

# Ch. 5 - Allocation

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



14 octobre 2020

# Table des matières

- 1 Introduction
- 2 Allocation statique
- 3 Allocation automatique
- 4 Allocation dynamique
- 5 Portée et durée de vie
- 6 Pointeurs intelligents

# 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

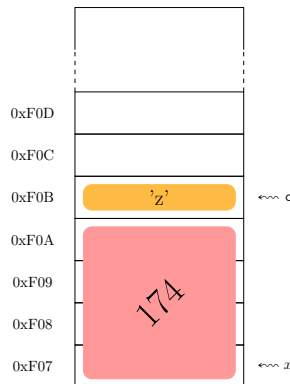
- Variables globales et variables déclarées avec le mot-clé `static`
- Portée
  - globale si la variable est globale
  - locale si la variable est déclarée au sein d'un bloc
- Durée de vie
  - Mémoire allouée au lancement du programme
  - Mémoire désallouée en fin de programme
- Valeurs par défaut
  - 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`

Segment de données



*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 1

## ■ Fichier `static.cpp`

```
1  const double PI = 3.14;
2
3  double circle_area(double r)
4  {
5      return PI * r * r;
6  }
7
8  int main()
9  {
10     cout << circle_area(2) << endl;
11 }
```

## Exemple 2

### ■ Fichier `static.cpp`

```
1  int countdown()
2  {
3      static int i = 6;
4      i--;
5      return i;
6  }
7
8  void boom()
9  {
10     bool stop = false;
11     while(! stop)
12     {
13         int j = countdown();
14         if(j >= 0)
15             cout << j << endl;
16         else
17             stop = true;
18     }
19     cout << "BOOM" << endl;
20 }
21
22 int main()
23 {
24     boom();
25 }
```

# 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 en C++

# Exemple de bonne utilisation

## ■ Fichier `good-static.cpp`

```
1  double random_double(double min, double max)
2  {
3      static random_device rd;
4      static mt19937 rng;
5
6      uniform_real_distribution<double> dist(min, max);
7      return dist(rng);
8  }
9
10 int main()
11 {
12     cout << random_double(0, 1) << endl;
13 }
```

- Les objets `rd` et `rng` sont instanciés et initialisés une unique fois
- Ils sont réutilisés à chaque appel de `random_double`

# Allocation automatique

# Classe d'allocation automatique

- Variables non globales allouées sans mot-clé ou avec `auto`, à la déclaration
- Portée toujours locale
- Durée de vie
  - Mémoire allouée à la déclaration
  - Mémoire désallouée automatiquement 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
  - Le plus grand espace allouable est déterminé par le système
  - 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

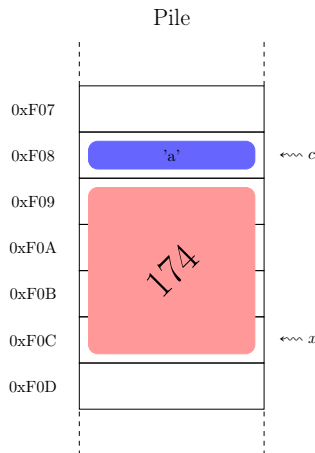
# Illustration

```

{
  ...
  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`, seuls les primitifs et les adresses d'objets sont alloués sur la pile

# Exemple

## ■ 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          cout << i << " " << j << endl;
9
10         i--;
11     } //j is destroyed
12     cout << i << endl;
13     //cout << j << endl; //j is out of the scope
14 } //i is destroyed
```

## Exemple 2

### ■ Fichier `automatic2.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++

# 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*
  - Le système d'exploitation *devrait* désallouer en fin de programme

### ■ Valeurs par défaut

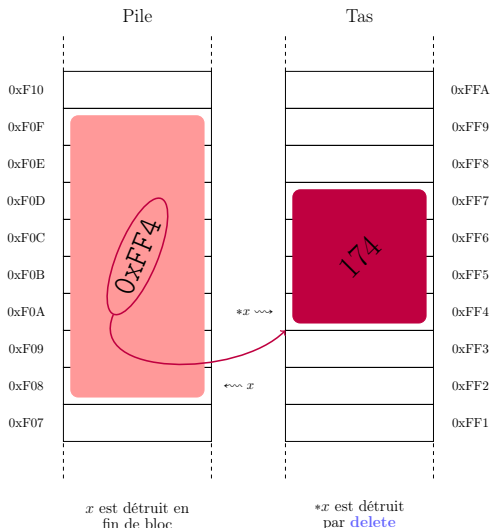
- Aucune : instanciation explicite nécessaire



# Illustration

```
{
  ...
  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

# 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`

# Exemple C++

## ■ Fichier dynamic-cpp.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 ?

```

# Exemple C

## ■ Fichier dynamic-c.cpp

```
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 }
```

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

## ■ Fichier `fct-alloc-c.c`

```
1  int main()
2  {
3      int * p = (int*) malloc(4 * sizeof(int));
4      print_int_array(p, 4); //undeterminate values
5
6      free(p);
7      print_int_array(p, 4); //undefined behaviour
8
9      p = (int*) calloc(4, sizeof(int));
10     print_int_array(p, 4); //0 0 0 0
11
12     for(int i = 0; i < 4; i++)
13         p[i] = i;
14     print_int_array(p, 4); //0 1 2 3
15 }
```

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

## ■ Fichier fct-alloc-c.c

```
1  int main()
2  {
3      int * p = (int*) malloc(4 * sizeof(int));
4
5      p = (int*) realloc(p, 2 * sizeof(int));
6      if(p) //check wether allocation succeeded
7      {
8          print_int_array(p, 2); //0 1
9          print_int_array(p, 4); //0 1 2 3 : not undeterminate
10     }
11     else
12         free(p);
13
14     p = (int*) realloc(p, 4 * sizeof(int));
15     if(p)
16     {
17         print_int_array(p, 2); //0 1
18         print_int_array(p, 4); //0 1 ? ?
19     }
20     else
21         free(p);
22 }
```

# 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 }
```



# 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 déréferencer
  - `*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 déréférencement
- `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 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 classe, il faut prendre des précautions particulières
  - 1 Destructeur
  - 2 Constructeur de copie (cf. Chapitre 8)
  - 3 Opérateur d'affectation (cf. Chapitre 8)

## Utilisation de `new` / `malloc`

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

Je veux faire un `new` en C++

■ **NON !**

# Utilisation de `new` en C++

## Je veux quand même faire un `new`

- 1 J'ai l'habitude en `Java`
  - En `Java`, on ne peut pas écrire sur la pile
  - En `Java`, il y a un garbage collector (compromis efficacité / sûreté)
- 2 Je veux éviter une copie de paramètre de fonction
  - Utilise le passage par référence
- 3 Je veux qu'une fonction modifie ses paramètres (effet de bord)
  - Utilise le passage par référence
- 4 Je veux avoir un attribut sans avoir à le recopier par le constructeur
  - Utilise une référence
- 5 Je veux activer le polymorphisme
  - Utilise une référence

# Moralité

## Hygiène de programmation en C++

- Utilisez des références quand vous pouvez
  - Utilisez des pointeurs quand vous devez
- 
- Exemple d'obligation : résoudre une dépendance cyclique

## Si on veut *vraiment* utiliser `new`

- Encapsulation dans des *pointeurs intelligents*
- 
- Cf. fin de chapitre



# 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 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`

```

1  int * pt1, * pt2, * pt3 = NULL; //statiques , globales
2  int global = 2; //statique , globale
3
4  int f()
5  {
6      int j = 42; //automatique , locale à f
7      pt1 = &j; //ok
8      int * k = (int*)malloc(sizeof(int)); //dynamique , locale
9      *k = 23; //ok : espace alloué
10     pt2 = k; //ok
11     static int l = 17; //statique , locale
12     pt3 = &l;
13     global = 3;
14 } //j et k sont désalloués (mais pas *k, ni l)
15
16 int main()
17 {
18     f();
19     printf("%p:", pt1); //ok
20     printf("%d\n", *pt1); //ko : j est désalloué
21     printf("%p:", pt2); //ok
22     printf("%d\n", *pt2); //ko : *k n'a pas été désalloué
23     printf("%p:", pt3); //ok
24     printf("%d\n", *pt3); //ok : l n'a pas été désalloué
25     printf("%d\n", global); //ok
26 }

```

# Illustration en C++

## ■ Fichier `life.cpp`

```
1  int *pt1, *pt2, *pt3 = nullptr; //statiques , globales
2  int global = 2; //statique , globale
3
4  int f()
5  {
6      int j = 42; //automatique , locale à f
7      pt1 = &j; //ok
8      int * k = new int(23); //dynamique , locale
9      pt2 = k; //ok
10     static int l = 17; //statique , locale
11     pt3 = &l;
12     global = 3;
13 } //j et k sont désalloués (mais pas *k, ni l)
14
15 int main()
16 {
17     f();
18     cout << pt1 << ":"; //ok
19     cout << *pt1 << endl; //ko : j est désaloué
20     cout << pt2 << ":"; //ok
21     cout << *pt2 << endl; //ok : *k n'a pas été désalloué
22     cout << pt3 << ":"; //ok
23     cout << *pt3 << endl; //ok : l n'a pas été désalloué
24     cout << global << endl; //ok
25 }
```

# Le cas des classes

- Souvent, l'adresse d'un objet est l'adresse du premier attribut
- Les attributs non dynamiques et non statiques ont la même classe d'allocation que l'objet
- Les attributs `static` sont statiques
- Attributs dynamiques
  - Données en classe d'allocation dynamique
  - Adresses de même classe d'allocation que l'objet
- Les fonctions membres sont généralement allouées dans le segment de code
  - Pas les fonctions `inline`
- Les remarques en terme de portée et de durée de vie sont valides sous ces conditions

# Illustration

## ■ Fichier class-alloc.cpp

```

1  struct Array {
2      int i;
3      int * arr;
4
5      Array(int i) : i(i) {
6          arr = new int[i];
7      }
8  }; //missing destructor... and other things
9
10 int main() {
11     Array a(2); //a automatic
12                 //i automatic
13                 //tab automatic
14                 //*tab dynamic
15
16     static Array b(2); //b static
17                       //i static
18                       //tab static
19                       //*tab dynamic
20
21     Array * c = new Array(2); //c dynamic
22                               //i dynamic
23                               //tab dynamic
24                               //*tab dynamic
25 }
```

# Les doubles pointeurs en C

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

# 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 }
```

■ Fichier `doubleptr.c`



## 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 déréférençant 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  $\text{pt}$  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 déréférençant 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`

# Pointeurs intelligents

# Nécessité en allocation dynamique

- Quand on sort d'un scope, il faut décider quoi faire de la mémoire allouée
- Utilisation du patron de classe, `unique_ptr`, `shared_ptr` et `weak_ptr`.
- Paramétré par le type de la variable dynamique à encapsuler
- Comportements « similaires » aux pointeurs / références, en classe automatique, implémenté via un patron de classe et la surcharge d'opérateurs (cf. Ch. 7).
- Chaque patron définit comment la mémoire doit être gérée à la destruction (automatique) du pointeur intelligent.
  - On détruit les données ?
  - On détruit les données si plus rien ne pointe dessus ?
  - On ne détruit rien ?
- Implémenté en comptant le nombre de références dans le constructeur à l'aide d'une variable statique.
- Inclure `memory.h` (C++uniquement)

# Pointeurs intelligents

- Quand un pointeur possédant un objet est détruit, il faut définir comment libérer la mémoire
- Trois types de pointeurs intelligents « principaux »
  - 1 `unique_ptr` : pointeur intelligent qui n'autorise qu'une possession unique de l'objet.
    - Copier et affecter le pointeur provoque une erreur de compilation.
    - Quand le pointeur est détruit, la donnée est détruite.
  - 2 `shared_ptr` : pointeur intelligent qui autorise des possessions multiples d'un même objet.
    - Les données pointées sont détruites si plus rien ne pointe dessus.
    - Le dernier pointeur possédant les données est détruit
  - 3 `weak_ptr` : pointeur intelligent qui ne « possède pas » d'objet.
    - Doit être converti en `shared_ptr` pour accéder l'objet (`lock()`).
    - Pratique pour une possession temporaire, quand l'objet peut être détruit n'importe quand par un facteur extérieur.
- Instanciation « à la volée » avec `new` ou via `make_unique`, `make_shared`

# Exemple `unique_ptr`

## ■ Fichier `unique.cpp`

```
1  int main()
2  {
3      int i = 2;
4      int * pti = &i;
5      unique_ptr<int> u1(&i);
6      {
7          //unique_ptr<int> u2(&i); //bad idea
8          //unique_ptr<int> u2 = u1; //compile error
9          unique_ptr<int> u2 = move(u1); //u2 owns, u1 invalid
10     }
11     cout << *pti << endl;
12     u1.reset(); //deletes memory (why ?!)
13     cout << i << endl; //seg fault
14 }
```



# Exemple `shared_ptr`

## ■ Fichier `shared.cpp`

```
1  int main()
2  {
3      shared_ptr<int> p1(new int(5));
4      weak_ptr<int> wp1 = p1; //p1 owns the memory.
5
6      {
7          shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
8          if(p2) //check if the memory still exists!
9          {
10             cout << "if_p2" << endl;
11         }
12     } //p2 is destroyed. Memory is owned by p1.
13
14     p1.reset(); //Memory is deleted.
15
16     shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
17     if(p3)
18     {
19         cout << "if_p3" << endl;
20     }
21 }
```

## Exemple de fuite mémoire : initialisation (1/2)

```
1 int f(shared_ptr<int> i, int j);  
2 int g();  
3  
4 f(shared_ptr<int> (new int (42)), g());
```

### ■ Ordre d'appel

- 1 Allocation dynamique de l'entier 42
- 2 Création du `shared_ptr<int>`
- 3 Appel de la fonction `g`
- 4 Appel de la fonction `f`

## Exemple de fuite mémoire : initialisation (2/2)

### Ordre d'appel

- 3 peut avoir lieu *avant* 1 et 2, et peut en particulier être appelé entre 1 et 2

### Problème potentiel : *g* lance une exception

- Le `shared_ptr` n'as pas encore eu le temps de posséder la mémoire
- Il ne peut pas la libérer
- Fuite mémoire

# Solution

## ■ Première idée

```
1  int f(shared_ptr<int> i, int j);  
2  int g();  
3  
4  shared_ptr<int> si (new int (42));  
5  f(si, g());
```

## ■ Meilleure idée

```
1  int f(shared_ptr<int> i, int j);  
2  int g();  
3  f(make_shared<int>(42), g());
```

## ■ Moralité : ne pas faire de `new`

# Exemple de fuite mémoire : cycle

## ■ Fichier `cycle.cpp`

```
1  class A
2  {
3      public:
4          shared_ptr<B> ptB;
5  };
6
7  class B
8  {
9      public:
10         shared_ptr<A> ptA;
11     };
12
13     int main()
14     {
15         shared_ptr<A> a(new A);
16         shared_ptr<B> b(new B);
17         cout << a.use_count() << ", " << b.use_count() << endl;
18         a->ptB = b;
19         cout << a.use_count() << ", " << b.use_count() << endl;
20         b->ptA = a;
21         cout << a.use_count() << ", " << b.use_count() << endl;
22         a.reset();
23         b.reset();
24         cout << a.use_count() << ", " << b.use_count() << endl;
25     }
```

# Sournoiserie

- Affichage à la ligne 17 : 1 1
- Affichage à la ligne 19 : 1 2
  - `b` et `ptB` pointent vers l'objet de type `B`
- Affichage à la ligne 21 : 2 2
- Affichage à la ligne 24 : 0 0

## Zombie

- `ptB` dans `A` fait survivre `B`
- `ptA` dans `B` fait survivre `A`

## Solution

- Utiliser `weak_ptr`

# Solution

## ■ Fichier `cycle-sol.cpp`

```

1  class A
2  {
3      public:
4          shared_ptr<B> ptB;
5  };
6
7  class B
8  {
9      public:
10         weak_ptr<A> ptA;
11     };
12
13     int main()
14     {
15         shared_ptr<A> a(new A);
16         shared_ptr<B> b(new B);
17         cout << a.use_count() << ", " << b.use_count() << endl;
18         a->ptB = b;
19         cout << a.use_count() << ", " << b.use_count() << endl;
20         b->ptA = a;
21         cout << a.use_count() << ", " << b.use_count() << endl;
22         a.reset();
23         b.reset();
24         cout << a.use_count() << ", " << b.use_count() << endl;
25     }

```

# Exemple complet

- Liste simplement chaînée

- 1 Fichier `linkedlist-new.cpp`
- 2 Fichier `linkedlist-smart.cpp`

- Classe interne de nœud

- Une donnée, un élément suivant
- Difficile d'utiliser des références
  - Initialement, la liste est vide (tête et queue)
  - La queue a toujours un successeur vide

- Remarque : aucune gestion explicite de la mémoire avec les pointeurs intelligents

- Pas de `new`, `delete`
- Pas de destructeur

- En C, on n'a pas le choix

- `malloc` et `free` manuels nécessaires
- Fichier `linkedlist-c.c`



# Remarque

## Hygiène de programmation C++

- 1 Ne faites pas de `new`
  - 1 Utilisez des références
  - 2 Utilisez des pointeurs intelligents
- 2 Utilisez `make_shared` et `make_unique`
  - 1 Évite de faire un `new` et de gérer la mémoire « à la main »
  - 2 Évite de créer des `shared_ptr` temporaires

## Hygiène de programmation de base

- Mettez à `nullptr` ou `NULL` un pointeur dont l'espace a été désalloué
  - Permet de vérifier que l'espace est « invalide »
  - Permet d'éviter les doubles `delete` et `free`