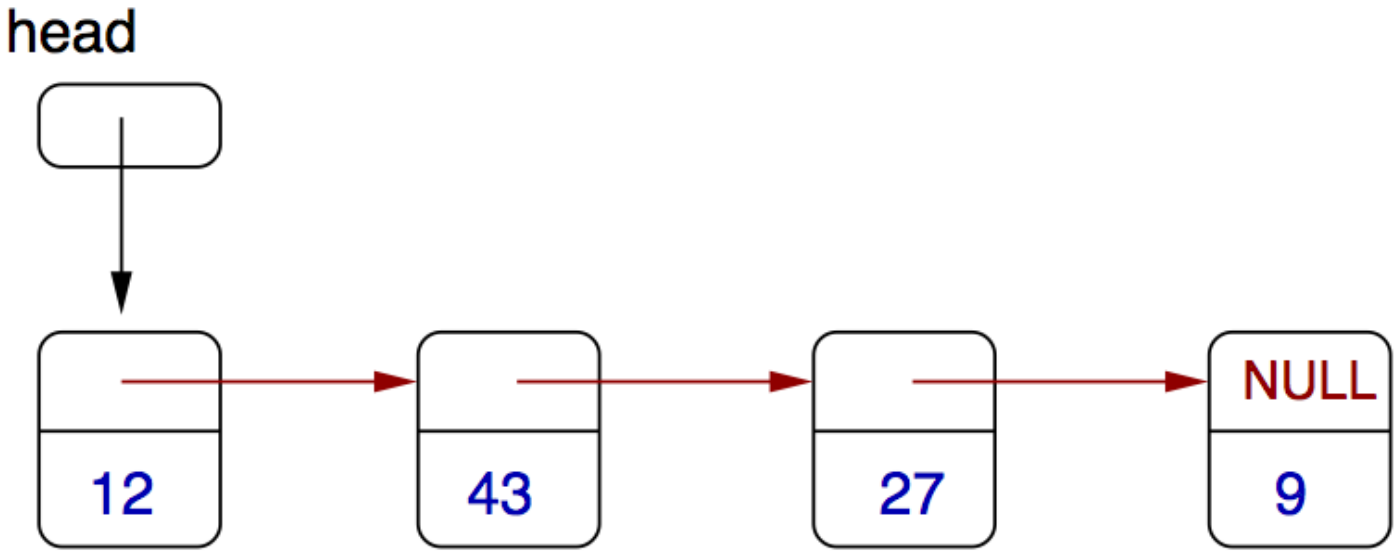
- Liste = séquence ordonnée d'objets de même type.
  Exemple : liste d'entiers $(12, 43, 27, 9)$

Liste chaînée - Définition

- Liste = séquence ordonnée d'objets de même type.
  Exemple : liste d'entiers $(12, 43, 27, 9)$
  - l'ordre : $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
  - la multiplicité : $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$

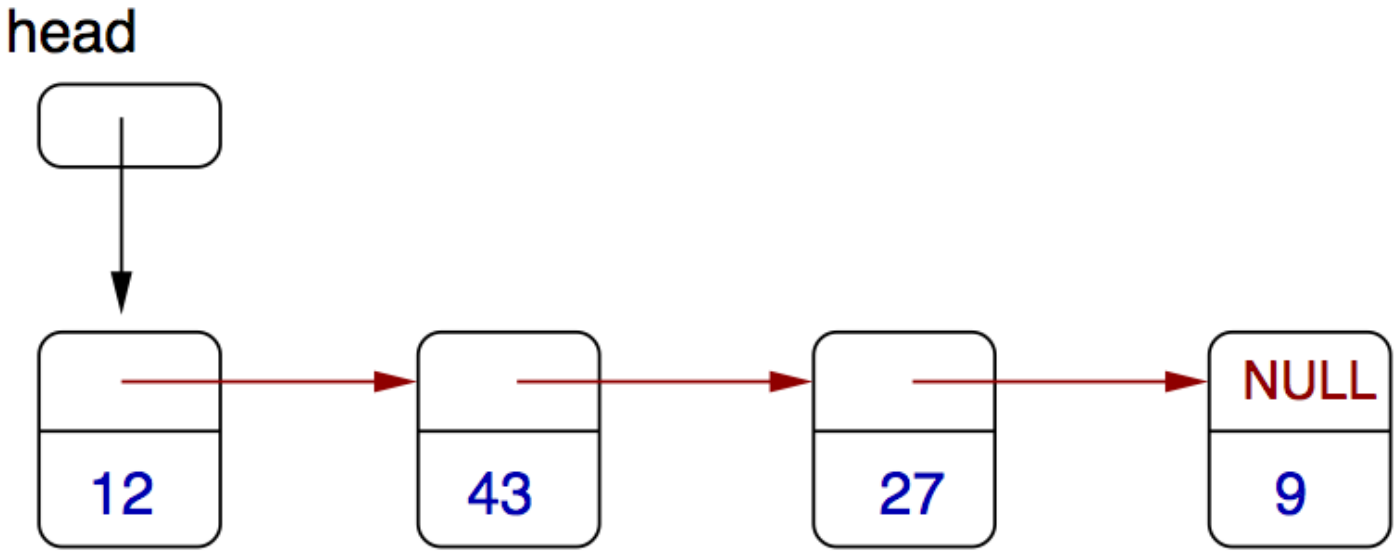
Liste chaînée - Définition

- Liste = séquence ordonnée d'objets de même type.
  Exemple : liste d'entiers $(12, 43, 27, 9)$
  - l'ordre : $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
  - la multiplicité : $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$



Liste chaînée - Définition

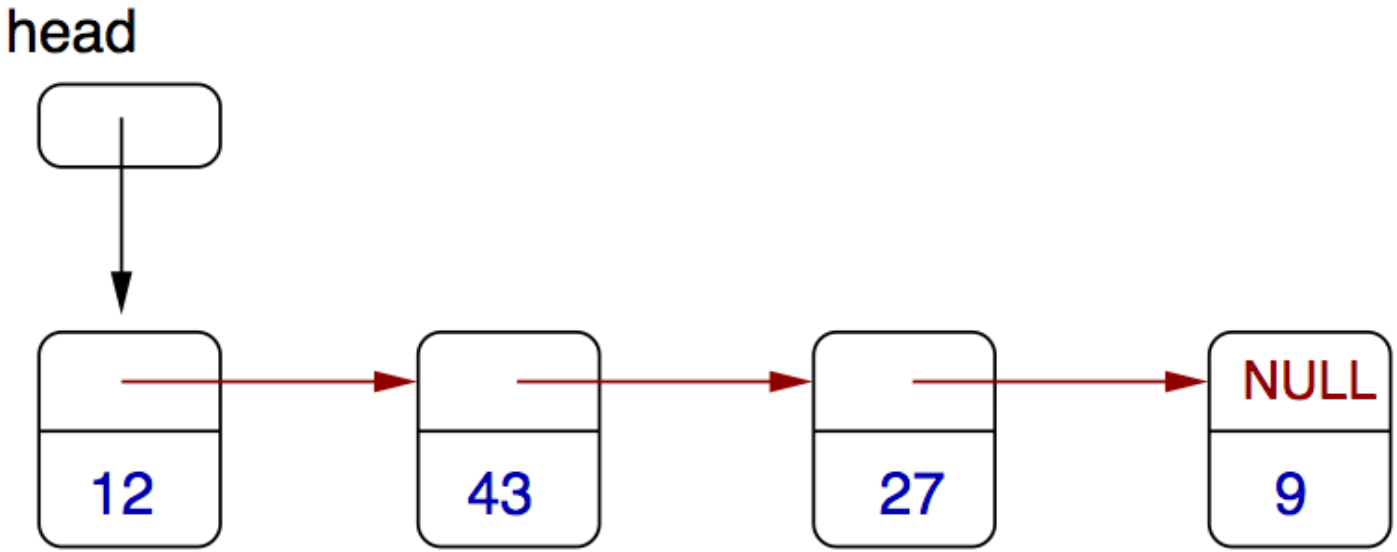
- Liste = séquence ordonnée d'objets de même type.
  Exemple : liste d'entiers $(12, 43, 27, 9)$
  - l'ordre : $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
  - la multiplicité : $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$



- Les éléments sont chaînés entre eux

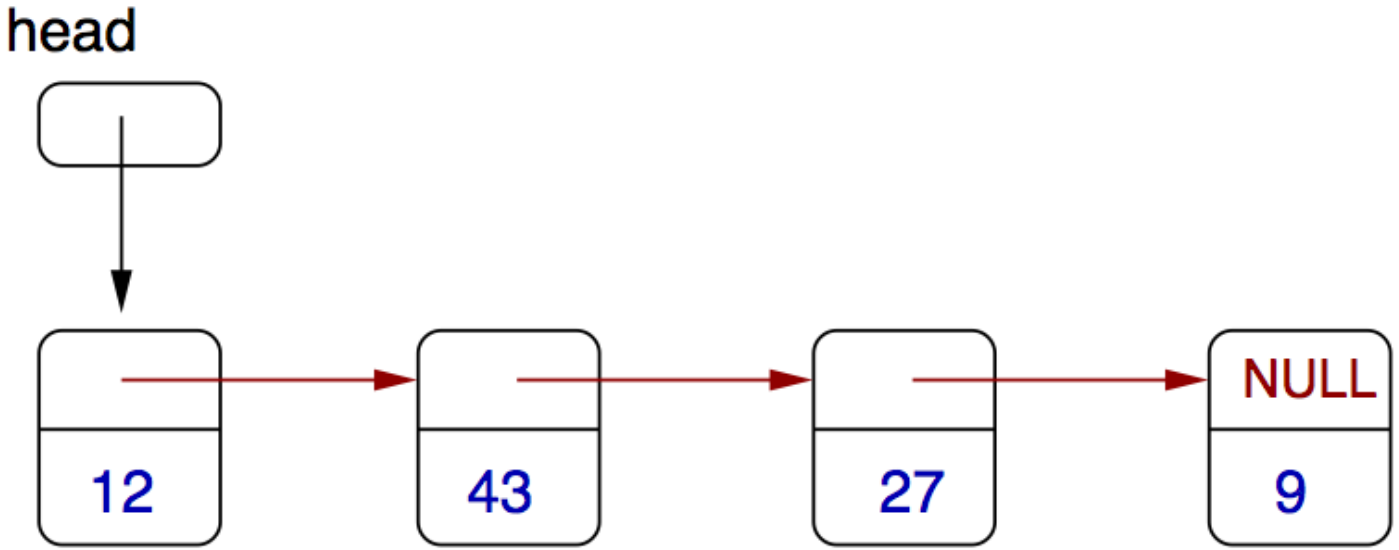
Liste chaînée - Définition

- Liste = séquence ordonnée d'objets de même type.
  Exemple : liste d'entiers $(12, 43, 27, 9)$
  - l'ordre : $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
  - la multiplicité : $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$



- Les éléments sont chaînés entre eux
- Chaque élément contient des informations sur l'objet et un pointeur vers un autre élément de la liste, ou un pointeur NULL

Liste chaînée - Caractéristiques



Avantages :

- Gestion d'une structure de données non-contiguë
- Insertion et suppression faciles (pas coûteuses)

Défauts :

- Accès aléatoire coûteux
- Mémoire pour les pointeurs

Liste chaînée - Déclaration/Initialisation

Pour représenter un maillon (`typedef`) :

```
typedef struct element {
 type1 champ1;
 ...
 typeN champN;
 struct element * suivant;
} Element;
...
Element *tete = NULL;
```

- Tous les éléments sont accessibles depuis la tête de liste

Liste chaînée - Opérations

Opérations courantes sur une liste chaînée :

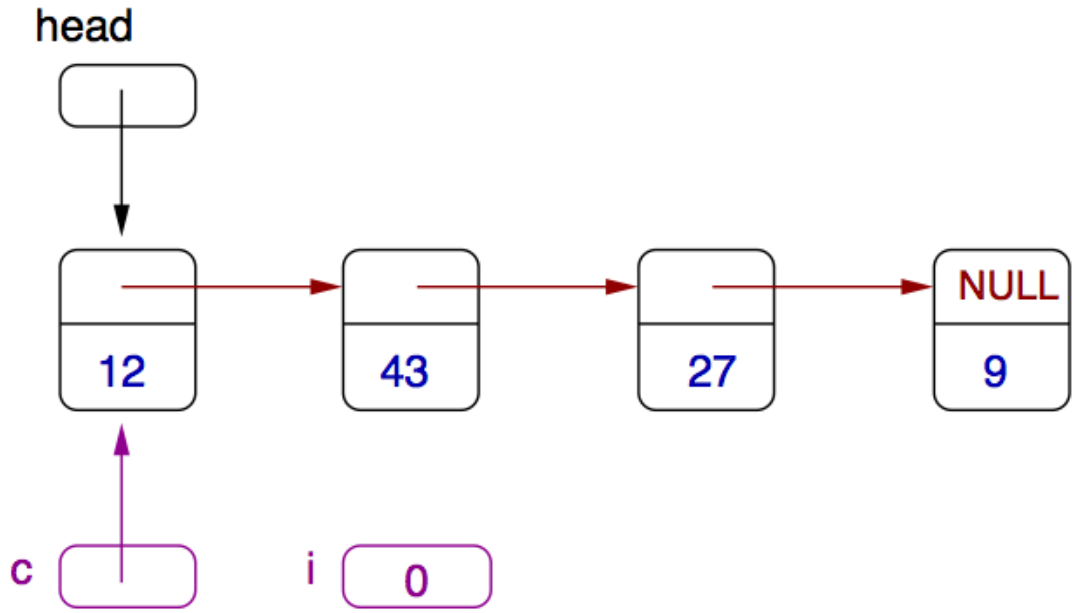
- calcul de la longueur d'une liste,
- recherche d'un élément,
- insertion d'un élément,
- suppression d'un élément,
- concaténation de deux listes,
- destruction d'une liste.

Toutes nos fonctions prennent une tête de liste en paramètre.

Calcul de la longueur

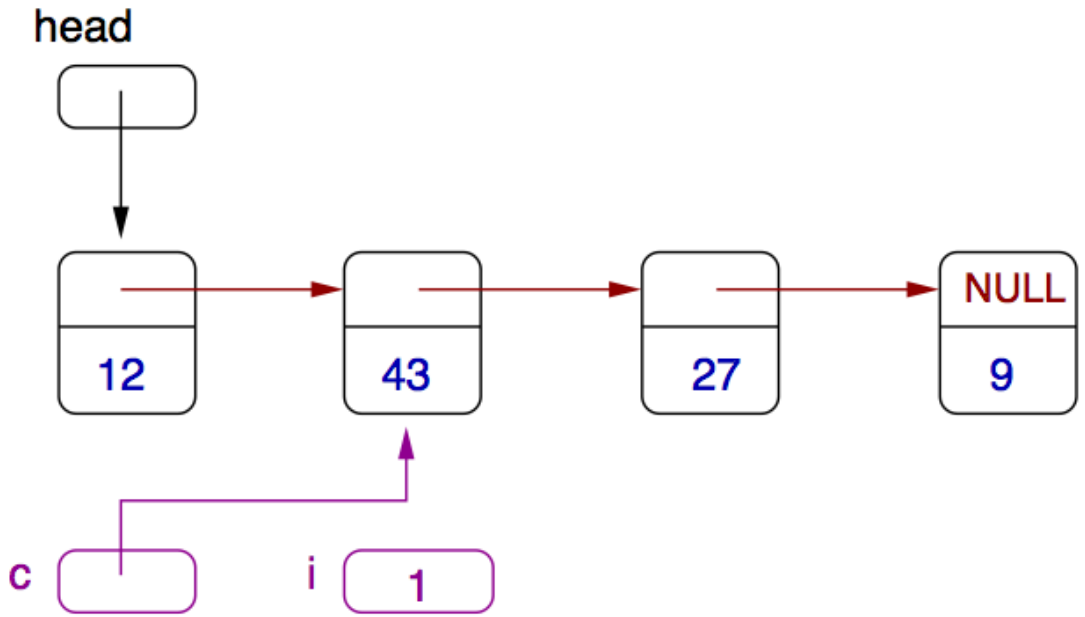
Quoi (spécification) : la longueur d'une liste

Comment (algorithme) :
on suit les pointeurs "suivant" jusqu'à rencontrer NULL et on compte le nombre d'éléments visités.



Calcul de la longueur (1)

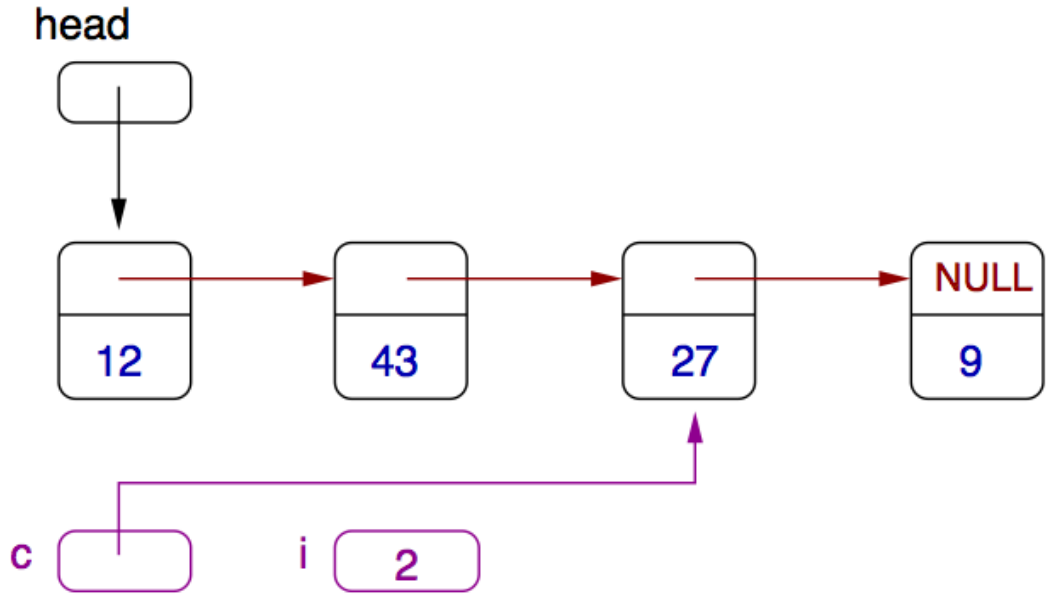
Comment (algorithme) :
on suit les pointeurs "suivant" jusqu'à rencontrer NULL et on compte le nombre de éléments visités.



Calcul de la longueur (2)

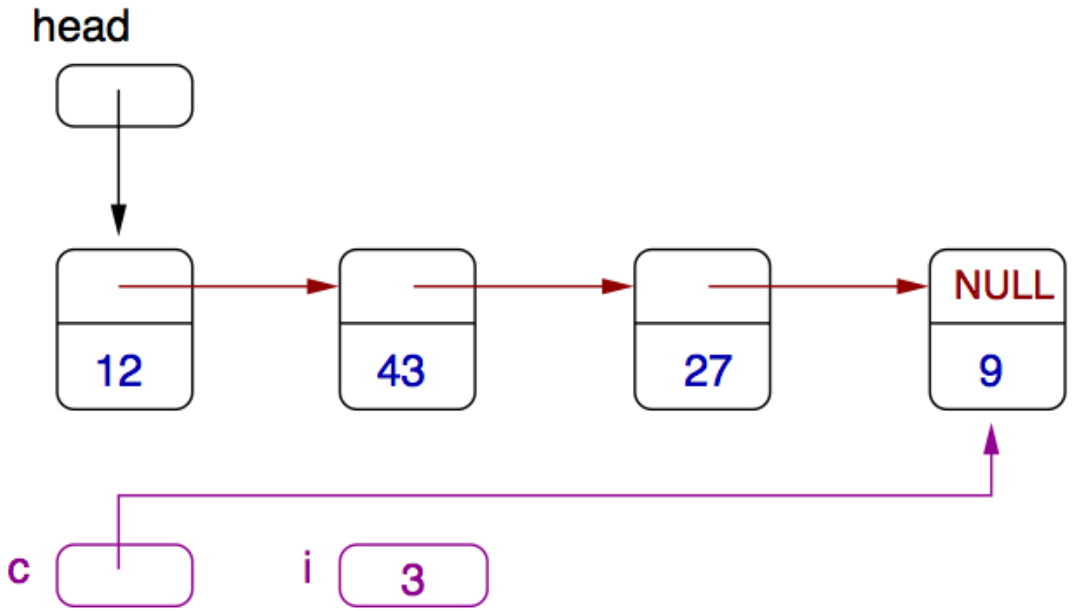
Comment (algorithme) :

on suit les pointeurs “suivant” jusqu'à rencontrer NULL et on compte le nombre de éléments visités.



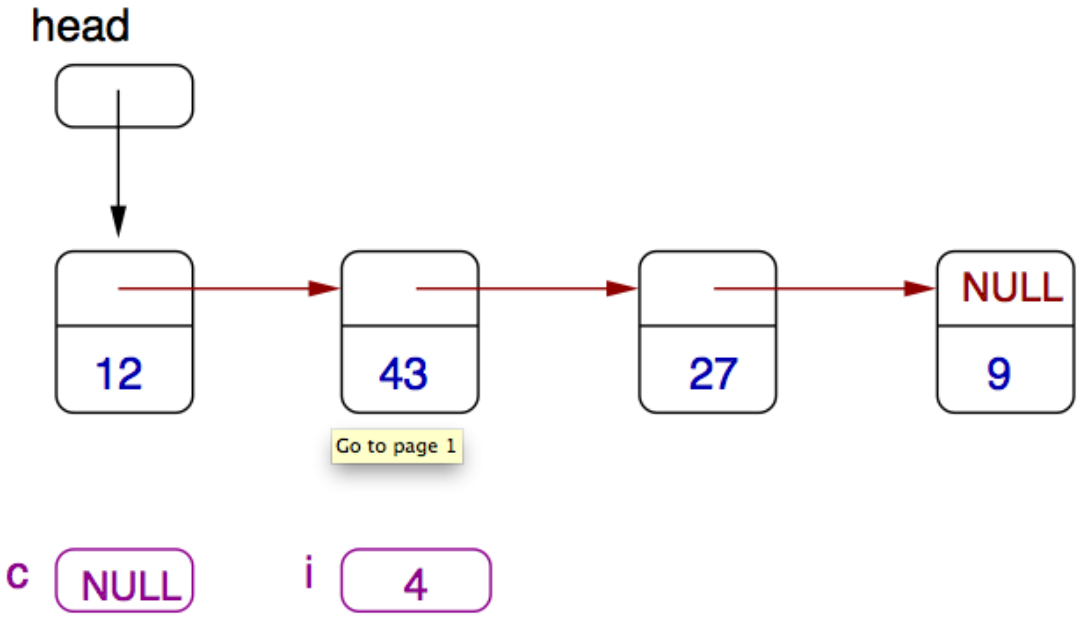
Calcul de la longueur (3)

Comment (algorithme) :
on suit les pointeurs “suivant” jusqu’à rencontrer NULL et on compte le nombre de éléments visités.



Calcul de la longueur (4)

Comment (algorithme) :
on suit les pointeurs “suivant” jusqu’à rencontrer NULL et on compte le nombre de éléments visités.



Recherche d'un élément

Quoi (spécification) : renvoie "`vrai`" si la liste contient un élément égal à "`elem`", "`faux`" sinon

Comment (algorithme) :
on suit les pointeurs "suivant" jusqu'à rencontrer NULL ou l'élément

Implémentation ?

Insertion : introduction

Insertion d'un élément en tête de liste :

- Allouer la mémoire pour le nouvel élément
- Remplir les champs de données du nouvel élément
- Le pointeur "`suivant`" du nouvel élément pointe vers la tête actuelle (ou NULL)
- Le pointeur "`tete`" pointe sur le nouvel élément pour maintenir la tête de la liste

Insertion : version naïve (à ne pas suivre)

Pour construire la liste : (12, 43, 27, 9) :

```
(1)  Element* tete;
(2)  tete = (Element*) malloc(sizeof(Element));
(3)  tete->donnees = 12;
(4)  tete->suivant = malloc(sizeof(Element));
(5)  tete->suivant->donnees = 43;
(6)  tete->suivant->suivant = malloc(sizeof(Element));
(7)  tete->suivant->suivant->donnees = 27;
(8)  tete->suivant->suivant->suivant = malloc(sizeof(Element));
(9)  tete->suivant->suivant->suivant->donnees = 9;
(10)  tete->suivant->suivant->suivant->suivant = NULL;
```

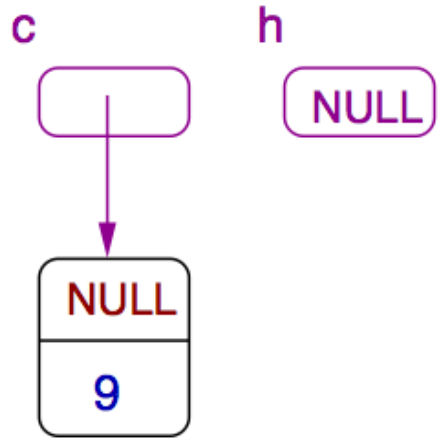
Remarque : Peu pratique. On préfère construire une liste par insertions successives.

Insertion en tête de liste

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```

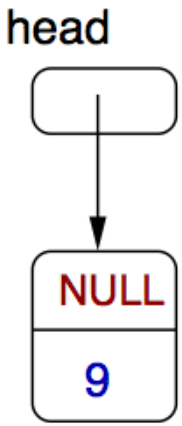
Insertion en tête de liste (2)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



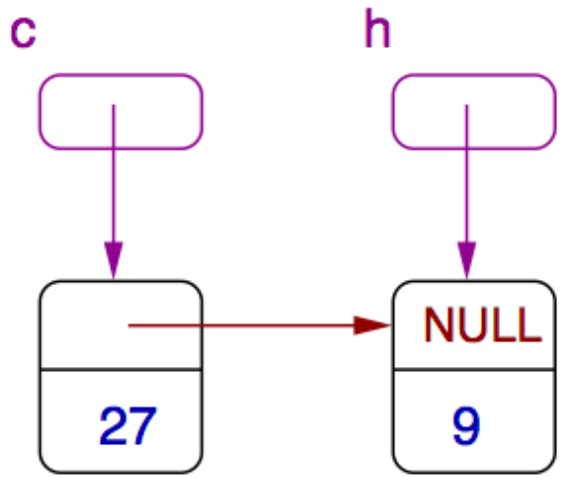
Insertion en tête de liste (3)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



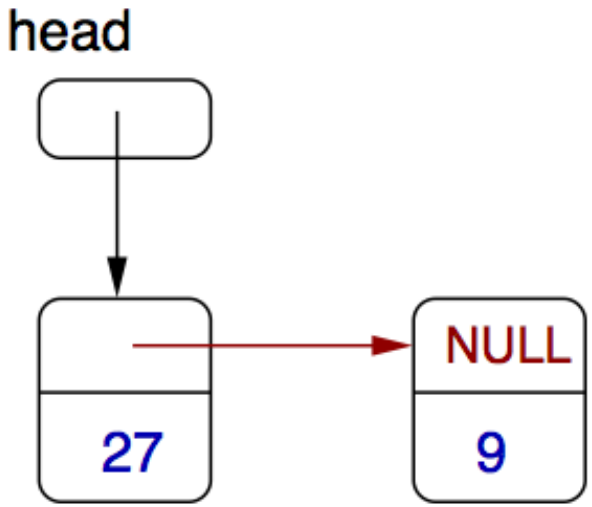
Insertion en tête de liste (4)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



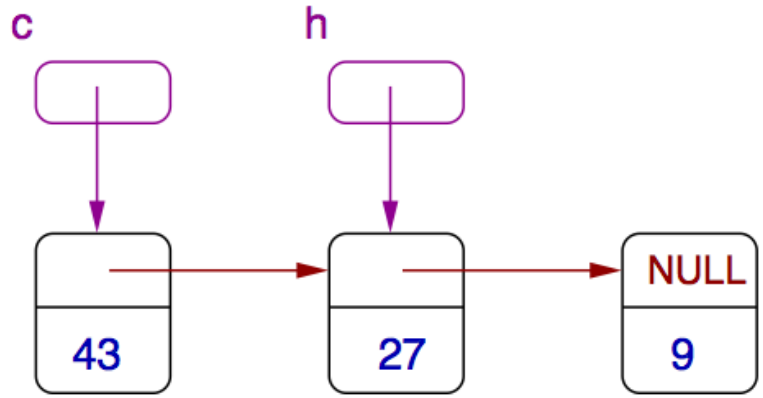
Insertion en tête de liste (5)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



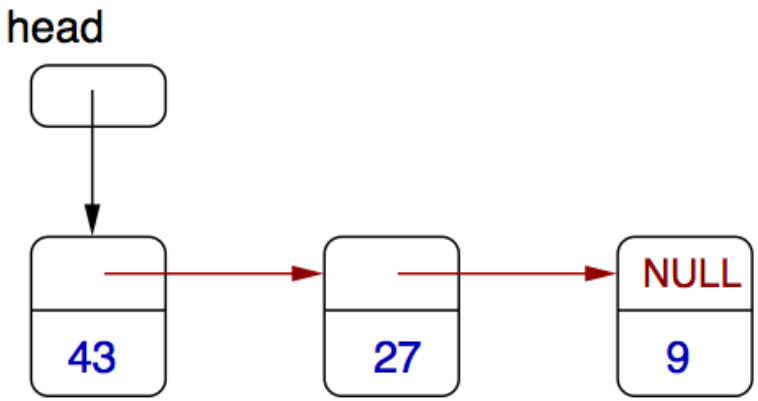
Insertion en tête de liste (6)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



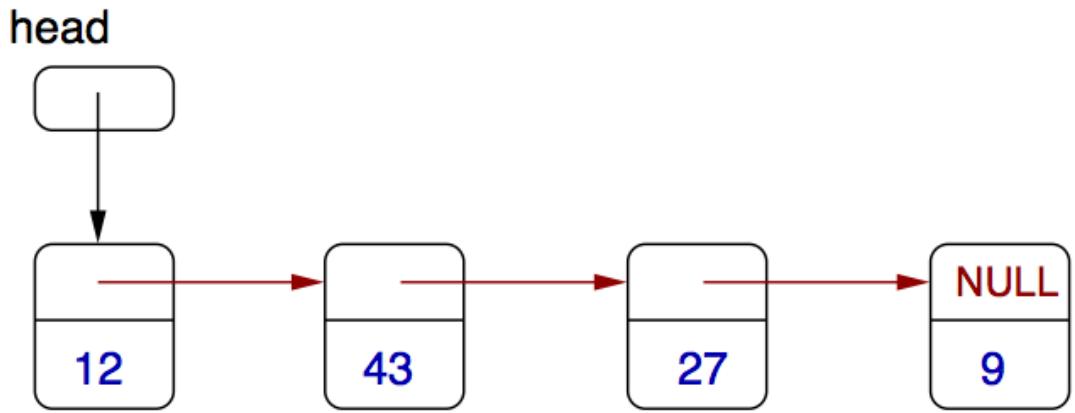
Insertion en tête de liste (7)

```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



Insertion en tête de liste (8)

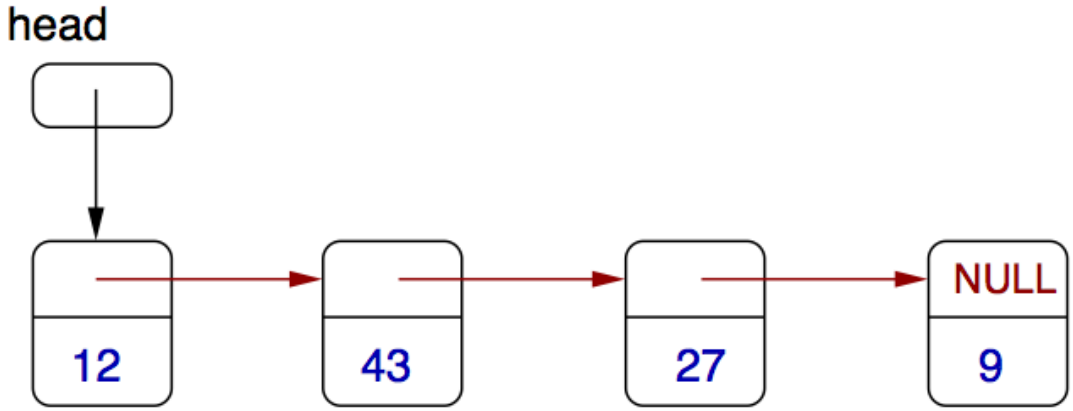
```
Element* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);}
```



Insertion en tête de liste

Remarques :

- la liste est dans l'ordre inverse de celui des insertions,
- "`insere`" fonctionne sur une liste vide ou non-vide,
- la tête de liste est modifiée à chaque insertion,
- le coût d'une insertion est constant.



Insertion en fin de liste

Insertion d'un élément en fin de liste non vide :

- Allouer la mémoire pour le nouvel élément
- Remplir les champs de données du nouvel élément
- Parcourir la liste pour obtenir le pointeur `dernier` vers le dernier élément
- "`suivant`" du nouvel élément pointe vers NULL
- "`dernier->suivant`" pointe vers le nouvel élément

Insertion après un élément

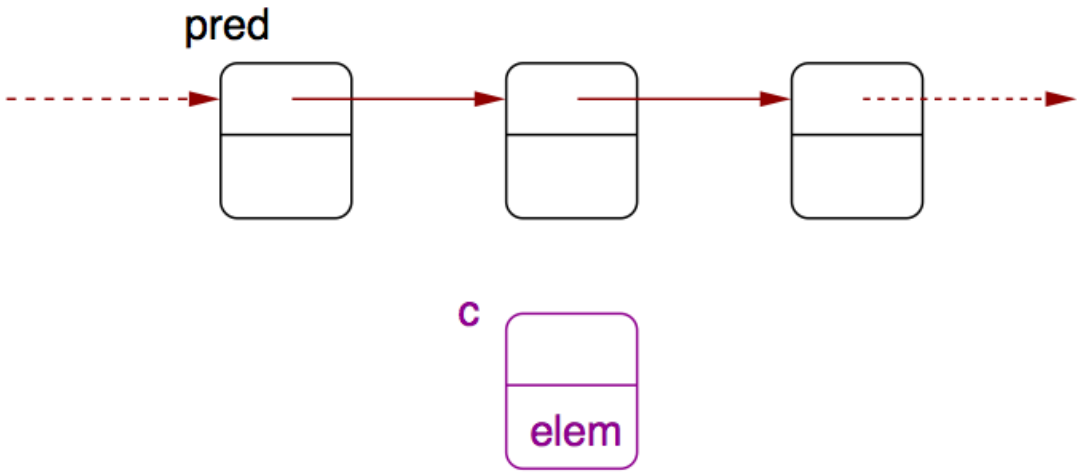
Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite insérer un élément après `pred`

Comment (algorithme) :

- Allouer la mémoire pour le nouvel élément et renvoie `c` vers cet élément
- Remplir les champs de données du nouvel élément
- `c->suivant` reçoit `pred->suivant`
- `pred->suivant` reçoit `c`



Insertion après un élément (2)

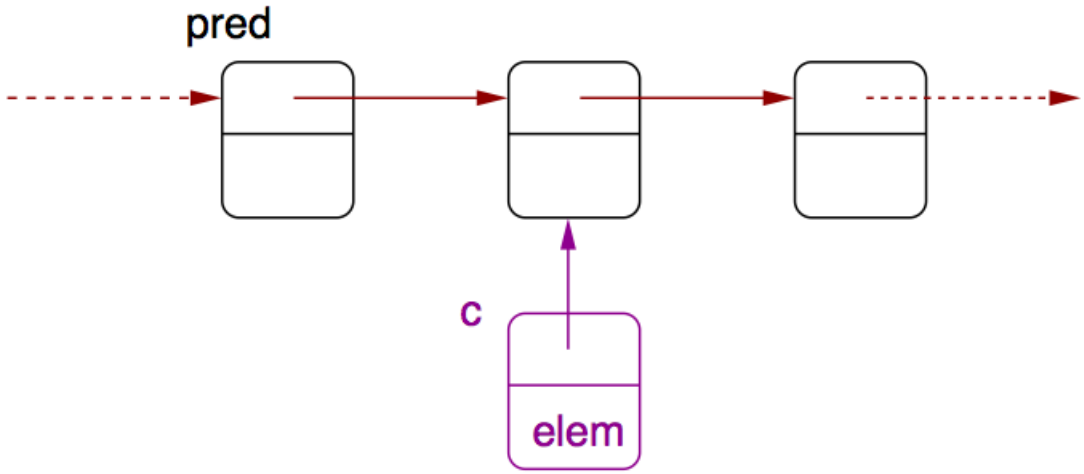
Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite insérer un élément après `pred`

Comment (algorithme) :

- Allouer la mémoire pour le nouvel élément et renvoie `c` vers cet élément
- Remplir les champs de données du nouvel élément
- `c->suivant` reçoit `pred->suivant`
- `pred->suivant` reçoit `c`



Insertion après un élément (3)

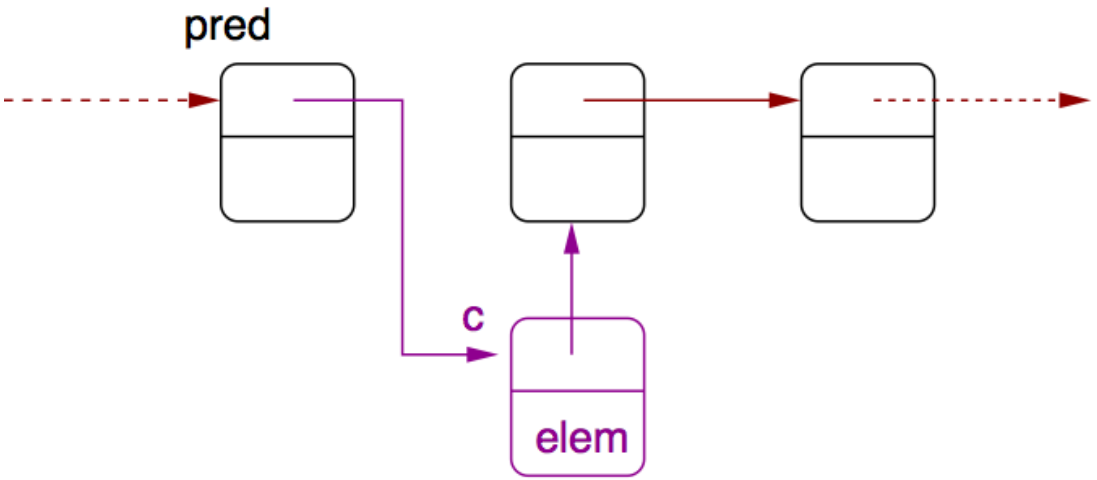
Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite insérer un élément après `pred`

Comment (algorithme) :

- Allouer la mémoire pour le nouvel élément et renvoie `c` vers cet élément
- Remplir les champs de données du nouvel élément
- `c->suivant` reçoit `pred->suivant`
- `pred->suivant` reçoit `c`



Suppression d'un élément

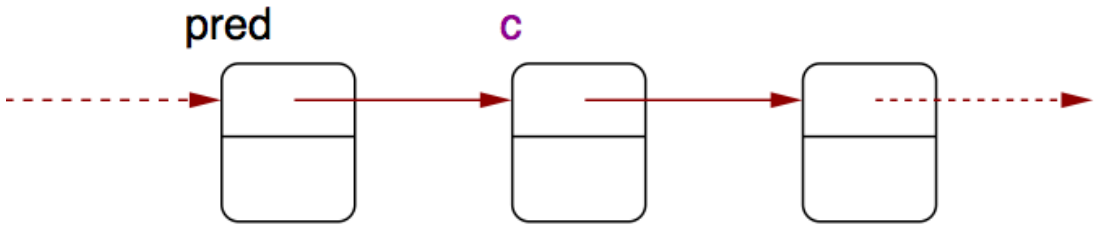
Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite supprimer un élément (pointé par `c`) après `pred`

Comment (algorithme) :

- `pred->suivant` reçoit `c->suivant`
- Libère la mémoire de l'élément pointé par `c`



Suppression d'un élément (2)

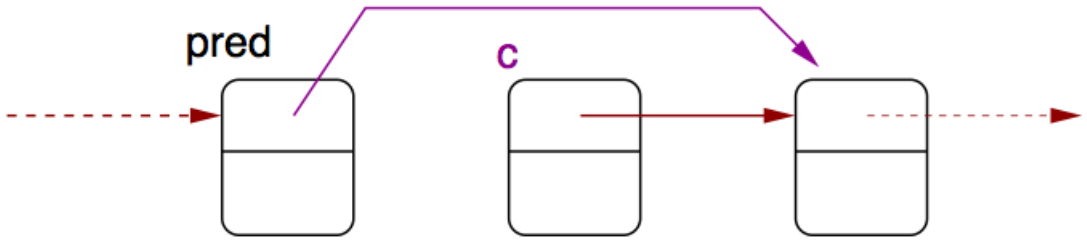
Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite supprimer un élément (pointé par `c`) après `pred`

Comment (algorithme) :

- `pred->suivant` reçoit `c->suivant`
- Libère la mémoire de l'élément pointé par `c`



Suppression d'un élément (3)

Quoi (spécification) :

Soit `pred` un pointeur vers un élément.

On souhaite supprimer un élément (pointé par `c`) après `pred`

Comment (algorithme) :

- `pred->suivant` reçoit `c->suivant`
- Libère la mémoire de l'élément pointé par `c`

Terminer le tableau

Pour la liste chaînée :

- Taille ?
- Mémoire ?
- Non-contiguïté ?
- Accès aléatoire ?
- Insertion/Suppression ?