

## Introduction à la Programmation des Algorithmes

### 3.4. Langage C – Types et allocation dynamique

François Fleuret

<https://fleuret.org/11x001/>



**UNIVERSITÉ  
DE GENÈVE**

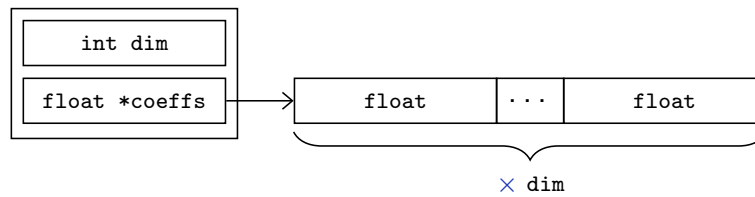
Comme nous l'avons vu, un pointeur ne fournit pas d'information sur la zone qu'il pointe, à part l'adresse du premier octet qui la compose.

La création de structures de données complexes se fait en combinant des pointeurs et des champs de données.

```

1 typedef struct {
2     int dim;
3     float *coeffs;
4 } vecteur;

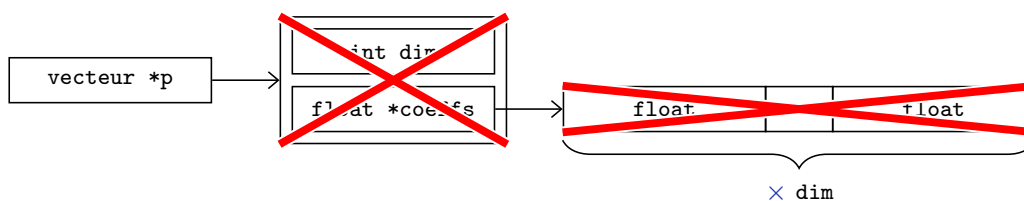
```



```

1 vecteur *cree_vecteur(int d) {
2     vecteur *p;
3     p = malloc(sizeof(vecteur));
4     p->dim = d;
5     p->coeffs = malloc(sizeof(float) * d);
6     return p;
7 }
8
9 void detruit_vecteur(vecteur *p) {
10    free(p->coeffs);
11    free(p);
12 }

```



```

1 void randomize_vecteur(vecteur *p) {
2     for(int k = 0; k < p->dim; k++)
3         p->coeffs[k] = 2 * (float) rand() / (float) RAND_MAX - 1;
4 }
5
6 float longueur_vecteur(vecteur *p) {
7     float s = 0;
8     for(int k = 0; k < p->dim; k++)
9         s += p->coeffs[k] * p->coeffs[k];
10    return sqrt(s);
11 }
12
13 int main(void) {
14     vecteur *p = cree_vecteur(100);
15     randomize_vecteur(p);
16     printf("longueur=%f\n", longueur_vecteur(v));
17     detruit_vecteur(p);
18     return 0;
19 }

```



L'opérateur d'affectation = se limite à faire une copie du pointeur sans dupliquer la variable pointée.

```

1 int main(void) {
2     vecteur *p, *q;
3     q = cree_vecteur(100);
4     p = q;
5     detruit_vecteur(p);
6     detruit_vecteur(q);
7     return 0;
8 }

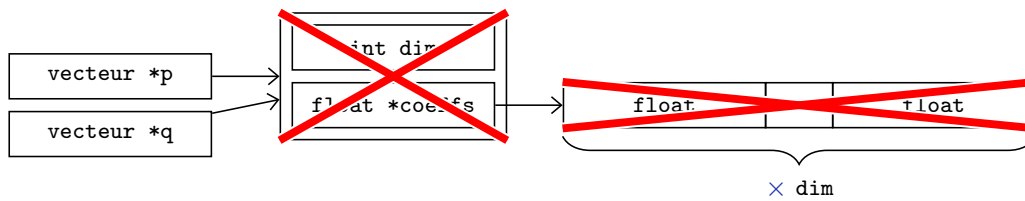
```

\*\*\* Error in `./a.out': double free or corruption

```

1  int main(void) {
2      vecteur *p, *q;
3      p = cree_vecteur(100);
4      q = p;
5      detruit_vecteur(p);
6      detruit_vecteur(q);
7      return 0;
8  }

```



Il faut programmer une **copie** d'un objet pointé.

```

1  vecteur *clone_vecteur(vecteur *p) {
2      vecteur *q = cree_vecteur(p->dim);
3      for(int k = 0; k < p->dim; k++)
4          q->coeffs[k] = p->coeffs[k];
5      return q;
6  }
7
8  int main(void) {
9      vecteur *p, *q;
10     q = cree_vecteur(100);
11     randomize_vecteur(q);
12     p = clone_vecteur(q);
13     detruit_vecteur(p);
14     detruit_vecteur(q);
15     return 0;
16 }

```

```

1  vecteur *somme_vecteur(vecteur *p, vecteur *q) {
2      if(p->dim != q->dim) abort();
3      vecteur *r = clone_vecteur(p);
4      for(int k = 0; k < p->dim; k++)
5          r->coeffs[k] += q->coeffs[k];
6      return r;
7  }
8
9  float produit_scalaire(vecteur *p, vecteur *q) {
10     if(p->dim != q->dim) abort();
11     float s = 0;
12     for(int k = 0; k < p->dim; k++)
13         s += p->coeffs[k] * q->coeffs[k];
14     return s;
15 }

```

```

1  int main(void) {
2      vecteur *p1, *p2, *p3;
3      p1 = cree_vecteur(100);
4      p2 = cree_vecteur(100);
5      randomize_vecteur(p1);
6      randomize_vecteur(p2);
7      p3 = somme_vecteur(p1, p2);
8      printf("%f %f\n",
9          produit_scalaire(p1, p3) + produit_scalaire(p2, p3),
10         produit_scalaire(p3, p3));
11     detruit_vecteur(p1);
12     detruit_vecteur(p2);
13     detruit_vecteur(p3);
14     return 0;
15 }

```

affiche

86.744965 86.744965

Contrairement à des langages plus récents, en particuliers ses extensions “orientées objets” C++ et Objective-C, le langage C ne permet pas facilement de gérer la destruction automatique de variables allouées dynamiquement.

Il n’y a pas de moyen simple de faire fonctionner ce code sans fuite de mémoire:

```
1  int main(void) {
2      vecteur *p1, *p2;
3      p1 = cree_vecteur(100);
4      p2 = cree_vecteur(100);
5      randomize_vecteur(p1);
6      randomize_vecteur(p2);
7      s = produit_scalaire(p1, somme_vecteur(p1, p2));
8      detruit_vecteur(p1);
9      detruit_vecteur(p2);
10     return 0;
11 }
```

Le vecteur créé par `somme_vecteur` ligne 7 n’est jamais libéré.

La manipulation dynamique de la mémoire permet même de créer des types dont la taille est dynamique.

Imaginons que l’on veuille récupérer des mesures  $X_0, X_1, \dots$  (températures, vitesses) sans savoir à l’avance combien il y en aura, et dont on voudra calculer la moyenne  $\hat{E}(X)$  et la variance  $\hat{V}(X)$ .

Avec les tableaux que nous avons vus jusqu’ici, nous devrions allouer une zone mémoire de la taille maximum possible.

Une alternative plus élégante consiste à agrandir dynamiquement le tableau quand c’est nécessaire.

```

1  typedef struct {
2      int nb, nb_max;
3      float *mesures;
4  } echantillon;
5
6  echantillon *cree_echantillon() {
7      echantillon *e = malloc(sizeof(echantillon));
8      e->nb = 0;
9      e->nb_max = 10;
10     e->mesures = malloc(sizeof(float) * e->nb_max);
11     return e;
12 }
13
14 void detruit_echantillon(echantillon *e) {
15     free(e->mesures);
16     free(e);
17 }

```

Le champ `nb_max` contient la taille du tableau `mesures`, et `nb` combien de valeurs ont déjà été stockées. La taille de `10` initiale ligne 9 est un choix arbitraire.

```

1  void ajoute_mesure(echantillon *e, float x) {
2      if(e->nb == e->nb_max) {
3          float *m = malloc(sizeof(float) * 2 * e->nb_max);
4          for(int k = 0; k < e->nb; k++) m[k] = e->mesures[k];
5          free(e->mesures);
6          e->mesures = m;
7          e->nb_max *= 2;
8      }
9      e->mesures[e->nb++] = x;
10 }

```

On teste si le tableau est plein ligne 2, si c'est le cas on alloue un tableau deux fois plus grands ligne 3, on y copie les valeurs stockées lignes 4, libère l'ancien tableau ligne 5, et met à jour les champs lignes 6–7.

Dans tous les cas on stock la nouvelle valeur ligne 9.

$$\hat{E}(X) = \frac{1}{N} \sum_{k=0}^{K-1} X_k$$

```

1  float moyenne(echantillon *e) {
2      if(e->nb == 0) abort();
3      float s = 0;
4      for(int k = 0; k < e->nb; k++)
5          s += e->mesures[k];
6      return s / e->nb;
7  }

```

$$\hat{V}(X) = \frac{1}{N-1} \sum_{k=0}^{N-1} (X_k - \hat{E}(X))^2$$

```

1  float variance(echantillon *e) {
2      if(e->nb < 2) abort();
3      float s = 0, m = moyenne(e), d;
4      for(int k = 0; k < e->nb; k++) {
5          d = e->mesures[k] - m;
6          s += d * d;
7      }
8      return s / (e->nb - 1);
9  }

```



```
1  int main(void) {
2      echantillon *e = cree_echantillon();
3      for(int k = 0; k < 10000; k++) {
4          ajoute_mesure(e, (float) rand() / (float) RAND_MAX);
5      }
6      printf("%f %f\n", moyenne(e), variance(e));
7      detruit_echantillon(e);
8      return 0;
9  }
```

affiche

0.497132 0.083105

## Chaînes de caractères

La conversion entre un octet et un caractère se fait avec une table. La plus standard étant la table ASCII (“American Standard Code for Information Interchange”) qui remonte aux années 60s.

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL	8	BS	9	HT	10	LF	11	VT	12	FF	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB	24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'	40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G	72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g	104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w	120	x	121	y	122	z	123	{	124		125	}	126	~	127	DEL

Des conventions de codage sur plusieurs octets permettent de représenter d’autres alphabets avec des milliers de caractères. Le standard est Unicode.

La convention en langage C est qu’une chaîne de caractères est représentée comme une suite de caractères en mémoire, chacun occupant un octet, suivi du caractère de code zéro. Il indique la fin de la chaîne.

```

1  int longueur_chaine(char *s) {
2      int l = 0;
3      while(s[l] != '\0') l++;
4      return l;
5  }
```

```

1 char *clone_chaine(char *s) {
2     int l = longueur_chaine(s);
3     char *t = malloc(sizeof(char) * (l + 1));
4     for(int k = 0; k <= l; k++) t[k] = s[k];
5     return t;
6 }
7
8 int main(void) {
9     char *a = "abcdefghijklmnopqrstuvwxy";
10    printf("l=%d\n", longueur_chaine(a));
11    char *b = clone_chaine(a);
12    printf("b=\"%s\"\n", b);
13    free(b);
14    return 0;
15 }

```

affiche

l=26

b="abcdefghijklmnopqrstuvwxy"

```

1 char *sous_chaine(char *s, int debut, int fin) {
2     int l = longueur_chaine(s);
3     if(debut < 0 || debut > fin || fin > l) abort();
4     char *t = malloc(sizeof(char) * (fin - debut + 1));
5     for(int k = 0; k < fin - debut; k++) t[k] = s[debut + k];
6     t[fin - debut + 1] = '\0';
7     return t;
8 }
9
10 int main(void) {
11     char *a = "abcdefghijklmnopqrstuvwxy";
12     char *c = sous_chaine(a, 3, 5);
13     printf("c=\"%s\"\n", c);
14     free(c);
15     return 0;
16 }

```

affiche

c="de"

Nous verrons dans un prochain cours que la plupart des opérations standard de copies de zone mémoires, de remplissage avec des constantes, et de manipulation de chaînes de caractères existent déjà dans les bibliothèques standard.

Il est [généralement] difficile d'écrire une version plus efficace et robuste que celle écrite par des programmeurs aguerris, et spécifiquement adaptée à un processeur / système.