Cours 8

Le type caractère (`char`)
Les chaînes de caractères
Les structures

Vincent Lepetit

vincent.lepetit@epfl.ch

Les chaînes de caractères

Nous avons déjà vu des chaînes de caractères dans les instructions `cout`:

```
cout << "bonjour" << endl;
```

`"bonjour"` correspond à une chaîne formée de 7 caractères:

`b`, `o`, `n`, `j`, `o`, `u`, `r`.

En C++ (et dans d'autres langages), il existe un type "chaîne de caractères".

En C, une chaîne de caractères est simplement un ***tableau de caractères***.

Nous allons voir:

- le **type caractère**,
- les tableaux de caractères,
- les fonctions C sur les tableaux de caractères.

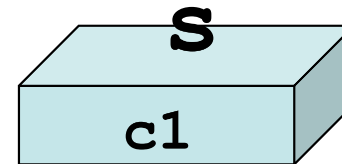
Le type caractère `char`

Le type `char` permet de représenter un caractère
→ une lettre, un chiffre, un symbole comme `* - + { }` …

On peut déclarer des variables de type `char`, de la même façon que les variables de type `int` ou `float`.

Déclaration:

```
char c1, c2;
```



Initialisation:

```
c1 = 's';
```

affecte à la variable `c1` le caractère `s`, **noté entre apostrophes** `'` (ou simples cotes).

Affectation:

```
c2 = c1;
```

Entrées/Sorties pour les variables de type `char`

Comme pour `int` ou `float`:

Affichage:

```
char c = 's';

cout << "Le caractere est " << c << "." << endl;
```

→ on obtient:

```
Le caractere est s.
```

Lecture:

```
char c1;
cin >> c1;
cout << "Vous avez tape " << c1 << endl;
```

Codage du type `char`

Le type `char` est codé sur un octet soit 8 bits

→ il peut donc coder 256 (=2^8) caractères différents.

Il existe des caractères spéciaux:

- le retour à la ligne qui se note `\n`,
- la tabulation qui se note `\b`,
- le caractère de fin de chaîne qui se note `\0`, etc…

Code ASCII (pour *American Standard Code for Information Interchange*):

Comme un ordinateur ne peut stocker que des valeurs numériques, il faut décider d'une convention pour savoir à quel caractère correspond une valeur donnée. La convention adoptée s'appelle le code ASCII.

- la valeur 65 représente le caractère `A` (majuscule);
- la valeur 66 représente le caractère `B` (majuscule);
- la valeur 97 représente le caractère `a` (minuscule);
- la valeur 32 représente l'espace;
- etc...

Codes ASCII

32	Espace	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	

Expressions avec des caractères

On peut écrire des expressions faisant intervenir des variables de type `char`.

Ceci est intéressant par exemple pour :

- transformer un caractère contenant un chiffre en sa valeur numérique, ou
- tester si un caractère est une lettre minuscule.

L'expression est évaluée en utilisant le code ASCII.

Mais en pratique on n'a pas besoin de connaître les codes ASCII, voir les 2 exemples suivants.

Transformer un caractère contenant un chiffre en sa valeur numérique

Exemple:

```
char c1 = '5';

int chiffre = c1 - '0';
```

→ `chiffre` contient la valeur 5, calculée comme la différence entre
le code ASCII du caractère `'5'` (53) et
le code ASCII du caractère `'0'` (48).

Tester si un caractère contient une lettre minuscule

Exemple:

```
char c2 = 'h', c3 = 'G';

if (c2 >= 'a' && c2 <= 'z')
  cout << "c2 contient une minuscule." << endl;

if (c3 >= 'a' && c3 <= 'z')
  cout << "c3 contient une minuscule." << endl;
```

affichera:

```
c2 contient une minuscule.
```

Les comparaisons du type `c2 >= 'a'` se font entre codes ASCII.

Obtenir le code ASCII d'un caractère

On peut convertir un caractère en `int` pour obtenir son code ASCII:

```
char c = 'A';
cout << int(c) << " -> " << c << endl;

c = char(66);
cout << int(c) << " -> " << c << endl;
```

affichera:

```
65 -> A
66 -> B
```

Les chaînes de caractères

Les chaînes de caractères

Déclaration, comme un *tableau de `char`*:

```
char nom[30];
```

Initialisation lors de la déclaration:

```
char nom[30] = "Dupont";
```

nom	D	u	p	o	n	t	\0	?	?	?	...

Par contre, l'initialisation en dehors de la déclaration n'est pas possible.

On n'a pas le droit d'écrire:

```
nom = "Dupont";
```

dans le corps du programme.

Les chaînes de caractères

Affichage:

```
cout << nom << endl;
```

Lecture au clavier:

```
cin >> nom;
```

Attention

On utilise des apostrophes (`'`), ou simple-cote, pour définir un caractère:

```
char lettre = 'D';
```

On utilise des guillemets (`"`), ou double-cote, pour définir une chaîne:

```
char nom[30] = "Dupont";
```

Longueur d'une chaîne

La chaîne

`char nom[30];`

nom	?	?	?	?	?	?	?	?	?	?	...

peut contenir *au maximum* 30 caractères, c'est-à-dire qu'elle va pouvoir contenir par exemple les chaînes:

`Du` ou
`pont` ou
`Dupont` etc...

Mais pour fonctionner correctement, `cout` et les fonctions de manipulation de chaînes (`strcmp`, `strlen`, ... voir transparents suivants) **doivent connaître le nombre de caractères** (2 dans le cas de `Du`, 4 dans le cas de `pont`, etc...).

Pour cela, le C utilise le caractère de fin de chaîne, qui est un caractère spécial noté `\0`, et qui a 0 comme code ASCII.

Caractère de fin de chaîne

Par exemple, si on fait:

```
char nom[30] = "Du";
```

`nom` contient: **nom**

D	u	**\0**	?	?	?	?	?	?	?	...

Si on fait:

```
char nom[30] = "Dupont";
```

`nom` contient: **nom**

D	u	p	o	n	t	**\0**	?	?	?	...

Le caractère `\0` est ajouté automatiquement à la déclaration.

De même, `cin` ajoute le caractère `\0` à la fin de la chaîne entrée par l'utilisateur.

Fonctions relatives aux chaînes

Nous allons voir quelques fonctions de manipulation de chaînes de caractères:

- **`strcmp`** qui permet de comparer deux chaînes;
- **`strcpy`** qui permet de copier des chaînes;
- **`strlen`** qui permet de calculer la longueur d'une chaîne.
- **`atoi`** et **`atof`** qui permettent de convertir une chaîne en une valeur entière ou flottante.

Pour utiliser ces fonctions, il faut inclure le fichier `string.h`, c'est-à-dire ajouter au début du programme:

```
#include <string.h>
```

Comparer des chaînes

Soit deux chaînes `mot1` et `mot2` déclarées suivant:

```
char mot1[100] = "bonjour";
char mot2[100] = "hello";
```

Supposons qu'on veuille tester si `mot1` est égal à `mot2`.

On ne peut pas faire:

```
if (mot1 == mot2)
   ...
```

De même on ne peut pas faire:

```
if (mot1 < mot2)
   ...
```

pour tester si `mot1` est avant `mot2` dans l'ordre alphabétique.

Ces 2 tests comparent les ADRESSES des premières lettres de `mot1` et `mot2`, PAS `mot1` et `mot2`:

`mot1` et `mot2` sont des tableaux, ils contiennent donc les adresses de leur premier élément !

Pour comparer des chaînes: `strcmp`

Il faut utiliser la fonction `strcmp` (pour *strings comparison*).

Cette fonction s'appelle en donnant en argument les deux chaînes à comparer;

```
int comp = strcmp(mot1, mot2);
```

et renvoie une valeur entière qui est:

- positive si `mot1` est après `mot2` dans l'ordre du code ASCII;
- nulle si `mot1` est égal à `mot2`;
- négative si `mot1` est avant `mot2`.

Pour comparer des chaînes: `strcmp`

Exemple:

```
int comparaison = strcmp(mot1, mot2);

if (comparaison < 0)
  cout << "Dans l ordre alphabetique." << endl;
else if (comparaison == 0)
  cout << "Egaux." << endl;
else
  cout << "Dans l ordre inverse." << endl;
```

Recopier des chaînes

Soit deux chaînes `mot1` et `mot2` déclarées suivant:

```
char mot1[100];
char mot2[100] = "hello";
```

Pour copier la chaîne `mot2` dans `mot1`, **on ne peut pas faire**:

```
mot1 = mot2;
```

Cette instruction copie l'ADRESSE du premier caractère de `mot2` dans `mot1`.

Pour recopier des chaînes: `strcpy`

Il faut utiliser la fonction `strcpy` (pour *string copy*).

```
strcpy(mot1, mot2);
```

copie `mot2` dans `mot1`.

Exemple:

```
char mot1[100] = "bonjour";
char mot2[100] = "hello";

cout << mot1 << " " << mot2 << endl;
strcpy(mot1, mot2);
cout << mot1 << " " << mot2 << endl;
```

va afficher:

```
bonjour hello
hello hello
```

Convertir une chaîne en une valeur numérique: `atoi` et `atof`

Exemple:

```
char ch1[100] = "123";
int i = atoi(ch1);
```

`i` contient maintenant la valeur `123`.

Pour les valeurs à virgule, on utilise `atof` (pour *alphanumeric to `float`*).

Exemple:

```
char ch2[100] = "123.45";
float f = atof(ch1);
```

Longueur d'une chaîne: `strlen`

La fonction `strlen` (pour *string length*) renvoie la longueur de la chaîne passée en paramètre.

Exemple:

```
char mot[100] = "hello";
int L;
```

après l'affectation:

```
L = strlen(mot);
```

la variable `L` contiendra 5 (la chaîne `"hello"` contient 5 caractères).

Accéder à un caractère de la chaîne

Comme une chaîne est un tableau de caractères, on peut accéder à un de ses caractères en utilisant des crochets `[ ]`:

Exemple:

```
char mot[100] = "hello";
char c;
```

```
c = mot[1];
```

affecte à `c` le deuxième caractère de `mot`, c'est-a-dire `'e'`.

```
mot[1] = 'a';
```

remplace le deuxième caractère de `mot` par `'a'`. `mot` contient alors `"hallo"`.

```
c = mot[5];
```

`c` contient le caractère de fin de chaîne `\0`.

Exemple 1

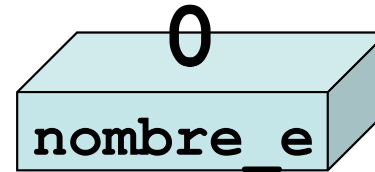
```
#include <iostream>
#include <string.h>
using namespace std;

int main(int argc, char ** argv)
{
  char ch1[100], ch2[100];

  cout << "Entrez une chaine: " << endl;
  cin >> ch1;
  cout << "Entrez une autre chaine: " << endl;
  cin >> ch2;

  cout << "La chaine " << ch1 << " a "
       << strlen(ch1) << " caracteres." << endl;

  if (strcmp(ch1, ch2) == 0)
    cout << "Les 2 chaines sont identiques." << endl;

  return 0;
}
```

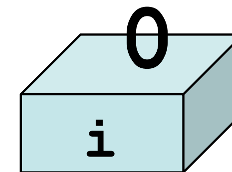
Exemple 2

Compter le nombre de *e* dans la chaîne de caractères `mot` :

```
int nombre_e = 0;

for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

Pas-à-pas

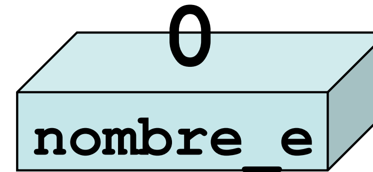


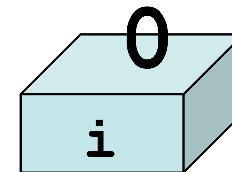



```
int nombre_e = 0;

for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot | e | x | e | m | p | l | e | **\0** | ? | ? | ...

Pas-à-pas



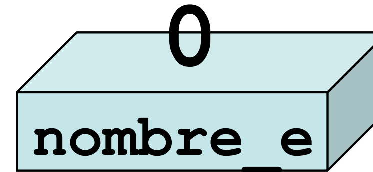


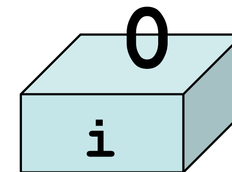

```
int nombre_e = 0;
                         7
for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot

e	x	e	m	p	l	e	**\0**	?	?	...

Pas-à-pas

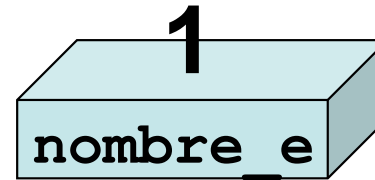


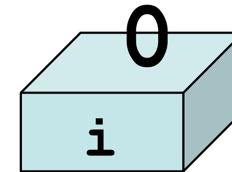



```
int nombre_e = 0;
                          7
for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot	e	x	e	m	p	l	e	\0	?	?	...

Pas-à-pas

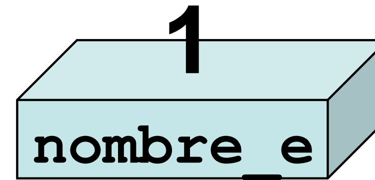


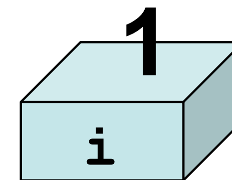



```
int nombre_e = 0;
                                7
for(int i = 0; i < strlen(mot); i++)
   if (mot[i] == 'e')
      nombre_e = nombre_e + 1;
```

mot	e	x	e	m	p	l	e	\0	?	?	...

Pas-à-pas

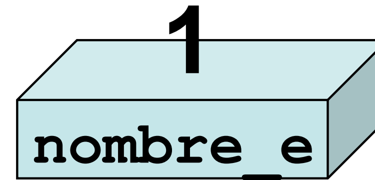


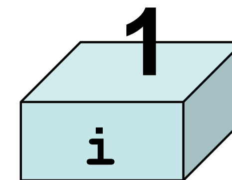



```
int nombre_e = 0;
                         7
for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot | e | x | e | m | p | l | e | \0 | ? | ? | ...

Pas-à-pas

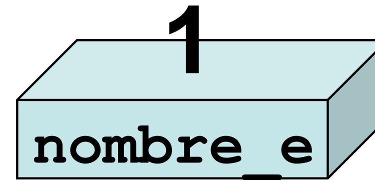


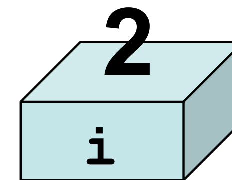



```
int nombre_e = 0;
                                     7
for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot	e	x	e	m	p	l	e	**\0**	?	?	...

Pas-à-pas



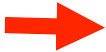



```
int nombre_e = 0;
                          7
for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```

mot	e	x	e	m	p	l	e	**\0**	?	?	...

Pas-à-pas

```
int nombre_e = 0;

for(int i = 0; i < strlen(mot); i++)
  if (mot[i] == 'e')
    nombre_e = nombre_e + 1;
```



Exemple 3

Fonction qui remplace chacune des voyelles d'un mot passé en paramètre par un caractère passé en paramètre:

```
void remplace(char * mot, char nc)
{
  for(int i = 0; i < strlen(mot); i++)
  {
    char c = mot[i];
    if (c == 'a' || c == 'e' || c == 'i' ||
        c == 'o' || c == 'u' || c == 'y')
      mot[i] = nc;
  }
}
```

cherche_base

Ecrivez une fonction `cherche_base` d'en-tête

```
int cherche_base(char * sequence, char base)
```

qui retourne le rang où apparaît pour la première fois la base représentée par le caractère `base` dans la séquence ADN `sequence`. Si la base n'apparaît pas, la fonction devra renvoyer -1 pour indiquer une erreur.

Exemples:

Après

```
int n = cherche_base("ATTGCC", 'C');
```

`n` doit contenir 4;

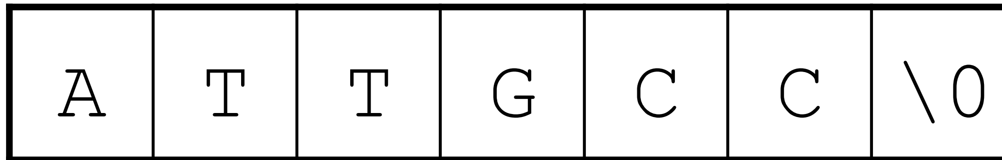
Après

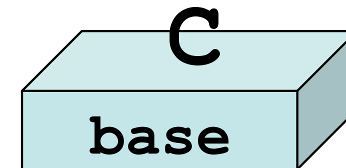
```
int n = cherche_base("GTTGCC", 'A');
```

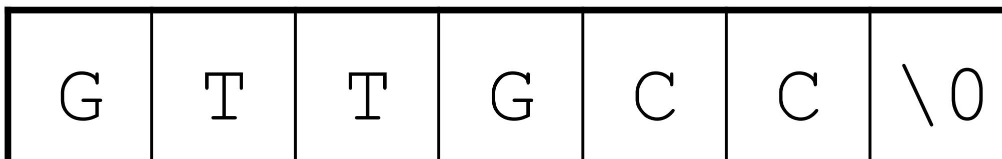
`n` doit contenir -1.

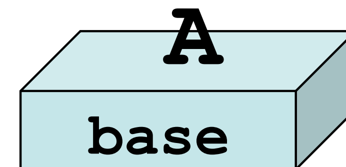
cherche_base

```
int cherche_base(char * sequence, char base)
{
  for(int i = 0; i < strlen(sequence); i++)
    if (sequence[i] == base)
      return i;

  return -1;
}
```







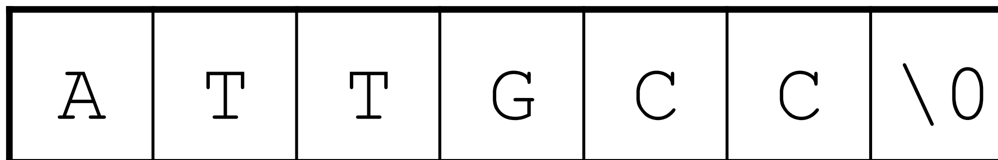


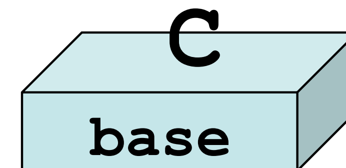
Attention

La fonction suivante ne marche pas:

```
int cherche_base(char * sequence, char base)
{
  for(int i = 0; i < strlen(sequence); i++)
    if (sequence[i] == base)
      return i;
    else
      return -1; // !!! FAUX
}
```

Dans le cas de l'exemple ci-dessous, on sort de la boucle quand `i` vaut 0 en renvoyant -1.





cherche_codon

Ecrivez une fonction `cherche_codon` d'en-tête

```
bool cherche_codon(char * sequence,
                   char base1, char base2, char base3)
```

qui retourne `true` si le codon défini par les 3 caractères `base1 base2 base3` apparaît dans la séquence ADN `sequence`, et `false` sinon.

Exemples:

Après

`bool ok = cherche_codon("ATTGCC", 'A', 'T', 'T');`

`ok` doit contenir `true`;

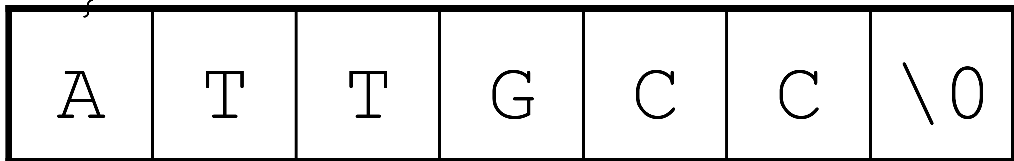
Après

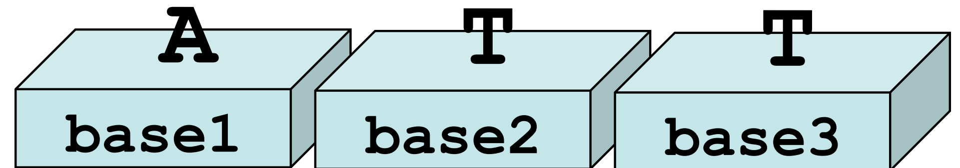
`bool ok = cherche_codon("ATTGCC", 'A', 'C', 'C');`

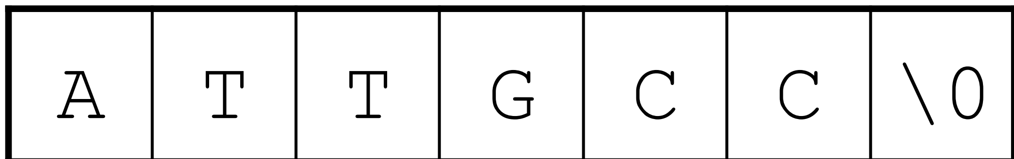
`ok` doit contenir `false`.

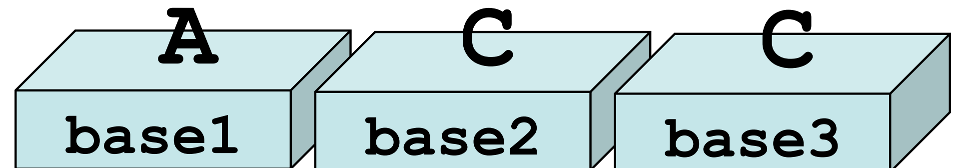
cherche_codon

```
bool cherche_codon(char * sequence,
                   char base1, char base2, char base3)
{
  for(int i = 0; i < strlen(sequence) - 2; i++)
    if (sequence[i    ] == base1 &&
        sequence[i + 1] == base2 &&
        sequence[i + 2] == base3)
      return true;

  return false;
}
```









complementaire

Écrivez une fonction qui retourne le complémentaire d'une séquence ADN:
A devient T, T devient A, C devient G, G devient C.

Utilisez le prototype (= en-tête) suivant:

```
char * complementaire(char * seq_originale)
```

→ Il faut allouer une nouvelle chaîne de caractères dans laquelle vous stockerez la séquence complémentaire.

→ Après avoir calculé la séquence complémentaire, la fonction retournera un pointeur sur la nouvelle chaîne.

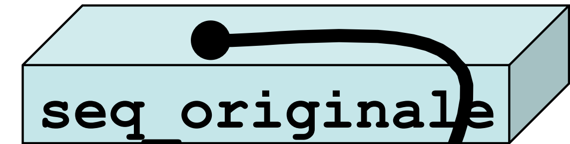
complementaire

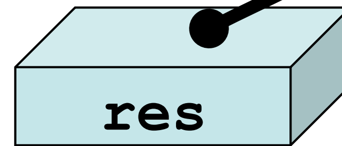
```
char * complementaire(char * seq_originale)
{
```

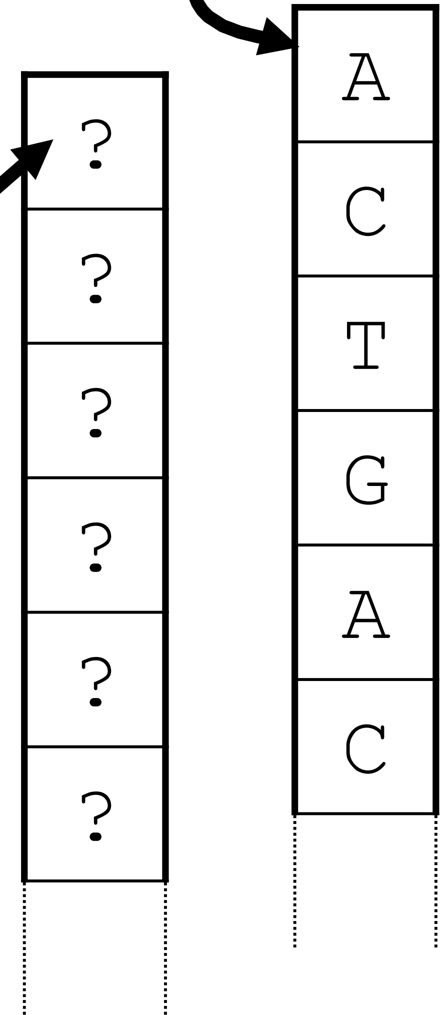
Commençons par allouer une chaîne de même taille que la chaîne `seq_originale` passée en paramètre.

Attention ! Il ne faut pas oublier de réserver une place de plus pour le caractère de fin de chaîne `'\0'` !

```
char * res = new char[strlen(seq_originale) + 1];
```





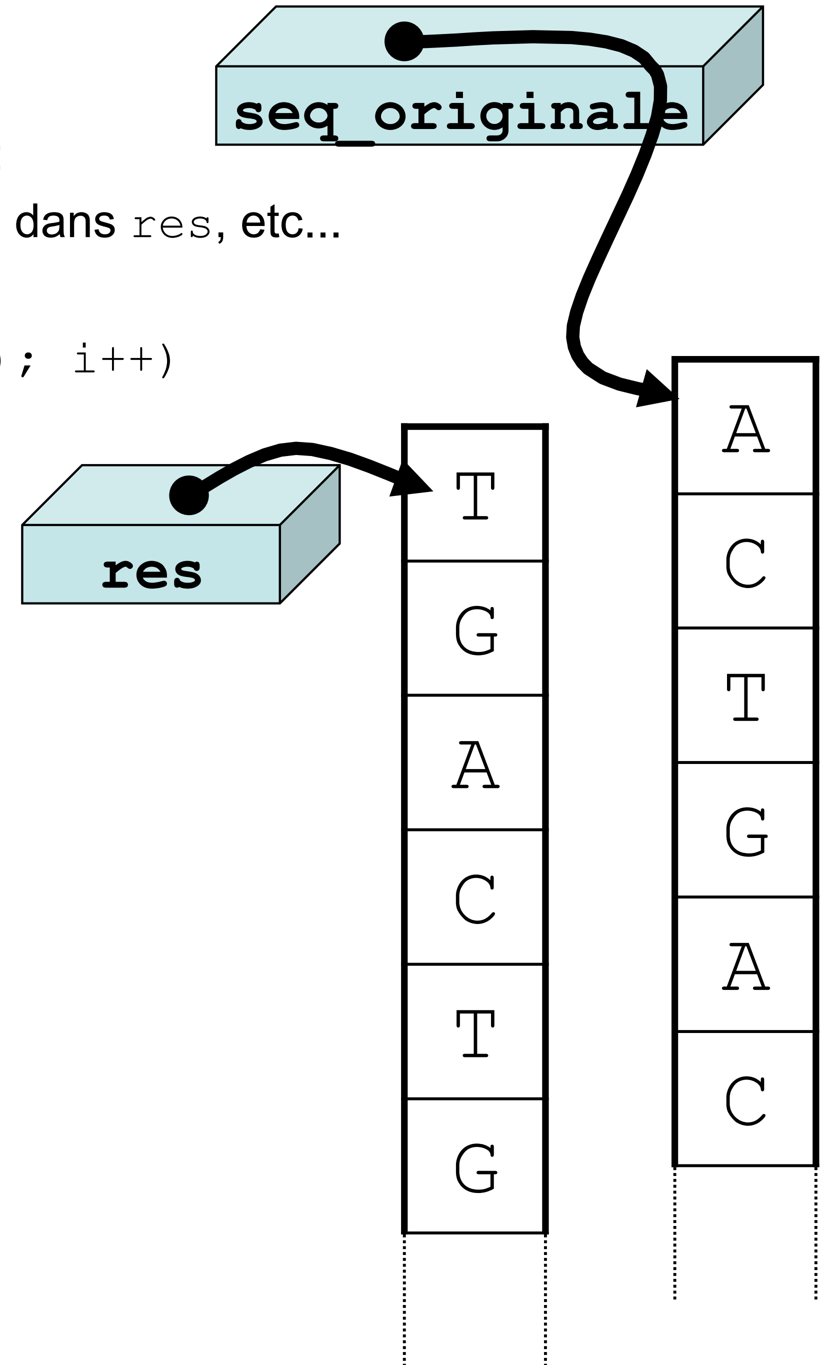


complementaire

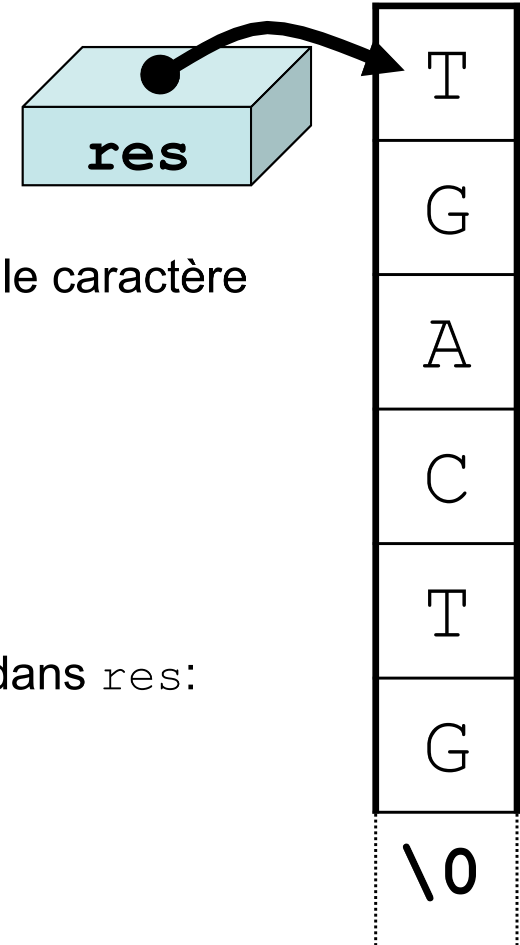
Parcourons les deux chaînes caractère par caractère:

Quand `seq_originale` contient un `A`, on place un `T` dans `res`, etc...

```
for(int i = 0; i < strlen(seq_originale); i++)
  switch(seq_originale[i])
  {
  case 'A':
    res[i] = 'T';
    break;
  case 'T':
    res[i] = 'A';
    break;

   (etc...)
  }
```



complementaire



Attention, il y a un piège !

Il faut encore dire où s'arrête la chaîne `res`, c'est-à-dire placer le caractère `'\0'` au bon endroit:

```
res[strlen(seq_originale)] = '\0';
```

Il reste à retourner l'adresse de la mémoire allouée, contenue dans `res`:

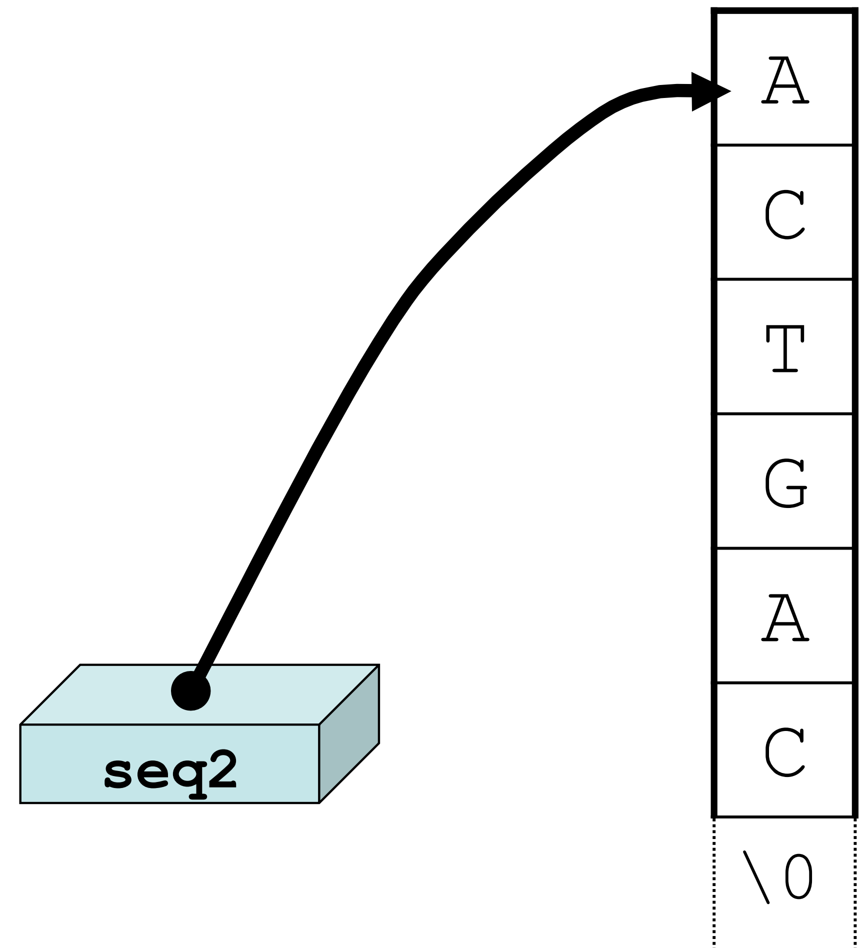
```
  return res;
```

complementaire

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
```

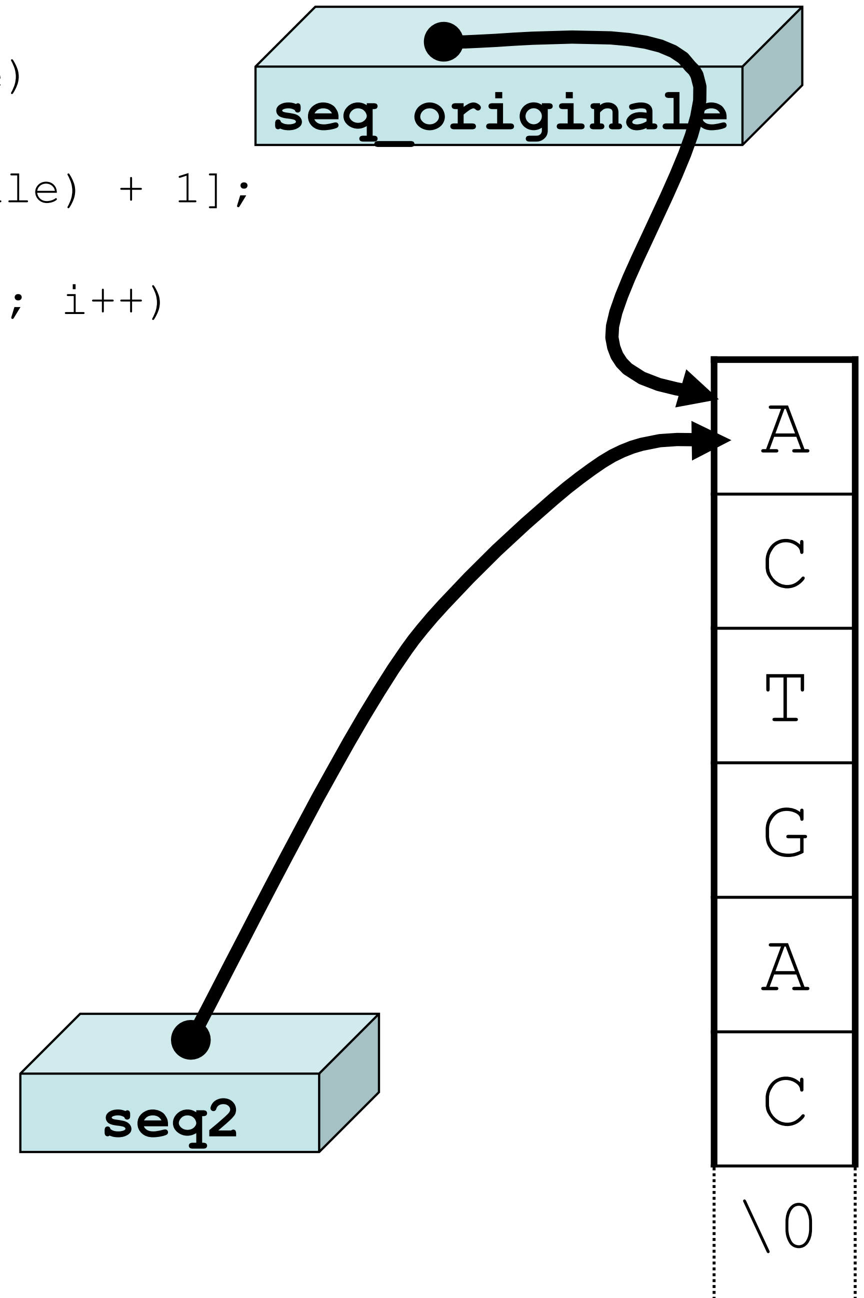
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



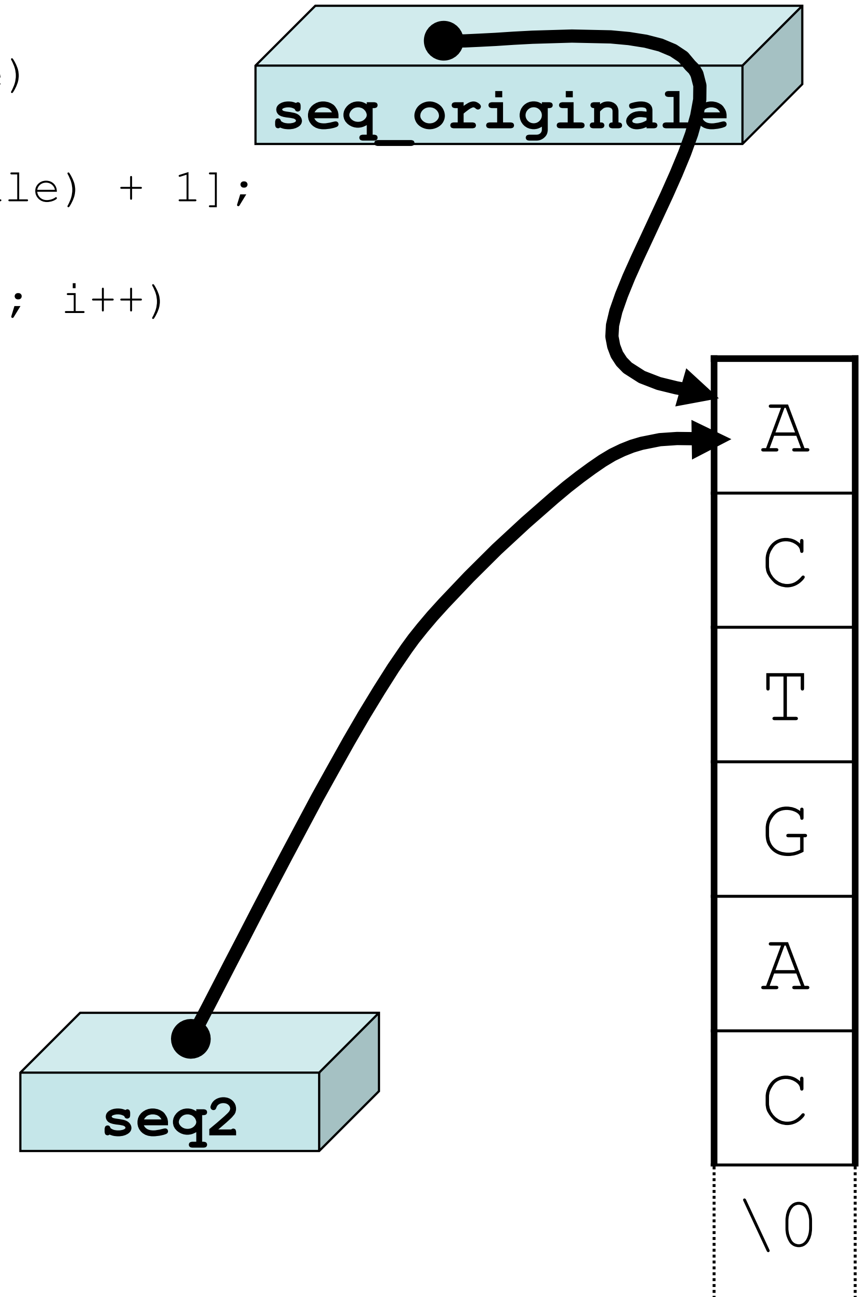
Appel de complementaire

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



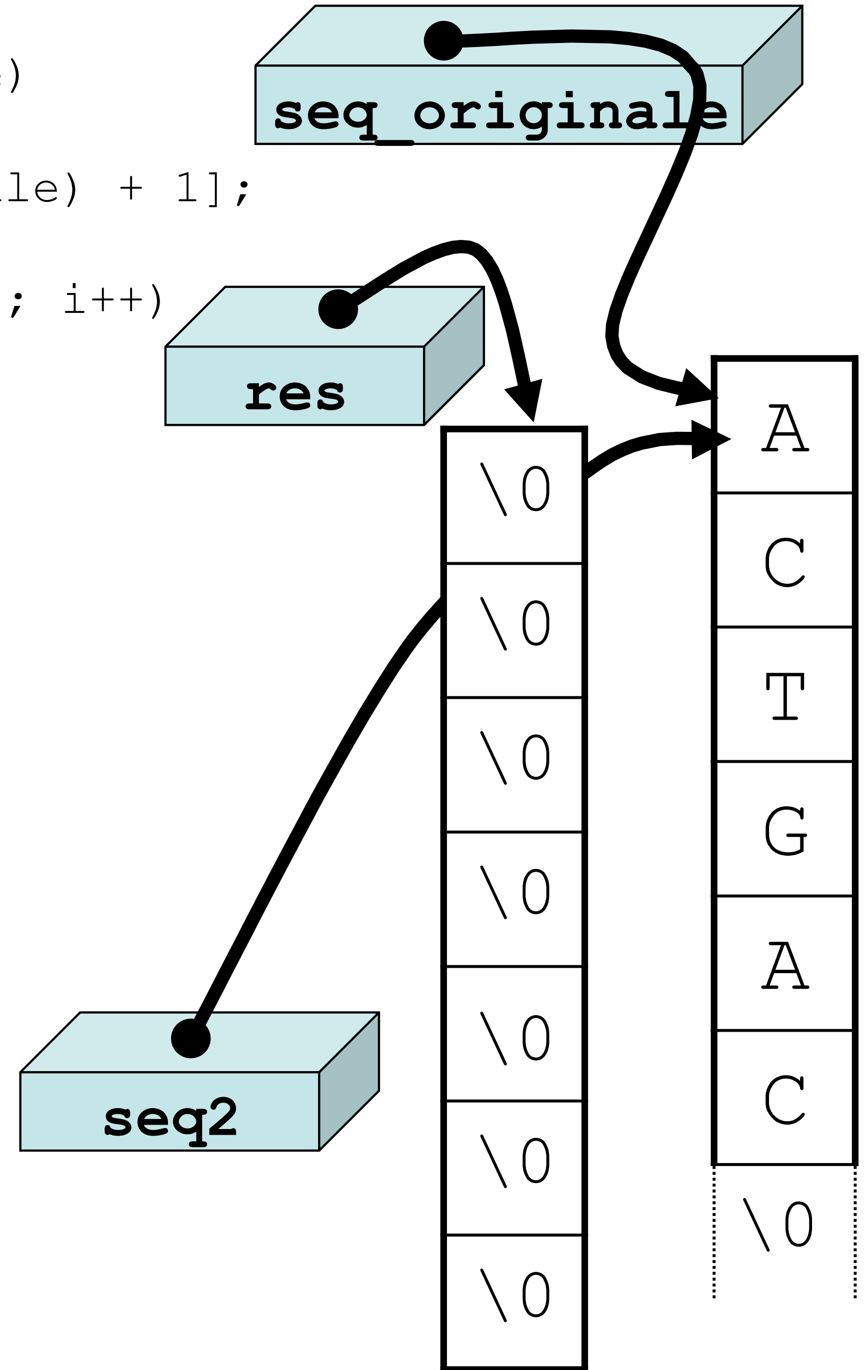
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



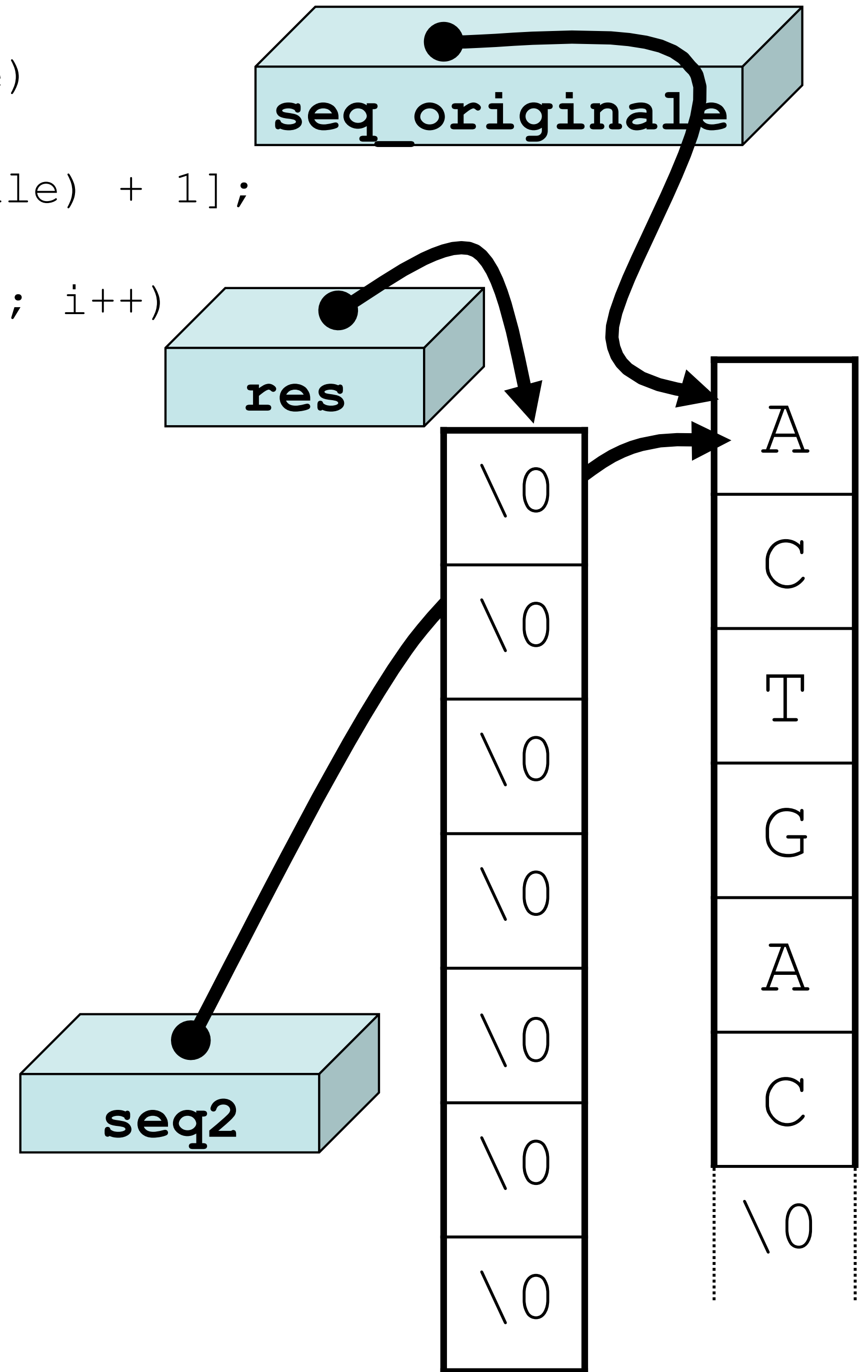
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



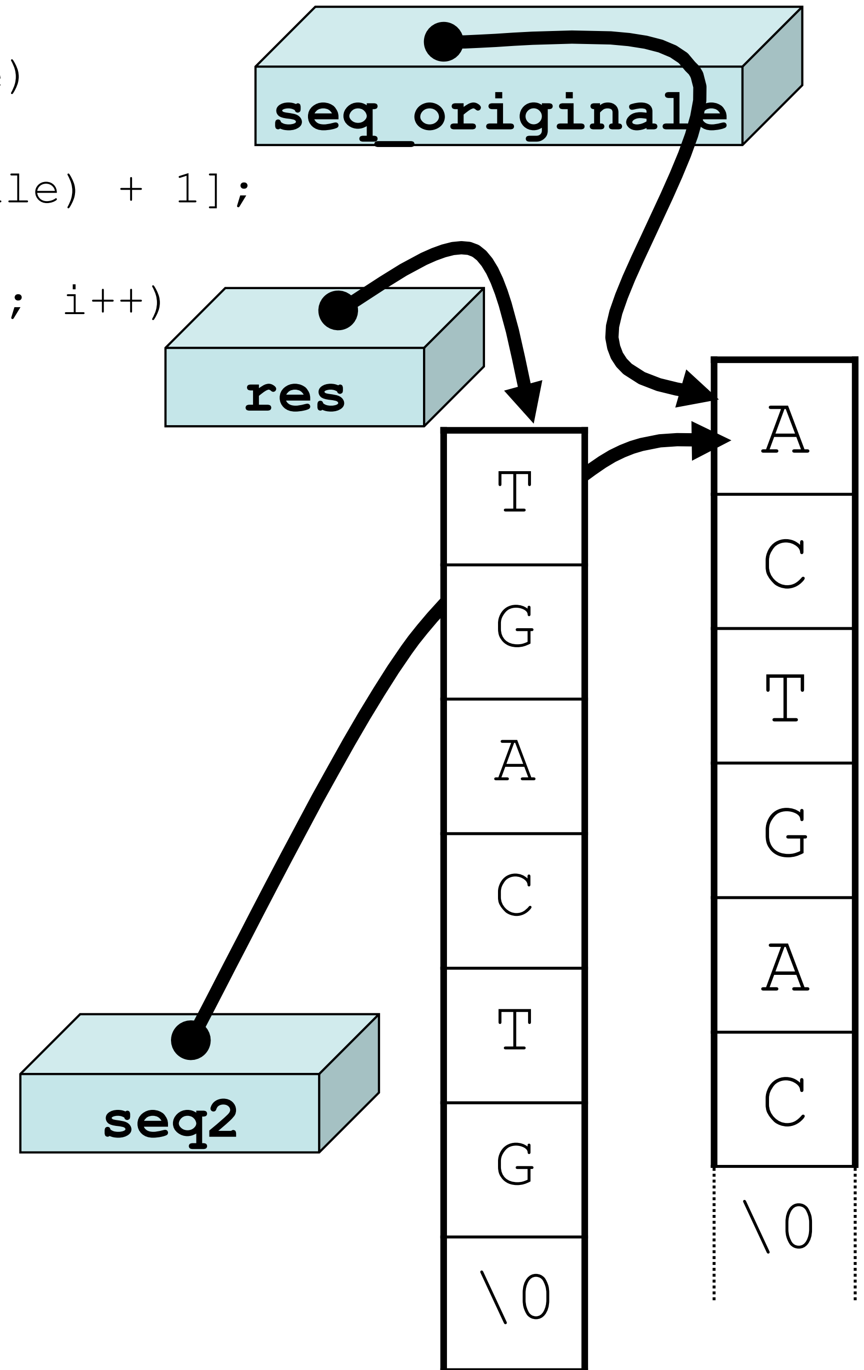
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



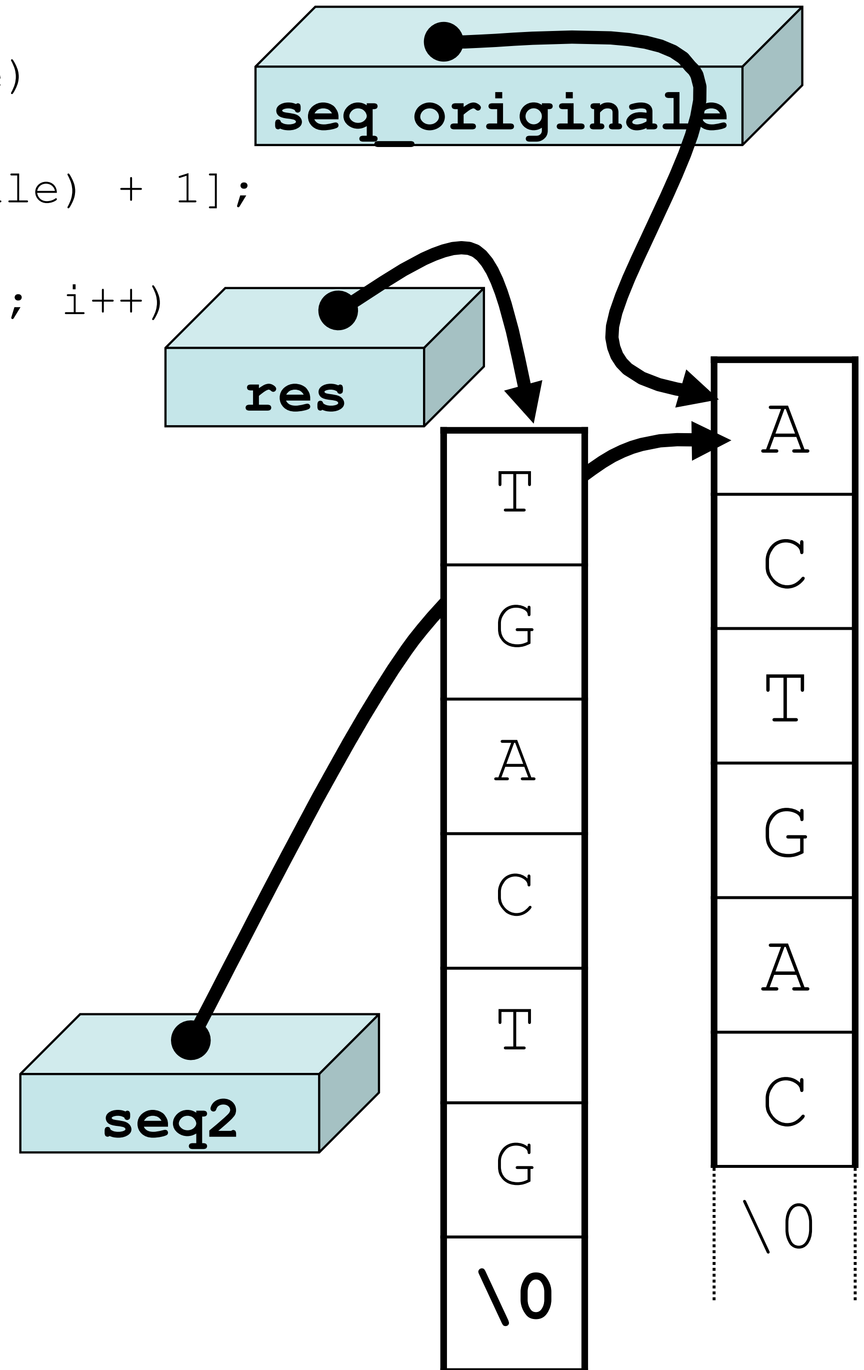
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



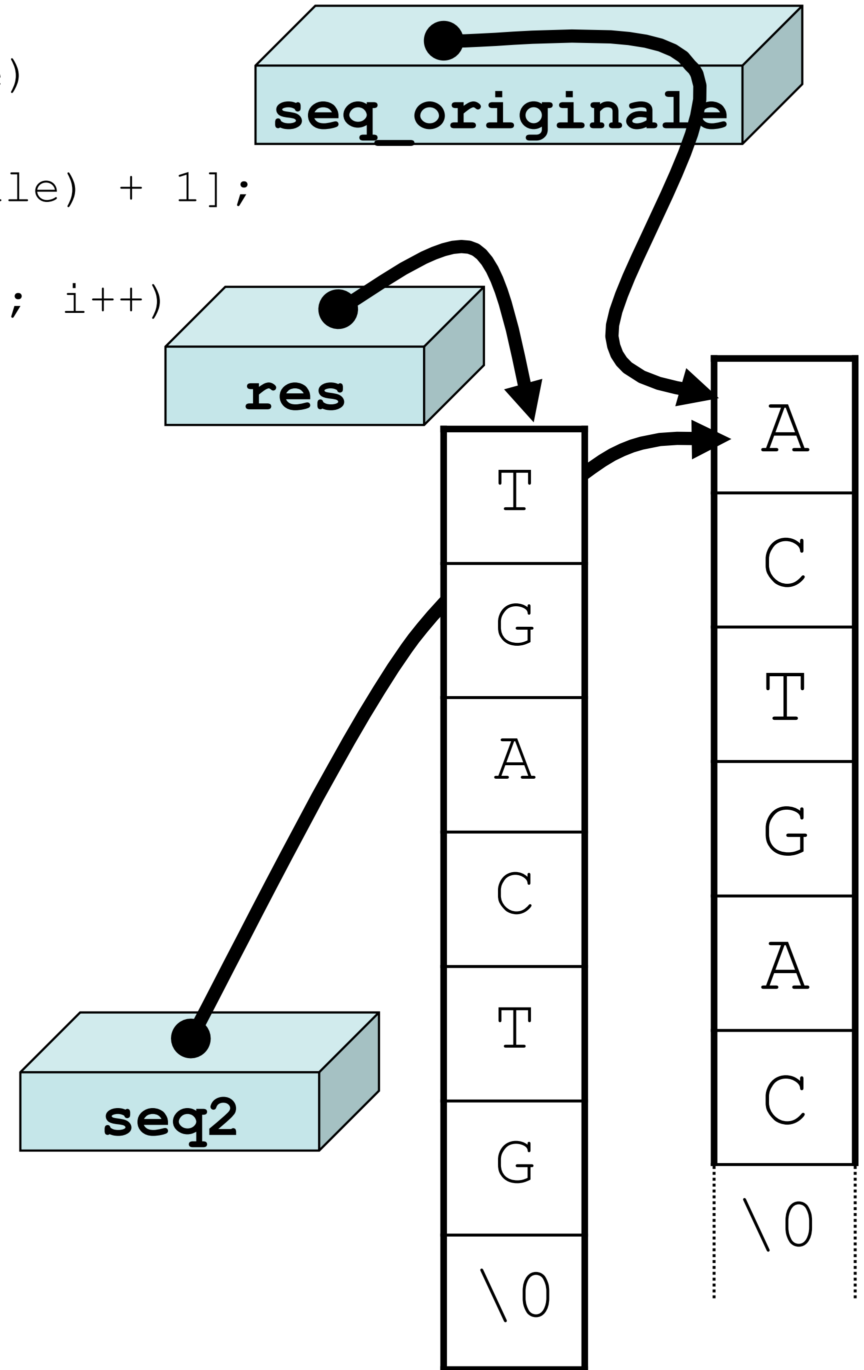
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
  char * seq3 = complementaire(seq2);
```



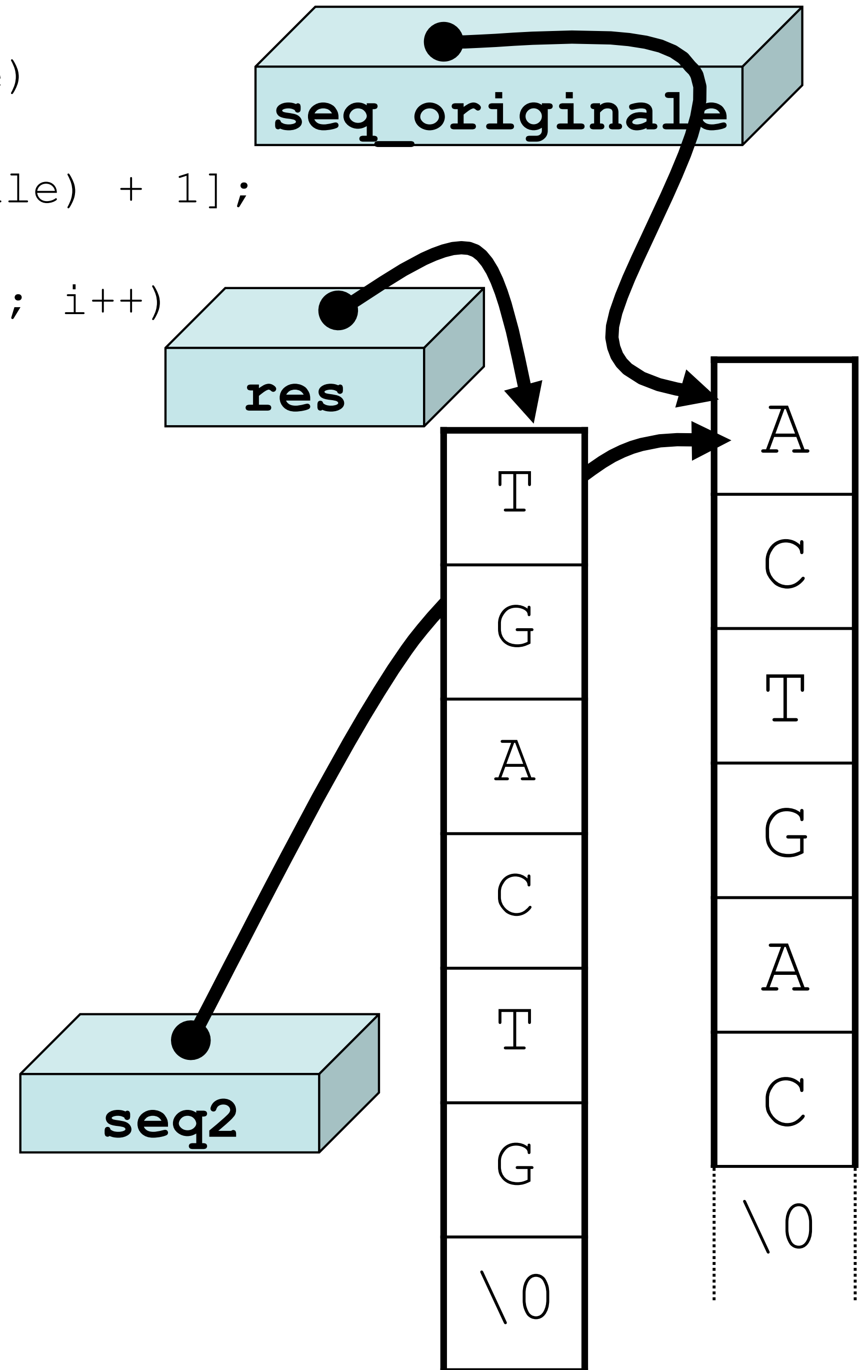
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
char * seq3 = complementaire(seq2);
```



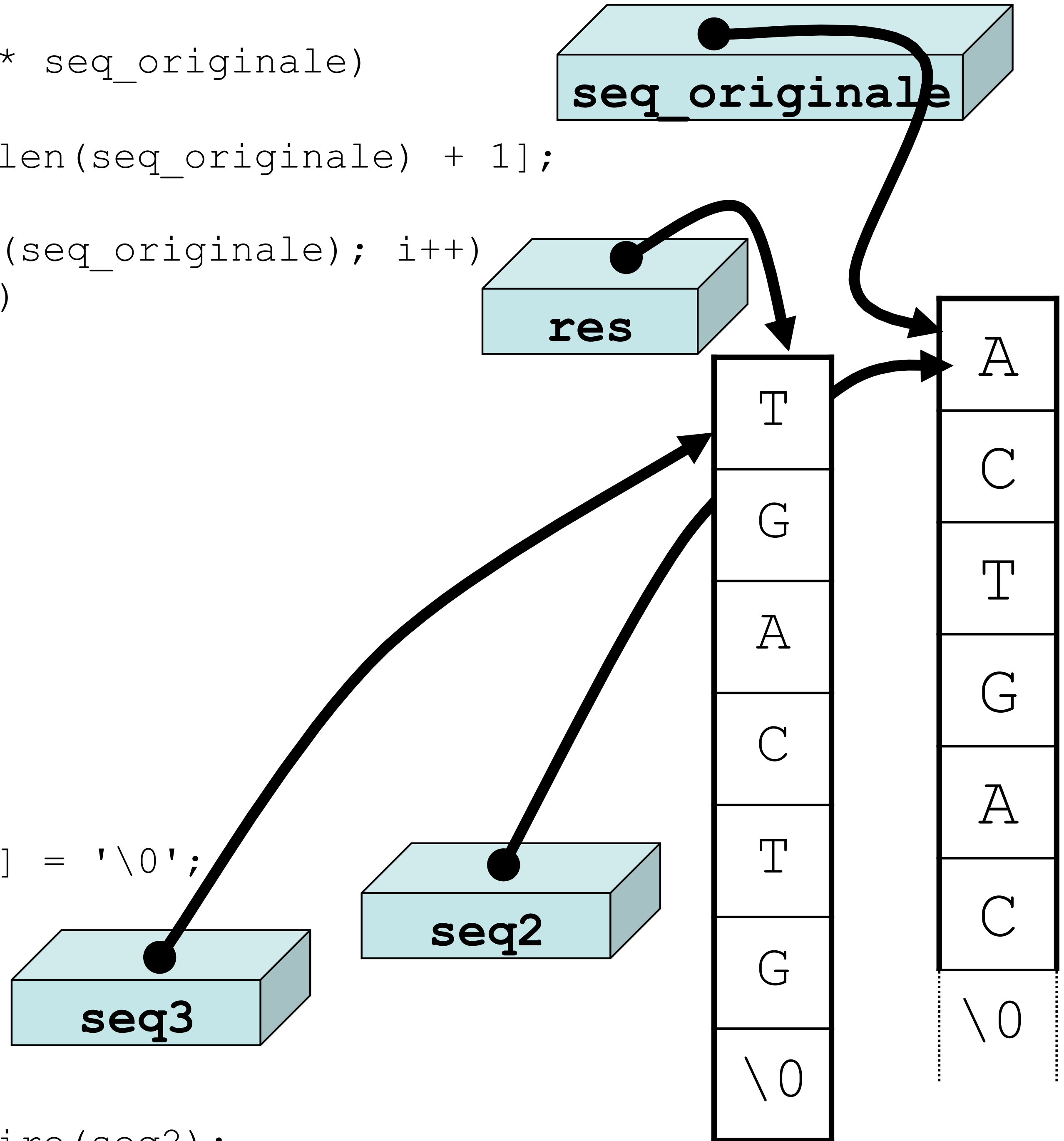
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
char * seq3 = complementaire(seq2);
```



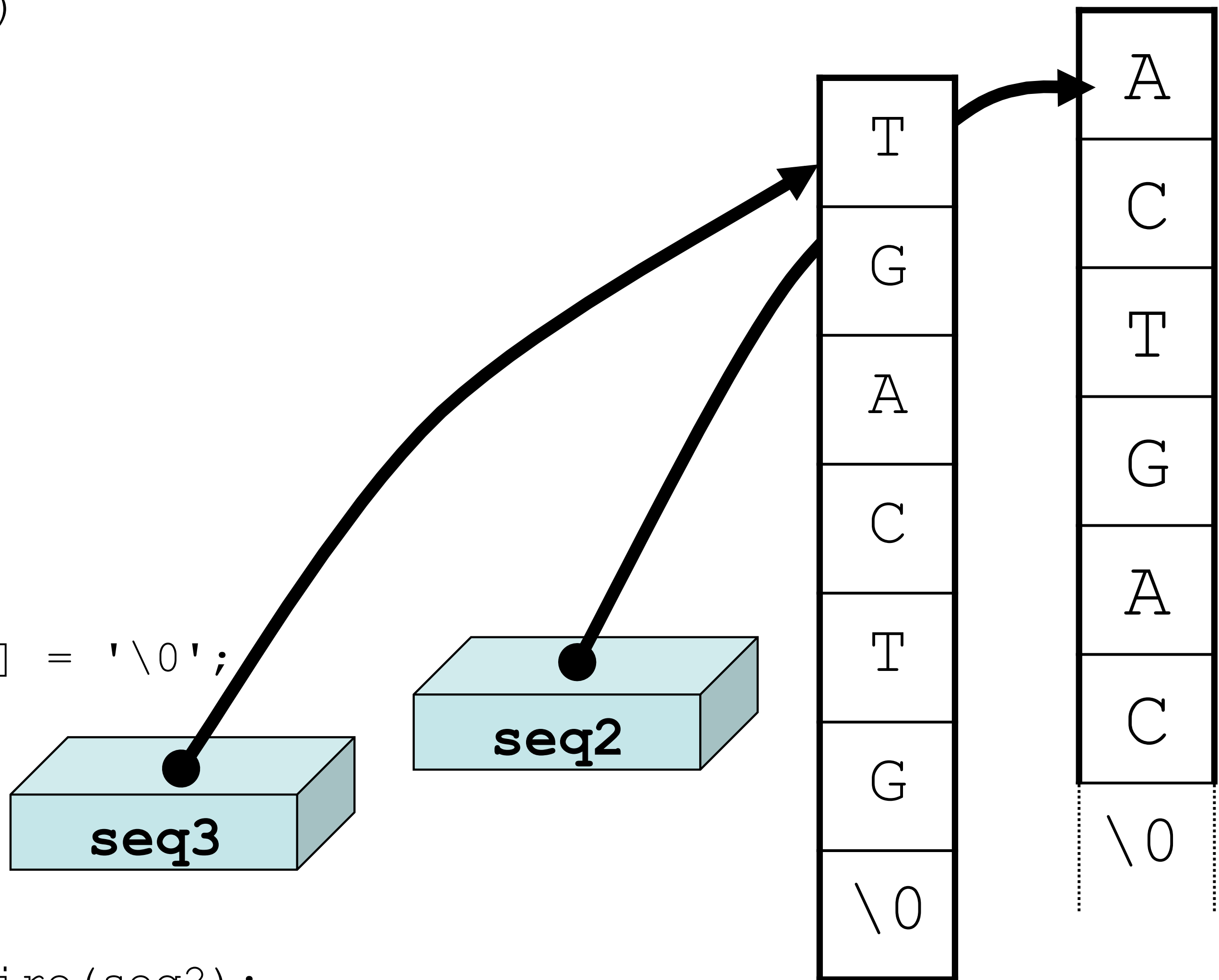
Appel de `complementaire`

```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
char * seq3 = complementaire(seq2);
```



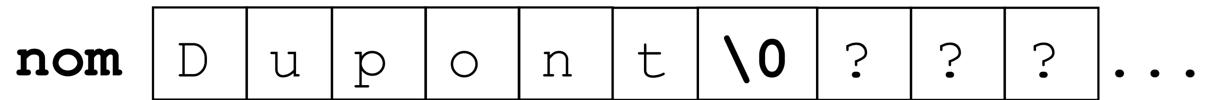
Appel de `complementaire`

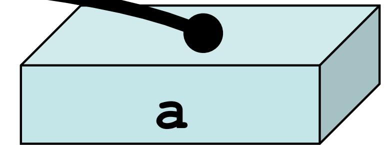
```
char * complementaire(char * seq_originale)
{
  char * res = new char[strlen(seq_originale) + 1];

  for(int i = 0; i < strlen(seq_originale); i++)
    switch(seq_originale[i])
    {
    case 'A':
      res[i] = 'T';
      break;
    case 'T':
      res[i] = 'A';
      break;

    (etc...)
    }

  res[strlen(seq_originale)] = '\0';

  return res;
}
(...)
APPEL:
char * seq3 = complementaire(seq2);
```



Comprendre la ligne

```
int main(int argc, char ** argv)
```





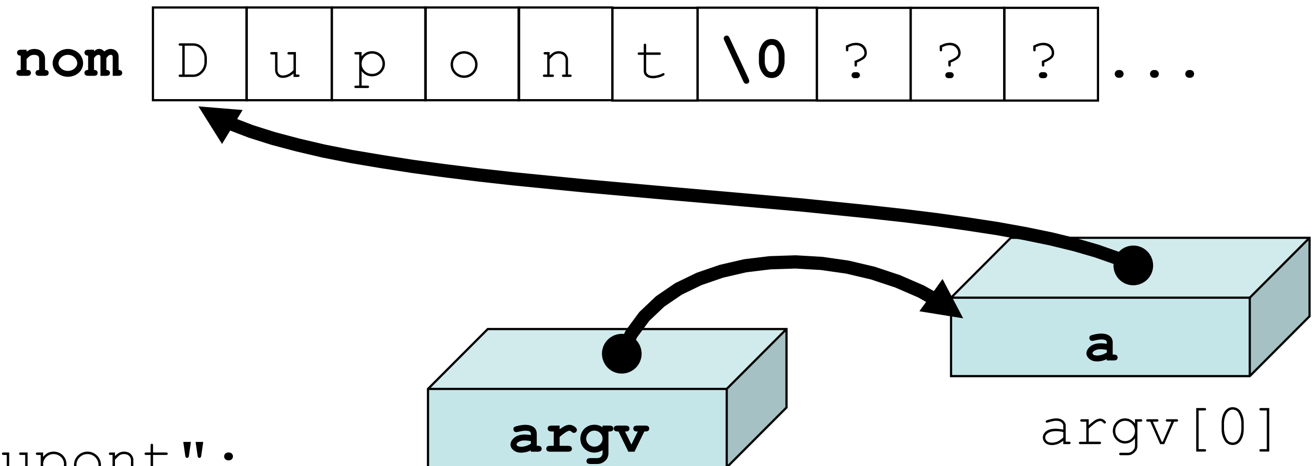
Si on fait:

```
char nom[30] = "Dupont";
char * a = nom;
```

`a` est un pointeur, et son type est `char *`.

```
cout << a << endl;
```

affiche `Dupont`




```
char nom[30] = "Dupont";
char * a = nom;
```

`a` est un pointeur, et son type est `char *`.

Mais `a` est une variable, et on peut définir un pointeur sur `a` en faisant:

```
char ** argv = &a;
```

`*argv` et `argv[0]` correspondent à `a`, donc à la chaîne `Dupont`:

```
cout << argv[0] << endl;
```

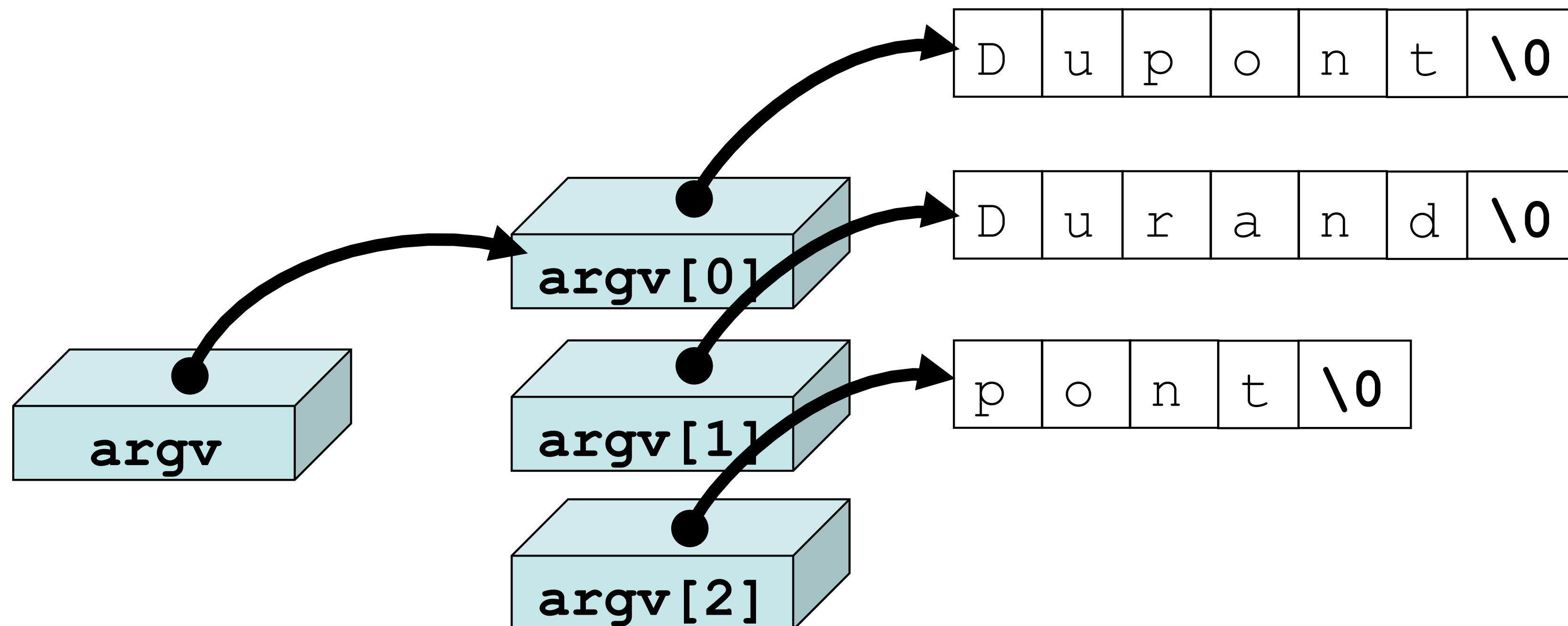
affiche `Dupont`

Supposons maintenant que l'on plusieurs pointeurs sur `char` (comme `a`) successifs en mémoire.

`cout << argv[0] << endl;` affiche `Dupont`

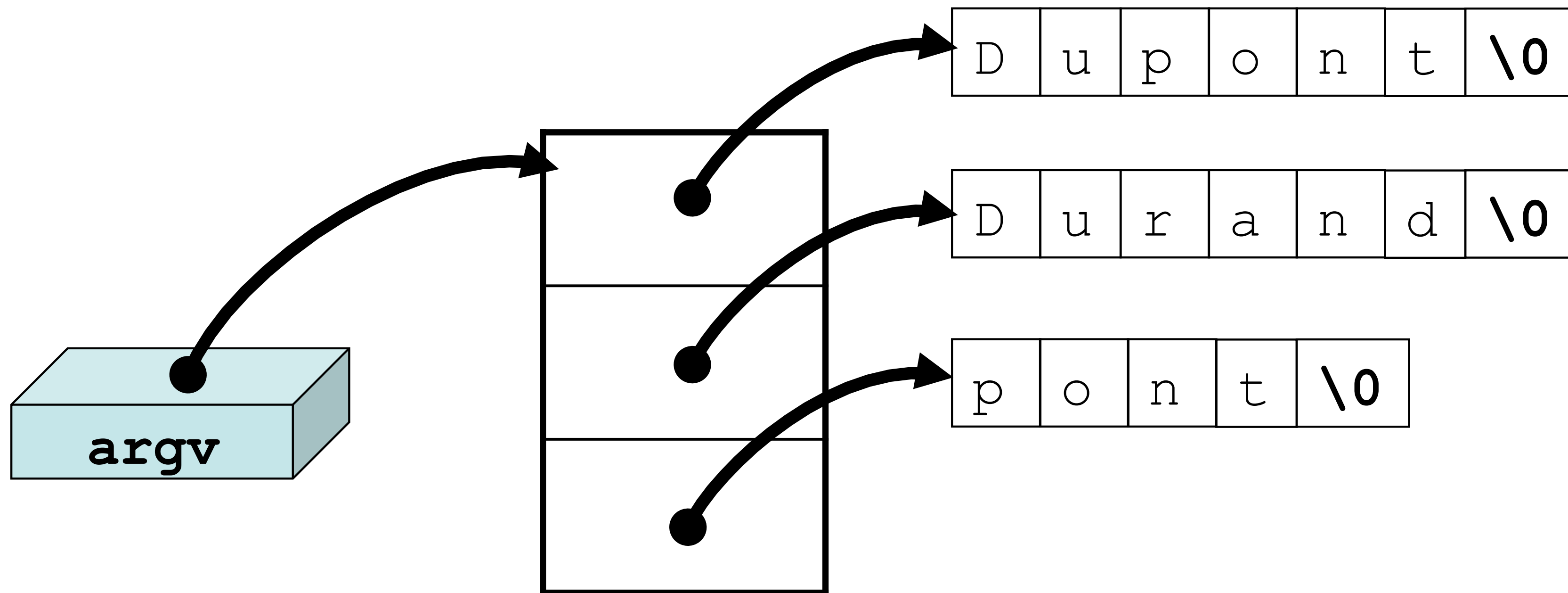
`cout << argv[1] << endl;` affiche `Durand`

`cout << argv[2] << endl;` affiche `pont`



`argv[0]`, `argv[1]`, et `argv[2]` sont des chaînes de caractères.

`argv` est un tableau de chaînes de caractères.



Dans le cas de la fonction `main`:

```
int main(int argc, char ** argv)
```

- `argv` est un tableau de chaînes de caractères;
- `argc` est la taille du tableau.

La première chaîne de caractères est le nom du programme exécutable.

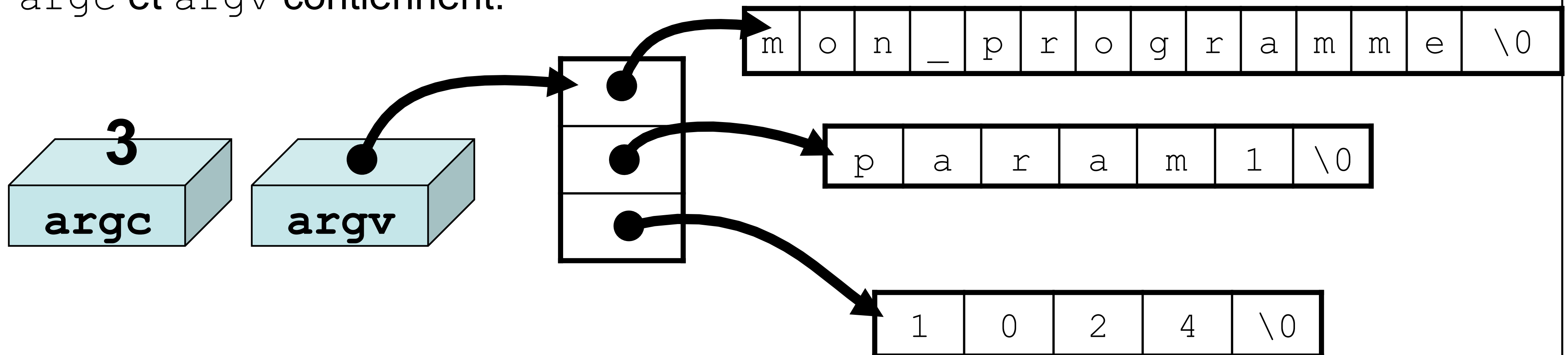
Les chaînes suivantes sont les paramètres entrés dans la fenêtre Terminal, après le nom du programme.

Par exemple, si on tape:

```
./mon_programme param1 1024
```

pour appeler le programme `mon_programme`,

`argc` et `argv` contiennent:



int main(int argc, char ** argv)

```
int main(int argc, char ** argv)
{
  cout << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << argv[i] << endl;

  return 0;
}
```

Si on appelle ce programme en tapant dans la fenêtre Terminal:

```
./mon_programme param1 1024
```

on obtient:

```
3
mon_programme
param1
1024
```

Entrées/Sorties pour les variables de type `char` **en C**

Affichage:

Le format d'affichage **`%c`** permet d'afficher une variable de type `char` (comme `%d` permet d'afficher une variable de type `int`). Ainsi, avec les instructions suivantes:

```
char c = 's';
printf("Le caractere est %c.\n", c);
```

on obtient:

```
Le caractere est s.
```

Lecture:

On peut utiliser la fonction `getchar()`:

```
char c1;
c1 = getchar();
```

lit un caractère au clavier et le range dans la variable `c1`.

Mais ***attention***, le mécanisme est un peu différent que pour la lecture d'entiers ou de flottants. Voir transparent suivant.

Les structures

Notion de structure de données

`int`, `float`, `char`... et les pointeurs sont des types ***scalaires***: ils permettent de stocker une valeur et une seule par variable.

Les tableaux sont une ***structure de données***.

Ils regroupent plusieurs valeurs de même type, chacune étant repérée par un indice.

En langage C, il existe une autre structure de données, qu'on nomme ***structure***.

Les structures permettent de regrouper plusieurs valeurs pouvant être de types différents.

Les ***éléments d'une structure*** sont appelés ***champs***, et sont repérés ***par leur nom***.

Les structures sont une version primitive des objets des langages objets, une notion très importante en programmation.

Déclaration d'une structure

Voici un exemple de déclaration de structure:

```
struct Chat
{
  float poids;
  char sexe; // 'M' ou 'F'
  int numero_tatouage;
};
```

Il s'agit d'un *nouveau type* appelé `Chat`:

Ce type `Chat` peut maintenant être utilisé pour déclarer des variables:

```
Chat felix, garfield;
```

→ Chacune des variables `felix` et `garfield` contiendra un flottant (`poids`), un caractère (`sexe`) et un entier (`numero_tatouage`);

Déclaration d'une structure

`struct` est un mot-clé indiquant la déclaration d'une structure

`Chat` est le nom de la structure.

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

`poids` est un des champs de la structure, de type `float`.

La liste des champs de la structure est délimitée par des accolades `{` et `}`.

L'accolade fermante `}` est suivie d'un point-virgule `;` (contrairement à l'accolade fermante du corps d'une fonction, d'une boucle ou d'un `if`).

Déclaration d'une structure

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

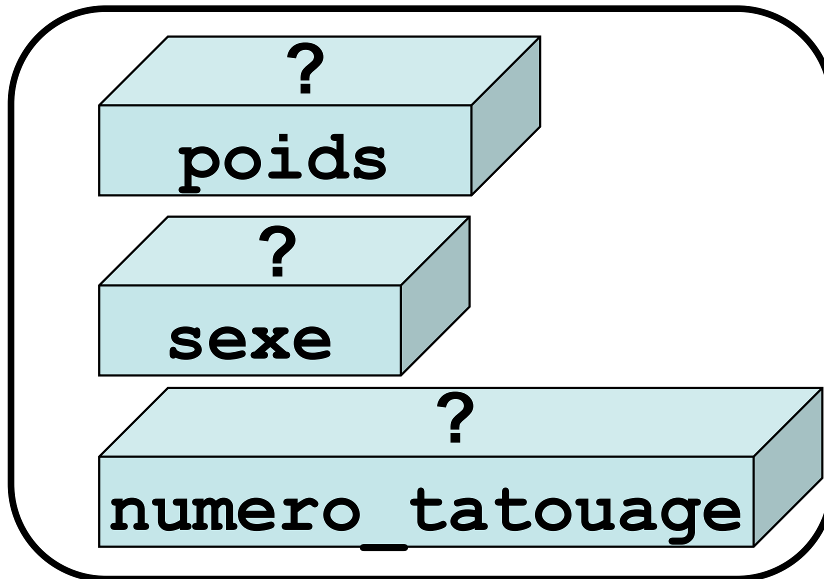
***Champs* de la structure**

Déclaration d'une variable de type structure

`Chat` est le nom de la structure.

`felix` est le nom de la variable.

```
Chat felix;
```



Exemple

```
#include <iostream>
using namespace std;

struct Chat
{
  float poids;
  char sexe; // 'M' ou 'F'
  int numero_tatouage;
};

int main(int argc, char ** argv)
{
  Chat felix;

  felix.numero_tatouage = 65328;
  felix.sexe = 'M';

  cout << "Entrez le poids du chat de tatouage "
       << felix.numero_tatouage << endl;
  cin >> felix.poids;

  cout << "Le chat " << felix.numero_tatouage
       << " pèse " << felix.poids << endl;

  return 0;
}
```

Déclaration de la structure `Chat`

Déclaration d'une variable `felix` de type `Chat`

Manipulation des champs de la variable `felix`

Utilisation des champs d'une structure

Chaque ***champ*** d'une structure peut être manipulé comme une variable:

```
felix.numero_tatouage = 65328;
felix.sexe = 'M';

cout << "Entrez le poids du chat de tatouage "
     << felix.numero_tatouage << endl;
cin >> felix.poids;

cout << "Le chat " << felix.numero_tatouage
     << " pèse " << felix.poids << endl;
```

Par exemple:

```
felix.numero_tatouage = 65328;
```

Un point `.` sépare le nom de la variable du nom du champ

`felix`: nom de la variable de type `Chat`

`numero_tatouage`: nom d'un champ de la structure `Chat`

`felix.numero_tatouage` se manipule alors comme toute variable de type `int`.

Utilisation globale d'une structure

Il est possible d'affecter à une variable de type structure le contenu d'une autre variable de même type.

Par exemple, si `chat` et `felix` sont de type `Chat`, on peut écrire:

```
chat = felix;
```

C'est la seule utilisation globale d'une structure autorisée par le C.

En particulier, la lecture au clavier d'une structure doit se faire champ par champ. **On ne peut pas faire:**

```
cin >> chat;
```

Il faut faire:

```
cin >> chat.poids;
cin >> chat.sexe;
cin >> chat.numero_tatouage;
```

Initialisation d'une structure lors de sa déclaration

Par exemple:

```
Chat felix = {2.5, 'M', 65328};
```

Les valeurs sont alors copiées champ par champ. Cet exemple revient donc à faire:

```
Chat felix;

felix.poids = 2.5;
felix.sexe = 'M';
felix.numero_tatouage = 65328;
```

Imbrication de structures de données

Les champs des structures que nous venons de voir étaient de type scalaire (ici `char`, `int` et `float`).

→ un champ peut être d'un type quelconque: *pointeur, tableau, structure…*

→ on peut aussi définir des tableaux de structures.

Structure comportant des tableaux

Soit les déclarations suivantes:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  int numero_tatouage;
};
Chat felix, garfield;
```

Les deux variables `felix` et `garfield` ont maintenant un champ supplémentaire: `nom`, un tableau de caractères (donc une chaîne de caractères).

La notation

```
  felix.nom[3]
```

désigne le quatrième caractère du `nom` de `felix`.

Tableaux de structures

Exemple de déclaration:

```
Chat elevage[50];
```

`elevage` est un tableau de 50 chats:

```
      elevage[5]
```

désigne le 6ème chat du tableau `elevage`.

```
      elevage[5].nom
```

désigne le nom du 6ème chat du tableau `elevage`.

Structures contenant d'autres structures

Exemple:

soit une structure permettant de représenter une date:

```
struct Date
{
  int jour, mois, annee;
};
```

→ on peut utiliser cette structure dans une autre structure:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
```

```
garfield.date_de_naissance.annee = 2005;
```

permet d'affecter 2005 à l'année de la date de naissance de `garfield`.

A quoi servent les structures ?

Elles simplifient la conception et l'écriture des programmes.

Elles permettent de représenter une entité complexe comme un chat, une personne, un produit, une date... et de faciliter sa manipulation en regroupant les données.

En C++, ce concept a été étendu pour donner les classes, qui regroupent à la fois les données et des fonctions pour manipuler ces données. C'est ce que vous verrez au deuxième semestre.

Exemples de structures

```
struct Date
{
  int jour, mois, annee;
};
```

```
struct Personne
{
  char nom[30], prenom[30];
  Date date_de_naissance;
};
```

```
struct Point
{
  float x, y;
};
```

```
struct Particule
{
  float position[3], vitesse[3];
  float masse, charge;
};
```

```
struct Etudiant
{
  char nom[100], prenom[100];
  char section[5];
  float note_bonus,
        note_examen,
        note_semestre;
};
```

```
struct Liste_Etudiants
{
  Etudiant * etudiants;
  int nb_etudiants;
  float moyenne_bonus,
        moyenne_examen,
        moyenne_semestre;
};
```

Exemple de l'intérêt des structures

Pour définir un ensemble de personnes avec un tableau de structures:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
Chat elevage[50];
```

A COMPARER avec la solution sans structure, avec un tableau par champ:

```
char nom[50][100];
float poids[50];
char sexe[50];
int jour_de_naissance[50], mois_de_naissance[50],
    annee_de_naissance[50];
```

etc...

qui est plus pénible à manipuler.

Exemple d'utilisation

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
Chat elevage[50];
(...)
for(int i = 0; i < 50; i++)
{
  cout << "Entrez le nom du chat " << i + 1 << endl;
  cin >> elevage[i].nom;
  cout << "Entrez son poids" << endl;
  cin >> elevage[i].poids;
}

for(int i = 0; i < 50; i++)
  cout << elevage[i].nom << " pèse " << elevage[i].poids << endl;
```

Structures et fonctions

Structure en paramètre d'une fonction

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
};

void affiche(Chat c)
{
  cout << c.nom << " est ";
  if (c.sexe == 'M')
    cout << "un male";
  else
    cout << "une femelle";
  cout << " et pèse " << c.poids << " kg." << endl;
}

int main(int argc, char ** argv)
{
  Chat chat1;
```

(...on remplit les champs de `chat1`*...)*

```
  affiche(chat1);

  return 0;
}
```

Structure en résultat d’une fonction

Une fonction peut retourner une valeur de type structure. Exemple:

```
Chat nouveau_male(char * nom, float poids)
{
  Chat ch;

  strcpy(ch.nom, nom);
  ch.sexe = ‘M’;
  ch.poids = poids;

  return ch;
}

...

Chat felix;
felix = nouveau_male("Félix", 0.5);
affiche(felix);
```

→ `Felix est un mâle et pèse 0.5 kg.`

Passage par valeur d'une structure

Le passage d'une structure par valeur a deux inconvénients:

- comme avant, la fonction ne peut pas modifier les champs de la structure;

- le passage de paramètre est coûteux si la structure comporte beaucoup de données:

A l'appel de la fonction `affiche`:

```
void affiche(Chat c)
{
  ...
}

  affiche(chat1);
```

- le champ `nom` de `chat1` est copié dans le champ `nom` du paramètre `c`;
- le champ `sexe` de `chat1` est copié dans le champ `sexe` du paramètre `c`;
- le champ `poids` de `chat1` est copié dans le champ `poids` du paramètre `c`.

Passage par valeur d'une structure

En règle générale, il vaut mieux éviter le passage d'une structure par valeur.

Nous verrons dans le cours suivant les pointeurs de structures en paramètre.

Qu'affiche ce programme ?

```
struct A
{
  int n;
  float f;
};

struct B
{
  A x;
  float f;
};

A f(int m, float g)
{
  A z;
  z.n = m;
  z.f = g;
  return z;
}
```

```
...

int n = 1;
A x;
x.n = 2;
x.f = 3;
cout << n << " " << x.n << endl;

B y;
y.x = x;
x.f = 2;
cout << y.x.f << endl;

x = f(1, 2);
cout << x.n << " " << x.f << endl;
```

Déclaration d'une variable de type structure **en C**

En C, il fallait répéter le mot-clé `struct` pour déclarer une variable:

```
struct Chat felix;
```

au lieu de

```
Chat felix;
```

en C++