Cours avancé de S.E.

Andreea Dragut & Vincent Risch

Univ. Aix-Marseille

2013

Plan général

1. Gestion de la mémoire
   - ⇒ Allocation (statique, dynamique)
   - ⇒ Gestion manuelle, automatique
   - Architecture système
   - Gestion système
   - Intéraction processus (IPC)
2. Gestion du temps
   - Temps partagé, ordonnanceur
   - Algorithmes
   - Cadre d'illustration

Plan de ce cours – Mémoire – Allocation, gestion

- Comparaison C/C++, Java, Python, Perl
- Allocation dynamique en C
- Allocation dynamique en C++
- Dangers et erreurs typiques de programmation
  - Fuites de mémoire
  - Pointeurs pendants
  - Problèmes avec les constructeurs, destructeurs, héritage, etc.

Comparaison C/C++, Java, Python, Perl

- tous ces langages – aujourd'hui – *standard* –
- utilisation mondiale dans beaucoup de domaines d'activité
- disponibles librement sur énormément de plateformes
- vastes bibliothèques de fonctions
- documentation, livres, cours – très riches
- immenses communautés d'utilisateurs

Comparaison C/C++, Java, Python, Perl

- C/C++ – contrôle très précis des ressources matérielles et système, dont notamment la mémoire
- C/C++ – avantage : choix précis de la stratégie/politique ;
- C/C++ – inconvénient : effort pour éviter des erreurs (fuites mémoire, pointeurs « perdus »etc.)
- Java, Python, Perl – automatisation de ces choix pour le programmeur – gestionnaire automatique de mémoire
- Java, Python, Perl – avantage : absence de ces soucis
- Java, Python, Perl – inconvénient : moins efficace, gestion automatique souvent trop générale pour tâche intensives
- Choix en fonction du cadre de l'application
  - scripts pour petits et moyens sites web, analyse de logs – Perl, Python
  - applications mono-utilisateurs, avec interface graphique – Java
  - calculs intensifs, gestion de gros flots de données, systèmes embarqués, noyau système d'exploitation – C/C++

Comparaison C/C++

- C – plus simple et prédictible
  - noyau système d'exploitation
  - systèmes embarqués
  - systèmes temps réel à contraintes très dures
- C++ – plus riche, davantage d'outils
  - applications complexes, calculs intensifs
  - néanmoins : petites et moyennes équipes de développeurs
  - si équipes plus grandes : avoir une sous-équipe centralisant la gestion mémoire

Allocation dynamique en C – plan

- En général – allocation mémoire en C
  - déclaration de variables
  - demande explicite d'espace mémoire – allocation dynamique
- notion de pointeur, left value, right value
- notion d'adresse de variable, pointeurs « statiques »
- allocation dynamique – fonctions C – `stdlib`

```
#include <stdlib.h>

void *malloc (size_t size);
void *calloc (size_t nelem, size_t elsize);
void *realloc(void  *ptr,   size_t size);
void  free   (void *ptr);
```

  - `malloc()` et `calloc()` – réservation espace
  - `realloc()` – déplacement bloc déjà réservé, pour changer de dimension
  - `free()` – libération de l'espace qui avait préalablement été alloué
- à faire et à ne pas faire

Détails allocation dynamique en C

- Rappel : qu'est-ce qu'une variable de type pointeur ?
  - une variable qui est censée contenir une adresse mémoire où l'on peut stocker un *type* d'information (entier, char, etc, et même un pointeur !)
  - type du pointeur – donné par le type d'information pointée.
  - comme tout autre variable – déclaré, initialisé, utilisé, manipulé
- Déclaration d'un pointeur :

```
TYPE  *nomPointeur; // la VARIABLE nomPointeur est de type TYPE *

char  *p;           // par exemple, ici p est un char *
                    // c'est-a-dire un POINTEUR vers un char

char **r;           // on peut iterer: ici r est un char **
                    // i.e. un POINTEUR vers un POINTEUR vers un char

char*  s;           // on peut aussi coller l'* au type qui est pointe
```

  - la simple déclaration NE MET AUCUNE VALEUR dedans
  - le pointeur aura seulement sa place en mémoire (quatre octets, si 32 bit, ou huit pour 64bit, etc.) pour stocker une adresse
  - avant de commencer à l'utiliser — il faut l'INITIALISER
    - adresse d'une autre variable – opérateur unaire `&` – e.g. appliqué à une left value
    - adresse *dynamiquement allouée* (et encore valide) – `malloc()`, etc.

Petite digression – left value, right value (pas de politique :-)

- L'opérateur = effectue l'assignation de valeur (i.e. d'une *right value*)
- où cela, donc ?
- dans un ENDROIT – en C et C++ on dit dans une *left value*

```
LEFT_VALUE = RIGHT_VALUE;
```

- des *left value* sont des :
  - noms des variables,
  - expressions avec l'opérateur unaire de déréférencement `*`

```
int   A ;// declaration d'une variable de type int
int  *pA;// declaration d'une variable de type pointeur d'int
A   = 1 ;// assignation de lq right value 1 a la left value A: A vaut 1
pA  = &A;// assignation de la right value adresse-de-A a la left value pA
*pA = 2 ;// assignation de la right value 2 a la left value dereferencement-de-pA
         // et maintenant A vaut bien entendu 2
```

- des *right value* – toute *expression* C/C++ valide
- élégance et simplicité syntaxique :
  - le type de (l'expression) `pA` est bien entendu un `int *`
  - et alors, le type de l'expression `*pA` est un `int` (déréférencement d'un `int *`)
- mais NE PAS CONFONDRE
  - l'opérateur `*` de déréférencement – on l'utilise pour des expressions
  - l'`*` de la déclaration `TOTO *` d'un type « *pointeur-de-*`TOTO` ».
- c'est juste une « coïncidence »

Nécessaires pour assurer le bonheur : déclaration, initialisation, utilisation

- encore une fois, pas à pas :

```
int  *pA; // pA n'a AUCUNE VALEUR qui AIT DU SENS
int    A; // A non plus n'a AUCUNE VALEUR qui AIT DU SENS
pA  = &A; // mais &A est bien defini -- c'est la ou se trouve A en memoire
          // et SEULEMENT MAINTENANT on peut utiliser *pA comme left value
*pA =  2; // en mettant en fait une valeur dans A
```

- mais tout ceci est tout de même statique – il n'y aura qu'« un seul `A` »
- comment faire cela dynamiquement (nombre de cases mémoire variant d'une l'exécution du programme à l'autre) ?

```
int  *pA;
int    N,K;
printf("Nombre d'elements svp :\n");
scanf("%d",&N); // adresse de N (car passage param par valeur)
pA   = (int *)malloc(N * sizeof(int)); // maintenant pA est dispo
       // (bien entendu si l'appel a malloc() a reussi)
       // pA pointe vers une zone de N int un apres l'autre
for(K=0; K<N; K++) {
 printf("Element no %d svp: \n",K);
 scanf("%d",pA+K); // adresse de l'int no. K du bloc reserve
 // ATTENTION: on n'en a recu que N, donc K peut aller de 0 a N-1
}
```

Des pointeurs, des pointeurs, toujours des pointeurs

- Rappel du rappel : comme avec toute autre variable, la *déclaration* **ne met aucune valeur** *dedans*
- donc on ne peut jamais faire

```
   int  *pA;            // et "tout de suite" apres
   *pA = quelqueChose; // car pA n'a pas de bonne valeur, donc
// *pA NE MENE NULLE PART ou on ait le droit d'aller
// et on ne peut pas faire non plus par exemple
   scanf("%d",pA);    // pour la meme raison
// et on ne peut faire encore moins
   scanf("%d",pA+2);  // parce que deux cases apres nulle part
// c'est toujours du nulle part (ailleurs)
// et donc on ne saurait faire non plus
   scanf("%d",&(pA[2]));
// et ni
   printf("%d",pA[2]); // pour la meme raison
```

- au passage : `pA+2` et `pA[2]` — la même chose
- il faut donc faire un coup de `malloc()`
- revenons donc dans le droit chemin

Pérenniser le bonheur : déclaration, initialisation, utilisation, libération

```
int  *pA;
int    N,K;
printf("Nombre d'elements svp :\n"); scanf("%d",&N);
pA   = (int *)malloc(N * sizeof(int));
if(pA == NULL) {
 printf("Gros probleme: pas assez de memoire dispo\n");
 exit(1);
}
for(K=0; K<N; K++) {
 printf("Element no %d svp: \n",K);scanf("%d",pA+K);
}
trierOrdreCroissant(pA,N); // par exemple
for(K=0; K<N; K++) { printf("%d ",pA[K]); }
printf("\n");
free(pA); // quand on n'a plus besoin de pA, on le LIBERE
```

- donc `free()` annonce au système d'allocation dynamique qu'il peut reprendre la main sur le bloc qu'il nous avait alloué
- mais il faut appeler `free()` avec un pointeur VALIDE (et une seule fois)
- et après `free()`, ne plus utiliser cette VALEUR du pointeur

Pourquoi tant de rigueur ? Règles de comportement, sinon gros boum !

- ne pas toucher (de mémoire) si on n'a pas demandé (et reçu) la permission
- bien rendre ce que nous avons emprunté (et ne plus y toucher après)
- ne pas aller au delà de ce qu'on nous a accordé
- ne pas demander « trop » d'espace mémoire (sans aussi le rendre des fois)
- être bien clean, réglo et cool avec les variables locales :
  - ne pas rendre l'ADRESSE d'une variable locale (ce sera du *nulle part*...)
  - en quittant la zone de validité, la variable « n'existe plus », mais AUCUNE OPÉRATION n'est effectuée avec sa VALEUR (e.g. pas de `free()`...)

```
char *maChaine(const char *P) {
 char *R = NULL;
 if(P != NULL) {
   R = (char *)malloc((strlen(P)+1)*sizeof(char));
   if(R != NULL) { strcpy(R,P); }
 }
 return R; // on rend la VALEUR de R; R lui-meme disparaitra
}
int main() {
 char *A;
 A = maChaine("Bonjour tout le monde");
 printf("%s\n",A); // A pointe la ou le malloc() nous a dit (plus haut)
 return 0; // que manque-t-il ici ?
} // ici ca pourrait aller, mais si c'etait dans une boucle......
```

Jeunes cadres encore plus dynamiques

- Supposons que nous `malloc()`-ions paisiblement une fois, mais
- qu'on n'avait pas bien prévu combien (e.g. lecture au clavier non bornée),
- et qu'alors on vient de remplir le tout trop vite.
- Comment « rajouter » alors de l'espace supplémentaire ?
- Utilisant e.g. la fonction `void *realloc(void *P, size_t C)`
  - elle essaie de trouver une nouvelle zone, de taille `C`, et
  - elle y copie ce qu'elle « sait » que se trouve (déjà alloué) à `P`
- Toujours `realloc(void *P, size_t C)` peut aussi diminuer à `C` la zone déjà allouée commençant à `P`
- Note : le gestionnaire de mémoire dynamique « sait ce qu'il fait » –
  - une fois avoir alloué une zone, il en connaît la taille
  - il garde une table de correspondance entre adresses allouées et tailles
  - donc il se souvient de tout
  - c'est ainsi qu'il suffit de donner juste le pointeur à `free()` (ou `realloc()`)
- On peut également utiliser des listes chaînées –
  - cela évite la recopie
  - mais on complique un peu la gestion
  - et on ralentit un tout petit peu le parcours
- Autre note : il existe une sœur de `malloc()` — la fonction `calloc()` — fait la même chose, et en plus y met des zéro partout

La persistence de la mémoire — comment éviter les fuites mémoire ?

- C'est quoi une fuite mémoire d'abord ?
- soit un programmeur (P) et son système (G) de gestion de la mémoire
  - P demande à G (et obtient) une zone (Z) de mémoire
  - P utilise Z pendant un bout de temps
  - ensuite P ne s'en sert plus, mais. . .
  - P oublie de le dire à G !
- et mieux encore, P fait tout cela dans une boucle, « beaucoup » de fois
- donc P n'arrête pas de demander de nouvelles zones Z
- sans jamais les rendre à G
- alors G finit tout simplement par arriver au bout du rouleau
- note au passage :
  - c'est là que Java, Perl et autres nous aident – on n'a pas besoin de s'en souvenir – cela fonctione « magiquement » ;
  - mais leur système générique peut ne pas convenir pour des cas de travail intensif (car il nettoie « quand il veut », « prenant son temps »)
  - on utilise alors le C ou C++ pour tout contrôler,
  - mais on doit « savoir ce qu'on fait »– besoin permanent de finir par rendre ce qu'on a emprunté.

La persistence de la mémoire — donnons un mauvais exemple

```
#include <stdio.h>
#include <stdlib.h>
int main() { // un tableau de chaines de caracteres
  int numberOfIter  = 5;
  int numberOfTabs  = 4;
  int eachTabLength = 4;
  int kIter;
  for(kIter = 0; kIter < numberOfIter; kIter++) {
    char **tab = (char**)malloc(numberOfTabs * sizeof(char*));
    int kTab;
    for(kTab = 0; kTab < numberOfTabs; kTab++) {
      tab[kTab] = (char *)malloc(eachTabLength * sizeof(char));
      int kElem;
      for(kElem = 0; kElem < eachTabLength-1; kElem++) {
        tab[kTab][kElem] = 'A' + kTab + kElem; //whatever
      }
      tab[kTab][eachTabLength-1] = '\0';//pour le printf()
    }
    for(kTab = 0; kTab < numberOfTabs; kTab++) { printf("%s ",tab[kTab]);}
    free(tab); printf("Free to go\n"); // PAS VRAIMENT !
  } // on n'a libere que le "premier niveau"; les chaines-memes non!
  printf("Good bye.\n"); return(0); // si on y arrive...
}
```

La persistence de la mémoire — qu'est-ce qui fallait rajouter ?

- donc, dans le programme précédent – allocation d'un tableau de tableaux de caractères :
  - un `malloc()` pour le tableau `tab` « maître »
  - une boucle avec un `malloc()` pour chaque tableau de caractères (élément de `tab`)
- alors les appels à `free()` – même structure (complémentaire)
  - d'abord une boucle pour chaque tableau de caractères –
  - pour lui faire `free()` dessus
  - seulement ENSUITE on peut libérer le tableau `tab` « maître »
- un peu comme la fermeture des paranthèses dans une expression
- ceci a l'air « simple », mais on peut avoir des situation beaucoup plus compliquées —
- très grand nombre de lignes de code entre le `malloc()` et le `free()`
- ou bien `malloc()` et `free()` dans des fonctions différentes, d'appels très profonds
- il faut donc faire TRÈS ATTENTION pour tout projet sérieux
- deux catégories générales d'erreurs mémoire :
  - pointeurs « perdus » (menant nulle-part – pas d'initialisation, ou inattention zone déjà libérée, ou encore variable qui n'est plus valide, etc.)
  - fuites mémoire (ne pas libérer et continuer à en demander)

Allocation dynamique en C++ – aperçu général

- langage beaucoup plus riche que C
  - opérateur `new` pour l'équivalent de `malloc()` d'un élément
  - opérateur `delete` pour l'équivalent de `free()` d'un élément
  - opérateur `new []` pour l'équivalent de `malloc()` de plusieurs éléments
  - opérateur `delete []` pour l'équivalent de `free()` de plusieurs éléments
  - constructeur, constructeur de recopie, opérateur =, notion de *référence*, polymorphisme etc.
- on peut ainsi obtenir de la mémoire dynamiquement de beaucoup plus de manières
- donc on peut aussi avoir des fuites mémoire ou pointeurs perdus avec beaucoup plus de manières :
  - utiliser `delete` après un `new []`
  - oublie de `delete` dans destructeur alors qu'on a `new` dans le constructeur
  - constructeur de recopie par défault pour classe à pointeurs données-membres, qui ne copie que les valeurs des pointeurs
  - constructeur qui construit à moitié en allouant, et lors d'une erreur, sort en exception sans libérer ce qu'il a alloué
  - rendre une référence d'une zone allouée dynamiquement (au lieu de rendre le pointeur)
  - rendre une référence d'une variable locale
  - passage d'objets dérivés par valeur (donc recopie), donc supprimant involontairement le polymorphisme
  - etc.

Tout d'abord, petit rappel général de C++

- classes –
  - généralisation de la notion de type de variable
  - regrouper données(-membres) et fonctions (méthodes) agissant dessus
  - permettre la réutilisation du code et son développement « incrémental »– héritage et polymophisme
  - notions de `pointeur` et `références` – essentielles
- objets –
  - « incarnations » des classes
  - zones mémoire complexes et « **intelligentes** »
  - doivent être attentivement
    - *construits*
    - *copiés*
    - *détruits*
- pourquoi tout cela ?
- un objet peut garder la trace d'autres objets
- comment ?
- avec des ... pointeurs dessus ! (liste, tableaux, arbre, etc.)
- et alors on doit dupliquer ou nettoyer *judicieusement* en profondeur
- en pensant aux CONSÉQUENCES de chaque CHOIX et de leur ORDRE :
  - on copie le tout ou pas ?
  - on nettoie le tout ou pas ?
  - etc.

Allocation dynamique en C++ – petite classe qui a l'air innocent

```
class MaPetiteChaine {
public:
  MaPetiteChaine(const char* data = "");
  virtual ~MaPetiteChaine();
  virtual const char* to_cstr() const;
private:
  char* m_dataP;
};
MaPetiteChaine::MaPetiteChaine(const char* dataP) :
  m_dataP(new char[strlen(dataP)+1]) {//<=== allocation
  strcpy(m_dataP,dataP);
}
MaPetiteChaine::~MaPetiteChaine() {
  //OOPS, oubli de liberer "m_dataP".   <=== fuite memoire
}
const char* MaPetiteChaine::to_cstr() const {
  return m_dataP;
}
int main() {
  MaPetiteChaine salut("Bonjour tout le monde");
  cout << salut.to_cstr() << endl;
}
```

Allocation dynamique en C++ – d'une erreur à l'autre

- il ne faut pas oublier de soigneusement libérer tout ce qu'on a alloué
- dans l'exemple ci-avant, le destructeur ne libere pas la mémoire
- corrigeons tout cela, et essayons maintenant de copier l'objet

```
MaPetiteChaine::~MaPetiteChaine() { delete [] m_dataP; }
int main() {
  MaPetiteChaine salut("Bonjour tout le monde");
  cout << salut.to_cstr() << endl;

       // on en fait une copie dynamiquement
  MaPetiteChaine* reSalutP = new MaPetiteChaine(salut);

       // on utilise la copie et l'original, tout baigne
  cout << reSalutP->to_cstr() << endl;
  cout << salut.to_cstr()     << endl;

  delete reSalutP; // on nettoie la copie
  reSalutP = 0;    // pour etre bien clean

       // et on devrait pouvoir encore utiliser l'original....
  cout << salut.to_cstr() << endl; // a l'air ok, mais BOUM !
}      // et encore BADABOUM ! mais qu'est-ce qu'on a fait ?..
```

Allocation dynamique en C++ – des pièges visibles et moins invisibles

- nous avons pourtant bien libéré la mémoire dans le destructeur
- et nous avons détruit *la copie* et non pas l'original
- pourquoi alors un BOUM si on essaie de se servir de l'original ?
- et pourquoi un BADABOUM à la fin ?
- parce que
  - la création de la copie ne se fait « magiquement »
  - le compilateur génère le code qui la fait pour nous
  - dans ce cas-ci, ce code ne fait pas « ce qu'il faut »
  - puisqu'en général, le code de recopie généré par le compilateur fait une copie « superficielle »
  - donc les deux objets ont leur donnée-membre pointant vers LA MÊME zone mémoire
  - et alors une fois libérée – il ne faudrait plus y toucher et ne pas la libérer une SECONDE FOIS
- il faut donc explicitement écrire le BON *constructeur de recopie*

```
MaPetiteChaine::MaPetiteChaine(const MaPetiteChaine &chaineIni) :
    m_dataP(0) {
    if(chaineIni . m_dataP && strlen(chaineIni . m_dataP)) {
        m_dataP = new char[strlen(chaineIni . m_dataP)+1];
        strcpy(m_dataP,chaineIni . m_dataP);
    }
}
```

- et le déclarer dans le classe également, bien entendu

Rappels – références, constructeurs en C++

- *Références* –
  - c'est presque « comme les pointeurs », mais « sans l(a plupart d)es ennuis »
  - on les déclare utilisant `&`, ainsi : `TYPE &rA;`
  - il faut les initialiser dans la déclaration (sauf si c'est un paramètre, ou valeur de retour, ou déclaration de donnée-membre dans la déclaration de la classe)
  - on les utilise comme les variable normales
- *Constructeur* –
  - peut prendre des arguments, ou pas
  - initialise l'objet **pendant sa création**
  - peut également allouer des ressources (mémoire, etc.)
  - il peut y en avoir plusieurs variantes (distinguées selon leurs arguments)
- *Constructeur de recopie* –
  - constructeur très particulier
  - un seul – prend un seul argument, de type const ref de la classe
  - but – créer un objet qui est la copie d'un autre objet
- *Opérateur d'assignation* =
  - but – rendre un objet **déjà existant** la copie d'un autre objet
  - prend également une const ref de la classe en tant qu'argument
- *Destructeur*
  - un seul, ne prend pas d'argument, appelé automagiquement avant la suppression de l'objet
  - doit libérer les ressources – e.g. mémoire allouée

Allocation dynamique en C++ – marche à suivre – premiers pas

- il ne faut pas oublier de soigneusement libérer tout ce qu'on a alloué (e.g. dans les destructeurs)
- le compilateur, pour aider, génère des versions *par défaut* de ces méthodes ci-énumérées
- très souvent, ces versions par défaut ne sont pas appropriées
- il ne faut alors pas oublier de les définir, au lieu de laisser le compilateur le faire pour nous (surtout le constructeur de recopie et l'opérateur d'assignation)
- les règles du C++ étant clairement définies, il faut pouvoir répondre à la question : recopie ou passage par référence pour toute partie de code
- ensemble avec la réponse à la question : tel objet est-il encore valide ou pas ? – permet d'éviter d'autres problèmes
  - ne pas rendre en retour de fonction une référence à une variable locale
  - ne pas rendre en retour de fonction la const ref reçue en en paramètre – si la fonction était appelée avec quelque chose de temporaire `maFonction("toto");` alors le temporaire s'en va
  - ne pas oublier de passer les paramètres par référence ou adresse pour bénéficier du polymorphisme
  - ne pas utiliser des tableaux de classes si on souhaite avoir du polymorphisme (il existe par ailleur de nombreux containeurs, bien efficace, dans la STL)
- Bon courage !