UNIVERSITÉ PARIS DESCARTES

SDA - Structures de Données et Algorithmes

Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Eloi Keita

T3 – Rôle des pointeurs et des références
T4 – Modes d'allocation et de gestion mémoire
T5 – Besoins d'allocation dynamique de la mémoire
T7 – new (C++), allocation de mémoire dynamique
T8 – delete (C++), désallocation en mémoire dynamique
T9 à T13 – Exemple (C++) : tableau statique et dynamique
T14 – calloc et malloc (C), allocation de mémoire
T15 – realloc (C), modification de la zone mémoire allouée
T16 – free (C), désallocation de mémoire dynamique
T17 – Fuite mémoire
T20 - Synthèse des règles de prototypage
T21 - Retour de fonction de type pointeurs ou référence
T26 - Pointeurs multiples

Cours n° 3
Pointeurs et allocation dynamique

Sommaire

1. GESTION DE MEMOIRE DYNAMIQUE

Rappels

Rôle des pointeurs et des références

Pointeurs/références et fonctions

Pointeurs et références sont deux mécanismes de passage des paramètres de sortie

C

La notion de référence n'est pas définie
Seul le pointeur permet un **effet de bord** sur le contexte d'exécution de l'appelant par le passage d'un **paramètre de sortie de type pointeur** pointant **sur le paramètre effectif**
Mécanisme peu lisible et lourd à gérer dans le code
(**référencement** au niveau appelant et **déréférencement** au niveau appelé)

C++

Pointeur et **références** sont définies
Le **paramètre de sortie référence** alias **paramètre effectif** permet un deuxième mode de passage permettant un effet de bord sur le contexte d'exécution de l'appelant
Le passage par **référence** est **préféré** (simplicité du code : à son appel et du corps de la fonction)

Pointeurs et tableaux statiques

C C++

```
int tab[20];  // tab est l'adresse de son premier élément (&tab[0]) en mémoire
// d'où le passage 1) d'un paramètre pointeur (int* tab) et 2) de sa taille dans une fonction
```

Pointeurs et gestion dynamique de la mémoire

C C++

Allocation dynamique et mécanisme d'indirection

1. GESTION DE MEMOIRE DYNAMIQUE

Rappel

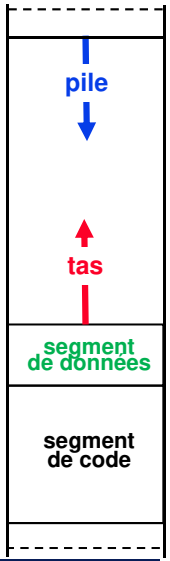
Modes d'allocation et de gestion de la mémoire

Trois modes d'allocation mémoire en C++ et C

Mémoire statique [données - *data segment*]
La mémoire est allouée en phase de compilation/édition de liens
(variables globales et variables statiques)
elle est automatiquement désallouée/libérée dès que le programme a fini de s'exécuter

Mémoire automatique [pile - *stack*]
La mémoire est automatiquement allouée, gérée et désallouée pendant l'exécution de la fonction
(adresse de retour, paramètres formels, variables locales et valeur de paramètre de retour)

Mémoire dynamique [tas - *heap*]
La mémoire est explicitement gérée par le programmeur
- **allouée** (en C++, `new` et en C, `malloc`, `calloc`, `realloc`)
- **libérée** (en C++, `delete` et en C, `free`)



Besoins d'allocation dynamique de mémoire 1/2

Optimisation de la mémoire utilisée pour la faisabilité d'une application

L'allocation mémoire automatique d'une variable par sa déclaration n'est possible que si l'on connaît la taille de la variable

Exemple : un tableau de relevé annuel de température

```
int t[365];
```

Lorsque l'on ne connaît pas cette taille

Exemple : une application qui gère une liste de clients dont on ne connaît pas à l'avance le nombre

1) Une solution simple est de dimensionner l'application
- Risque d'un mauvais dimensionnement
- Pas d'optimisation de la mémoire utilisée (sur-dimensionnement)

2) La solution optimale est l'allocation dynamique de mémoire
- L'allocation est explicitement demandée (codée dans le source) suivant les besoins
- L'allocation mémoire est faite pendant l'exécution du programme (d'où le nom d'allocation dynamique)

Besoins d'allocation dynamique de mémoire 2/2

- Adaptation dynamique des variables allouées selon les besoins de l'application

  Exemple :
  Saisie de chaînes de caractères de type mot (de taille variable)
  Dans le dictionnaire
  de « à » [1 caractère]
  à « anticonstitutionnellement » [25 caractères]
  Cette connaissance impose de dimensionner la chaîne de caractères à 25
  Rem : en français, la moyenne de la longueur des mots est de l'ordre de 5
  Problème de surdimensionnement… qui peut être problématique

- De nombreuses structures de données emploient tout naturellement l'allocation dynamique de mémoire

  Exemple :
  Les listes, les arbres…

new – allocation de mémoire dynamique C++

Syntaxe de l'allocation en mémoire dynamique

d'une variable

```
<id_type>* <id_variableDyn> = new <id_type>;
```

Demande système d'allocation d'une zone mémoire contenant une variable de type `id_type` en mémoire dynamique

```
Date* d = new Date;
```

d'un tableau

```
<id_type>* <id_tableauDyn> = new <id_type> [nb_Item];
```

Demande système d'allocation d'une zone mémoire contenant un tableau de `nb_Item` de type `id_type` en mémoire dynamique

```
Date* tDates = new Date[15];
```

Valeur de retour

si l'allocation réussit, `new` retourne l'adresse de début de la zone allouée

sinon (place mémoire insuffisante) `i)` `new` lance d'une exception *bad-alloc (sortie pg.)*

`ii)` `new(nothrow)` retourne `NULL`

```
Date* d = new(nothrow) Date;
if (d == NULL) {
   cout<<"Allocation mémoire impossible"; exit(1);
}
// d est l'adresse de la zone allouée
```


delete – désallocation en mémoire dynamique C++

Syntaxe de la désallocation en mémoire dynamique

d'une variable

```
delete <id_variableDyn>;
```

Demande au système la désallocation de la zone mémoire allouée à la variable `id_variableDyn` en mémoire dynamique

```
delete d;
```

d'un tableau

```
delete [] <id_tableauDyn>;
```

Demande au système la désallocation de la zone mémoire allouée au tableau `id_tableauDyn` en mémoire dynamique

```
delete [] tDates;
```

Aucune valeur de retour

- Une zone désallouée peut dès lors être utilisée par le système. Elle ne doit plus être accédée (en lecture ou écriture) via son adresse d'allocation antérieure
- A chaque instruction `new` (initialisant un pointeur **p**) doit correspondre une instruction `delete` (adaptée au pointeur **p**)

Tableau statique et tableau dynamique

C++

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

int main (){
   Date noel = {25, 12, 2016}; Date premierAn = {1, 1, 2017};
   Date retourIUT = {3, 1, 2017};

   Date tabSta [] = {noel, premierAn};

   Date* tabDyn = new(nothrow) Date[3]; // allocation du tableau
                                        // dynamique tabDyn
   if (tabDyn == NULL) { //allocation impossible
      printf("Allocation mémoire impossible.\n");
      exit(1);           //sortie de programme avec statut d'erreur
   }

1  tabDyn[0]= noel;
   tabDyn[1]= premierAn;
   tabDyn[2]= retourIUT; // date de retour à l'IUT

2  delete [] tabDyn;
3  tabDyn = NULL;
4  return 0;
}
```

// Extrait de main03.cpp
#include <iostream>
using namespace std;

points d'arrêt

Version en C
// main03.c

Tableau en mémoire dynamique – Point d'arrêt 1 1/4

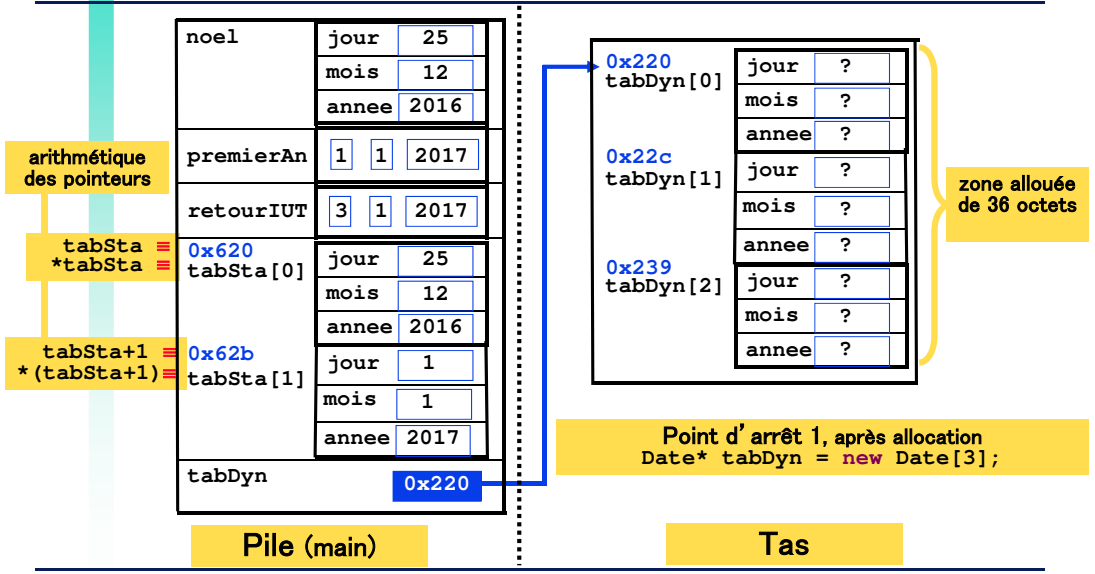


Tableau en mémoire dynamique – Point d'arrêt 2 2/4

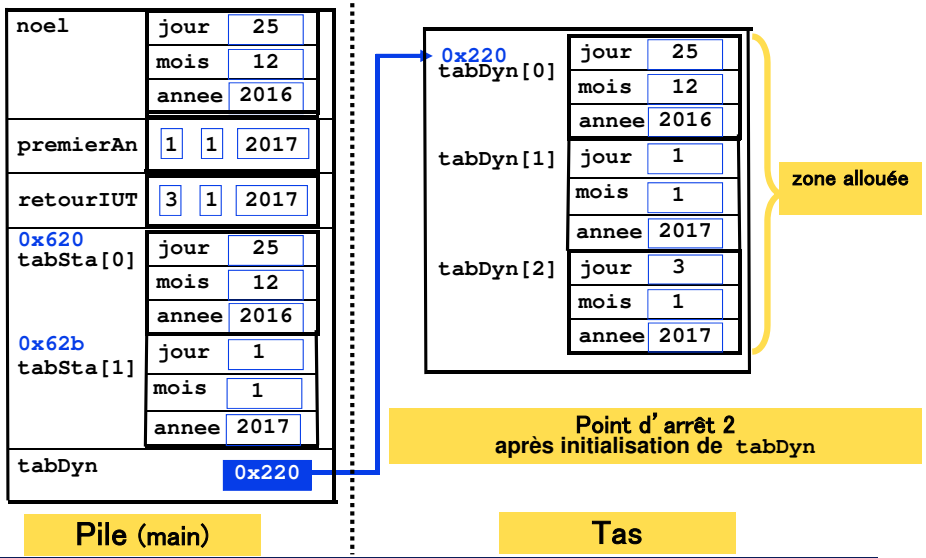


Tableau en mémoire dynamique – Point d'arrêt 3 3/4

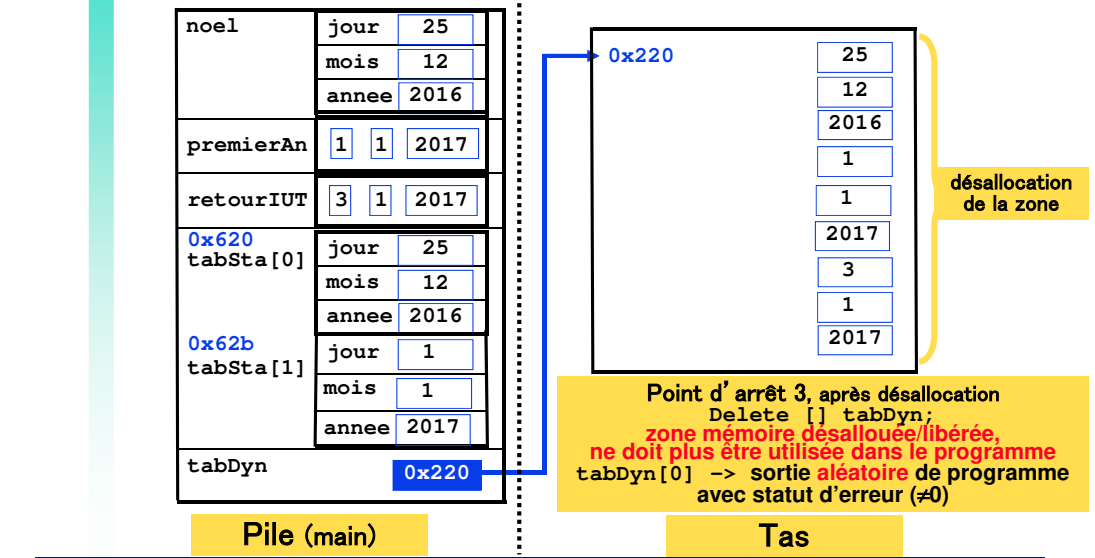
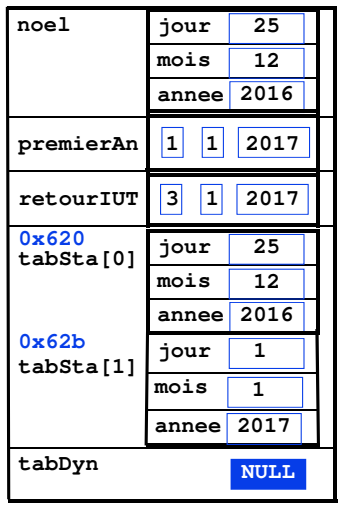
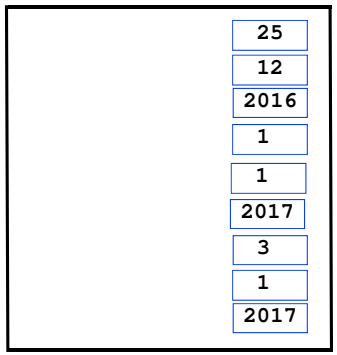


Tableau en mémoire dynamique – Point d'arrêt 4 4/4



Pile (main)



Point d'arrêt 4, après
`tabDyn = NULL;`
Ex: `tabDyn[0]` -> sortie de programme avec statut d'erreur (≠0)

Tas

calloc et malloc – allocation de mémoire

C

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

Syntaxe de l'allocation mémoire

```
void* calloc(size_t nbItem, size_t tailleItem);
```

Demande système d'allocation d'une zone de taille `nbItem * tailleItem` octets consécutifs en mémoire dynamique. La zone mémoire est initialisée à 0.

variable → `Date* d = (Date*) calloc(1, sizeof(Date));`
tableau → `Date* tDates = (Date*) calloc(15, sizeof(Date));`

```
void* malloc(size_t tailleOctets);
```

Demande système d'allocation d'une zone de `tailleOctets` octets consécutifs en mémoire dynamique

variable → `Date* d = (Date*) malloc(sizeof(Date));`
tableau → `Date* tDates = (Date*) malloc(15 * sizeof(Date));`

Valeur de retour

si l'allocation réussit, `calloc` ou `malloc` retourne l'adresse (différente de `NULL`) de début de la zone allouée

sinon (place mémoire insuffisante ou taille de zone demandée <= 0) retourne `NULL`

Attention : `calloc` ou `malloc` retourne une valeur de type pointeur générique
Pour pouvoir utiliser la zone mémoire, faire une conversion de type explicite
Ne pas désallouer par `delete` une zone allouée par `calloc` , `malloc`

realloc – modification de la zone mémoire allouée

C

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

Syntaxe de modification de l'allocation en mémoire dynamique

```
void* realloc(void* ptr, size_t taille);
```

Demande système de réallocation/extension d'une zone mémoire dynamique d'adresse (de début) `ptr` déjà allouée (par `calloc`/`malloc`/`realloc`) à `taille` octets consécutifs en mémoire dynamique
Si `taille` est inférieure à la taille originelle, la zone est tronquée à la nouvelle taille
Si `ptr` est NULL, `realloc` équivaut à `malloc(taille)`
Si `taille` vaut `0`, `realloc` équivaut à `free(ptr)`

Valeur de retour

si l'allocation réussit, `realloc` retourne l'adresse (différente de `NULL`) de début de la zone réallouée.
Cette adresse peut être différente de celle fournie en argument.
Dans ce cas, le contenu de l'ancienne zone est copié à la nouvelle adresse et l'ancienne zone est automatiquement désallouée

sinon (place mémoire insuffisante ou *taille=0*) `realloc` retourne `NULL`

Attention : Ne pas désallouer par `delete` une zone allouée par `realloc`

free – désallocation de mémoire dynamique

C

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

Syntaxe de la désallocation de mémoire dynamique

```
void free(void* ptr);
```

Demande système de désallocation de la zone mémoire allouée à l'adresse `ptr` en mémoire dynamique
La zone a déjà été allouée (`ptr` initialisé par `calloc`/`malloc`/`realloc`)

```
free(d);
free(tDates);
```

Mêmes remarques que pour l'opérateur `delete`

- Une zone désallouée ne doit plus être accédée (en lecture ou écriture)
  Ne pas utiliser le pointeur `ptr` après l'instruction `free(ptr);`
- A tout appel de fonction d'allocation de mémoire (`calloc`/`malloc`/`realloc`) doit correspondre un et un seul appel à la fonction `free` avec pour argument le pointeur `ptr`
- Ne pas désallouer par `free` une zone allouée par `new`

Fuite mémoire, suivi de la gestion des données

Une **fuite mémoire** se produit lorsque le programmeur perd la trace (**le pointeur**) de la zone de mémoire dynamique allouée par l'opérateur **new** ou l'une des fonctions d'allocation (**calloc**, **malloc**, **realloc**)

```
void fuiteMemoire() {
   char* s = new char[…];
   // instructions sans appel de delete[]
   …
   return;
}
```

Après l'exécution de fuiteMémoire
le contexte de la pile d'exécution est désallouée
l'adresse de la zone allouée (s) est par conséquent perdue

L'**allocation dynamique** des données doit être **signalée** dans la documentation des fonctions pour une **bonne réutilisation** incluant la gestion de la mémoire dynamique (cf. primitive de traitement de chaîne (en C) `strdup`, …)

```
char* f() {
   char* ca = new char[…];
   // sans appel de delete []
   return ca;
}
```

Après l'exécution de f
il faudra désallouer la mémoire dynamique
C'est à signaler dans la documentation

Des pièges et des fuites…

```
void piege(int* p) {
  std::cout << *p << std::endl;
  delete p;
}

int main() {
  int a;
  piege(&a);
  int* p = new int;
  *p = 21;
  piege(p);
  std::cout << *p << std::endl;
  return 0;
}
```

L'évolution de la pile d'exécution du programme vous permettra de répertorier les différents problèmes

Un gros problème de collision…

```
void egoiste() {
  int* p = new int;
  *p = 12;
}

int main() {
  for ( ; ; )
    egoiste();
  return 0;
}
```

Un schéma de l'évolution de la pile d'exécution du programme vous aidera encore à identifier la collision

L'exécution du programme est également très explicite sur le type de collision

Note (1) : Tableau (sauf tableau de caractères (délimités par '\0') passé en paramètre **obligatoirement** accompagné du **passage** de sa **taille**

Synthèse des règles de prototypage

		Nature du paramètre p	Langage C++	Langage C
entrée	[in]	de **type natif T** (bool, char, int float, double…)	variable `void f(T p, …)`	
		de **type-utilisateur T** taille >> 4 octets	**référence** vers une constante `void f(const T& p, …)`	**pointeur** vers une constante `void f(const T* p, …)`
		tableau(1) **d'éléments de type T**	**pointeur** vers une constante `void f(const T* p, …)` ou `void f(const T p[], …)`	
sortie	[out] ou [in, out]	**type natif T** (bool, char, int, …)	**référence** `void f(T& p, …)`	**pointeur** `void f(T* p, …)`
		de **type-utilisateur T**		
		Tableau(1) **d'éléments de type T**	**pointeur** `void f(T* p, …)` ou `void f(T p[],…)`	
	retour	de **type natif T** ou **type-utilisateur**(2) **T**	`T f(…)`	

Note (2) en cas de taille >> 4 octets, **Préférez** au paramètre de retour un **paramètre formel [out]**

Retour de fonction de type pointeur ou référence
De la faiblesse des compilateurs… (1/3)

```
int* ad_locale() {
  int locale = 0;
  return &locale;
}

int& ref_locale() {
  int locale = 0;
  return locale;
}

int main() {
  int* p1 = ad_locale();
  *p1 = 10;
  int& r1 = ref_locale();
  r1 = 11;
return 0;
}
```

Ces fonctions sont-elles cohérentes ?

L'évolution de la pile d'exécution du programme principal vous donnera la réponse

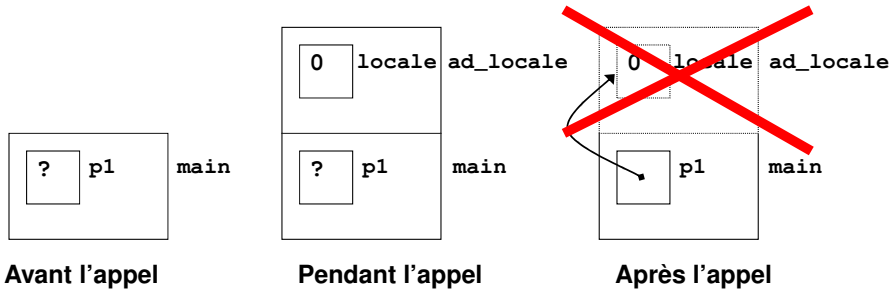
Retour de fonction de type pointeur ou référence
De la faiblesse des compilateurs… (2/3)

```
int* ad_param(int param) {
  return &param;
}

int& ref_param(int param) {
  return param;
}

int main() {
  int* p2 = ad_param(12);
  *p2 = 13;
  int& r2 = ref_param(14);
  r2 = 15;
  return 0;
}
```

Ces fonctions sont-elles cohérentes ?

L'évolution de la pile d'exécution du programme principal vous donnera la réponse

Retour de fonction de type pointeur ou référence
De la faiblesse des compilateurs… (3/3)



Retour de fonction de type pointeur ou référence

```
int* ad_table(int* t, int size, int i) {
   assert(i > 0 || i < size);
   return &t[i];
}

int& ref_table(int* t, int size, int i) {
   assert(i > 0 || i < size);
  return t[i];
}

int main() {
  const int SIZE = 4;
  int tab[SIZE];
  int* tmp;
  tmp = ad_table(tab, SIZE, 0);
  *tmp = 12;
  *ad_table(tab, SIZE, 1) = 13;
  int& r = ref_table(tab, SIZE, 2);
  r = 14;
  ref_table(tab, SIZE, 3) = 15;
  for (int i = 0; i < SIZE; ++i)
    cout << "tab[" << i << "] = " << tab[i] << endl;
  return 0;
}
```

tab=&tab[0]

?
?
?
?

Retour de fonction de type pointeur ou référence

```
int* ad_table(int* t, int size, int i) {
   assert(i > 0 || i < size);
   return &t[i];
}

int& ref_table(int* t, int size, int i) {
   assert(i > 0 || i < size);
  return t[i];
}

int main() {
  const int SIZE = 4;
  int tab[SIZE];
  int* tmp;
  tmp = ad_table(tab, SIZE, 0);
  *tmp = 12;
  *ad_table(tab, SIZE, 1) = 13;
  int& r = ref_table(tab, SIZE, 2);
  r = 14;
  ref_table(tab, SIZE, 3) = 15;
  for (int i = 0; i < SIZE; ++i)
    cout << "tab[" << i << "] = " << tab[i] << endl;
  return 0;
}
```

tab=&tab[0]	
	12
	13
	14
	15

Pointeurs multiples

Un pointeur est une variable implantée en mémoire
Un pointeur a une adresse, on peut lui appliquer l'opérateur &
Un pointeur peut donc être pointé par un pointeur,
ce qui permet la déclaration d'un double pointeur (valeur de type pointeur sur pointeur)

Cas d'un double pointeur

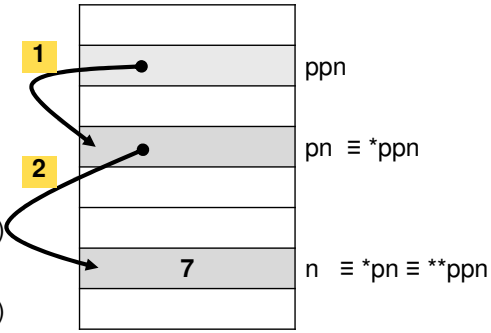
```
int n=7;
int* pn=&n; //int *pn=&n;
int** ppn=&pn; //int **pn=&n;
```

`ppn` est un double pointeur sur un entier

`*ppn` est une référence (nom de variable) d'un pointeur sur un entier

`(int*)`

`**ppn` est une référence (nom de variable) d'un entier



Ce que vous avez appris aujourd'hui…

Gestion de la mémoire dynamique

Allocation et **désallocation** (en C++ et en C) d'une **zone de mémoire dynamique adaptée à vos besoins**

La pratique vous apprendra à éviter les pièges classiques de la gestion de la mémoire dynamique qui peuvent être très coûteux en temps de déboguage

- Allouer une zone-mémoire et utiliser son contenu avant de l'avoir initialisé

```
Date* d = new Date;
cout << d->mois << endl; // d->mois ≡ (*d).mois
```

- Libérer une zone-mémoire et continuer à utiliser son contenu

```
delete d;
cout << d->mois << endl;
```

- Allouer une zone mémoire et la perdre en perdant la valeur de son pointeur

```
Date* d = new Date;
d = new Date;
```

- Lire ou écrire en dehors des limites de la zone mémoire

```
Date* tDates = new Date[15];
cout << tDates[15].mois << endl;
```