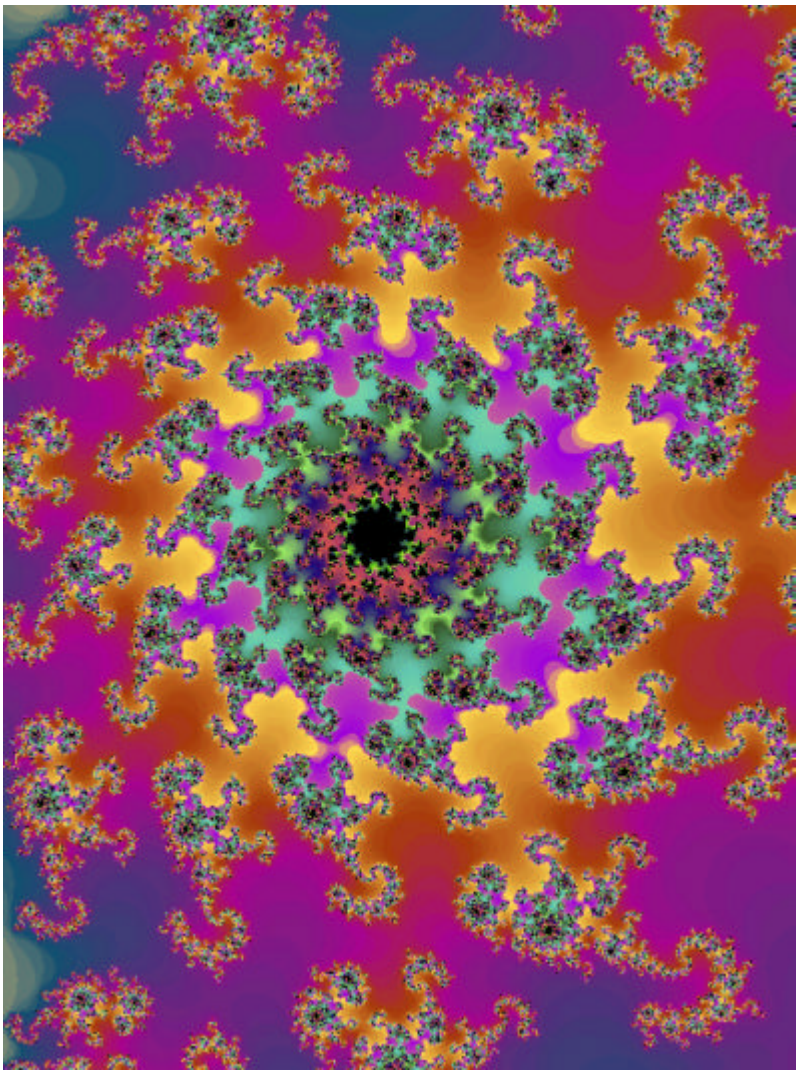
CHAPITRE 9

Instructions



Un programme C++ est constitué d'une quantité arbitraire de fonctions ("Fonctions", page131), elles-même constituées d'une suite d'instructions exécutées séquentiellement. Une instruction peut être simple ou composée. Chaque instruction doit impérativement se terminer par un point-virgule. Le point-virgule est un **terminateur.** (Rappelons qu'en PASCAL, le point-virgule est également utilisé, mais il joue le rôle de séparateur).

Une instruction composée est englobée par les caractères { et } (un peu à la manière du BEGIN...END de PASCAL ou MODULA). Les parenthèses {} définissent un **bloc d'exécution**. Par la même occasion, ils définissent une instruction composée. Il n'est pas indispensable de terminer un bloc d'exécution par un point-virgule; ainsi :

```
int      toto = 1024;           // ; obligatoire
while (toto)                    // ; incorrect, car il
                                // terminerait l'exécution de
                                // while (compilerait
                                // correctement néanmoins)
     {
     cout<<"Toto égale : "<<toto<<endl;
     toto--;                    // ; obligatoire
     }                          // ; facultatif
cout<<"Toto égale zéro"<<endl;  // ; obligatoire
```

9.1 *Conditions (if, else, switch)*

Ces instructions permettent le test d'une condition, et la sélection d'une instruction ou d'une autre selon le résultat du test. La condition est booléenne; comme le type booléen n'existe pas en tant que tel en C++, n'importe quel entier (`char, short, int, long, unsigned`...) peut convenir.

9.1.1 *Instruction if ( )*

L'instruction `if` permet de tester l'expression immédiatement suivante, écrite entre parenthèses. L'expression entre parenthèses est évaluée comme une variable entière, et le résultat de l'évaluation est comparé à la valeur 0. Si le résultat est différent de 0 (correspond à TRUE), alors l'instruction suivant immédiatement le `if ()` est exécutée, dans le cas contraire, cette même instruction est sautée. Dans ce dernier cas, il se peut qu'une clause `else` soit exécutée en lieu et place ("Instruction else", page117).

N'importe quelle expression livrant un résultat convertible en un entier peut être utilisée comme condition du `if ()`.

```
int  a, b, c;
c = 4;
b = 3;
if ((a = (b * c) / 2) == 6)
     cout<<"On verifie que (4 * 3) / 2 == 6 "<<endl;
```

9.1.2 *Instruction else*

L'instruction else ne se rencontre qu'en conjonction avec if. Il s'agit de l'alternative à if, si la condition contenue dans la clause if () n'est pas remplie.

```
int      i1, i2;
cout<<"Entrer un nombre"; cin>>i1;
cout<<"Entrer un deuxième nombre"; cin>>i2;
if (i1 > i2)
     cout<<"Le premier nombre est plus grand que le deuxième"<<endl;
else
     {
     if (i1 == i2)
          cout<<"les deux nombres sont égaux"<<endl;
     else
          cout<<"Le deuxième nombre est plus grand que le premier"<<endl;
     }
```

La clause `else` se rapporte toujours à l'instruction `if` immédiatement précédente. Si l'on désire rapporter une clause `else` à une instruction `if` antérieure à la précédente, il est nécessaire d'utiliser des blocs d'instructions { ... }.

```
int  a, b, c;
a = 2;
b = 3;
c = 4;
if (++a == b)      // -> TRUE
     {             // { necessaire pour definir un bloc
                   // en prevision du else.
     if (a == --c) // -> TRUE
          cout<<“On a verifie une evidence”<<endl;
     }             // Fin du bloc...
else
     cout<<“++2 != 3... Etrange ...”<<endl;
```

9.1.3 *Instruction switch ()*

`switch(`*`expression`*`) instruction` permet le choix parmi un nombre arbitraire de possibilités, sans devoir recourir à une série de tests `if ()`. `switch ()` ressemble, par sa fonctionnalité, à l'instruction CASE OF de PASCAL. Bien que ce ne soit pas mandatoire, `switch ()` n'a véritablement de sens que si l'instruction correspondante est une instruction composite. L'expression doit être entière.

`switch()` se rencontre presque toujours en conjonction avec trois autres instructions :

`case` expression : instruction

`break;`

`default` : instruction;

Une expression `case` est nécessaire pour chaque condition devant être testée. L'instruction `default` permet de prendre les mesures nécessaires si aucune des conditions définies par les clauses `case` n'est remplie.

Le programme donné en exemple çi-après teste si un caractère entré au clavier est ou non un digit (0..9). La manière de réaliser ce test n'est vraisemblablement pas optimale, mais est à considérer comme un exemple d'utilisation de l'instruction `switch ()`.

```
void main()
     {
     char ch;
     cin>>ch;
     switch(ch)
          {
          case ‘0’:
          case ‘1’:
          case ‘2’:
          case ‘3’:
          case ‘4’:
          case ‘5’:
          case ‘6’:
          case ‘7’:
```

```
        case ‘8’:
        case ‘9’:
                  cout<<”Vous avez entré un digit : ”<<ch<<endl;
                  break;   // nécessaire !
        default :
                  cout<<”Ceci n’est pas un  digit : ”<<ch<<endl;
                       // Instruction break pas nécessaire ici.
        }
}
```

On remarque l’utilisation de l’instruction `break`, qui ne semble à priori pas évidente. En C et en C++, l’exécution du code, suite à une clause `case` remplie commence effectivement par l’instruction suivant la clause `case`, mais ne se termine pas automatiquement à l’apparition de la prochaine clause `case` ! Les habitués du langage PASCAL sont invités à se concentrer pour éviter des ”fautes d’habitudes” ! L’exécution continue en effet jusqu’à la fin de l’instruction `switch ()`, à moins que l’instruction `break` ne soit rencontrée entre temps, ce qui provoque une rupture de la séquence, et une terminaison prématurée de l’instruction `switch ()`.

L’instruction `default`, qui permet de prendre les mesures nécessaires si aucune clause `case` n’est remplie, est purement optionnelle. Il est nécessaire de la mettre en dernier lieu. L’instruction `break` n’est alors plus nécessaire, puisque le déroulement normal du programme (sans `break`) aboutit à la fin de l’instruction `switch ()`.

9.2 *Itérations (while, do...while, for())*

Une itération introduit une répétition d'actions, associée à une ou plusieurs conditions contrôlant le nombre de répétitions. Il y a, dans C++, trois types d'instructions générant des itérations. Toutes ces itérations peuvent être avortées prématurément au moyen de l'instruction `break`, rencontrée précedemment dans le contexte de l'instruction `switch ()`. De manière similaire, il est toujours possible d'abandonner l'itération en cours pour passer directement à la suivante au moyen de l'instruction `continue`.

9.2.1 *Variables locales à l'itération*

Une variable peut être locale à un bloc d'instructions; ainsi :

```
if (test)
     {
     int i; // i n'est défini que dans le bloc {}
     }
```

Pour toutes les instructions d'itération existait, jusqu'à la normalisation de C++ par ANSI, la possibilité de définir une variable dans l'en-tête de l'itération, comme par exemple :

```
for (int i = 0; i < 10; i++) cout<<i<<endl;
```

De manière un peu surprenante, la variable i avait alors une durée de vie qui excédait celle du bloc d'instructions contrôlé par l'itération. ANSI a corrigé ce défaut, et les nouveaux compilateurs restreignent la visibilité strictement au corps de l'itération. Ainsi

```
for (int i = 0; i < 10; i++) cout<<i<<endl;
cout<<i<<endl;
```

aura pour effet une erreur de compilation dans le cas d'un compilateur conforme au nouveau standard, mais passera sans problèmes sur un compilateur un peu plus ancien. Malheureusement, cette modification, pour cosmétiquement justifiée qu'elle puisse paraître, introduit d'innombrables problèmes lorsque l'on désire recompiler du code existant.

9.2.2 *Instruction while ()*

L'instruction `while` (*condition*) *instruction* exécute *instruction* tant que *condition* est vraie ("est évaluée comme entier positif non nul"). Le petit fragment de programme suivant lit un fichier séquentiellement et en imprime le contenu sur `cout`.

```
ifstream  f;
f.open("tmp.tmp");
if (f.fail())
     {
     cerr<<"Could not open file tmp.tmp"<<endl;
     exit(1);      // Interruption abortive du programme
```

```
     }
while (!f.eof())   // Jusqu'à la fin du fichier ...
     {
     char ch = f.get();      // Lire un caractère du fichier...
     cout<<ch;
     }
cout<<endl<<"Fin de fichier"<<endl;
```

`while ()` est très semblable à la construction WHILE de PASCAL. Pour les habitués de PASCAL, les deux instructions peuvent être considérées comme équivalentes.

9.2.3 *Instruction do ... while ()*

Si `while ()` est semblable au `WHILE` de PASCAL, alors `do...while ()` ressemble au `REPEAT ... UNTIL` de PASCAL. La seule différence avec `while ()` est que l'expression terminant l'itération est évaluée **après** l'exécution de l'instruction.

Une nuance néanmoins pour les adeptes de PASCAL qui auraient une trop grande tendance à ramener les langages "étrangers" à leur dialecte favori : `REPEAT ... UNTIL` effectue l'itération **jusqu'à ce** que la condition soit remplie, alors `do ... while ()` effectue l'itération **pendant** que la condition est remplie. D'un point de vue purement linguistique, cela semble d'ailleurs assez clair...

L'exemple çi-dessous lit des caractères depuis le terminal et les stocke sur un fichier, jusqu'à ce qu'un caractère particulier (ETX, CONTROL-C) soit lu. Le fichier est alors fermé.

```
char      ch;
const     char      ETX = 3;

ofstream  f;            // Fichier de sortie
f.open("tmp.tmp");      // Association de f au nom "tmp.tmp"
if (f.fail())
     {
     cerr<<"Could not open tmp.tmp"<<endl;
     exit(1);      // Avortement ...
     }
do
     {
     cin>>ch;
     if (ch != ETX)
          f<<ch;
     }
while (ch != ETX);
f.close();
```

9.2.4 *Instruction for ()*

La boucle for () de C++ ressemble syntactiquement à son homologue PASCAL, mais en diffère profondément par son implémentation. `for ()` permet en effet de spécifier :

- la condition initiale
- la condition de continuation de l'itération
- que faire après chaque itération

La boucle `for ()` est certainement l'instruction d'itération la plus populaire parmi les programmeurs C++, en raison de la grande flexibilité qu'elle permet. Syntactiquement, on définit la boucle for comme :

```
for (Instruction d'initialisation;
     Condition de continuation d'itération;
     Instruction à exécuter après chaque itération)
     instruction;
```

Dans sa forme la plus simple, `for()` implémente une boucle infinie :

```
for(;;) { cout<<"Boucle infinie"<<endl; }
```

On rencontre fréquemment la macro-instruction suivante :

```
#define DOFOREVER (for(;;))
```

Les exemples d'utilisation de `for ()` sont innombrables. Ainsi, pour exécuter une boucle mille fois, on pourra utiliser :

```
for (int i = 0; i < 1000; i++) { ... /* boucle à effectuer */ ... }
```

`for ()` peut remplacer une itération `do ... while ()`, comme dans l'exemple suivant, où une fonction externe, appelée endOfLoop(), livre un résultat qui est utilisé dans l'itération `for ()`

```
int i;
for (i = 0; !endOfLoop(); i++) { ... }
```

Pour les habitués de PASCAL, il est nécessaire de tenir compte de quelques détails dans la boucle `for ()`. Tout d'abord, la variable servant à l'itération a une durée de vie qui peut excéder la durée de vie de la boucle `for ()`. Ainsi, la séquence d'instructions suivante est toujours vraie, contrairement à PASCAL, où la variable d'itération n'a de valeur définie que pendant l'itération. :

```
int i;
for (i = 0; i < 10; i++) cout<<i<<endl;
if (i == 10) ... // Forcément vrai !
```

Il est possible de modifier, à l'intérieur de la boucle, la valeur de la variable d'itération. Contrairement à PASCAL, la nouvelle valeur sera prise en compte dès l'itération suivante. De

fait, utiliser le terme de ”variable d’itération” en C++ dans le cadre de la boucle for () est abusif. La boucle `for ()` n’a en réalité pas vraiment de variable d’itération, mais

- une instruction permettant l’initialisation de l’itération. Cette instruction peut être vide, ou consister en une simple assignation de valeur à une éventuelle variable d’itération.
- une instruction contrôlant le déroulement de l’itération. Cette instruction n’a pas nécessairement de relations avec l’instruction d’initialisation. En particulier, la fin de l’itération pourrait être provoquée par un événement étranger au programme.
- une instruction permettant d’effectuer les opérations nécessaires après chaque itération. Cette instruction n’a pas forcément de relations avec les deux autres instructions.

9.3 *Labels, expressions, instructions composées*

Ce sous-chapitre regroupe les instructions les plus élémentaires de C++. En particulier, l'instruction nulle, n'impliquant aucune opération, est à classer dans ce groupe, bien qu'elle ne soit pas mentionnée de manière particulière dans le texte. L'instruction nulle est représentée par un point-virgule seul.

9.3.1 *Labels*

Un label est une étiquette que l'on "colle" à un endroit particulier du programme. La seule utilisation de label est pour définir la destination d'une instruction `goto` ("goto", page126). Le label doit être suivi de ":" pour être reconnu comme tel.

```
label : instruction;
```

Une utilisation - même peu fréquente - de labels dénote presque toujours une maîtrise insuffisante du langage. Cette critique est valable en C++ aussi bien qu'en n'importe quel langage structuré.

9.3.2 *Déclarations*

Une déclaration sert à définir un identificateur. Sous sa forme la plus élémentaire, une déclaration indique simplement au compilateur qu'un identificateur donné existe, et qu'il sera défini ailleurs (généralement plus loin dans le programme)

```
struct    aVeryComplexStructThatWillBeDefinedSomewhereElse;
```

La déclaration sert aussi à la définition de variables :

```
int       anInteger;
char      *aString;
float     aFloatValue;
```

9.3.3 *Affectations*

Une instruction d'affectation permet l'assignation d'une valeur à un identificateur. L'identificateur doit désigner une variable (ou, pour l'initialisation, une constante) de même type, ou d'un type compatible, que la valeur que l'on veut lui assigner.

Les exemples suivants sont des exemples valides d'affectations :

```
int a;
a = 1;
a = a + 1;              // a == 2
a *= 4;                 // a == 12
```

9.3.4 *Appels de fonctions*

Un appel de fonction est implémenté, en C++, par l'identificateur de la fonction suivi entre parenthèses, de la liste d'arguments que l'on désire passer à cette fonction. Si l'on ne passe aucun argument, alors les parenthèses doivent tout de même être mentionnées :

```
openFile(”FILE.TXT”);
abort();
```

9.3.5 *Instructions composées*

En principe, les diverses constructions du langage (`if`, `while`, etc...) n'admettent qu'une seule instruction. Comme il est difficile d'implémenter quelque action que ce soit au travers d'une seule instruction, C++ permet de définir des instructions composées, formées d'une séquence arbitrairement complexe d'instructions simples. Ces instructions composées forment ce que l'on appelle un **bloc d'instructions**. Un bloc d'instructions débute par une parenthèse C gauche ({, ou accolade gauche), suivi d'une liste d'instructions élémentaires, et se termine par une parenthèse C droite (}, ou accolade droite). Il n'est pas indispensable de terminer un bloc d'instructions par un point-virgule. Le fragment de code çi-dessous réalise exactement la même chose que l'exemple donné dans “Instruction while ()”, page120, mais sous une forme légèrement différente.

```
char ch;
ifstream f;
f.open(”tmp.tmp”);      // Ouverture d’un fichier
if (f.fail())           // Le fichier ne peut être ouvert
                        // Instruction composée
     {
     cerr<<”Cannot open tmp.tmp”<<endl;
     exit(1);
     }
do
                        // Le fichier a pu être ouvert
                        // Instruction composée
     {
     ch = f.get();
                        // Si EOF, on n’imprime pas le caractère -->
                        // pas d’instruction composée.
     if (ch != EOF) cout<<ch;
     }
while (!f.eof());
f.close();
```

9.4 *Sauts (break, return, goto, continue)*

Les sauts permettent le transfert inconditionnel de l'exécution d'un programme d'un point vers un autre.

9.4.1 *break*

`break` permet la terminaison d'une instruction itérative, comme `while`, `do ... while`, `for()`. Il est également utilisé dans le cadre de l'instruction `switch ()` pour interrompre l'exécution séquentielle des diverses clauses `case`.

9.4.2 *return*

`return` termine l'exécution d'une fonction. Si la fonction n'est pas censée retourner de valeur (fonction void, ou procédure), `return` termine simplement l'exécution, quel que soit l'état d'exécution de la fonction considérée; si aucune terminaison anticipée n'est désirée, alors `return` peut être omis dans ce cas particulier. Si la fonction est censée retourner une valeur (fonction typée), l'instruction `return` doit être suivie d'une expression de même type que le type retourné par la fonction. Dans ce cas, `return` *expression* est mandatoire.

```
#include <limits.h>

int minValue(int *array, int sizeOfArray)

     {
     int minVal = INT_MAX;
     for (int i = 0; i < sizeOfArray; i++)
          if (array[i] < minVal) minVal = array[i];
     return minVal;
     }
```

9.4.3 *goto*

On dit souvent que l'instruction `goto` de PASCAL est à proscrire. Cette proposition est encore plus valable en C++, à tel point que la plupart des exemples d'utilisation de `goto` sont franchement artificiels.

```
goto      label;
```

L'instruction goto accompagné d'un identificateur d'étiquette (label) permet le transfert inconditionnel du contrôle à l'instruction spécifiée par le label.

9.4.4 *continue*

Dans le cadre d'une instruction itérative (`for`(), `while`, `do...while`()), il peut être utile de terminer prématurément l'itération en cours pour passer à la suivante, par exemple

lorsque le programme peut se rendre compte rapidement que toute analyse supplémentaire est superfétatoire. C'est exactement ce qu'implémente l'instruction `continue`. L'itération en cours est abandonnée, et l'itération suivante est entamée. L'utilisation de `continue` en dehors du cadre d'une instruction d'itération est prohibée.

9.5 *Conversions de types*

C++ offre la possibilité de convertir une variable d'un certain type en un autre type. Cette opération est souvent appelée *casting*. Il existe des conversions implicites, que le compilateur effectue automatiquement si le besoin s'en fait sentir, et des conversions explicites, imposées par le programmeur.

Certaines conversions sont qualifiées de "sûres", alors que d'autres ne le sont pas. Un cas typique de conversion peu sûre est celle impliquant une troncature, comme par exemple :

```
float     pi = 3.1415926;
int       k;
k    = pi;     // Conversion implicite float ->> int
```

Normalement, le compilateur prévient le programmeur lorsqu'il rencontre une conversion de ce type, au moyen d'un *warning.*

Selon le type de machine, une conversion peut être sûre ou non. Ainsi, sur une machine donnée, short peut avoir la même dimension que int, rendant ainsi la conversion `int` ->> `short` parfaitement sûre, alors que le transport sur une autre machine, pour laquelle `short` est représenté par 16 bit, et `int` par 32 bit, va générer un *warning* de la part du compilateur C++.

9.5.1 *Conversions implicites*

Une conversion implicite est générée de manière automatique, sans que le programmeur aie à s'en inquiéter. Il en va ainsi des exemples suivants :

```
char c = 3;                  // int ->> char
int       anInt = c;         // char ->> int

int  val = 3.1415926;        // float ->> int. warning !
```

Dans certaines opérations arithmétiques, le compilateur doit effectuer des conversions implicites qui peuvent parfois conduire à des résultats inattendus. Ainsi :

```
int val = 3.1415926;         // val = 3;
float     fval = val + 3.1415926;      // fval = 6.1415926
```

Dans l'exemple précédent, val est tout d'abord converti en un double (= 3.0) , puis on lui additionne 3.1415926. Par contre :

```
val = val + 3.1415926;       // val = 6.0
```

Le résultat est évalué à 6.1415926, puis reconverti en int, ce qui donne 6. Deux conversions ont donc eu lieu. En principe, les opérations arithmétiques sont effectuées dans la précision de la variable la plus précise, le résultat étant tronqué par la suite. Ainsi :

```
int       i = 10;
i *= 2.3;          // i = 23, et non 20.
```

9.5.2 *Conversions explicites*

Une conversion explicite est déterminée par l'utilisateur, au moyen de la syntaxe suivante :

```
type      (expr);   // expr est converti en type
```

Alternativement, on peut également utiliser :

```
(type)    expr;     // expr est converti en type
```

Une utilisation fréquente de la conversion explicite apparaît lorsque l'on utilise des pointeurs void* pour écrire du code générique. Un pointeur sur n'importe quel type peut être implicitement converti en un pointeur void*, mais l'inverse n'est pas possible, parceque la conversion n'est potentiellement pas sûre. Le programmeur doit explicitement signaler au compilateur quel est le type résultat, c'est à dire quelle est la conversion à appliquer.

```
int       ival;
void      *pv;
pv = &ival;        // Conversion implicite
cout<<*pv<<endl;   // Erreur, on ne peut déréférencer un void*
cout<<*((int*) pv)<<endl;        // Ok conversion explicite
```

On utilise aussi les conversions explicites pour éviter tous les tests de type. Un cas fréquent où la conversion est nécessaire est pour convertir un pointeur sur un objet constant en un pointeur sur un objet non-constant. Ce cas se présente en particulier lorsque l'on a besoin de passer une chaîne de caractères (déclarée comme `const char* str = "Toto"`, par exemple) à une fonction externe demandant un `char*` comme paramètre. Ce cas se présente en particulier lors de l'utilisation de fonctions provenant d'autres fournisseurs, qui ont développé leurs fonctions de manière à ce qu'elles soient également utilisables par des versions plus vieilles de compilateurs C, ne connaissant pas le mot réservé `const`.

Enfin, il est parfois utile d'utiliser une conversion explicite pour éviter certaines ambiguïtés dans le cas où plusieurs conversions sont possibles, et pour documenter le programme.

9.6 *Test*

1. Le segment de programme suivant est-il correct ? Dans le cas contraire, comment réécrire ce code ?

```
int  estUnNombrePair(int unParametre)
     {
     if (unParametre % 2) return 0;
     }
```

2. Ecrire une petite procédure qui convertit tous les caractères d'une chaîne passée en paramètre en majuscules. On admettra que l'on dispose d'une machine où les caractères sont représentés en USASCII (code ITU no 7).
3. Que vaut la variable a après l'appel de la procédure dans les trois cas mentionnés ?

```
void      procedure(int& a)
     {
     switch (a)
          {
          case 0 : a++; break;
          case 1 : a++;
          case 2 : a++; break;
          case 3 : a = 10;
          default :
               a = 0;
          }
     );

void main()
     {
     int a;
     a = 1;
     procedure(a);
     procedure(a);
     procedure(a);
     };
```