TD-2
Procédures, fonctions, visibilité, vie et mort des variables - Binaire

Procédures, fonctions

De faon simpliste, une procédure (ou une fonction) est un moyen de factoriser un code souvent utilisé et d'utiliser ce bloc simplement à un autre endroit du programme en l'appelant par son nom. En réalité c'est un mécanisme plus intressant que le simple recopiage car on peut utiliser le nom de la procédure dans le code même de la procédure que l'on est en train de définir.

Il reste une précision à donner sur les liens tissés entre les paramètres formels et les paramètres effectifs passés aux fonctions et procédures.

Exercice 1. :
Que fait le programme suivant ?

```
#include <iostream>

int f(int x){
  x++; // x=x+1;
  return x;
}

int g( int &x){
  x=x+1;  // x++;
  return x;
}

int main(){
  int n;
  n=5;
  cout << f(n) << endl;
  cout << f(n) << endl;
  cout << g(n) << endl;
  cout << g(n) << endl;
}
```

Exercice 2. :
Reprenons la carte bleue et ses trois champs : **montant** qui modélise l'argent disponible, **code** qui contient le code secret, **type_carte** qui est soit visa, soit mastercard. Il est usuel pour une utilisation de type commercial comme celle-ci de fournir au gestionnaire des facilités pour contrôler les valeurs des différents champs. Peut-on écrire des *accesseurs* et *modificateurs* pour chacun de ces trois champs ? Un accesseur permet de savoir ce qu'un champ donné contient. Un modificateur permet de modifier le contenu d'un champ donné. Si oui, les écrire. A chaque fois se poser la question : fonction ou procédure ? Paramètre valeur, paramètre référence ?

Au cours de l'exercice précédent vous avez pu constater que l'on peut discuter le fait d'écrire un programme sous la forme d'une procédure ou d'une fonction. Cette hésitation est naturelle, en effet *toute procédure peut être transformée en fonction, et toute fonction encodée sous la forme d'une procédure.*

Le plus simple est de transformer une procédure en fonction : il suffit de l'obliger à retourner une valeur sans intérêt arbitraire.

Pour la transformation de fonction en procédure, il faut préserver la valeur de résultat de l'ancienne fonction. Elle doit être récupérée quelque part, et donc une variable doit être modifiée. Pour qu'une fois la procédure terminée la variable ne soit pas effacée, il faut qu'elle corresponde à une variable déjà existante, et ce lien ne peut être tissé qu'à l'aide d'un passage par référence.

Néanmoins, il existe des raisons compréhensibles où on préfère écrire une fonction plutôt qu'une procédure. Non ?

Exercice 3. Reprendre une fonction et une procédure de l'exercice précédent pour les transformer respectivement en procédure et en fonction.

Variables

Une variable est une zone réservée de la mémoire, sa taille est liée à son type, et on la repère en utilisant un nom. Les variables sont crées lors de déclarations, ou lors d'appels de fonctions lorsque celles ci sont des paramtres. Le mcanisme de passage par rfrence peut faire qu'a un moment donné de l'excution une variable ait plusieurs noms. On peut alors se demander lequel utiliser ?

Plus gnralement la vie des variables le long d'une excution peut tre décrite par deux notions :

- Sa Persistance : quand une variable existe t'elle dans la mmoire ? (est-elle vivante ?)
- La visibilité : quand une variable peut elle être lgalement utilisée ?

Par exemple une variable globale est vivante pendant toute la dure du programme. Les variables locales et les paramtres des fonctions existent le temps de l'excution du bloc correspondant.

Dans le programme suivant *i* est vivante pour *intro*, mais n'est pas visible :

```
#include <iostream>
void intro(){
 cout << ‘‘ i vaut ‘‘ << i << endl;
}
void main(){
 int i=5;
 intro();
 cout << i << endl;
}
```

Il existe des cas ou des variables, bien que visibles, ne peuvent tout de mm pas tre utilises :

```
#include <iostream>
int i=5;
void intro(){
 int i=6;
 cout << ‘‘ i vaut ‘‘ << i << endl;
}
void main(){
 intro();
 cout << i << endl;
}
```

On parle alors de recouvrement. Comment pouvez vous l'empcher dans l'exemple prcdent ?

Il existe d'autres commandes plus exotiques, comme la directive *static* : elle rend une variable persistante de manire permanente, sans changer les règles de visibilité. On utilisera rarement cette dclaration, mais à titre d'exercice, que fait le programme suivant :

```
#include <iostream>
int compteur(){
 static int compt=0;
 compt++;
 return (compt);
}
void main(){
 int i;
 for(i=0;i<10;i++){
  cout << compteur() << `` '';
 }
}
```

Les Blocs

Un bloc est un morceau de code délimité par { et }, qui peut contenir des dclarations de variables, dites locales. En voici un exemple :

```
#include <iostream>
void main(){
 int i=5,j=7;
    {
    int buffer;
    buffer=i; i=j;j=buffer;
    }
 cout << i << `` `` << j << endl;
}
```

La variable $buffer$ a la mme dure de vie que le bloc qui la dclare. Ainsi, au moment de l'affichage, on ne pourrait plus y faire rfrence.

Par contre, un bloc ne doit pas tre vu comme une fonction, puisque le bloc a accs aux variables i et j.

Les boucles $for, while, do$ sont des cas particulier de blocs, ainsi on peut crire :

```
int i=5;
for (int i=0; i<10;i++){cout << i << endl;}
cout << i << end;
```

A titre de rappel, on dispose des commandes d'chappement suivantes :

- *return* : sort de la fonction, procdure en cours
- continue : sort de la boucle pour valuer la prochain itration
- break : sort dfinitivement de la boucle

Exercice 4. Simplifier le programme suivant ?

```
# include <iostream>
int i;
void main(){
 for (i=0;i<10;i++){
  if (i< 8) continue;
  else {cout << i; break;}
  cout << i << ' ' << endl;
 }
}
```

Exercice 5. Peut on rendre toutes les variables visibles accessibles ? Peut on rendre toutes les variables vivantes visibles ?

TP Nombres Binaires

L'objet de ce TP est de comprendre et manipuler les reprsentations des nombres dans une autre base que la base dcimale. Vous serez galement amené à apprcier le temps de rflexion par rapport au temps de programmation. Nous vous conseillons de ne pas vous prcipiter, le crayon est un meilleur allié que le clavier. Ce TP vous donnera aussi une ide de l'exercice qui vous sera demandé lors de l'examen sur machine.

Codage binaire des nombres positifs

En notation décimale :

$$56013 = 5 \times 10^4 + 6 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 3 \times 10^0$$

La base du systéme décimal : 10, les chiffres 0,1,...,9. La base du systéme binaire : 2, les chiffres 0,1. Exemple :

$$(1011)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 == 8 + 2 + 1 = (11)_{10}$$

La même suite de chiffres donne deux entiers différents en fonction de la base utilisée. Pour enlever toute ambiguité on notera la base comme l'index.

Exercice 6. : Que vaut en décimal le nombre binaire '100010' ? Quel est le plus grand entier qu'on peut représenter en binaire à l'aide de 8 bits? Même question pour n bits?

Exercice 7. Soit b un entier non négatif avec la représentation binaire $(b_n b_{n-1} \dots b_1 b_0)_2$. Quelle est la représentation binaire de $b/2$ (division entière)? Quel est le reste de la division de b par 2? Comment s'écrit 122 en binaire ?

Exercice 8. Déduire de l'exercice précédent l'algorithme puis le programme de transformation d'un entier non négatif en binaire.

Binaire : Addition

$$\begin{aligned}
0+0 &= 0\\
0+1 &= 1\\
1+0 &= 1\\
1+1 &= (10)_2\\
1+1+1 &= (11)_2
\end{aligned}$$

Exemple : $\begin{array}{r} 11011 \\ +\ 1011 \\ \hline 100110 \end{array}$

Exercice 9. Soit x et y deux variables booléennes (deux bits) et $(z_1 z_0)_2$ le résultat de l'addition binaire $x+y$. Ecrire deux expressions booléennes : la première qui exprime le bit du résultat z_0 et la deuxième qui donne le bit de retenu z_1. Ecrire un programme qui réalise cette addition.

Binaire : Multiplication

La multiplication de nombres binaires peut s'effectuer à l'aide d'additions et de décalages binaires.

$$\begin{array}{r} 1101 \\ \times\ 101 \\ \hline 1101 \\ +\ 1101 \\ \hline 1000001 \end{array}$$

Exercice 10. Transformer 122 et 34 en binaire, puis effectuer la multiplication.

Exercice 11. Soit s une chaîne de caractères contenant un nombre binaire. Ecrire une fonction ou procédure qui multiplie le nombre contenu dans s par le nombre élevé à la puissance de 2 donnée par l'utilisateur. Par exemple, si on donne 3, le nombre est multiplié par 8.

Binaire : Commandes C++ – opérateurs de décalage

Les deux opérateurs de décalage sont le décalage à droite et le décalage à gauche. Il s'agit de décalages binaires. Ils sont quasi instantanés en C++. Regardons sur un exemple.

On a le programme suivant (avec les bons `#include`):

```
int main(){
  unsigned long x, y, z;

  x = 37;       /* en binaire : 100101 */
  y = x << 2;   /* en binaire : 10010100 */  /* decalage a gauche */
  z = x >> 2;   /* en binaire : 1001 */      /* decalage a droite */

  cout << "x = " << x << "\n";
  cout << "y = " << y << "\n";
  cout << "z = " << z << "\n";
}
```

Après compilation, son exécution donne :

```
x = 37    y = 148     z = 9
```

Il faut se méfier de certaines limitations, en particulier, un entier (même long) a un nombre limité de chiffres (64 s'il n'est pas signé). Si on dépasse ce nombre, on repart de l' "autre côté", par exemple, le code suivant[1] :

```
int main(){
  unsigned long x, y;

  x = 18446744073709551615;
  cout << "x = " << x << "\n";
  y = x+1;
  cout << "y = " << y << "\n";
}
```

donne comme résultat d'exécution :

```
x = 18446744073709551615      y = 0
```

Ainsi, quand on utilise des décalages à gauche, on risque d'avoir des résultats bizarres si l'on ne fait pas attention.

L'intérêt des décalages pour notre problème est assez évident, puisque multiplier un nombre par 2^n revient à le décaler à gauche de n. Noter qu'on peut faire le décalage avec des `char`, puisque un `char` est une suite de bits.

[1]le nombre `x` n'a pas été choisi au hasard : $\texttt{x} = 2^{64} - 1$.

Remarque non intégrées dans ce sujet : on a vu que les tableaux sont toujours passés en arguments comme des références. Ce n'est pas le cas des structures. Si par exemple un tableau est un champs d'une structure, il est passé par .... valeur, et donc recopié !

Correction

ex 1 : Il faut expliquer les passages par référence et par valeur (vu en If121). Si une majorité d'étudiants n'ont pas compris, ils peuvent mettre le nez dedans en TP et jouer à s'en convaincre. L'affichage obtenu est : 6 6 6 7.

ex 2 : Ici on touche une limitation du savoir actuel des étudiants et le besoin de pouvoir plus. En C Une fonction ne retourne pas un tableau. Pour avoir un accesseur au code de la carte, il faut utiliser les pointeurs (ou les objets), les pointeurs seront vus plus tard.

```
#include <iostream>

typedef enum geree_par {visa,mastercard};
typedef int code[4];

typedef struct carte_bleue {
  int montant;
  code code_s;
  geree_par type_carte;
};

/* ici accesseur */
int acc_montant(carte_bleue c){
  return c.montant;
}

/* function ne retourne pas un tableau; pas d'accesseur avec code */
void  affiche_code(carte_bleue c){
  int i;

  cout << "montant : ";
  for (i=0; i<4 ; i++)
    cout << c.code_s[i];
  cout << endl;
}

/* accesseur */
geree_par acc_gestion(carte_bleue c){
  return c.type_carte;
}

void affiche_type(geree_par g){
  cout << "geree par : ";
  switch(g) {
   case visa    : cout << "visa";    break;
   case mastercard   : cout << "mastercard";   break;
  }
  cout << endl;
}

void modif_montant(carte_bleue &c, int m){
  c.montant=m;
}

void modif_code(carte_bleue &c, code s){
  int i;

  for (i=0; i<4 ; i++)
```

```
  c.code_s[i]=s[i];
}

void modif_gestion(carte_bleue &c, geree_par g){
  c.type_carte=g;
}

int main(){
  carte_bleue c;
  code nc = {1,1,1,1};

  c.montant = 100;
  c.code_s[0]=1;
  c.code_s[1]=4;
  c.code_s[2]=1;
  c.code_s[3]=8;
  c.type_carte=visa;
  cout << "montant : " << acc_montant(c) << endl ;
  affiche_code(c);
  affiche_type(acc_gestion(c));
  modif_montant(c, 200);
  modif_code(c, nc);
  modif_gestion(c, mastercard);
  cout << "montant : " << acc_montant(c) << endl ;
  affiche_code(c);
  affiche_type(acc_gestion(c));
}
```

ex 3 : Pour une fonction en procédure, il suffit d'ajouter un paramtre réfrence à la procédure et d'y mettre la valeur que la fonction devait retourner.

Pour une procédure en fonction, il suffit de faire retourner quelque chose, même une constante. Ce ne sera pas très joli, car on risque d'avoir une fonction qui modifie ses propres arguments (si la procédure avait plusieurs paramètres références par exemple), bien loin de la fonction au sens mathématiques du terme.

Attention, les appels changent!

ex 4 : la derniere ligne n'est jamais executee. On peut meme tout simplifier en disant que ca ne fait qu'un seul *cout*

ex 5 : Je dirais oui, et non ?, non a cause des fonctions recursives, et le oui parce qu'il suffit de donner des noms differents, non ?

ex 6 : 100010 en binaire vaut 34.

Se rappeler que $2^{n-1} + \cdots + 2 + 1 = 2^n - 1$.

ex 7 : Directement de la définition: $b/2 = (b_n b_{n-1} \ldots b_1)_2$ et le reste est égal à b_0.

122 s'écrit 1111010 en binaire.

ex 8 :

```
#include <iostream>
#include <string>

int main(){
  int a;
  string s = "";

  do {
    cout << "Entrer un entier positif : ";
    cin >> a;
```

```
  }
    while (a <0);

    while (a!=0) {
      if ((a%2) == 1)
s = "1" + s;
      else
s = "0" + s;
      a = a / 2;
      // cout << " a " << a << " s " << s << endl; ajoute pour les tests
    }
    cout << "en binaire : " << s << endl;
}
```

ex 9 : Soit $(z_1 z_0)_2 = x + y$. Alors

$$\begin{aligned} z_0 &= (\text{ non } x \text{ et } y) \text{ ou } (x \text{ et } \text{ non } y) \\ z_1 &= x \text{ et } y \end{aligned}$$

Dans la sol on voit que C traite false et 0 (true et 1) indifféremment.

```
#include <iostream>

int main(){
  bool x,y,z0 = false,z1=false;

  x = 0;
  y = true;
  z0 = (x && !y) || (!x && y);
  z1 = (x && y);
  cout << "x " << x << "y " << y << "z0 " << z0 << "z1 " << z1;
}
```

ex 10 : 122 s'écrit 1111010 en binaire. 100010 en binaire vaut 34.

Le produit des deux nombres en binaire vaut 1000000110100, soit 4148.

ex 11 : Ceci pour montrer qu'on ne fait que décaler vers la gauche!

```
#include <iostream>
#include <string>

void multiplie(string &s, int n){
  int i;

  for (i=1;i<=n;i++)
    s = s + "0";
}

int convertit(string s){
  int l = s.length();
  int n = 0;
  int i;
  for (i=0; i<l; i++)
    if (s[i]=='0') n = n*2;
    else n = n*2 + 1;
  return n;
}

int main(){
  char a;
```

```
  int p;
  string s = "";

  cout << "Entrer un nombre binaire (!= (0 ou 1) pour terminer) : ";
  cin >> a;
  while ((a =='0') || (a=='1')) {
    s = s + a;
    cin >> a;
  }

  cout << "Entrer une puissance de 2 : ";
  cin >> p;
  // on pourrait mettre une boucle pour s'assurer que p est >=0

  if (!s.empty()) multiplie(s, p);
  cout << "final : " << s << " en decimal " << convertit(s) << endl;
}
```