Récursivité des actions [rc]
Exercices de cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Exercices de cours (Solution)

Mots-Clés Récursivité des actions ■
Difficulté ●●○ (3 h) ■

1 Appréhender le cours

1.1 Fonction factoriel / pgfactoriel

Objectif

Le factoriel d'un entier naturel n est défini par la relation de récurrence :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \end{cases}$$

Donnez le type de récursivité, la condition d'arrêt et la récurrence.

Solution simple

C'est une récursivité simple de condition d'arrêt $0! = 1$ et de récurrence :

$$n! = n \cdot (n-1)!$$

Déduisez une fonction récursive `factoriel(n)` qui calcule et renvoie le factoriel de `n` (entier).

Validez votre fonction avec la solution.

Solution C++ @[pgfactoriel.cpp]

```
/**
  Fonction naïve récursive
  @param[in] n - un entier
  @return Factoriel de n
*/

int factoriel(int n)
{
  return (n <= 0 ? 1 : n * factoriel(n - 1));
}
```

Explicitez le calcul de `fac(3)` (abrégé de `factoriel(3)`).

Solution simple

Chaque fonction s'exécute dans son environnement de données associé : lors de l'exécution du calcul de `fac(0)|, il y a quatre variables \lstinline`n@ définies avec quatre valeurs différentes dans chaque environnement de données. Précisons le déroulement des calculs :

1. Le calcul de `fac(3)` est lancé (étape 1).

2. Pour évaluer la valeur de `3*fac(2)`, le calcul de `fac(3)` se suspend pour connaître la valeur de `fac(2)` (étape 2).
3. Le calcul de `fac(2)` se suspend à son tour pour évaluer `fac(1)` (étape 3), lequel se suspend également pour évaluer `fac(0)`.
4. Grâce à la condition d'arrêt, `fac(0)| renvoie 1: cette valeur remplace \lstinline`fac(0)@ dans le calcul suspendu de `fac(1)`.
5. Le calcul de `fac(1)` peut reprendre là où il était suspendu et s'effectuer, `fac(1)` renvoie 1, et le calcul de `fac(2)` peut reprendre et s'effectuer pour produire 2, puis le calcul de `fac(3)` reprend et s'effectue pour produire 6.

Finalement, on a calculé :

```
fac(3) -> 3*fac(2) -> 3*2*fac(1) -> 3*2*1*fac(0) -> 3*2*1*1 -> 6
```

Écrivez une fonction récursive terminale `facRt(n,a)` qui calcule et renvoie le factoriel de `n` (entier), l'entier `a` étant le résultat.

Écrivez une fonction maître `factorielRt(n)` qui lance la fonction récursive terminale.

Validez vos fonctions avec la solution.

Solution C++ @[pgfactoriel.cpp]

```
/**
  Fonction récursive terminale
  @param[in] n - un entier
  @param[in] a - un entier
  @return Factoriel de n
*/

int facRt(int n, int a)
{
  return (n <= 0 ? a : facRt(n - 1,a * n));
}

/**
  Fonction récursive terminale maître
  @param[in] n - un entier
  @return Factoriel de n
*/

int factorielRt(int n)
{
  return facRt(n,1);
}
```

Écrivez un programme qui saisit un entier et teste les fonctions.

1.2 Boucles récursives / pgboucles

Écrivez le **profil** d'une procédure récursive `afficherBoucleUp(n1,n2)` qui affiche les entiers de `n1` à `n2` en ordre croissant.

Donnez le type de récursivité, la condition d'arrêt et la récurrence.

Solution simple
C'est une récursivité simple de condition d'arrêt `n1>n2` et de récurrence `n1+1`.

Déduisez le corps de la procédure récursive.
Exemple :

```
afficherBoucleUp(-3,5) ==> -3 -2 -1 0 1 2 3 4 5
```

Validez votre procédure avec la solution.

Solution C++ @[pgboucles.cpp]

```
/**
  Procédure afficherBoucleUp
  @param[in] n1 - un entier
  @param[in] n2 - un entier
*/

void afficherBoucleUp(int n1, int n2)
{
  if (n1 <= n2)
  {
    cout<<n1<<endl;
    afficherBoucleUp(n1 + 1,n2);
  }
}
```

En utilisant les mêmes principes, écrivez une procédure récursive `afficherBoucleDn(n1,n2)` qui affiche les entiers de `n1` à `n2` en ordre décroissant. Exemple :

```
afficherBoucleDn(-3,5) ==> 5 4 3 2 1 0 -1 -2 -3
```

Validez votre procédure avec la solution.

Solution C++ @[pgboucles.cpp]

```
/**
  Procédure afficherBoucleDn
  @param[in] n1 - un entier
  @param[in] n2 - un entier
```

```
*/

void afficherBoucleDn(int n1, int n2)
{
  if (n1 <= n2)
  {
    afficherBoucleDn(n1 + 1,n2);
    cout<<n1<<endl;
  }
}
```

Écrivez un programme qui saisit deux entiers et teste les procédures.

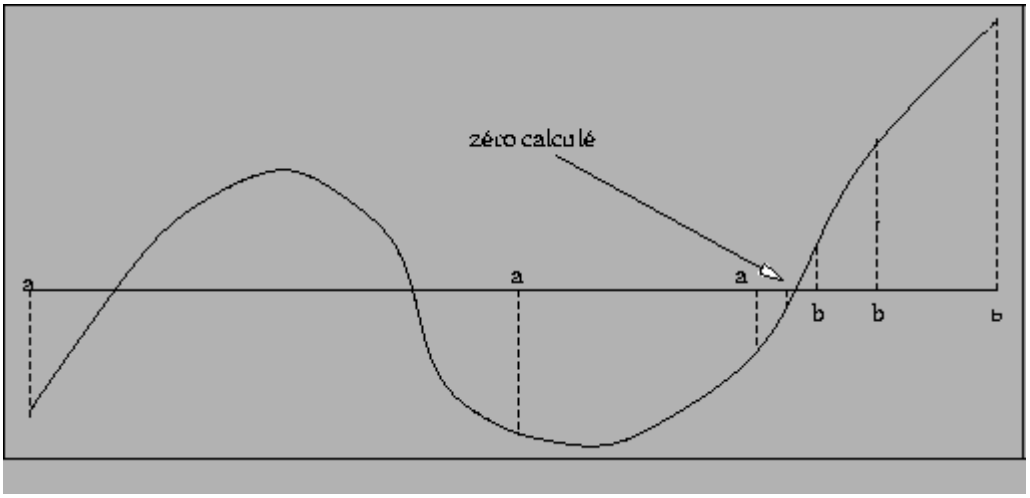
Validez votre programme avec la solution.

Solution C++ @[pgboucles.cpp]

```
int main()
{
  int n1, n2;
  cout<<"n1? ";
  cin>>n1;
  cout<<"n2 (>=n1)? ";
  cin>>n2;
  afficherBoucleUp(n1,n2);
  afficherBoucleDn(n1,n2);
}
```

1.3 Recherche d'un zéro / pgzerodich

Objectif
On cherche à calculer un zéro d'une fonction réelle continue f sur un intervalle $[a, b]$, prenant des valeurs de signes opposés aux extrémités. Le théorème des valeurs intermédiaires assure l'existence d'un zéro. L'idée de la dichotomie est de chercher un zéro sur $[a, (a+b)/2]$ ou bien sur $[(a+b)/2, b]$, selon le signe de $f((a+b)/2)$.



Écrivez une fonction récursive `zerodich(a,b,epsilon)` qui calcule et renvoie un zéro d'une fonction réelle continue `f` sur un intervalle `[a,b]` à `epsilon` près.

Validez votre fonction avec la solution.

Solution C++ @[pgzerodich.cpp]

```
/**
  Suppose que f(a)*f(b) < 0
  @param[in] a - un réel
  @param[in] b - un réel
  @param[in] epsilon - précision
  @return le zéro de f entre [a,b] à epsilon près
*/

double zerodicho(double a, double b, double epsilon)
{
  double c = (a + b) / 2.0;
  double fc = f(c);
  if (std::abs(fc) < epsilon)
  {
    return c;
  }
  else if (f(a) < fc)
  {
    return zerodicho(a,c);
  }
  else
  {
```

```
    return zerodicho(c,b);
  }
}
```

1.4 Carré d'un entier / pgcarrei

Écrivez une fonction récursive `carrei(n)` qui calcule et renvoie le carré d'un entier positif `n` par la méthode des impairs.

Orientation

Analysons le problème :

Étape 1 Commençons par chercher l'expression commune à tous les appels récursifs de la fonction considérée. Prenons quelques exemples :

```
4^2 = 1 + 3 + 5 + 7
3^2 = 1 + 3 + 5
2^2 = 1 + 3
1^2 = 1
```

Nous pouvons les réécrire de la manière suivante :

```
4^2 = 3^2 + 7
3^2 = 2^2 + 5
2^2 = 1^2 + 3
1^2 = 1
```

ce qui donne l'expression commune à chaque étape : le carré d'un valeur `n` est `(2*n-1)` additionné au carré de `(n-1)`.

Vérifions :

```
carrei(4) = carrei(3) + 7
carrei(3) = carrei(2) + 5
```

Étape 2 Trouvez le point d'arrêt de la récursivité.
On connaît la valeur de 1^2 qui vaut 1. Donc 1 est notre point d'arrêt.

Étape 3 Vérifiez qu'à chaque appel, la fonction se rapproche du point d'arrêt.
Le point de départ de notre calcul est une valeur n strictement positive. A chaque étape, on diminue n de 1, donc on finira bien par atteindre la valeur $n = 1$.

Validez votre fonction avec la solution.

Solution C++ @[pgcarrei.cpp]

```cpp
/**
  Carré par la méthode des impairs
  @param[in] n - un entier naturel
  @return le carré de n
*/

int carrei(int n)
{
  return (n == 1 ? 1 : 2 * n - 1 + carrei(n - 1));
}
```

Écrivez un programme qui teste votre fonction.

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgcarrei.cpp]

```
int main()
{
  int val;
  cout<<"Votre entier positif? ";
  cin>>val;
  cout<<carrei(val)<<endl;
}
```

2 Appliquer le cours

2.1 Fonction divisible / pgdivisible

Propriété
Un entier n est divisible par un entier p, si (et seulement si) le reste de la division entière de n par p (c.-à-d. le modulo) est nul.

Dans le cas où $0 < b \leq a$, écrivez une fonction récursive `divisible1(a,b)` qui teste et renvoie `Vrai` si un entier `a` est divisible par un entier `b`.

De même, dans le cas où $a < b$ (avec $b > 0$), écrivez une fonction récursive `divisible2(a,b)` qui teste et renvoie `Vrai` si un entier `a` est divisible par un entier `b`.

Validez vos fonctions avec la solution.

Solution C++ @[pgdivisible.cpp]

```
/**
  Prédicat de divisibilité (cas 0 <= b <= a)
  @param[in] a - un entier
  @param[in] b - un entier
  @return Vrai si a est divisible par b
*/

bool divisible1(int a, int b)
{
  return (a <= b ? a == b : divisible1(a - b,b));
}
```

```
/**
  Prédicat de divisibilité (cas a <= b avec b positif)
  @param[in] a - un entier
  @param[in] b - un entier
  @return Vrai si a est divisible par b
*/

bool divisible2(int a, int b)
{
  return (a >= b ? a == b : divisible2(a + b,b));
}
```

Déduisez une fonction maître `divisible(n,p)` qui teste et renvoie `Vrai` si un entier `n` est divisible par un entier `p`.

Validez votre fonction avec la solution.

Solution C++ @[pgdivisible.cpp]

```
/**
  Prédicat de divisibilité: cas général
  @param[in] n - un entier
  @param[in] p - un entier
  @return Vrai si n est divisible par p
*/

bool divisible(int n, int p)
{
  int a = n, b = p;
  if (b < 0)
  {
    a = -a;
    b = -b;
  }
  if (b == 0)
  {
    return false;
  }
  else if (a >= b)
  {
    return divisible1(a,b);
  }
  else
  {
    return divisible2(a,b);
  }
}
```

Écrivez un programme qui saisit deux entiers puis teste votre fonction.

Testez.

```
Vos deux entiers? 125632 256
==>divisible(a,b) vaut Faux
```

Validez votre programme avec la solution.

Solution C++ @[pgdivisible.cpp]

```
int main()
{
  int a, b;
  cout<<"Vos deux entiers? ";
  cin>>a>>b;
  cout<<"==> divisible(a,b) vaut "<<divisible(a,b)<<endl;
}
```

2.2 Jeu ”devinez le nombre” / pgdeviner

Objectif
Le jeu « devinez le nombre de 1 à 100 » est le suivant :

- La personne #1 choisit secrètement un entier X de 1 à 100
- La personne #2 propose à la personne #1 un entier P de 1 à 100, qui lui répond
  - C'est exact si $X = P$
  - Plus bas si $X < P$
  - Plus haut si $X > P$
- La personne #2 doit répéter jusqu'à ce que le nombre est deviné.

Quelle est la stratégie la plus efficace pour trouver l'entier mystère ?

Solution simple
Utiliser une recherche dichotomique.

Écrivez le profil et le début de la procédure récursive `deviner(vmin,vmax,mystere)` du jeu : elle demande à l'utilisateur un entier compris dans `[vmin..vmax]`, l'entier `mystere` étant le nombre à deviner.

Écrivez les trois cas de la récursion.
Affichez l'invite :

```
Devinez l'entier entre [vmin] et [vmax]?
```

Validez votre procédure avec la solution.

Solution C++ @[pgdeviner.cpp]

```
/**
  Procédure deviner
  @param[in] vmin - entier vmin
  @param[in] vmax - entier vmax
  @param[in] mystere - entier mystere
*/

void deviner(int vmin, int vmax, int mystere)
{
  int nombre;
  cout<<"Devinez l'entier entre "<<vmin<<" et "<<vmax<<"? ";
  cin>>nombre;
  if (nombre == mystere)
  {
    cout<<"C'est exact"<<endl;
  }
  else if (mystere < nombre)
  {
```

```
    cout<<"Plus bas"<<endl;
    deviner(vmin,nombre - 1,mystere);
  }
  else
  {
    cout<<"Plus haut"<<endl;
    deviner(nombre + 1,vmax,mystere);
  }
}
```

Écrivez un programme qui lance le jeu entre 1 et 100 en tirant au hasard l'entier mystère.

Outil C++

La fonction `rand()`, définie dans la bibliothèque `<random>`, renvoie un entier pseudo-aléatoire positif ou nul. Utilisez le modulo pour projeter l'entier dans l'intervalle souhaité.

Outil C++

L'initialisation du générateur de nombres pseudo-aléatoires (ici avec l'horloge système) s'effectuera dans le **programme principal** avec l'instruction :

```
srand(time(NULL));
```

La procédure `srand` est définie dans la bibliothèque `<random>` et la fonction `time` dans la bibliothèque `<ctime>`.

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgdeviner.cpp]

```
int main()
{
  srand(time(0));
  deviner(1,100,(rand() % 100) + 1);
}
```

2.3 Procédure quorest / pgquorest

Propriété
La relation de la division entière est :

$$a = q_{n^*}\, b + r_{n^*} \text{ avec } 0 \leq r_{n^*} < b$$

les suites étant définies par :

$$\begin{cases} q_n = q_{n-1} + 1 \\ r_n = r_{n-1} - b \\ q_0 = 0,\ r_0 = a \end{cases}$$

Prouvez la relation.

Solution simple
En effet, à l'initialisation $a = 0 \cdot b + a = a$. Et si l'on suppose la relation vraie jusqu'à l'ordre k, à l'ordre suivant :

$$\begin{aligned} a &= (q_k + 1)\, b + (r_k - b) \\ &= q_k\, b + r_k \text{ qui est vraie} \end{aligned}$$

Écrivez une procédure récursive `quorest(a,b,quotient,reste)` qui calcule dans `quotient` le quotient entier de la division d'un entier `a` par un entier `b` et dans `reste` le reste de cette division. Les entiers `a` et `b` sont supposés positifs.

Validez votre procédure avec la solution.

Solution C++ @[pgquorest.cpp]

```
/**
  Procédure quorest
  @param[in] a - un entier
  @param[in] b - un entier
  @param[out] qt - entier quotient de a par b
  @param[out] rt - entier reste de a par b
*/

void quorest(int a, int b, int& qt, int& rt)
{
  if (a < b)
  {
    qt = 0;
    rt = a;
  }
  else
  {
```

```
    quorest(a - b,b,qt,rt);
    ++qt;
  }
}
```

Écrivez un programme qui saisit deux entiers positifs puis calcule et affiche l'opération de la division entière.

Testez. Exemple d'exécution :

```
Deux entiers positifs? 50 6
50 = 8 * 6 + 2
```

Validez votre programme avec la solution.

Solution C++ @[pgquorest.cpp]

```
int main()
{
  int a, b;
  cout<<"Vos deux entiers positifs? ";
  cin>>a>>b;
  int qt, rt;
  quorest(a,b,qt,rt);
  cout<<a<<" = "<<qt<<" * "<<b<<" + "<<rt<<endl;
}
```

2.4 Nombre encadré / pgcnombre

Objectif
Cet exercice demande deux entiers puis affiche le premier entier, entouré d'autant de paires de crochets `"["` et `"]"` qu'indiqué par la valeur du deuxième nombre (supposé positif ou nul). Exemples d'exécution :

```
42 3
==> [[[42]]]
```

```
24 0
==> 24
```

Quelle est la condition d'arrêt ?

Solution simple
C'est que `nc` (nombre de crochets) vaut zéro.

Et la récurrence ?

Solution simple
C'est :

1. Affichez un crochet ouvrant
2. Appelez récursivement avec `nc-1` paires de crochets
3. Affichez un crochet fermant

Écrivez une procédure récursive `afficherCNombre(n,nc)` qui affiche un entier `n` entouré de `nc` (entier) de paires de crochets `"["` et `"]"`.

Validez votre procédure avec la solution.

Solution C++ @[pgcnombre.cpp]

```
/**
  Procédure afficherCNombre
  @param[in] n - un entier
  @param[in] nc - nombre de crochets
*/

void afficherCNombre(int n, int nc)
{
  if (nc == 0)
  {
    cout<<n;
  }
  else
```

```
  {
    cout<<"[";
    afficherCNombre(n,nc - 1);
    cout<<"]";
  }
}
```

Écrivez un programme qui saisit deux entiers et teste la procédure.

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgcnombre.cpp]

```
int main()
{
  int n;
  cout<<"Votre entier? ";
  cin>>n;
  int nc;
  cout<<"Nombre de crochets? ";
  cin>>nc;
  afficherCNombre(n,nc);
}
```

2.5 Coefficients binomiaux / binome

Propriété

Les coefficients du binôme $\binom{n}{k}$ $(1 \leq k < n)$ sont calculables récursivement selon :

- si $k = 0$ ou $k = n$: retourner 1
- sinon retourner $\binom{n-1}{k-1} + \binom{n-1}{k}$

Donnez le type de récursivité, la condition d'arrêt et la récurrence.

Solution simple

C'est une récursivité multiple de conditions d'arrêts $p = 0$ ou $p > n$ et de récurrence la formule ci-dessus.

Écrivez une fonction récursive `binome(n,p)` qui calcule et renvoie le coefficient binomial $\binom{n}{p}$.

Validez votre fonction avec la solution.

Solution C++ @[PGrecursivite.cpp]

```
/**
  Fonction binome
  @param[in] n - un entier naturel
  @param[in] p - un entier naturel
  @return le binomial(n,p)
*/

int binome(int n, int p)
{
  if (p == 0 || p == n)
  {
    return 1;
  }
  else
  {
    return (binom(n - 1,p) + binom(n - 1,p - 1));
  }
}
```

Quelle est la complexité de la traduction directe de cet algorithme ?

Solution simple

Il est exponentiel car pour chaque étape de calcul il faut recalculer plusieurs fois les coefficients binomiaux d'ordre inférieur.

Proposez un autre algorithme **récursif** ayant cette fois une complexité polynomiale.

Aide simple

Essayez de mettre en évidence une autre relation de récurrence en vous aidant du fait que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

$$\binom{2}{0}\quad\binom{3}{1}\quad\binom{2}{1}\quad\binom{4}{2}\quad\binom{1}{0}\quad\binom{2}{1}\quad\binom{5}{3}\quad\binom{3}{2}\quad\binom{1}{1}\quad\binom{2}{2}\quad\binom{3}{3}\quad\binom{1}{0}\quad\binom{4}{3}\quad\binom{2}{1}\quad\binom{3}{2}\quad\binom{1}{1}\quad\binom{2}{2}$$

Solution simple

En développant la relation :

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!\cdot(n-k)!} = \frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k\cdot(k-1)\cdot\ldots\cdot 1}\\ &= \frac{n}{k}\cdot\frac{(n-1)\cdot\ldots\cdot(n-k+1)}{(k-1)\cdot\ldots\cdot 1}\end{aligned}$$

En terme de récurrence, on peut établir la nouvelle relation :

$$\binom{n}{k} = \binom{n-1}{k-1}\cdot\frac{n}{k} \text{ si } k>0 \quad \text{et} \quad \binom{n}{0} = 1$$

La double récursion est ainsi supprimée et il y a au plus $k+1$ appels de fonction. L'algorithme est polynomial en $O(k)$.

Écrivez une fonction récursive `binomeRec(n,k)` qui calcule et renvoie le coefficient binomial $\binom{n}{k}$ qui évite la double récursion.

Validez votre fonction avec la solution.

Solution C++

```
unsigned binomeRec(unsigned n, unsigned k)
{ return (0 == k) ? 1 : binomeRec(n-1, k-1) * n / k; }
```

2.6 Chiffres d'un entier / pgchiffres

Objectif
Cet exercice demande un entier puis affiche la suite de ses chiffres de la droite vers la gauche puis de la gauche vers la droite. Exemple d'exécution :

```
Votre entier? 12354
==> De droite à gauche
45321
==> De gauche à droite
12354
```

Écrivez une procédure récursive `afficherChiffreDG(n)` qui affiche la suite des chiffres d'un entier `n` de la droite vers la gauche.

Validez votre procédure avec la solution.

Solution C++ @[pgchiffres.cpp]

```
/**
  Procédure afficherChiffreDG
  @param[in] n - un entier
*/

void afficherChiffreDG(int n)
{
  if (n > 0)
  {
    cout<<n % 10<<endl;
    afficherChiffreDG(n / 10);
  }
}
```

De même, écrivez une procédure récursive `afficherChiffreGD(n)` qui affiche la suite des chiffres d'un entier `n` de la gauche vers la droite.

Validez votre procédure avec la solution.

Solution C++ @[pgchiffres.cpp]

```
/**
  Procédure afficherChiffreGD
  @param[in] n - un entier
*/

void afficherChiffreGD(int n)
{
  if (n > 0)
  {
```

```
    afficherChiffreGD(n / 10);
    cout<<n % 10<<endl;
  }
}
```

Écrivez un programme qui saisit un entier et teste vos procédures.

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgchiffres.cpp]

```
int main()
{
  int n;
  cout<<"Votre entier? ";
  cin>>n;
  cout<<"==> De droite à gauche:"<<endl;
  afficherChiffreDG(n);
  cout<<"==> De gauche à droite:"<<endl;
  afficherChiffreGD(n);
}
```

2.7 Code du coffre-fort / pgcoffrefort

Afin de protéger l'accès au coffre-fort, un nombre aléatoire est généré pour servir de code à la prochaine ouverture. Ce nombre aléatoire fait moins de 10 chiffres. De plus, le nombre de chiffres pairs de ce nombre est rajouté comme chiffre des unités.

Écrivez une version récursive `nbChiffPairs(nb)` qui calcule le dernier chiffre, à rajouter au nombre donné, pour former le code du coffre-fort.

Solution simple

Si `nb` vaut zéro, alors c'est 1. Sinon il faut analyser chaque chiffre de `nb` et regarder s'il est pair : si oui, il y en a un en plus. D'où il convient de définir une fonction récursive `nbChiffPairsRec(nb)` qui calcule le nombre de chiffres pairs d'un nombre donné, supposé non nul positif pour faire ce calcul.

Validez votre fonction avec la solution.

Solution C++ @[pgcoffrefort.cpp]

```
/**
  Fonction récursive
  @param[in] nb - un entier
  @return le nombre de chiffres pairs de n
*/

int nbChiffPairsRec(int nb)
{
  if (nb == 0)
  {
    return 0;
  }
  else
  {
    return ((nb % 10) % 2 == 0 ? 1 : 0) + nbChiffPairsRec(nb / 10);
  }
}

/**
  Fonction maître
  @param[in] nb - un entier
  @return le nombre de chiffres pairs de n
*/

int nbChiffPairs(int nb)
{
  return (nb == 0 ? 1 : nbChiffPairsRec(nb));
}
```

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgcoffrefort.cpp]

```
int main()
{
  int code;
  cout<<"Votre code (entiers ayant moins de 10 chiffres)? ";
  cin>>code;
  int nc = nbChiffPairs(code);
  cout<<"Nombres de chiffres pairs de "<<code<<" : "<<nc<<endl;
}
```

3 Approfondir le cours

3.1 Fonction 91 de Mac-Carthy / carthy

Définition
Introduite par MAC-CARTHY pour montrer certains pièges de la récursivité, considérons la fonction récursive :

$$F(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ F(F(n+11)) & \text{sinon} \end{cases}$$

Écrivez une fonction `carthy(n)` de la définition.

Validez votre fonction avec la solution.

Solution C++ @[PGrecursivite.cpp]

```
/**
  Fonction carthy
  @param[in] n - un entier
  @return carthy(n)
*/

int carthy(int n)
{
  return (n > 100 ? (n - 10) : carthy(carthy(n + 11)));
}
```

Trouvez la valeur de F pour tout n.

3.2 Fonction de Morris / morris

Définition

La fonction dite « de Morris » est définie par (m et n entiers naturels) :

$$\text{morris}(m,n) = \begin{cases} 1 & \text{si } m = 0 \\ \text{morris}(m-1, \text{morris}(m,n)) & \text{sinon} \end{cases}$$

Écrivez une fonction `morris(m,n)` de la définition.

Validez votre fonction avec la solution.

Solution C++ @[PGrecursivite.cpp]

```
/**
  Fonction morris
  @param[in] m - un entier naturel
  @param[in] n - un entier naturel
  @return morris(m,n)
*/

int morris(int m, int n)
{
  return (m == 0 ? 1 : morris(m - 1,morris(m,n)));
}
```

Quel est le problème de cette fonction ?

Solution simple

Une preuve trop rapide de terminaison de cette fonction conduirait à écrire :

- On considère l'ordre lexicographique sur $\mathbb{N}^2$.
- Le calcul de `morris(m,n)` fait appel au calcul de `morris(m-1,X)` et $(m-1, X) \prec (m, n)$ pour tout X.
- La fonction termine lorsque son premier paramètre est nul.

Mais $X = morris(m,n)$. Alors l'exécution de `morris(1,0)` ne termine pas car ce calcul conduit à l'exécution de `morris(0,morris(1,0))`, donc à nouveau au calcul de `morris(1,0)` et ainsi de suite. En effet, dans la plupart des langages de programmation, les paramètres sont évalués avant d'être passés à la fonction (on parle « d'appel par valeurs ») : le langage évalue en priorité les expressions les plus profondes d'une expression donnée. Bref, il faut être attentif !

3.3 Identité et fonction unité / identite

Définition

Voici une « définition » farfelue mais correcte de l'application identité Id sur les entiers naturels :

$$\begin{cases} Id(0) = 0 \\ Id(n) = n * Un(n-1), \forall n \geq 1 \end{cases}$$

où Un désigne la fonction constante qui a tout n associe 1, que l'on peut définir de la manière suivante :

$$\begin{cases} Un(0) = 1 \\ Un(n) = Id(n) - Id(n-1), \forall n \geq 1 \end{cases}$$

Écrivez des fonctions mutuellement récursives `Id(n)` et `Un(n)` avec `n` entier positif ou nul.

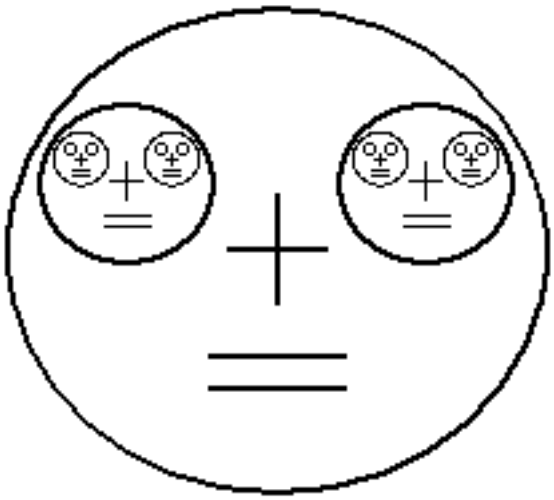
Validez votre fonction avec la solution.

Solution C++ @[PGrecursivite.cpp]

```
/**
  Fonction Id
  @param[in] n - un entier
  @return Id(n)
*/

int Id(int n)
{
  return (n == 0 ? 0 : n * Un(n - 1));
}

/**
  Fonction Un
  @param[in] n - un entier
  @return Un(n)
*/

int Un(int n)
{
  return (n == 0 ? 1 : Id(n) - Id(n - 1));
}
```

3.4 0 + 0 = la tête à Gogo / pgtete

Objectif
Comme vous le savez, 0+0=0.



On pourrait aussi dire 0=(0+0). Dans ce cas, on peut aussi aller un peu plus loin, et puisque 0 vaut (0+0), remplacer les 0 de (0+0) par leur valeur, et obtenir :

```
0 = ((0 + 0) + (0 + 0))
```

Rien n'empêche de continuer et d'écrire :

```
0 = (((0 + 0) + (0 + 0)) + ((0 + 0) + (0 + 0)))
```

Cela devient vite fatiguant de le faire à la main. Cet exercice le fait pour vous. Exemples d'exécution :

```
Un entier? 0
==> 0 = 0
```

```
Un entier? 2
==> 0 = ((0 + 0) + (0 + 0))
```

Quelle est la condition d'arrêt ?
Quelle est la récurrence ?

Solution simple
La condition d'arrêt est que `n` vaut zéro et la récurrence est :

1. Affichez une parenthèse ouvrante
2. Appelez récursivement avec `n-1`
3. Affichez un plus
4. Appelez récursivement avec `n-1`
5. Affichez une parenthèse fermante

Écrivez une procédure récursive `afficherTeteToto(n)` qui affiche `n` fois les zéros à droite de l'égalité « 0=0 » par leur valeur « (0+0) ».

Validez votre procédure avec la solution.

Solution C++ @[pgtete.cpp]

```
/**
  Procédure afficherTeteToto
  @param[in] n - un entier
*/

void afficherTeteToto(int n)
{
  if (n == 0)
  {
    cout<<"0";
  }
  else
  {
    cout<<"(";
    afficherTeteToto(n - 1);
    cout<<"+";
    afficherTeteToto(n - 1);
    cout<<")";
  }
}
```

Écrivez un programme qui saisit un entier et lance la procédure.

Testez.

Validez votre programme avec la solution.

Solution C++ @[pgtete.cpp]

```
int main()
{
  int n;
  cout<<"Un entier? ";
  cin>>n;
  cout<<"==> 0 = ";
  afficherTeteToto(n);
}
```

4 Références générales

Comprend [Felea-PG1 :c3 ;ex85], [Franceioi-AL1 :c6 :ex1], [Franceioi-AL1 :c6 :ex4] ■