Encore des pointeurs

Prototype de fonction

Quelle différence entre

```
void toto(int* t);
void toto(int t[]);
void toto(int t[10]);
```

Prototype de fonction

Quelle différence entre

```
void toto(int* t);
void toto(int t[]);
void toto(int t[10]);
```

Au final, les 3 fonctions manipulent des pointeurs.

Prototype de fonction

Quelle différence entre

```
void toto(int* t);
void toto(int t[]);
void toto(int t[10]);
```

Au final, les 3 fonctions manipulent des pointeurs.

La dernière fonction requiert nécessairement un tableau (non dynamique) de 10 entiers.

Tableau multidimensionnel

```
int t[10][20];
```

- le compilateur alloue une zone mémoire permettant de stocker de manière contiguë 10 tableaux de 20 entiers,
- toute référence à `t` est convertie en pointeur de tableaux de 20 `int`.

Tableau multidimensionnel

```
int t[10][20];
```

- le compilateur alloue une zone mémoire permettant de stocker de manière contiguë 10 tableaux de 20 entiers,
- toute référence à `t` est convertie en pointeur de tableaux de 20 `int`.

Dans le passage d'un tableau multi-dimensionnel à une fonction, il est nécessaire d'indiquer la taille des dimensions à l'exception de la première.
L'adresse de `t[i][j]` est
`(adresse de t) + (i*20 + j) * sizeof(int).`

```
void fonction(int tab[][20]){
    ...
}
```

Tableau dynamique à 2 dimensions

```
int largeur = 7;
int hauteur = 3;

int** tableau = malloc(hauteur * sizeof(int*))
for(i=0; i<hauteur; i++)
    tableau[i] = malloc(largeur * sizeof(int));
for(i=0; i<hauteur; i++)
    for(j=0; j<largeur; j++)
        tableau[i][j] = 0;

/***** libération mémoire *****/
for(i=0; i<hauteur; i++)
    free(tableau[i]);
free(tableau);
```

Pointeur de tableau

```
int main(void)
{
    int t[10][4];
    int (*p)[4] = t; /* p pointe sur t[0]. *p <=> t[0]. */

    (*p)[0] = 0; /* t[0][0] = 0 */
    (*p)[1] = 0; /* t[0][1] = 0 */

    (*(p + 1))[0] = 1; /* t[1][0] = 1 */
    (*(p + 1))[1] = 1; /* t[1][1] = 1 */

    return 0;
}
```

Tableau de pointeurs

Utilisation : tableau à 2 dimensions où toutes les lignes n'ont pas la même taille.

- solution 1 : tableau de pointeurs de tableaux

```
int (*tab[3])[]
```

Tableau de pointeurs

Utilisation : tableau à 2 dimensions où toutes les lignes n'ont pas la même taille.

- solution 1 : tableau de pointeurs de tableaux

```
int (*tab[3])[]
```

- solution 2 (préférable) :

```
int ligne1[5];
int ligne2[2];
int ligne3[8];
int* tab[3] = {ligne1, ligne2, ligne3};
```

Tableau de pointeurs

Utilisation : tableau à 2 dimensions où toutes les lignes n'ont pas la même taille.

- solution 1 : tableau de pointeurs de tableaux

```
int (*tab[3])[]
```

- solution 2 (préférable) :

```
int ligne1[5];
int ligne2[2];
int ligne3[8];
int* tab[3] = {ligne1, ligne2, ligne3};
```

Tableau de pointeurs

Utilisation : tableau à 2 dimensions où toutes les lignes n'ont pas la même taille.

- solution 1 : tableau de pointeurs de tableaux

```
int (*tab[3])[]
```

- solution 2 (préférable) :

```
int ligne1[5];
int ligne2[2];
int ligne3[8];
int* tab[3] = {ligne1, ligne2, ligne3};
```

Affichage du tableau :

```
for (i = 0; i < 3; i++){
  for (j = 0; j < taille[i]; j++)
     printf("%d ", tab[i][j]);
  printf("\n");
}
```

Avec un tableau de pointeurs sur des tableaux :

```
int ligne1[5];
int ligne2[2];
int ligne3[8];
int (*tab[3])[] = ???;

for (i = 0; i < 3; i++){
  for (j = 0; j < taille[i]; j++)
    printf("%d ", ???);
  printf("\n");
}
```

Avec un tableau de pointeurs sur des tableaux :

```
int ligne1[5];
int ligne2[2];
int ligne3[8];
int (*tab[3])[] = {&ligne1, &ligne2, &ligne3};

for (i = 0; i < 3; i++){
  for (j = 0; j < taille[i]; j++)
    printf("%d ", (*tab[i])[j]);
  printf("\n");
}
```

Un autre outil pour Debugger : Valgrind

Valgrind est un logiciel qui permet entre autres :

- vérifier les accès en lecture/écriture ;
- contrôler les fuites mémoires ;
- vérifier que l'on n'utilise aucune variable non initialisée.

Utilisé pour les langages C et C++ (ne fonctionne pas sous windows).

Utilisation de Valgrind

À la compilation, ajouter l'option `-g` pour afficher les lignes où surviennent les erreurs.

Utilisation de Valgrind

À la compilation, ajouter l'option `-g` pour afficher les lignes où surviennent les erreurs.

Éxécuter le programme dans Valgrind :

```
$ valgrind ./programme
```

Erreur d'écriture (1)

```
#include<stdlib.h>

int main(void)
{
    int *p = NULL;
    *p = 0;
    return 0;
}
```

Erreur d'écriture (2)

```
#include<stdlib.h>

int main(void)
{
  int* p = malloc(3 * sizeof(int));
  if (p != NULL)
    {
      p[3] = 0;
    }
  return 0;
}
```

Erreur d'écriture (3)

```
#include<stdlib.h>

int main(void)
{
  int *p = malloc(sizeof(int));
  if (p != NULL)
    {
        free(p);
        *p = 1;
    }
  return 0;
}
```

Fuite mémoire

```
#include<stdlib.h>

int main(void)
{
  int *p = malloc(sizeof(int));
  if (p != NULL)
    free(p);
  p = malloc(sizeof(int));
  p = NULL;
  return 0;
}
```

Valeur non initialisée

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

int main(void)
{
  int* i = malloc(sizeof(int));
  if(*i != 0)
    {
      printf("Hello, world !\n");
    }
  return 0;
}
```

Valgrind pour profiler son code

L'utilisation de l'option `--tool=callgrind` génère un fichier `callgrind.out.pid` à la fin de l'exécution qui contient des informations sur les temps d'exécutions de chaque section du programme.

Valgrind pour profiler son code

L'utilisation de l'option `--tool=callgrind` génère un fichier `callgrind.out.pid` à la fin de l'exécution qui contient des informations sur les temps d'exécutions de chaque section du programme.

Le logiciel `kcachegrind` permet de visualiser ces données.