Ch. 5 - Allocation

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

11 octobre 2021

Table des matières

Introduction

Les classes d'allocation

- En C / C++, il est possible d'allouer une variable de plusieurs manières
- Ces manières sont appelées *classes d'allocation*
- Il existe trois classes d'allocation
  1. Statique (variables globales et `static`)
  2. Automatique (variables locales)
  3. Dynamique (allocation avec `new` et `malloc`)
- Chaque classe définit
  - habituellement la zone mémoire où l'espace est alloué
  - la durée de vie de la mémoire allouée
  - une potentielle valeur par défaut

Allocation statique

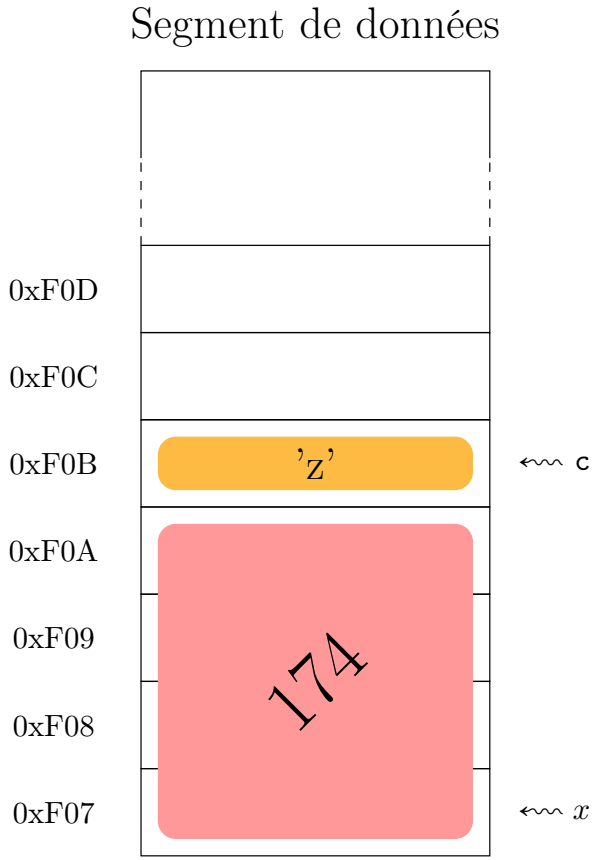
Classe d'allocation statique

- Variables globales et variables déclarées avec le mot-clé `static`
- Portée
  - globale si la variable est globale non `static`
  - locale à l'unité de traduction si la variable est globale `static`
  - locale à un bloc si la variable est déclarée au sein d'un bloc
- Durée de vie du programme
- Valeurs par défaut
  - Zéro pour les types numériques et `char` (+ tableaux)
  - Les objets doivent être instanciés
- Stockage
  - L'endroit où sont stockées les données est déterminé à la compilation
  - Le stockage des données est réalisé à la compilation
  - Très haute performance : allocation une fois avant l'exécution
  - Habituellement dans le segment de données

Illustration

```
{
   ...
   static int x = 174; //&x = 0xF07
   ...
}
...
char c = 'z';//global
```

`sizeof(int) = 4`



x et c sont détruits en fin de programme

Métaphore

- Enregistrer des données dans le segment de données est comme mettre des marchandises dans un gros sac sous vide
  - La taille du sac est exactement celle des marchandises stockées
    - Taille du segment = somme des tailles des données
  - Les marchandises sont les unes à côté des autres, il n'y a pas d'espace libre
    - Données contiguës en mémoire

Attention

- `static` en `Java` a donc une signification très différente en `C` / `C++`.
- Les « constantes » globales déclarées avec `#define` *ne* sont *pas* allouées

Exemple de bonne utilisation

- Fichier `static.c`
  - Même principe en `C++` : fichier `static.cpp`

```
1  const char * policy = "azertyuiopqsdfghjklmwxcvbn0123456789!?.";
2  //unsigned l = strlen(policy); //can't do
3
4  unsigned policy_length()
5  {
6      static unsigned length = 0;
7      static bool computed = false;
8
9      if(! computed)
10     {
11         length = strlen(policy);
12         computed = true;
13     }
14
15     return length;
16 }
```

- Les variables `length` et `computed` sont instanciées et initialisées une unique fois
- Ils sont réutilisés à chaque appel de `policy_length`

Avantages et inconvénients

Avantages

- Allocation très rapide
- Portée « illimitée »
- Durée de vie illimitée

Inconvénients

- Portée (semi) globale
  - Organisation de code
- Durée de vie illimitée

Remarques à propos du segment de données

- La mémoire est allouée tant qu'il reste de la place dans le segment de données
  - La taille maximale du segment de donnée est déterminée par le système (`ulimit -d`)
  - S'il n'y a plus de place, l'allocation est rejetée
- Le segment de donnée n'est pas « protégé » par la portée
  - Le programmeur est responsable de son utilisation

Hygiène de programmation

- Éviter les variables globales en C
  - En C++, les namespaces résolvent une partie des problèmes

Allocation automatique

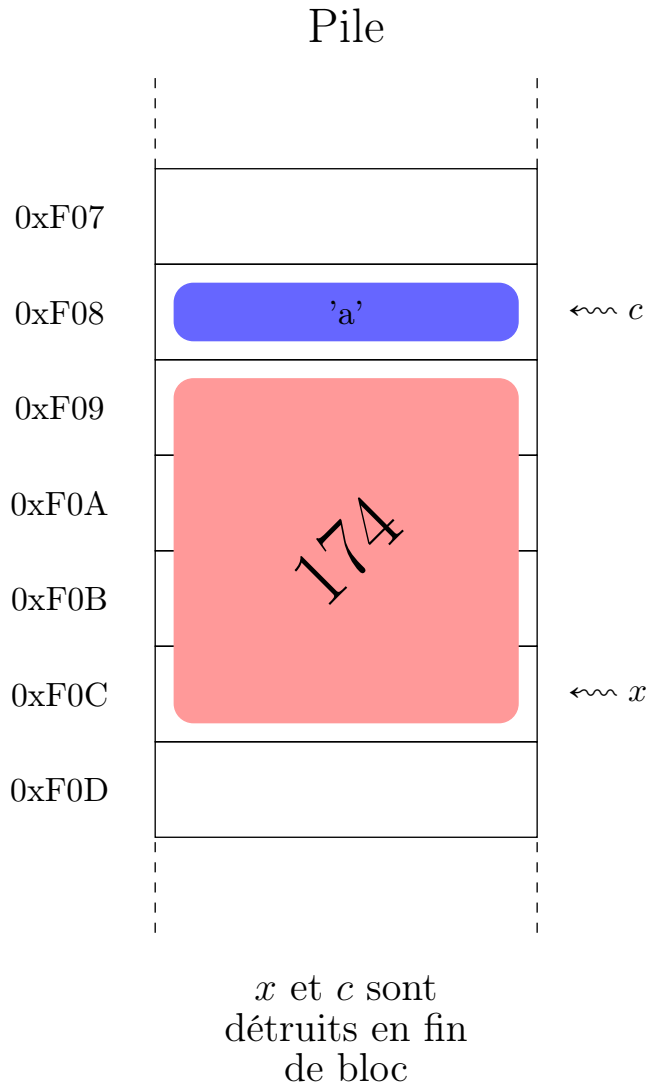
Classe d'allocation automatique

- Variables non globales déclarées sans mot-clé (ou avec `auto` en C++)
- Portée toujours locale
- Durée de vie
  - Mémoire allouée à la déclaration
  - Mémoire désallouée implicitement en fin de bloc
- Valeurs par défaut
  - Aucune (valeur indéterminée)
- Habituellement, les données sont stockées sur la pile
  - Stockées au sommet de la pile
    - Registre `rsp`, adresses décroissantes
  - Haute performance : on sait *à la compilation* où stocker
    - Pas de calcul ou d'appel système nécessaire à l'exécution
  - L'affectation des données est au pire réalisée à l'exécution

Illustration

```
{
    ...
    int x = 174; //&x = 0xF0C
    char c = 'a';
    ...
}
```

`sizeof(int) = 4`



x et c sont détruits en fin de bloc

Métaphore

- Enregistrer des données sur la pile est comme stocker des caisses au fond d'un puits
  - Pour stocker, on lâche la caisse qui tombe au fond du puits
    - Données enregistrées au sommet de la pile
  - On sait *a priori* où va la caisse
    - À la compilation, on sait où la donnée est enregistrée
    - Au sommet de la pile, relativement à `rsp`
  - Très rapide de stocker : on lâche la caisse
    - Aucun calcul n'est effectué à l'exécution
- En `Java` et en `C#`, seuls les primitifs et les adresses d'objets sont alloués sur la pile

Exemple

- Fichier `automatic.c`
  - Même principe en `C++` : fichier `automatic.cpp`

```
1  int main()
2  {
3    int i = 5; //most likely on stack
4    while(i >= 0)
5    {
6      int j = i + 2; //most likely on stack
7
8      printf("%d %d\n", i, j);
9
10     i--;
11   }
12   printf("%d\n", i);
13   //printf("%d\n", i); //j is out of the scope
14 }//i is destroyed
```

Exemple 2

- Fichier `automatic2.c`
  - Même principe en `C++` : fichier `automatic_2.cpp`

```
1  int explodeStack(int i)
2  {
3    int j = i * 2; //creates j most likely on the stack
4
5    if(i != 1)
6      explodeStack(j); //backup registers on stack and call (infinite recursion)
7  }
8
9  int main()
10 {
11   explodeStack(10);
12 }
```

Avantages et inconvénients

Avantages

- Allocation rapide
- Portée limitée
- Durée de vie limitée
- Allocation et désallocation implicite

Inconvénients

- Portée limitée
- Durée de vie limitée

Remarques à propos de la pile

- La mémoire est allouée tant qu'il reste de la place dans la pile
  - La taille maximale de la pile est déterminée par le système (`ulimit -s`)
  - S'il n'y a plus de place, l'allocation est rejetée
- La pile n'est pas protégée en écriture (pour un même programme)
  - Le programmeur est responsable de son utilisation
  - Une sortie de tableau peut corrompre son état
    - Variables corrompues
    - Pile d'exécution corrompue

Hygiène de programmation

- Utiliser au maximum en `C` et `C++`

Allocation dynamique

Classe d'allocation dynamique (1/2)

- Allocation avec `new` / `malloc` et destruction avec `delete` / `free`
  - Allouent un espace mémoire et retournent l'adresse vers l'espace alloué
  - L'adresse des données est en classe automatique
  - Les données sont en classe dynamique
- Portée
  - l'adresse allouée de la donnée est locale au bloc
  - les données sont accessibles globalement via leur adresse
- Les données sont habituellement stockées dans le tas
  - Ces données ne sont pas contiguës en mémoire
  - L'endroit où sont stockées les données est déterminé à l'exécution
  - Le stockage des données est réalisé à l'exécution
  - Performance plus faible : appel système est effectué (à l'exécution) pour déterminer où stocker les données

Classe d'allocation dynamique (2/2)

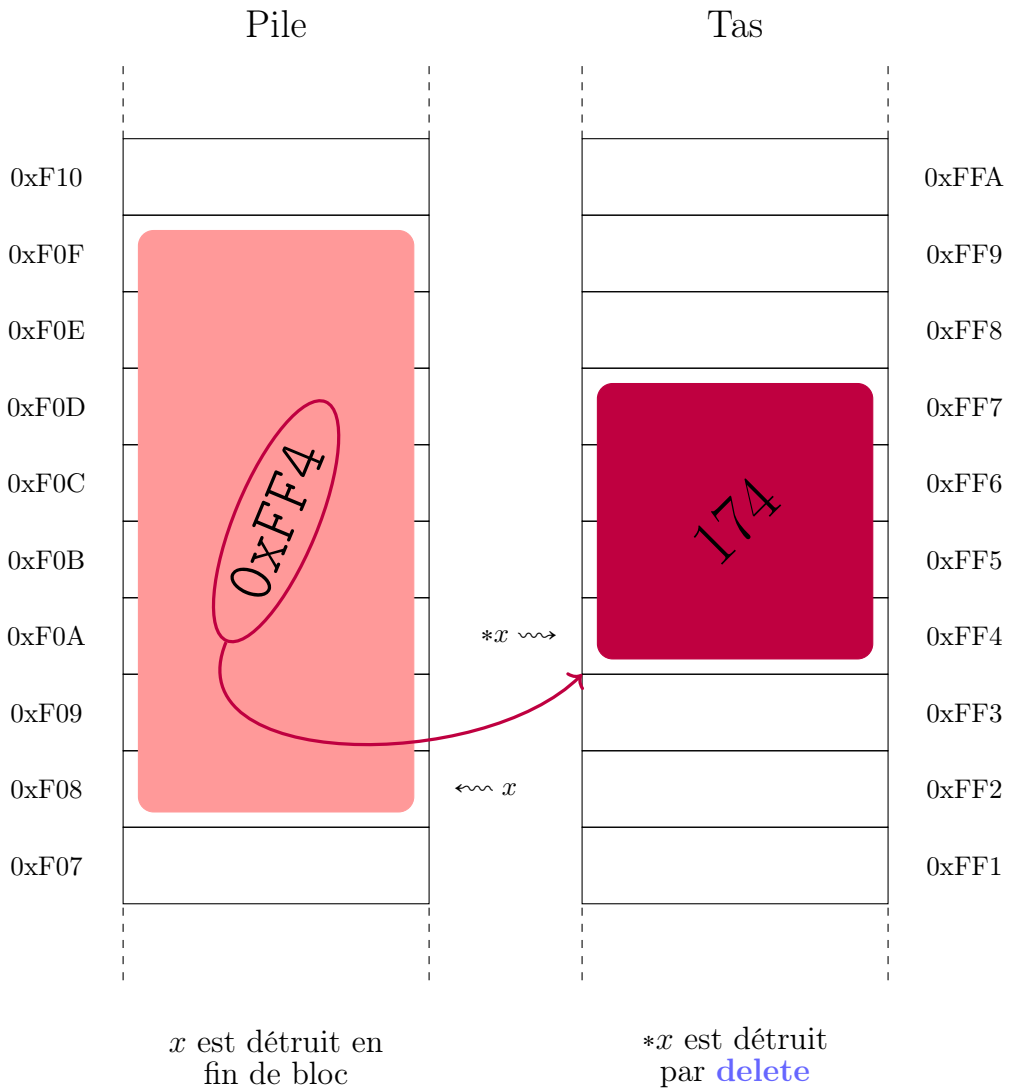
- Durée de vie
  - Mémoire allouée à l'instanciation par `new` / `malloc`
    - À la compilation, on sait où va l'adresse des données allouées
    - ... mais pas les données allouées
  - L'adresse des données est désallouée en sortie de bloc
    - Car elle est de classe automatique
  - Mémoire désallouée explicitement par `delete` / `free`
    - Si on ne le fait pas : fuite mémoire
    - Les ressources sont *perdues* (allouées mais inaccessibles)
    - Le système d'exploitation *devrait* désallouer en fin de programme
- Valeurs par défaut
  - Aucune : instanciation explicite nécessaire

Illustration

```
{
   ...
   int * x = (int*) malloc(sizeof(int));
   *x = 174;
   ...
}
```

```
{
   ...
   int * x = new int(174); //&x = 0xF08
                           //&*x = 0xFF4
   ...
}
```

```
sizeof(int) = 4
sizeof(int*) = 8 //x64
```



Métaphore

- Enregistrer des données dans le tas est comme stocker des marchandises dans un entrepôt
  1. Quand on veut stocker des marchandises, on demande au concierge si c'est possible
     - Appel système effectué pour l'allocation
  2. Le concierge cherche un endroit dans l'entrepôt où stocker
     - Le système d'exploitation cherche un espace mémoire libre
  3. S'il y a de la place, le concierge stocke la caisse et donne un ticket décrivant où elle est stockée
     - Le système d'exploitation alloue un espace mémoire et retourne l'adresse de cet espace
  4. S'il n'y a pas de place : ko
     - Cela ne veut pas dire que la mémoire est saturée
     - Elle est peut-être simplement fragmentée

Exemple C

- Fichier `dynamic.c`

```
1  int main()
2  {
3      Point * p = (Point*) malloc(sizeof(Point));
4      p->x = 1; p->y = 2;
5      print_point(*p);
6
7      free(p);
8      print_point(*p);
9
10     Point * p2 = NULL; Point * pp2 = NULL;
11     Point p3;
12
13     {
14         p2 = (Point*) malloc(sizeof(Point));
15         p2->x = 3; p2->y = 4;
16         print_point(*p2);
17         pp2 = p2;
18
19         p3 = *p2;
20         //free(p2); //uncomment
21     }
22
23     print_point(*p2);
24     print_point(*pp2);
25     print_point(p3);
26 }
```

Exemple C++

- Fichier `dynamic.cpp`

```
1  int main()
2  {
3    Point * p = new Point(2,3);
4    cout << p->getX() << " " << p->getY() << endl;
5    delete p;
6    cout << p->getX() << " " << p->getY() << endl;//unpredictable is p is deleted
7
8    Point * p2 = nullptr;
9    Point * pp2 = nullptr;
10   Point p3;//(0,0)
11
12   {
13     p2 = new Point(3,4);
14     cout << p2->getX() << " " << p2->getY() << endl;
15     pp2 = p2;
16     p3 = *p2;
17     //delete p2;//uncomment
18   }
19
20   cout << p2->getX() << " " << p2->getY() << endl;
21   cout << pp2->getX() << " " << pp2->getY() << endl;
22   cout << p3.getX() << " " << p3.getY() << endl;
23 }   //Q : does it leak ?
```

Fonction d'allocation dynamique en C

1. `malloc(size_t)` : alloue la mémoire sur le tas
   - Retourne l'adresse vers l'espace alloué
   - La mémoire a une valeur indéterminée
2. `calloc(size_t n, size_t m)` : alloue des « tableaux »
   - Mets la mémoire à zéro
   - À cause de l'alignement, la taille allouée n'est pas toujours *nm*
3. `realloc(void*, size_t)` : réalloue de la mémoire préalablement allouée par `malloc`, `calloc` ou `realloc`
   - Contracte ou étend un emplacement
   - Déplacement / copie possible

Remarque importante

- Avec `realloc`, si l'allocation échoue, l'ancien emplacement *n'est pas libéré*

- Nécessaire d'inclure `stdlib.h`

Illustration des fonctions d'allocation en C (1/2)

- Fichier `fct-alloc-c.c`

```c
void print_int_array(int * array, int size)
{
    for(int i = 0; i < size; i++)
        printf("%d ", array[i]);
    printf("\n");
}

int main()
{
    int * p = (int*) malloc(4 * sizeof(int));
    print_int_array(p, 4); //undeterminate values

    free(p);
    print_int_array(p, 4); //undefined behaviour

    p = (int*) calloc(4, sizeof(int));
    print_int_array(p, 4); //0 0 0 0

    for(int i = 0; i < 4; i++)
        p[i] = i;
    print_int_array(p, 4); //0 1 2 3
}
```

Illustration des fonctions d'allocation en C (2/2)

- Fichier `fct-alloc-c.c`

```c
int main()
{
    int * p = (int*) malloc(4 * sizeof(int));

    p = (int*) realloc(p, 2 * sizeof(int));
    if(p) //check wether allocation suceeded
    {
        print_int_array(p, 2); //0 1
        print_int_array(p, 4); //0 1 2 3 : not undeterminate
    }
    else
        free(p);

    p = (int*) realloc(p, 4 * sizeof(int));
    if(p)
    {
        print_int_array(p, 2); //0 1
        print_int_array(p, 4); //0 1 ? ?
    }
    else
        free(p);
}
```

Exemple

- Fichier `paginate.cpp`

```
1  int main()
2  {
3    long long unsigned int j = 0;
4    while(true)//this is going to hurt
5    {
6      new int[250];//should weight 1kb
7      j++;
8      if(j % 100 == 0)
9        cout << j << "kb allocated" << endl;
10   }
11 }
```

- On peut suivre l'évolution de la mémoire sur le moniteur système
- Sous Linux, le mécanisme de pagination est utilisé
  - On peut voir la mémoire swap prendre le relai

Rappel sur les pointeurs

- Les pointeurs *sont* des adresses
- Ces adresses peuvent correspondre à un espace alloué, ou non
- Accéder à un espace qui n'a pas été alloué amène à un comportement indéterminé
  - Zéro
  - Ancienne valeur
  - Erreur de segmentation
- Affecter une valeur à un pointeur change la valeur de l'adresse
  - Pas ce que pointe l'adresse
- Pour changer ce qui est pointé, il faut déférencer
  - `*pt = 42;`

Exemple

- `int * pt = 3;`
  1. Alloue un espace de 8 bytes (x64) sur la pile
  2. Donne à cet espace la valeur `0x00_00_00_00_00_00_00_03`
  3. L'instruction `int i = *pt;` provoque probablement une erreur de segmentation
- `int * pt = NULL;` et `int * pt = nullptr` se comportent de la même manière
  - Erreurs de segmentation au déférencement
- `int * pt = (int*)malloc(sizeof(int)); *pt = 3;`
  1. Alloue un espace `pt` de 8 bytes (x64) sur la pile
  2. Alloue un espace *s* de `sizeof(int)` sur le tas
  3. Affecte à `pt` l'adresse de *s*
  4. Affecte à l'adresse pointée par `pt` (dans *s*) la valeur 3
- `int * pt = new int(3);` est l'équivalent en `C++`

Avantages et inconvénients

Avantages

- Portée relativement illimitée
- Durée de vie illimitée
- Émulation de passage par référence
- Possibilité d'allouer de larges quantité de mémoire

Inconvénients

- Allocation plus lente
- Destruction manuelle nécessaire
  - Attention aux fuites mémoires et double free
- Autres contraintes (copie, affectation)

Remarques

- La mémoire est allouée tant qu'il reste de la place en mémoire
  - La taille maximale du tas est déterminée par le système
    - S'il n'y a plus de place (`ulimit -m`), l'allocation est rejetée
    - S'il n'y a plus de place en mémoire, les mécanismes de `swap` et de pagination du système *devraient* prendre le relais
- La mémoire n'est pas protégée en écriture (pour un même programme)
  - Le programmeur est responsable de son utilisation
  - Une sortie de tableau peut corrompre son état
    - Variables corrompues

Hygiène de programmation

- Limiter son utilisation en `C` / `C++`

Risques liés à `new` / `malloc`

1. Il faut libérer manuellement toute mémoire allouée
   - Sinon, il y a une fuite mémoire
   - La mémoire n'est pas toujours libérée dans le scope ou la classe où elle est allouée
   - Risque de double `delete` / `free` : erreur de segmentation
2. Un pointeur (en particulier ceux issus de `new` / `malloc`) peut être `NULL` ou `nullptr`
   - Pas une référence
3. En cas d'allocation dynamique pour un attribut de structure, il faut prendre des précautions particulières
   - Les mêmes précautions qu'avec des classes en `C++`, mais sans les mécanismes `C++`

Utilisation de `new` / `malloc`

- En `C`, utiliser l'allocation dynamique est parfois indispensable
  - Cf. section suivante

Je veux faire un `new` en `C++`

. NON !

- Il y a d'autres mécanismes
  - Pointeurs intelligents, références de rvalue

Hygiène de programmation

- Mettez toujours à `nullptr` ou `NULL` un pointeur dont la cible a été désallouée
  - Tester la nullité permet de connaître l'état du pointeur
  - Permet d'éviter les doubles `delete` et `free`

Portée et durée de vie

Récapitulatifs

- En C / C++, pas de notion de segment de données, pile ou tas
- Les classes d'allocation ne définissent que la durée de vie
- Allocation statique
  - Portée locale, semi-globale ou globale
  - Alloué en début de programme, détruit à la fin
- Allocation automatique
  - Portée locale
  - Alloué à la déclaration, désalloué en sortie de bloc
- Allocation dynamique
  - Portée « locale / globale »
  - Alloué et désalloué explicitement

Illustration en C

- Fichier `life.c` (même principe en C++ (`life.cpp`))

```c
int * addr_auto, * addr_dyn, * addr_stat = NULL; //statiques, globales
int global = 2; //statique, globale
int f() {
    int j = 42; //automatique, locale à f
    addr_auto = &j; //ok
    int * pt = (int*)malloc(sizeof(int)); //dynamique, locale
    *pt = 23; //ok : espace alloué
    addr_dyn = pt; //ok
    static int l = 17; //statique, locale
    addr_stat = &l;
    global = 3;
} //j et pt sont désalloués (mais pas *pt, ni l)
int main() {
    f();
    printf("%p:", addr_auto); //ok (dangling)
    printf("%d\n", *addr_auto); //KO : j est désaloué
    printf("%p:", addr_dyn); //ok
    printf("%d\n", *addr_dyn); //ok : *addr_dyn n'a pas été désalloué
    printf("%p:", addr_stat); //ok
    printf("%d\n", *addr_stat); //ok : l n'a pas été désalloué
    printf("%d\n", global); //ok
    free(addr_dyn);
    printf("%p:", addr_dyn); //ok (dangling)
    printf("%d\n", *addr_dyn); //KO : *addr_dyn est désalloué
}
```

Doubles pointeurs

Les doubles pointeurs en C

- En C, on a parfois besoin d'utiliser des doubles pointeurs
  - Classiquement, lorsque l'on dissocier des allocations

Exemple

- On veut créer une fonction qui alloue un tableau d'entiers
- On peut soit
  - laisser le compilateur créer le pointeur qui contiendra l'adresse de l'espace alloué
  - fournir le pointeur qui contiendra l'adresse de l'espace alloué

- Parfois, l'application que l'on fait des pointeurs « force » le programmeur à le fournir

Idée stupide

- On retourne l'adresse d'une variable locale

```
1  int* allocate_auto(int size)
2  {
3      int tab[size];
4      for(int i = 0; i < size; i++)
5      tab[i] = i;
6      return tab;
7  }
```

- Erreur de segmentation !
- Fichier `doubleptr.c`

Premier cas : on laisse le compilateur faire

1. On déclare le pointeur, on alloue et on affecte le pointeur avec `malloc`
2. On affecte l'espace alloué avec l'opérateur `[]`
3. On retourne le pointeur

```
1  int* allocate(int size)
2  {
3    int* pt = (int*)malloc(size * sizeof(int));
4    for(int i = 0; i < size; i++)
5      pt[i] = i; //*(pt + i * sizeof(int)) si pt est void*
6
7    return pt;
8  }
9
10 int main()
11 {
12   int * pt = allocate(5); //crée un pointeur pour stocker 5 entiers
13   for(int i = 0; i < 5; i++)
14     printf("%d ", pt[i]);
15   printf("\n");
16
17   free(pt);
18 }
```

Deuxième cas : on fournit le pointeur

1. On déclare le pointeur dans `main`, on alloue et on affecte le pointeur avec `malloc` dans une fonction
2. On affecte l'espace alloué en déférençant le pointeur
3. On retourne le pointeur

```
1  void allocate(int* pt, int size)
2  {
3    pt = (int*)malloc(size * sizeof(int));
4    for(int i = 0; i < size; i++)
5      pt[i] = i;
6  }
7
8  int main()
9  {
10   int * pt = NULL;
11   allocate(pt, 5);
12   for(int i = 0; i < 5; i++)
13     printf("%d ", pt[i]);
14   printf("\n");
15
16   free(pt);
17 }
```

- Erreur de segmentation !

Le nœud du problème

1. On crée `pt` dans `main`
2. On lui affecte la valeur `NULL`
   - Macro, valeur zéro
3. On appelle `allocate` en passant `pt` en paramètre
4. `pt` est passé par *valeur*
   - On passe une copie `pt'` de `pt` à `allocate`
5. On alloue un espace dans `allocate`, et on affecte `pt'` à l'adresse de cet espace
6. On affecte des valeurs dans l'espace alloué
7. On retourne dans `main`
   - `pt` est toujours à `NULL`
8. On déférence `pt`
9. Erreur de segmentation

Solution

Idée

- Il faudrait passer `pt` par adresse

- Il faut donc prendre l'adresse d'un pointeur
  - L'adresse d'un type `T` est de type `T*`
  - L'adresse d'un type `T*` est de type `T**`
- Double pointeur
- En `C++`, les doubles pointeurs sont très souvent inutiles car on possède les références
  - On évite d'utiliser une grande quantité de pointeurs grâce à ce concept

Deuxième cas bis : on utilise un double pointeur

1. On déclare le pointeur dans `main`, on le passe par adresse à `allocate`
2. On affecte l'espace alloué en déférençant l'adresse du pointeur
   - Ainsi, on a un effet de bord dans `main`

```
1  void allocate(int** pt, int size)
2  {
3    *pt = (int*)malloc(size * sizeof(int));
4    for(int i = 0; i < size; i++)
5      (*pt)[i] = i;
6  }
7
8  int main()
9  {
10   int * pt = NULL;
11   allocate(&pt, 5);
12   for(int i = 0; i < 5; i++)
13     printf("%d ", pt[i]);
14   printf("\n");
15
16   free(pt);
17 }
```

- Fichier `double-ptr.c`