heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Analyse et programmation 2

Concepts avancés

heig-vd

Thèmes abordés

- La récursivité
  - Une approche de programmation élégante pour certains problèmes
- Les classes de stockage
  - Comment préciser où doit se trouver une variable…
- Accélérer les programmes
  - L'optimisation de code

heig-vd

La récursivité
Définition

- On parle de récursivité lorsque
  - Un algorithme s'utilise lui même.
  - Ce concept se traduit en programmation par une fonction qui s'appelle elle même.
- Le langage C autorise les appels récursifs.

heig-vd

La récursivité
Un premier exemple : le calcul de factorielles

- Rappel : définition de la factorielle comme produit de facteurs
  - n! = 1 x 2 x 3 x … x n
  - 5! = 1 x 2 x 3 x 4 x 5 = 120
- Définition de la factorielle comme suite récurrente

  $U_0 = 1$
  $U_n = n * U_{n-1}$

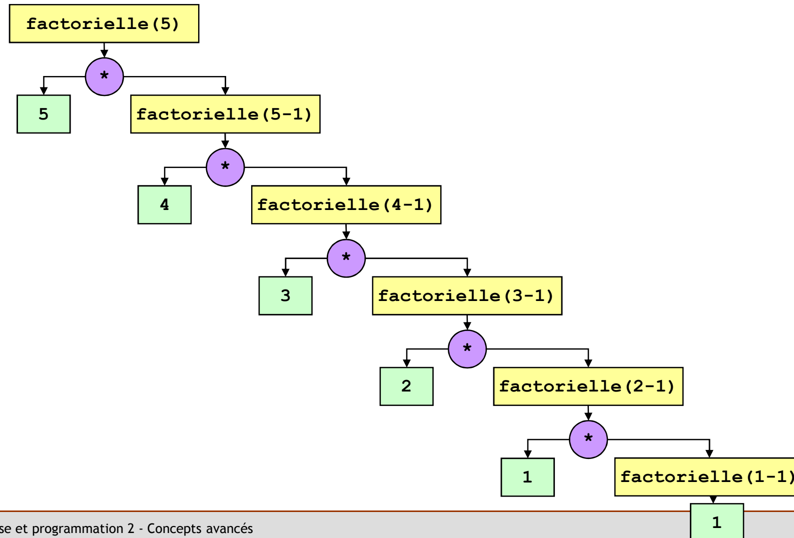
  5! = 5 x 4! = 5 x 4 x 3! = 5 x 4 x 3 x 2! = 5 x 4 x 3 x 2 x 1! = 5 x 4 x 3 x 2 x 1 x 1
- Programmation récursive

```
long factorielle(long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorielle(n - 1);
}
```

heig-vd

La récursivité

Déroulement de l'exécution

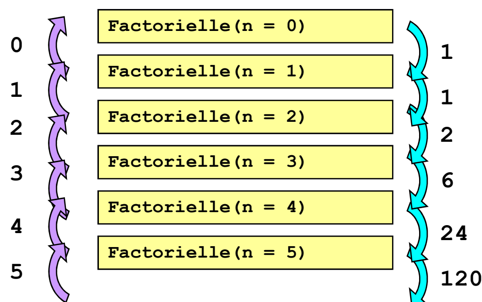


heig-vd

La récursivité

Effet en mémoire : paramètres et variables locales

Appels Retours Etat de la Pile



Adresse	Pile
A020	
A01C	n = 0
A018	n = 1
A014	n = 2
A010	n = 3
A00C	n = 4
A008	n = 5
A004	? ? ?
A000	? ? ?

heig-vd

La récursivité

Effet en mémoire - analyse

- C'est toujours la même fonction factorielle qui s'exécute
  - Mais avec à chaque fois un paramètre effectif différent sur la pile.
  - Il y a donc simultanément plusieurs valeurs du même paramètre formel n sur la pile
    - Chacun correspondant à un appel imbriqué de factorielle.
    - Un nouvel exemplaire des variables locales est également recréé sur la pile pour chaque appel.
- Conséquences possibles
  - Si la fonction est récursive trop longtemps, la pile « déborde ».
  - Elle va écrire au delà de la zone maximale réservée.
    - Sur un micro contrôleur, peut même effacer le programme.
    - Sur un PC, le système d'exploitation détecte l'anomalie et tue le processus.

heig-vd

La récursivité

Règles à respecter

- Une fonction récursive doit s'appeler
  - En transmettant des valeurs modifiées des paramètres reçus.
  - Sinon, récursion infinie.
- Elle doit contenir au moins une condition de sortie
  - Donc un test conditionnel avec un retour sans appel récursif.

heig-vd

La récursivité
Dérécursivisation

- Il est toujours possible de rendre un algorithme récursif non récursif
  - La récursivité consomme plus d'espace sur la pile.
    - paramètres, variables locales, adresse de retour.
  - La copie de toutes ces valeurs prend du temps.
    - factorielle(15) : itératif 75 ns, récursif 125 ns (+ 66 %) sur Pentium M 1.66 GHz
- Algorithmes simples comme les suites récurrentes
  - L'utilisation de variables d'accumulation permet d'éliminer la récursivité.
- Algorithmes plus complexes
  - On peut rendre un algorithme non récursif en gérant une « pile » par programme.
  - Cela élimine l'appel récursif.
- Quand les performances ne sont pas critiques
  - L'algorithme récursif est élégant, vite écrit et plus vite mis au point.

heig-vd

La récursivité
Dérécursivisation - Illustration sur la factorielle

```
long factorielle(long n)
{
    long i;
    long resultat; // variable d'accumulation
    resultat = 1;
    for (i = 1; i <= n ; i++)
        resultat = resultat * i;
    return resultat;
}
```

La récursivité
Application typiques

- Nous avons déjà vu
  - Le calcul des suites récurrentes.
- Autres domaines d'application typique
  - Le parcours des structures de données récursives (APR2).
    - Comme les structures arborescentes.
  - Exemple
    - Afficher la liste des fichiers d'un répertoire et de ses sous répertoires.

La récursivité
Exemple : la suite de Fibonacci

- Suite numérique définie par le mathématicien Fibonacci
  - Représente la croissance d'une population de lapins sous certaines hypothèses.
  - Possède des propriétés mathématiques intéressantes.
  - Illustration très célèbre de la récursivité :
    - Hello World de la récursivité
    - … mais (très très) peu d'utilité dans la pratique informatique …
- Formulation mathématique
  - $f(0) = 0$
  - $f(1) = 1$
  - $f(n) = f(n - 1) + f(n - 2)$

heig-vd

La récursivité

Exemple : la suite de Fibonacci - le code source en C

- Code source

```
long fibonacci(long n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- Résultats
  - `n :              0   1   2   3   4   5   6   7   ...`
  - `Fibonacci(n) :   0   1   1   2   3   5   8   13  ...`

heig-vd

La récursivité

Exemple : le parcours récursif du contenu d'un répertoire

- Nous ne verrons que la formulation algorithmique en pseudo code

```
ListerContenu(Dossier)
DEBUT
    AfficherNom("Dossier : ", Dossier)
    Fichier = ChercherPremierFichier(Dossier)
    TANT QUE Fichier != Vide FAIRE
        SI EstUnDossier(Fichier) ALORS
            ListerContenu(Fichier)
        SINON
            AfficherNom(Fichier)
        FIN SI
        Fichier = ChercherFichierSuivant(Dossier)
    FIN TANT QUE
FIN
```

heig-vd

La récursivité

La récursivité croisée

- Récursivité simple
  - Une fonction qui s'appelle elle même.
- Récursivité croisée
  - Une fonction A qui appelle une fonction B qui appelle la fonction A.
- Déclaration
  - Déclarer les prototypes utilisés avant l'appel des fonctions.

heig-vd

La récursivité

La récursivité croisée - exemple

```
long fonction_b(long n);

long fonction_a(long n)
{
    if (n == 0)
        return 1;
    else
        return 2 * fonction_b(n - 1);
}

long fonction_b(long n)
{
    if (n == 0)
        return 1;
    else
        return 3 * fonction_a(n - 1);
}
```

heig-vd

La récursivité

Recommandations

- Veiller à rendre limpide le code récursif
  - Les solutions récursives ne sont pas toujours triviales à analyser.
  - Utilisez la récursivité après une analyse soigneuse et une formulation mûre.
- Attention aux récursions infinies
  - Un programme qui rentre dans une récursion infinie devient incontrôlable.
  - Il finit par planter par débordement de pile.

heig-vd

Les classes de stockage

Compléments sur les déclarations de variable

- Structure générale d'une déclaration de variable

```
classe type identificateur = valeur_initiale ;
```

- Classe est une combinaison des formes suivantes
  - **`extern`**
  - **`static`**
  - **`const`** ou **`auto`** ou **`register`**
  - **`volatile`**

heig-vd

Les classes de stockage
extern

- Utilisation
  - Avant une déclaration de variable.
  - Avant un prototype de fonction.
- Signification
  - Dans les deux cas, indique au compilateur que l'objet qui suit existe quelque part.
  - Une déclaration commençant par extern ne crée rien.
  - Elle donne seulement une indication d'existence d'un objet au compilateur.

heig-vd

extern
Les variables globales partagées

- Une variable globale
  - Est visible à partir de sa déclaration seulement.
  - Dans le fichier où elle est déclarée seulement.
- Il est cependant possible de déclarer son existence
  - En utilisant le mot réservé **extern**.

```
extern double coefficient_global;
```

  - Signification pour le compilateur
    - Il existe une variable **coefficient_global** de type **double**.
    - Elle est déclarée ailleurs.
- Surtout intéressant avec la compilation séparée.

heig-vd

extern

Les variables globales partagées - illustration

```
extern double coefficient_global;
double f(double x);

int main()
{
    . . .
}

double f(double x)
{
    double resultat;
    resultat = coefficient_global * x;
    coefficient_global = x;
    return resultat;
}

double coefficient_global =1.0;
```


heig-vd

Les classes de stockage

static

- Utilisation
  - Avant la déclaration d'une variable locale ou globale.
  - Avant un prototype de fonction.
- Signification
  - Pour les variables déclarées à l'intérieur des fonctions
    - La variable devient une variable globale.
    - Invisible hors de la fonction.

```
// initialisation faite une seule fois au démarrage
static int j = 4;
```

  - Pour les variables globales déclarées hors d'une fonction
    - La variable globale est invisible hors du module.
  - Pour les fonctions
    - La fonction est invisible hors du module.

static

Les variables globales « cachées »

- Les variables globales
  - Elles sont visibles par toutes les fonctions.
  - Elles utilisent un nom qui n'est plus ensuite disponible pour d'autres variables.
  - Risque de conflit
    - Si deux fichiers sources d'un même projet déclarent chacun une variable globale de même nom.
    - Problème à l'édition de lien.
- Solution : les variables « static »

static

Les variables globales « cachées »

- Les variables « static »
  - Elles sont déclarées à l'intérieur d'une fonction.
  - La déclaration est précédée par le mot « static ».
  - Elles fonctionnent comme des variables globales.
  - Mais elles sont inaccessibles hors de la fonction.
  - Il n'y a plus de risque de conflit de nom (le nom est caché).
- Déclaration

```
double f(double x)
{
    // l'initialisation n'a lieu qu'une fois :
    static double coefficient_global =1.0;
    double resultat;
    resultat = coefficient_global * x;
    coefficient_global = x;
    return resultat;
}
```

heig-vd

static

Les fonctions « cachées »

```
// Fichier testfloat.c

float max(float a, float b)
{
    return a > b ? a : b;
}

void testfloat()
{
    float a, b;
    printf("a:");
    scanf("%f", &a);
    printf("b:");
    scanf("%f", &b);
    printf("Max : %f\n", max(a, b));
}
```

```
// Fichier testint.c

int max(int a, int b)
{
    return a > b ? a : b;
}

void testint()
{
    int a, b;
    printf("a:");
    scanf("%d", &a);
    printf("b:");
    scanf("%d", &b);
    printf("Max : %d\n", max(a, b));
}
```

```
// Fichier main.c

void testfloat();
void testint();

int main()
{
    testfloat();
    testint();
    _getch();
}
```

Nom dupliqué dans le code objet
Erreur de liaison

heig-vd

static

Les fonctions « cachées »

```
// Fichier testfloat.c

static float max(float a, float b)
{
    return a > b ? a : b;
}

void testfloat()
{
    float a, b;
    printf("a:");
    scanf("%f", &a);
    printf("b:");
    scanf("%f", &b);
    printf("Max : %f\n", max(a, b));
}
```

```
// Fichier testint.c

static int max(int a, int b)
{
    return a > b ? a : b;
}

void testint()
{
    int a, b;
    printf("a:");
    scanf("%d", &a);
    printf("b:");
    scanf("%d", &b);
    printf("Max : %d\n", max(a, b));
}
```

```
// Fichier main.c

void testfloat();
void testint();

int main()
{
    testfloat();
    testint();
    _getch();
}
```

Fonction static, cachée
Plus d'erreur de liaison

heig-vd

Les classes de stockage
const

- Utilisation
  - Avant la déclaration d'une constante locale ou globale.
- Signification
  - L'objet est non modifiable.
    - Interdit la modification de l'objet déclaré par du code C.

```
const int nombre_essai = 3;
```


heig-vd

Les classes de stockage
auto

- Utilisation
  - Avant la déclaration d'une variable locale ou globale.
- Signification
  - L'objet est une variable locale créée sur la pile.
    - Sans préciser, c'est le mode utilisé pour toutes les variables locales.
    - Conséquence : on n'utilise pas auto explicitement dans la pratique.

```
// initialisation faite à chaque entrée dans la fonction
auto int j = 4;
int j = 4; // ces deux formes sont équivalentes
```

heig-vd

Les classes de stockage
register

- Utilisation
  - Avant la déclaration d'une variable locale.
- Signification
  - L'objet est une variable locale.
  - On demande au compilateur de placer cette variable dans un registre CPU plutôt que sur la pile.
  - Conséquence : amélioration des performances.
  - S'il n'y a pas assez de registres disponibles, traité comme auto.
  - Les compilateurs optimiseurs décident automatiquement de placer des variables dans les registres CPU.

```
register int j;
```


heig-vd

Les classes de stockage
volatile

- Utilisation
  - Avant la déclaration d'une variable.
- Signification
  - L'objet est une variable.
  - Son contenu est lié à du contrôle de périphériques externes.
    - Les valeurs lues peuvent changer entre 2 lectures successives.
    - Les valeurs écrites ont un effet sur le matériel.
  - Le compilateur n'effectue aucune optimisation avec les variables de ce type.
    - Lorsqu'on utilise la valeur, elle est relue à chaque fois, même si une valeur récente se trouve dans un registre CPU.
    - Lorsqu'on affecte la variable, elle est immédiatement écrite.

```
volatile int digital_outputs;
```

heig-vd

Optimisation du code
Présentation

- Qu'est ce que l'optimisation
  - Transformer un code pour accélérer son exécution.
  - Ou pour diminuer l'espace mémoire qu'il occupe.
- Quand faut-il optimiser un algorithme ?
  - Quand les performances atteintes ne sont pas satisfaisantes.
  - Quand la cible ne peut pas contenir le programme développé.
- Que faut-il optimiser ?
  - Localiser les portions de code critiques.
  - Se focaliser sur ces portions.
  - 80 % du temps d'exécution est passé dans 20 % du code.
  - Il serait absurde de tout optimiser.
  - Un code optimisé peut devenir moins lisible.

heig-vd

Optimisation du code
Comment optimiser ?

- Remplacer des opérations lentes par des rapides
  - Calculer en `int` plutôt qu'en `float`
  - Selon le compilateur
    - `i++` plutôt que `i = i + 1`
    - `i << 2` plutôt que `i = i * 4`
- Simplifier des expressions
  - Calcul du polynôme $y = a * x^3 + b * x^2 + c * x + d$
    - y = a * x * x * x + b * x * x + c * x + d // 6 multiplications, 3 additions
    - y = x * (x * (x * a + b) + c) + d // 3 multiplications, 3 additions
- Précalculer des valeurs

```
// 4 calculs de sin(x) :
y = sin(x) + 1 / sin(x) - sin(x) * sin(x)
// 1 seul calcul de sin(x) :
a = sin(x)
y = a + 1 / a - a * a
```

heig-vd

Optimisation du code
Comment optimiser ?

- Trouver un algorithme plus efficace
  - Exemple : calculer la somme des entiers de 1 à 100’000
  - Algorithme 1

```
long i;
const long n = 100000;
long somme = 0;
for (i = 1; i <= n; i++)
    somme += i;
// 470 µs (sur Pentium M 1.6 GHz)
```

  - Algorithme 2

```
const long n = 100000;
somme = n * (n + 1) / 2;
// même résultat, mais 5 ns (sur Pentium M 1.6 GHz)
// Donc, va 100’000 fois plus vite
```


heig-vd

Optimisation du code
Les compilateurs optimiseurs

- Les compilateurs font automatiquement certaines optimisations
  - Avant d’optimiser son code, étudier ce que le compilateur fait déjà
  - Tenir compte de la plateforme cible
  - Il serait dommage de polluer le code sans en tirer aucun profit.
- Code compilé - Visual studio 2005

```
    int i = 1;
004113FE  mov         dword ptr [i],1
     i = i << 2;
00411405  mov         eax,dword ptr [i]
00411408  shl         eax,2
0041140B  mov         dword ptr [i],eax
     i = i * 4;
0041140E  mov         eax,dword ptr [i]
00411411  shl         eax,2
00411414  mov         dword ptr [i],eax
```

heig-vd

Optimisation de code
Les techniques efficaces

- Les techniques d'optimisation
  - A utiliser quand il y a vraiment des problèmes de performances.
- Les compilateurs d'aujourd'hui savent souvent bien optimiser
- Les problèmes d'optimisation dépendent de la cible
  - Multiplication double sur un PC : 5 ns
  - Sur un micro contrôleur 8 bits : >1 ms, donc 200'000 fois plus lent.

heig-vd

Qu'avons-nous appris ?

- La récursivité
  - Approche élégante pour certains algorithmes.
  - Sera approfondie en APR 2
- Les classes de stockage
  - Préciser dans quelle mémoire doit se trouver une variable.
  - Accès aux périphériques matériels (volatile).
- L'optimisation de code
  - Gains potentiels énormes en trouvant un bon algorithme au départ.
  - D'autres gains peuvent être réalisés en choisissant bien les types de données et les instructions sur certaines plateformes.
  - Certains compilateurs savent déjà bien optimiser.

Vos questions