MOOC Init Prog C++
Corrigés semaine 4

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 7 : Fonctions simples

1)

```
double min2(double a, double b)
{
  if (a < b) {
    return a;
  } else {
    return b;
  }
}
```

2) Sans utiliser `min2` :

```
double min3(double a, double b, double c)
{
  if (a < b) {
    if (a < c) {
      return a;
    } else {
      return c;
    }
  } else {
    if (b < c) {
      return b;
    } else {
      return c;
    }
  }
}
```

Pour calculer le minimum de 3 nombres, on peut aussi calculer d'abord le minimum des 2 premiers, et ensuite calculer le minimum de ce résultat et du dernier nombre, ce qui permet de réutiliser `min2` :

```
double min3(double a, double b, double c)
{
  return min2(min2(a, b), c);
}
```

Exercice 8 : Sapin

```
#include <iostream>
using namespace std;

void etoiles(int nb_etoiles)
{
  for(int i(0); i < nb_etoiles; ++i) {
    cout << "*";
  }
}

void espaces(int nb_espaces)
{
  for(int i(0); i < nb_espaces; ++i) {
    cout << " ";
  }
}

void triangle(int nb_lignes)
{
  for(int i(0); i < nb_lignes; ++i) {
    espaces(nb_lignes - i);
    etoiles(2 * i + 1);
    cout << endl;
  }
}

void triangle_decale(int nb_lignes, int nb_espaces)
{
  for(int i(0); i < nb_lignes; ++i) {
    espaces(nb_espaces + nb_lignes - i);
    etoiles(2 * i + 1);
    cout << endl;
  }
}

void sapin()
{
  triangle_decale(2, 2);
  triangle_decale(3, 1);
  triangle_decale(4, 0);

  // le tronc:
  triangle_decale(1, 3);
}

/* Cette fonction permet d'afficher les 3 parties du beau sapin.
 * no_ligne_fin a le meme role que le parametre nb_lignes
 * des fonctions triangle et triangle_decale.
 * no_ligne_debut definit la ligne du haut du trapeze.
 */
void trapeze_decale(int no_ligne_debut, int no_ligne_fin, int nb_espaces)
{
  for(int i(no_ligne_debut); i < no_ligne_fin; ++i) {
    espaces(nb_espaces + no_ligne_fin - i);
    etoiles(2 * i + 1);
    cout << endl;
  }
}

void beau_sapin()
{
  trapeze_decale(0, 3, 2);
  trapeze_decale(1, 4, 1);
  trapeze_decale(2, 5, 0);
  cout << "    |||" << endl;
}

/* Cette fonction n'etait pas demandee.
 * Notez qu'on peut avoir deux fonctions de meme nom, si leurs
 * parametres sont differents
 */
void beau_sapin(int nb_etages)
```

```
{
  for(int i(0); i < nb_etages; ++i) {
    trapeze_decale(i, 3 + i, nb_etages - i);
  }
  espaces(2 + nb_etages);
  cout << "|||" << endl;
}

int main()
{
  beau_sapin(10);
  return 0;
}
```

Exercice 9 : Calcul approché d'une intégrale (niveau 2)

Cet exercice correspond à l'exercice n°15 (pages 33 et 214) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

```
#include <iostream>
#include <cmath>
using namespace std;

double f(double x);
double integre(double a, double b);
double demander_nombre();

int main()
{
  double a(demander_nombre());
  double b(demander_nombre());

  // on définit la précision de l'affichage à 12 chiffres après la virgule
  cout.precision(12);

  cout << "Integrale de f(x) entre " << a
       << " et " << b << " :" << endl;
  cout << integre(a,b) << endl;
  return 0;
}

double f(double x) { return x*x; }
// double f(double x) { return x*x*x ; }
// double f(double x) { return 1.0/x ; }
// double f(double x) { return sin(x); }

double integre(double a, double b)
{
  double res;
  res =  41.0 * ( f(a) + f(b) )
      + 216.0 * ( f((5*a+b)/6.0) + f((5*b+a)/6.0) )
      +  27.0 * ( f((2*a+b)/3.0) + f((2*b+a)/3.0) )
      + 272.0 * f((a+b)/2.0) ;
  res *= (b-a)/840.0;
  return res;
}

double demander_nombre()
{
  double res;
  cout << "Entrez un nombre réel : ";
  cin >> res;
  return res;
}
```

Exercice 10 : Fonctions logiques

On peut remarquer que A ET A = A. NON(A ET A) vaut donc NON(A). On peut alors écrire la fonction `non` ainsi :

```
bool non(bool A)
{
  return non_et(A, A);
}
```

Comme NON(NON(C)) = C, on a NON(NON(A ET B)) = A ET B. On peut donc écrire la fonction `et` ainsi:

```
bool et(bool A, bool B)
{
  return non(non_et(A, B));
}
```

NON(A OU B) = NON(A) ET NON(B). Donc, (A OU B) = NON(NON(A OU B)) = NON(NON(A) ET NON(B)). On peut donc écrire la fonction `ou` ainsi:

```
bool ou(bool A, bool B)
{
  return non_et(non(A), non(B))
}
```

Ces relations sont intéressantes en électronique: il suffit d'un seul type de composant (effectuant la fonction `non_et`) pour réaliser n'importe quel circuit logique.

Exercice 11 : recherche dichotomique

> Cet exercice correspond à l'exercice n°34 (pages 83 et 256) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Aucune difficulté majeure pour cet exercice qui est niveau 2 uniquement parce que l'énoncé est moins détaillé.

Voici le code correspondant :

```
#include <iostream>
#include <limits>
using namespace std;

int demander_nombre(int min, int max);
unsigned int cherche(unsigned int borneInf, unsigned int borneSup);

constexpr unsigned int MIN(1);
constexpr unsigned int MAX(100);

// ----------------------------------------------------------------------
int main()
{
  cout << "Pensez à un nombre entre " << MIN << " et " << MAX << "."
       << endl << endl;
  const unsigned int solution( cherche(MIN, MAX) );
  cout << endl << "Votre nombre était " << solution << '.'
       << endl;
  return 0;
}

/* ----------------------------------------------------------------------
 * Recherche d'un nombre par dichotomie dans un intervalle [a b]
 * Entrée : les bornes de l'intervalle
 * Sortie : la solution
 * ---------------------------------------------------------------------- */
unsigned int cherche(unsigned int a, unsigned int b)
{
  //  cout << "[ " << a << ", " << b << " ]" << endl;

  if (b < a) {
    cerr << "ERREUR: vous avez répondu de façon inconsistante !" << endl;
    return b;
  }

  unsigned int pivot((a+b)/2);
  char rep;

  do {
    cout << "Le nombre est il <, > ou = à " << pivot << " ? ";
    cin >> rep;
  } while ((rep != '=') and (rep != '<') and (rep != '>'));

  switch (rep) {
  case '=':
    return pivot;
  case '<':
    return cherche(a, pivot-1);
  case '>':
    return cherche(pivot+1, b);
  }
}
```

Le seul petit truc auquel il faut penser est peut être le `switch` : **oui !** on peut faire un `switch` sur un caractère.

Exercice 12 : Recherche approchée de racine

```
#include <iostream>
#include <cmath>
using namespace std;

double const epsilon(1e-6);

 double f(double x) { return (x-1.0)*(x-1.5)*(x-2.0); }

 double df(double x) { return (f(x+epsilon)-f(x))/epsilon; }

 double itere(double x) { return x - f(x) / df(x); }

int main()
{
  double x1,x2;
  cout << "Point de départ ? ";
  cin >> x2;

  do {
    x1 = x2;
    cout << "  au point " << x1 << " : "<< endl;
    cout << "     f(x)  = " << f(x1) << endl;
    cout << "     f'(x) = " << df(x1) << endl;
    x2 = itere(x1);
    cout << "     nouveau point = " << x2 << endl;
  } while (abs(x2-x1) > epsilon);

  cout << "Solution : " << x2 << endl;

  return 0;
}
```
