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

T03 – Qualité d'analyse d'une fonction
T05 – Formalisme Javadoc
T06 – Documentation de fichier physique
T07 – Documentation de fonction
T10 – Pointeurs
T11 – Références
T13 – Rôle des Pointeurs/références dans les fonctions
T14 – Exécution de programme et gestion mémoire
T15 – Mécanisme d'exécution d'une fonction
T16 – Paramètres de sortie passés par référence
T18 – Paramètres de sortie passés par adresse
T22 – Comparaison des modes de passage
T24 – Règle d'optimisation
T27 – Synthèse des règles de prototypage

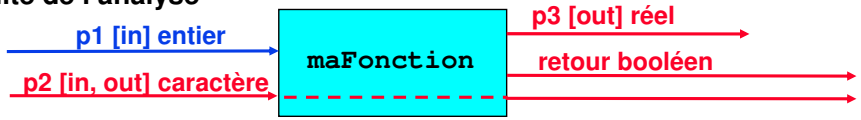
Cours n° 2
Pointeurs, références et prototypage des fonctions

Sommaire

1. ANALYSE DE FONCTION

Qualité de l'analyse d'une fonction

Qualité de l'analyse



Rôle définir le rôle fonctionnel correspondant au traitement de la fonction

Interface définir l'ensemble de ses paramètres formels [in], [out et [in,out]
– les entrées nécessaires au traitement [in]
– les résultats (1) calculés lors du traitement et (2) à transmettre à l'appelant via les paramètres formels [out] et/ou [in,out]

Qualité du nommage des entités (variables, fonctions, types, …)

Qualité du choix du prototype parmi tous les prototypes possibles
(1) ergonomie de l'appel
(2) optimisation du temps d'exécution à l'appel de la fonction

Qualité du choix du type
typer les paramètres au plus proche des données pour
(1) éviter des préconditions
(2) optimiser le temps d'exécution (en évitant la recopie) et la place mémoire

2. DOCUMENTATION DU LOGICIEL

Documentation de code et génération de document

Documents de la production logicielle

Parmi tous les documents à produire en fin de projet (industriels/académique), plusieurs documents techniques (livrables) sont liés au code-source :
la production brute en informatique

Utilité de la documentation technique

Documentation livrée au « client » pour la maintenance (évolutive ou adaptative) des produits logiciels
Capitalisation des connaissances : réutilisation (de tout ou partie) des produits logiciels développés

Intérêt

Produire automatiquement cette documentation à partir des commentaires du code-source

Principe

Utiliser des étiquettes sémantiques (tags) incluses dans les commentaires du code qui permettent d'extraire l'information « pertinente »
Exemples d'utilitaires : Javadoc, Doxygen, …
Formats usuels : HTML (documentation on-line hypertexte) [cf. bibliothèques]
rtf (Rich Text Format), Word, Latex… [cf. livrables de projets]

Formalisme Javadoc
Tags interprétés par Javadoc et Doxygen

Spécifiques aux unités de compilation (fichiers physiques)

`@author` identification de l'auteur
`@version` version (n°, date de création/modif.)

`@file` **nom du fichier physique**
`@brief` **rôle de l'unité de compilation**

Spécifiques aux fonctions

`@brief` **rôle de la fonction**

`@param` nom et description de l'argument
`@return` description du résultat de retour
`@exception` nom et description de l'exception lancée

`@param[in]` **paramètre d'entrée**
`@param[out]` **paramètre de sortie**
`@param[in,out]` **entrée-sortie**

`@pre` **préconditions**
`@post` **postconditions**

Toute entité

Tags additionnels de Doxygen

`@see` identification de lien avec d'autres entités
`@deprecated` indique un élément obsolète, la raison de l'obsolescence et ce qu'il faut utiliser à la place
`@since` version à laquelle l'élément a été ajouté

Documentation de fichier physique – Cartouche

Imposé Déclaration de commentaire /** … */ et **libellés des tags**
Libre Présentation à l'intérieur du commentaire « Javadoc » (/** … */)

Exemple de documentation

```
/**
 * @file gestionHoraire.cpp
 * @brief Utilitaires de gestion d'horaires
 * @author l'équipe pédagogique
 * @version 1.0  08/11/2014
 */
```

Cartouche: documentation du fichier physique `GestionHoraire.cpp`

Tout fichier physique doit avoir un cartouche
`.cpp` **programme principal ou corps de composant – Notion différée**
`.h` **entête (header) de composant – Notion différée**

Documentation de fonction

- **Déclaration de commentaire** `/** … */` **et libellés des tags sont imposés**
- **Présentation libre à l'intérieur du commentaire « Javadoc »** (/** … */)

```
Solution minOf(const int* t, unsigned int size,
               unsigned int i, unsigned int j);
```

Documentation associée au prototype de la fonction ou à la définition de la fonction

```
/** @brief Minimum d'un tableau sur un intervalle [i, j]
 *  @param[in] le tableau
 *  @param[in] sa taille
 *  @param[in] debut de l'intervalle
 *  @param[in] fin de l'intervalle
 *  @return la solution (minimum et index)
 *  @pre size>0, i>=0, i<j et j<size
 */
Solution minOf(const int* t, unsigned int size,
               unsigned int i, unsigned int j);
```

```
struct Solution  {
        int min;
        unsigned int inc
};
```

Remarque
– **Autant de** `@param` **que de paramètres formels,** donnés dans l'ordre des déclarations des paramètres
– **@ return donné en dernier** en cas de paramètre de retour

Règles de prototypage

pour le passage des paramètres d'entrées et de sortie

Connaissances acquises en IAP

Règle de prototypage – Passage des paramètres d'entrée et de sortie

C

R1.		Nature du Paramètre	Prototype	Nature de l'appel pE : paramètre effectif de type T
Paramètre d'entrée p [in] T	**Paramètre formel p de type T (tableaux exclus)**			
	C++ C	Variable	`void f(T p);`	`f(pE);` (variable) `f(2.0);` (littéral) `f(2.*pE);` (expression) *T réel*
Paramètre de sortie p [out] T *ou* p [in, out] T	**Paramètre formel p de type T (tableaux exclus)** [out]/[in,out]			
	C++	*référence*	*non définie*	*non définie*
	C++ C	Pointeur	`void f(T* p);`	`f(&pE);`
	Paramètre de retour [out]			
	C++ C	paramètre de retour	`T f();`	`T res; res=f();`

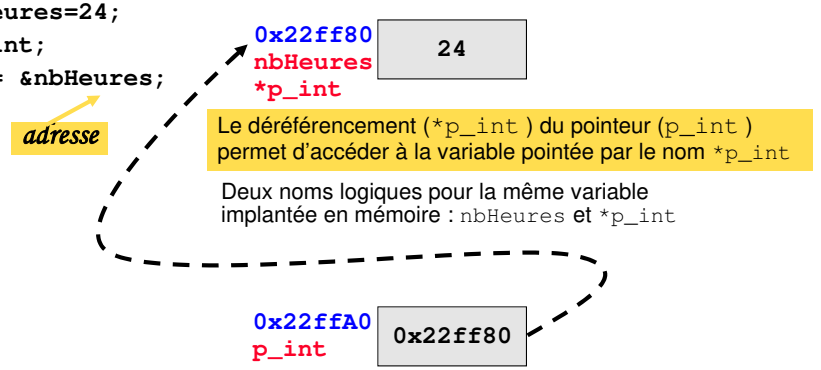
Par ascendance des langages : règle également applicable au C++

Déclaration et initialisation de pointeur
`[const]<id_type>* <id_variable>[=&<id_variable>];`

Rappel – Pointeurs et déréférencement

C C++

```
int nbHeures=24;
int *p_int;
*p_int = &nbHeures;
```



Le déréférencement (`*p_int`) du pointeur (`p_int`) permet d'accéder à la variable pointée par le nom `*p_int`

Deux noms logiques pour la même variable implantée en mémoire : `nbHeures` et `*p_int`

```
int *p_int = &nbHeures;
cout << p_int;     // affiche 0x22ff80,
cout << &p_int;    // affiche 0x22ffA0
cout << *p_int;    // affiche 24
```

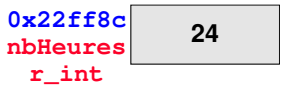

Déclaration et initialisation de référence
`[const]<id_type>& <id_variable>=<id_variable>;`

Références

C++

```
int nbHeures = 24;
int& r_int = nbHeures;
```

la référence *la variable référencée*



Déclaration de référence avec initialisation obligatoire
Une fois initialisée, une référence ne peut plus référencer une autre variable

Deux noms logiques pour la même variable implantée en mémoire
`r_int` est un alias de `nbHeures`

Déclaration et initialisation de référence
`[const]<id_type>& <id_variable>=<id_variable>;`

Références

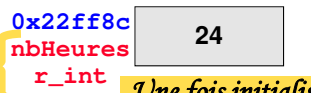
C++

```
int nbHeures = 24;
```

Déclaration et initialisation obligatoire

la référence *la variable référencée*

```
int& r_int = nbHeures;
```



2 noms logiques pour la même variable implantée en mémoire
r_int est un alias de nbHeures

Impossible de déclarer une référence sans l'initialiser

Une fois initialisée, une référence ne peut plus référencer une autre variable

```
int duree = r_int; // duree vaut 24
```


```
nbHeures = 12;      // r_int vaut 12
r_int++;            // r_int et nbHeures vallent 13
r_int = duree;      // r_int et nbHeures vallent 24
```

Déférencement inutile contrairement au pointeur

Rôle des pointeurs et références dans les fonctions

Pointeurs et références sont deux mécanismes de passage des paramètres de sortie ([out] ou [in,out])

C

La notion de référence n'est pas définie,

Seul le **pointeur** permet de passer un paramètre de sortie pour un effet de bord

Mécanisme peu lisible et lourd à gérer

(référencement au niveau appelant et déréférencement au niveau appelé)

C++

Pointeur et références sont définies

La **référence** est **préférée** (au pointeur)

le choix d'un paramètre formel **référence** est **préféré au pointeur**

- simplicité du développement de code des fonctions et de l'appel
- lisibilité du code (côté appelant et côté appelé)

Exécution de programme et gestion mémoire

Au lancement d'un programme

- **Chargement du code compilé [code segment]**
- **Allocation de l'espace mémoire nécessaire pour stocker les variables globales et statiques [data segment]**
- **Initialisation des variables globales qui le nécessitent**
- **Appel de la fonction main**

A chaque appel de fonction

- **Interruption d'exécution du programme appelant**
- **Mise en place du contexte d'exécution propre à la fonction dans la pile d'éxécution [stack]**

  Remarque : plusieurs contextes d'exécution peuvent être empilés dans la pile d'exécution (ajout d'un contexte d'exécution dès qu'une fonction en appelle une autre)
- **Exécution de la fonction dans son contexte d'exécution**

Mécanisme d'exécution d'une fonction

Chaque appel de fonction interrompt le programme appelant

1. **Allocation sur la pile d'exécution de l'espace mémoire nécessaire pour stocker**
   - **l'adresse de retour du programme appelant [code segment]**
   - **les paramètres formels (tous les arguments de la fonction listés entre parenthèses)**
   - **les variables locales à la fonction (variables déclarées dans le corps de la fonction)**
2. **Copie par valeur des paramètres effectifs dans les paramètres formels**
3. **Exécution des instructions du corps de la fonction**
   Visibilité limitée aux variables du contexte d'exécution
   (paramètres formels et variables locales)
   (et aux variables globales [data segment])
4. **Après exécution de la dernière instruction de la fonction (return/fin de bloc fonction)**
   - **sauvegarde du paramètre de retour (éventuel) dans un registre**
   - **désallocation de la mémoire allouée sur la pile jusqu'à l'adresse de retour**
   - **adresse de retour du programme dépilée**
   - **paramètre de retour (éventuel) empilé (*push*) dans la pile d'exécution**
5. **Retour au programme appelant : saut à l'adresse de retour**
   - **le résultat (paramètre de retour) est alors dépilé (*pop*) : la fonction est évaluée**
     la pile revient à son état d'origine (avant l'appel de la fonction) :
     plus de visibilité des paramètres formels et des variables locales de la fonction
   - **le programme continue à s'exécuter en séquence**

C++

Pile d'exécution – Paramètre de sortie passé par référence

```
// main01.cpp
#include <iostream>
using namespace std;

void saisie (float& px);
int main() {                          L10
  float pi = 3.14;                    L11
  cout << "pi = " << pi << endl;      L12
  saisie(pi);                         L13
  cout << "pi = " << pi << endl;      L14
  return 0;                           L15
}
void saisie(float& px) {              L20
  cout << "px =  " << px << endl;     L21
  cout << "entrez un nombre : ";      L22
  cin >> px;                          L23
  cout << "px = " << px << endl;      L24
  return;                             L25
}
```

```
pi = 3.14
px =  3.14
entrez un nombre : 3.14159
px = 3.14159
pi = 3.14159
```

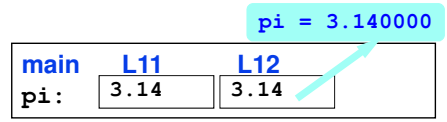
Pile d'exécution – Paramètre de sortie passé par référence

C++

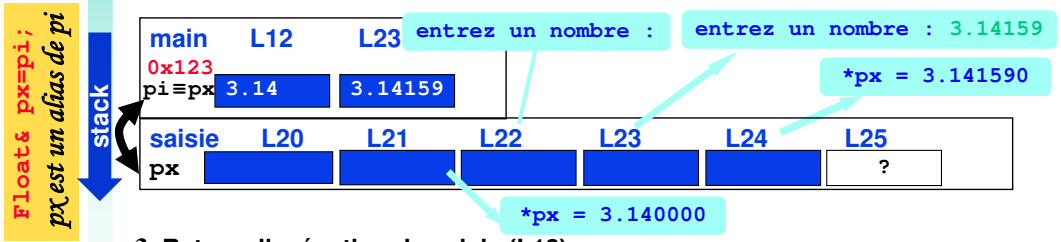
```
void saisie(float& px) {          L20
  cout << "px = " << px << endl;  L21
  cout << "entrez un nombre : ";  L22
  cin >> px;                      L23
  cout << "px = " << px << endl;  L24
  return;                         L25
}

int main() {                      L10
  float pi = 3.14;                L11
  printf("pi = %f\n",pi);         L12
  saisie(pi);                     L13
  printf("pi = %f\n",pi);         L14
  return 0;                       L15
}
```

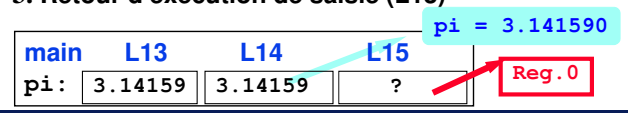
1. Exécution du programme (main)



2. Interruption d'exécution du main (L13) - appel de saisie



3. Retour d'exécution de saisie (L13)



Pile d'exécution – Paramètre de sortie passé par adresse

C C++

```
// main01.c
#include <stdio.h>

void saisie (float* px);
int main() {                      L10
  float pi = 3.14;                L11
  printf("pi = %f\n",pi);         L12
  saisie(&pi);                    L13
  printf("pi = %f\n",pi);         L14
  return 0;                       L15
}
void saisie(float* px) {          L20
  printf("*px = %f\n", *px);      L21
  printf("entrez un nombre : ");  L22
  scanf("%f", px);                L23
  printf("*px = %f\n", *px);      L24
  return;                         L25
}
```

```
pi = 3.140000
*px = 3.140000
entrez un nombre : 3.14159
*px = 3.141590
pi = 3.141590
```

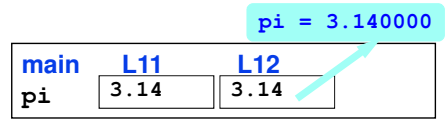

Pile d'exécution – Paramètre de sortie passé par adresse

C C++

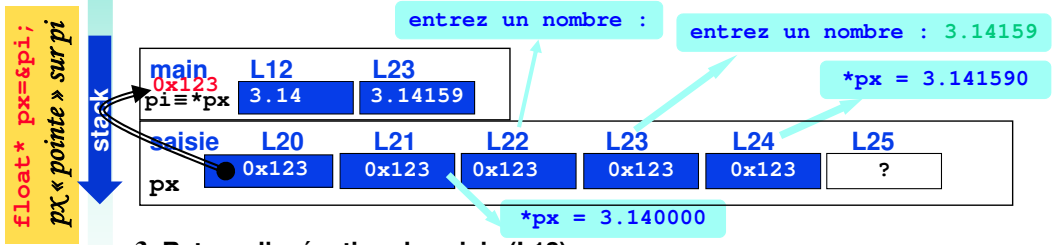
```
void saisie(float* px) {          L20
  printf("*px = %f\n", *px);      L21
  printf("entrez un nombre : ");  L22
      fflush(stdout);
  scanf("%f", px);                L23
  printf("*px = %f\n", *px);      L24
  return;                         L25
}

int main() {                      L10
  float pi = 3.14;                L11
  printf("pi = %f\n",pi);         L12
  saisie(&pi);                    L13
  printf("pi = %f\n",pi);         L14
  return 0;                       L15
}
```

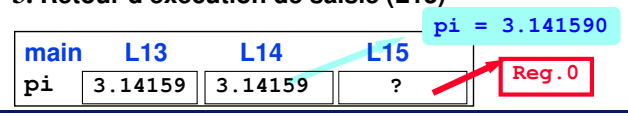
1. Exécution du programme (main)



2. Interruption d'exécution du main (L13) - appel de saisie



3. Retour d'exécution de saisie (L13)



Règle de prototypage – Passage des paramètres d'entrée et de sortie

C C++

R1.		Nature du Paramètre	Prototype	Nature de l'appel pE : paramètre effectif de type T
Paramètre d'entrée p [in] T	**Paramètre formel p de type T (tableaux exclus)**			
	C++ C	variable	`void f(T p);`	`f(pE);` (variable) `f(2.0);` (littéral) `f(2.*pE);` (expression) *T réel*
Paramètre de sortie p [out] T *ou* p [in,out] T	**Paramètre formel p de type T (tableaux exclus)**			
	[out]/[in,out] C++	référence	`void f(T& p);`	`f(pE);` (variable référencée)
	C++ C	pointeur	`void f(T* p);`	`f(&pE);` (adresse du paramètre effectif)
	Paramètre de retour			
	[out] C++ C	paramètre de retour	`T f();`	`T res; res=f();`

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

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(…)`	

Comparaison des modes de passage

```
x=3.14  y=9.81
x=9.81  y=3.14
```

C Passage par pointeur

```
// Extrait de main02.c

void swap(float* u, float* v);

int main() {
   float x = 3.14, y = 9.81;
   printf("x=%.2f  y=%.2f\n", x, y);
   swap(&x, &y);
   printf("x=%.2f  y=%.2f\n", x, y);
}

void swap(float* u, float* v) {
    float aux = *u;
    *u = *v;
    *v = aux;
}
```

référencements à l'appel

déréférencements dans le code de la fonction

C++ Passage par référence

```
// Extrait de main02.cpp

void swap(float& u, float& v);

int main() {
   float x = 3.14, y = 9.81;
   cout<<"x="<<x<<" " <<"y="<<y<<endl;
   swap(x, y);
   cout<<"x="<<x<<" " <<"y="<<y<<endl;
}

void swap(float& u, float& v) {
   float aux = u;
   u = v;
   v = aux;
}
```

Lisibilité
- à l'appel de la fonction
- du code de la fonction

Règle d'optimisation

pour le prototypage des fonctions

et s'appliquant aux paramètres d'entrée

Règle d'optimisation – Prototypage des paramètres d'entrée [in] (1/2)

Chaque appel de fonction ralentit l'exécution d'un programme

Principale raison

le **mécanisme** de **passage par valeur**
recopie des paramètres effectifs dans les paramètres formels

Dans le cas d'un paramètre d'entrée,
la recopie du paramètre effectif est fonction de sa taille mémoire
Elle peut être coûteuse en mémoire et en temps

Cette **règle** est a priori **applicable** pour tout paramètre **non natif**
dont la taille mémoire est supérieure à 4 octets (adresse pour un PC -32 bits)

Exemple :

```
struct AdPostale {char rue[20], ville[20]; int codePostal;};
void afficher(AdPostale a);              // ? octets recopiés
void afficher(const AdPostale* a);       // ? octets recopiés
void afficher(const AdPostale& a);       // ? octets recopiés
```

Règle d'optimisation – Prototypage des paramètres d'entrée ([in]) (2/2)

Objectif de la règle

(1) **Optimiser** la **recopie** en utilisant
- en **C++,** un passage par **référence** (alias)
- en **C,** un passage par **adresse** (4 octets) du paramètre

(2) **Protéger** le **paramètre d'entrée** par le modifieur **`const`**
toute modification du paramètre dans le code
entraîne une erreur à la compilation

Cette **règle** est a priori **applicable** pour tout paramètre **non natif**
dont la taille mémoire est supérieure à celle d'une adresse (4 octets)
(PC à 32 bits d'adressage)

Remarque Même problème pour le paramètre de retour
On préfèrera dans le cas d'un type de retour >> 4 octets
un paramètre formel de sortie [out]


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

2/2

Exemples d'analyse de fonction pour le prototypage en C++

Analyse d'une fonction donnant le minimum de 2 dates (date la plus ancienne)

d1 [in] date
d2 [in] date
→ **min** → leMin [out] date

Choix possibles pour le prototypage C++

```
a) Date min(const Date& d1, const Date& d2);
ou
b) void min(const Date& d1, const Date& d2, Date& leMin);
```

Même principe d'appel que précédemment

Préférence Ici, le paramètre de retour implique 12 octets recopiés au retour de l'appel de la fonction
La solution (b) est acceptable

Lorsque la **taille** du type considéré est **bien supérieure**, **préférence** du passage par le **paramètre formel**

C Même analyse pour le prototype en remplaçant la référence (`&`) par le pointeur (`*`)

Note (1) : Tableau (sauf tableau de caractères)
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	[in, out] ou [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**

Ce que vous avez appris aujourd'hui…

dans le cadre de ce dernier cours sur les fonctions…

La qualité d'analyse d'une fonction

Le formalisme Javadoc à base de tags sémantique pour la documentation de fonction et des entités de compilation (cartouches)

La spécificité et le rôle des **pointeurs** et des **références** dans les fonctions relativement **aux effets de bord** en Langage C et C++

La comparaison de leur mode de passage (pointeur et référence) et du code de l'appelé

La **règle d'optimisation** pour les paramètres passés en entrée

Et enfin, **la synthèse des règles de prototypage** et d'optimisation en langage C et en C++

La semaine prochaine :
l'allocation dynamique de mémoire