Table des matières

A. Blocs d'instructions et conditions

1. Qu'est ce qu'un bloc d'instructions ?

a. Définition

- Un bloc d'instruction est UNE instruction composée de plusieurs instructions qui se suivent.
- En C (et tous les langages dérivés) il est délimité avec les opérateurs { } (accolades ouvrante et fermante).
- Un bloc peut contenir d'autres blocs imbriqués
- Dans un fichier source il ne peut pas y avoir d'instruction en dehors d'un bloc (sauf directives macro-processeur et déclarations de variables globales ou de fonctions). Pour être valides toutes les instructions doivent être localisées dans un bloc. Le bloc au sommet est celui de la fonction main() qui réunit in fine toutes les instructions du programme.

b. Exemple

```
#include <stdio.h>
#include <stdlib.h>

int main()
{  //------------------------- ouv bloc main B1
int x,pasx;    // déclarations de variables locales au bloc main
               // elles sont visibles (accessibles) dans ce bloc
               //et tous les sous blocs

   {  //----------ouv B2
      int c;
      x=0;
      c=rand()%256;
      pasx=rand()%5;
   }  // ---------ferm B2
   x=640;

   {  // ---------ouv B3
      //c=10; // provoque erreur, c non visible dans ce bloc
      x/=2;
      pasx=15;

   }  // ---------ferm B3
   x+=pasx;
   printf("x vaut : %d\n",x);  // affichage ?

   return 0;

}  //-------------------------ferm bloc main
```

Les blocs posent la question de la visibilité des variables. En fait la durée de vie des variables en générale est associée au bloc dans lequel elles sont déclarées. C'est pourquoi elles sont visibles dans tous les sous blocs imbriqués et invisibles en revanche dans les blocs de niveau supérieur ou de même niveau mais séparés.

c. Utilité d'un bloc d'instructions

Dans l'exemple ci-dessus les blocs sont inutiles : ils n'influent en rien sur le déroulement linéaire du programme, il n'y a aucune modification de la succession des opérations et les supprimer revient au même sauf pour la déclaration de la variable c.

L'utilité des blocs est de permettre de rompre avec cette linéarité et d'introduire grâce à des instructions données par le langage :

- des sauts de bloc, ce sont les trois instructions if , if-else, if -else if -else
- des branchements, c'est l'instruction switch
- et des répétitions, ce sont les trois instructions de boucles while, do-while et for.

Les blocs permettent également de généraliser des portions de code qui se répètent dans un programme avec éventuellement des valeurs différentes. C'est ce que nous appelons une fonction. Une fonction est un bloc d'instruction doté d'une entrée de valeur et d'une sortie de valeur.

2. Définir une condition

a. Pourquoi une condition ?

Les sauts conditionnels permettent d'exécuter ou de ne pas exécuter un bloc d'instruction selon qu'une condition est remplie ou non. De même pour les boucles, un bloc d'instructions est répété tant qu'une condition fixée pour sa répétition reste vraie.

Par exemple pour l'intruction if, première forme du saut conditionnel, l'exécution du bloc est soumise à une condition de la façon suivante :

```
Si (condition vraie) Alors
    faire bloc d'instructions
FinSi
```

Si et seulement si la condition est remplie, c'est à dire est "vraie", alors les instructions du bloc qui en dépend sont exécutées.

Qu'est ce qu'une condition vraie ?

C'est une expression dont la valeur est différente de 0. A l'inverse une condition est considérée comme fausse si l'expression vaut 0.

b. Comment définir une condition ?

Définir une condition ? c'est écrire une expression qui sera évaluée comme vraie ou fausse. La plupart des conditions sont élaborées à partir de comparaisons de variables. Les comparaisons bits à bits sont parfois utilisées également.

c. Les opérateurs de comparaison

Il est possible de comparer entre elles des variables ou n'importe quelles expressions du point de vue de leurs valeurs. Soit deux expressions a et b (des variables ou des opérations) elles ont chacune une valeur et les comparaisons suivantes sont possibles :

```
a  >   b    a strictement supérieur à b
a  <   b    a strictement inférieur à b
a  >=  b    a supérieur ou égal à b
a  <=  b    a inférieur ou égal à b
a = =  b    a égal b (test d'égalité)
a  !=  b    a différent de b (test d'inégalité)
```

Le résultat de chaque expression vaut :

0 si c'est faux
1 si c'est vrai.

A votre avis, qu'imprime le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=10, b=5;
   printf("%d ",a < b);
   printf("%d ",a+b > 12);
```

```
   a = 5;
   printf("%d ",a >= b);
   printf("%d ",a == b);

   b = 4;
   printf("%d ",a <= b);
   printf("%d ",a == b);
   printf("%d ",a+3 != b*2);
   return 0;
}
```

Réponse :

```
0 1 1 1 0 0 0
```

Autre exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a,b;
   printf("%d ",56 < rand() );   //  1 si vrai, 0 si faux

   printf("%d ",a=rand() + b=rand() >= 1000 );
                                 // 1 si vrai, 0 si faux
   printf("%d ",a*b == 45);      // 1 si oui, 0 si non
         return 0;
}
```

Les résultats dépendent des retours de la fonction rand().

d. L'opérateur unaire NON !

L'opérateur ! placé à gauche d'une expression donne vrai si l'expression vaut 0 (faux) et faux si l'expression est différente de 0 (vraie). Par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=10, b=20;
   printf("%d ", !(a < b)); // imprime 0
   printf("%d ", !(a > b)); // imprime 1
   return 0;
}
```

Il est souvent utilisé dans des conditions à la place d'une égalité à 0, par exemple :

```
int a=10;

    printf("%d ", a==0);// imprime 0

    // et peut être remplacé par l'expression :

    printf("%d ", !a);   // qui imprime également 0
```

e. Priorités des opérateurs NON et comparaison

Toutes les opérations arithmétiques ainsi que décalages, cast, sizeof, NON et complément à 1 sont prioritaires par rapport aux comparaisons. En revanche les comparaisons sont prioritaires sur ET, OU inclusif et exclusif et affectations combinées (Annexe 1 : Priorité et associativité des opérateurs)

Par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=10, b=0, c=2;
   printf("a+b < c*10 vaut %d\n", a+b < c*10);
   printf("c*10 >= 66 vaut %d\n", c*10 >= 66);
   printf("a+b < c*10 >= 66 == 80 vaut %d\n",
          a+b < c*10 >= 66 == 80);
   return 0;
}
```

```
La première ligne donne 1
la seconde  0
la troisième 0
```

3. Mise en pratique : opérateurs de comparaison et NON

Exercice 1
Faire un programme qui affiche les résultats de ce qui suit :

```
int a, b, c;
    srand(5);
    Imprimez les résultats de :
    a=rand()%256;
    b=rand()%256;
    c=rand()%256;

    a<128
    b>128
    c==223
    a < b >= rand()%2 == 1
    a= b !=c +rand()%50;
    b= a==c
    c= rand()%10 < rand()%10 >= rand()%10 != rand()%10 ==rand()%10
```

Exercice 2
Qu'imprime le programme suivant ?

```
int main()
{
int a=10, b=0, c=2;
   printf("a+b < c*10 vaut %d\n", a+b < c*10);
   printf("c*10 >= 15 vaut %d\n", c*10 >= 15);
   printf("a+b < c*10 >= 15 == 1 vaut %d\n",
          a+b < c*10 >= 15 == 1);
   printf("a+b < c*10 <= 15 == 1 vaut %d\n",
          a+b < c*10 <= 15 == 1);
   return 0;
```

```
}
```

B. Sauts Conditionnels

1. L'instruction if

SI et seulement si l'expression est vraie ALORS le bloc des instructions associées au if est exécuté. Il ne peut y avoir qu'un seul bloc associé au if.

```
if ( expression1 vraie){
    bloc instructions;
}
```

Par exemple, soit une variable a, dans un programme on peut écrire :

```
if ( a >=100 ){  // test
    printf( "a est superieure ou egale à 100 \n"); // instruction
}
```

Derrière un if il ne peut y avoir qu'un seul bloc c'est à dire une seule instruction. S'il y a plusieurs instructions il faut d'ouvrir et fermer le bloc mais s'il y en a qu'une c'est inutile. L'exemple précédent peut s'écrire sans erreur à la compilation :

```
if ( a >=100 )
    printf( "a est superieure ou egale à 100 \n");
```

Attention à l'indentation, c'est inutile pour la machine en revanche c'est absolument nécessaire professionnellement pour rendre son code lisible.

Dans le cas d' une succession de if, chacun fait l'objet d'une évaluation :

```
if ( expression1 vraie){
    bloc instructions 1;
}
if ( expression2 vraie){
    bloc instructions 2;
}
if ( expression3 vraie){
    bloc instructions 3;
}
```

Seuls les blocs dont la condition est vraie seront effectués : aucun, quelques uns ou tous.

Des if peuvent être imbriqués, dans ce cas ils ne seront peut-être pas tous effectués, la série s'arrêtera au premier test faux rencontré :

```
if ( expression1 vraie){
    bloc instructions 1;
    if ( expression2 vraie){
        bloc instructions 2;
        if ( expression3 vraie){
            bloc instructions 3;
        }
    }
}
```

Si (et seulement si) expression1 est vrai, alors les instructions 1 du bloc sont exécutées et expression2 est évaluée.

Si (et seulement si) expression2 est vrai, alors les instructions 2 du bloc sont exécutées et expression3 est évaluée.

Si (et seulement si) expression3 est vraie, alors les instructions 3 du bloc sont exécutées.

2. Le couple d'instructions if-else

SI l'expression est vraie le bloc des instructions associées au if est exécuté, SINON c'est le bloc des instructions suivantes qui est exécuté. Le else doit être collé à la fin du bloc du if. Il ne peut pas y avoir des instructions entre les deux :

```
if (expression1 vraie){
    bloc instructions 1;
}
else{
    bloc instructions 2;
}
```

Si expression1 est vrai (valeur différente de 0) alors le bloc des instructions 1 est exécuté

Mais si expression1 est faux (vaut 0) alors c'est le bloc des instructions 2 qui est exécuté.

Quelque soit la valeur de l'expression1, un bloc d'instructions sera effectué, soit le premier, soit le deuxième. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=rand()%100;

          if (a<50)
                printf("a inferieur à 50\n");
          else
                printf("a supérieur ou égal à 50\n");
          return 0;
}
```

Une valeur est tirée entre 0 et 100. Le programme indique si la valeur est inférieure ou supérieure ou égale à 50.

Comme pour le if seul, il peut y avoir une succession de if-else, ou des if-else dans des if, if-else… Le else se rapporte toujours au if du bloc précédent, celui immédiatement au dessus.

```
if (expression1 vraie){
    bloc instructions1;
    if (expression2 vraie){
       bloc instructions2;
   }
}
else{
    bloc instructions3;
}
```

Si expression1 est vraie le bloc instructions1 est exécuté ensuite si expression2 est vraie le bloc instructions2 est exécuté et c'est fini. Sinon, c'est à dire si expression1 est faux alors c'est le bloc instructions3 qui est exécuté.

3. La forme contractée du if-else, opérateur conditionnel ?:

Soit par exemple la séquence suivante :

```
int a, b;

    a=rand()%10;
   b=rand()%10;

   if (a<b)
       a*=2;
    else
       b*=2;
```

Deux variables a et b sont initialisées avec des valeurs aléatoires entre 0 et 9 compris. Les deux variables sont comparées et la plus petite est multipliée par deux.

La comparaison peut s'écrire de la façon suivante :

```
(a < b) ? (a*=2) : (b*=2);
```

Ce qui se lit et s'exécute ainsi :

```
est ce que a < b ?  si oui faire a*=2 sinon faire b*=2
```

La forme générale est :

```
(expression test) ? si vraie instruction1 : si faux instruction2 ;
```

4. La cascade d' instructions if-else if-else

Lorsque l'on a une suite de if simples, tous les tests sont effectués et toutes les instructions peuvent être exécutées s'ils sont tous vrais. Dans le cas du if suivi d'un else, ce sera l'un ou l'autre des deux blocs uniquement qui sera exécuté.

Tout en restant dans le principe exclusif du if-else, il est possible d'allonger la liste des tests grâce à l'instruction "else if", c'est le troisième principe du saut :

```
if(expression1 vraie){
    instructions1;
}
else if (expression2 vraie){
    instructions2;
}
else if (expressions3 vraie){
    instructions3;
}
else{
    instructions4;
}
```

Si expression1 est vrai,
    le bloc des instructions1 est exécuté,

Sinon si expression2 est vrai,
    c'est le bloc des instructions2 qui est exécuté,

sinon si expression3 est vrai,
    c'est le bloc des instructions3 qui est exécuté,

et sinon
    c'est le bloc des instruction4 qui est exécuté.

Il ne peut y avoir qu'un seul bloc exécuté. Dés qu'un test vrai est trouvé les évaluations s'arrêtent. Si aucun test n'a donné vrai, alors c'est le bloc d'instruction du else final qui est exécuté.

5. Expérimentation : les sauts conditionnels (les trois if)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

*****************************************************************
2 : SAUTS CONDITIONNELS DE BLOCS / 3 possibilités

   Forme 1 :

      Si (condition vraie) Alors
         faire bloc d'instructions
      FinSi

   COMMENT DEFINIR UNE CONDITION ?
   -> les opérateurs de comparaison : >, >=, <, <=, ==, !=
   -> les expressions sont évaluées VRAIE (valeur 1) ou FAUX
   (valeur 0)

   REMARQUE :
   En C
   - toute expression qui vaut 0 peut être considérée comme fausse
   - et toute expression qui vaut autre chose que 0 peut être
     considérée comme vraie

*****************************************************************/

int main()
{
int d1,d2;
   srand(time(NULL));
   // tirage aux dés
   d1=1+rand()%6;
   d2=1+rand()%6;
   if (d1>d2)
      printf("le de 1 gagne : d1=%d, d2=%d\n",d1,d2);
   if (d2>d1)
      printf("le de 2 gagne : d1=%d, d2=%d\n",d1,d2);

   // pour voir, affichage de la valeur du test
   printf("d1>d2 vaut %d,d1<d2 vaut %d\n",d1>d2, d1<d2);

   //REMARQUE :
   //si une seule instruction, les { } ne sont pas nécessaires

   return 0;
}

/*****************************************************************
SAUTS CONDITIONNELS DE BLOCS

   Forme 2 :

      Si (condition vraie) Alors
```

```
         faire bloc d'instructions 1
      Sinon
         faire bloc d'instructions 2
      FinSi

µ*************************************************************/
/*
int main()
{
int d1,d2;
   srand(time(NULL));
   // tirage aux dés
   d1=1+rand()%6;
   d2=1+rand()%6;
   if (d1>d2)
      printf("le de 1 gagne : d1=%d, d2=%d\n",d1,d2);
   else
      printf("le de 2 gagne : d1=%d, d2=%d\n",d1,d2);

   return 0;
}
*/
/
*****************************************************************
SAUTS CONDITIONNELS DE BLOCS

   Forme 3 :

      Si (condition 1 vraie) Alors
         faire bloc d'instructions 1
      sinon Si (condition 2 vraie) Alors
         faire bloc d'instructions 2
      sinon Si (condition n vraie) Alors
         faire bloc d'instructions n
      Sinon
         faire bloc d'instructions par défaut
      FinSi

*************************************************************/
/*
//TEST 1
int main()
{
int d1,d2;
   srand(time(NULL));
   // tirage aux dés
   d1=1+rand()%6;
   d2=1+rand()%6;
   if (d1==d2)
      printf("de 1 et de 2 execo : d1=%d, d2=%d\n",d1,d2);
   else if (d1>d2)
      printf("le de 1 gagne : d1=%d, d2=%d\n",d1,d2);
   else
      printf("le de 2 gagne : d1=%d, d2=%d\n",d1,d2);

   return 0;
}
*/
/*
// TEST 2
```

```
int main()
{
int d;
   srand(time(NULL));
   // tirage aux dés
   d=1+rand()%6;

   if (d==1)
      printf("d vaut 1 : %d\n",d);
   else if (d==2)
      printf("d vaut 2 : %d\n",d);
   else if (d==3)
      printf("d vaut 3 : %d\n",d);
   else if (d==4)
      printf("d vaut 4 : %d\n",d);
   else
      printf("d vaut 5 ou 6 : %d\n",d);


   return 0;
}
*/
```

6. Mise en pratique : les sauts conditionnels

Exercice 1
Écrivez un programme qui demande deux entiers et indique lequel est le plus petit, lequel est le plus grand.

Exercice 2
Faire le programme qui lit deux variables au clavier et les affiche en ordre croissant

exercice 3
Faire un programme qui demande un entier et indique s'il est pair ou impair

Exercice 4 :
Donnez l'affichage de chacune des trois séquences suivantes pour :

a égal à 100,
a égal à 1

Séquence 1	Séquence 2	Séquence 3

`if ( a < 50 )`
`  printf("1");`
 	
`if ( a < 40 )`
`  printf("2");`
 	
`if ( a < 30 )`
`printf("3");`
 	
`if ( a < 20 )`
`printf("4");`
 	
`if ( a < 10 )`
`printf("5");`
	`if ( a < 50 )`
	`    printf("1");`
	`else`
	`    printf("2");`
	
	`if ( a < 30 )`
	`    printf("3");`
	`else`
	`    printf("4");`
		`if ( a < 50 )`
		`   printf("1");`
		`else if ( a < 40`
		`)`
		`   printf("2");`
		`else if ( a < 30`
		`)`
		`   printf("3");`
		`else if ( a < 20`
		`)`
		`   printf("4");`
		`else if ( a < 10`
		`)`
		`   printf("5");`

		`else` `  printf("0");`

Exercice 5
Faire un programme pour jouer à pile ou face. Au départ l'utilisateur choisit pile ou face, le programme lance la pièce et donne le résultat gagné ou perdu.

Exercice 6
Écrire le jeu du nombre caché : un nombre est caché, le joueur essaie de trouver lequel. Le programme indique si le joueur a gagné ou de combien le nombre donné par le joueur est trop grand ou trop petit.

Exercice 7
Deux créatures hargneuses possèdent un même pouvoir mais lorsqu'elles se rencontrent celle qui en possède le plus détruit l'autre. Imaginer une solution en C et faites un programme d'illustration.

Exercice 8
L'utilisateur entre deux signes choisi parmi + et - et le programme indique le signe du produit (celui donné par une multiplication avec les deux signes). Programmez.

Exercice 9
Écrire un programme qui lit trois variables au clavier est affiche le maximum des trois.

Exercice 10
Une entreprise vends deux types de produits. Les produits de type A qui donnent lieu à une TVA de 5,5% et les produits de type B qui donnent lieu à une TVA de 19,6%. Écrire un programme qui lit au clavier le prix hors taxe d'un produit, saisit au clavier le type du produit et affiche le taux de TVA et le prix TTC du produit.

Exercice 11
Un personnage arrive devant une porte. Un gardien est devant l'entrée. Il pose une question au personnage. Si le personnage donne la bonne réponse il peut passer, sinon il est détruit.
Faire un programme qui traduise cette situation en langage C.

C. Branchements

1. Branchement sélectif : switch, case et break

Une suite de if, else if, else comme :

```
if (i==0){
   instructions0;
}
else if (i==1){
   instructions1;
}
else if (i==7){
   instructions7;
}
else if (i==55){
   instructions55;
```

```
}
else{
   instructions_n;
}
```

peut être remplacée par un switch qui est un aiguillage. Il fonctionne de la façon suivante :

```
switch(valeur_expression){

    case expression_constante_1 :
           instructions1;
           break;

   case expression_constante_2 :
           instructions2;
           break;

   case expression_constante_3 :
           instructions3;
           break;

   default :
           instructions_n;
           break;
}
```

Le bloc des instructions à exécuter est décidé à partir de la valeur de l'expression en paramètre du switch :

- si ce paramètre vaut expression_constante_1, instructions1 sont exécutées
- si ce paramètre vaut expression_constante_2, instructions2 sont exécutées
- si ce paramètre vaut expression_constante_3, instructions3 sont exécutées
- si ce paramètre a une autre valeur que les différents cas proposés, instructions_n sont exécutées.

Il n'y a pas de limite au nombre de cas possibles. Mais chaque cas est identifié par une valeur constante c'est-à-dire non variable après le mot clé case.

L'instruction break provoque une sortie du bloc du switch. Sans le break, Toutes les instructions des cas qui suivent le cas d'entrée sont effectuées.

Notre suite de if, if else peut s'écrire :

```
switch( i ){

    case 0 :
           instructions0;
           break;
    case 7 :
           instructions7;
           break;
    case 55 :
           instructions55;
           break;
    default :
           instructions_n;
}
```

Selon la valeur de la variable i les instructions du cas correspondant sont effectuées jusqu'au break. Si i ne correspond à aucun des cas alors c'est le cas par défaut qui est effectué et s'il n'y a pas de cas par défaut aucune instruction n'est effectuée.

Chaque cas peut proposer plusieurs instructions et correspond à une suite d'instructions. De ce fait ce n'est pas un bloc d'instructions (qui est considéré par la machine comme une instruction unique, composée) et les accolades sont inutiles.

Exemple, une interface pour une base de données commerciale propose quatre actions : 'a' pour Afficher la liste des clients, 'b' pour afficher les données d'un client, 'c' pour saisir un client et 'q' pour quitter. Nous pouvons avoir une séquence du type :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int choix;

   // menu utilisateur
   printf( "a : Afficher la liste des clients\n"
           "b : Afficher les donnees d'un client\n"
           "c : Saisir les donnees d'un client\n"
           "q : quitter\n");
   // récupération du choix de l'utilisateur
   scanf("%c",&choix);

   // aiguillage sur les instructions correspondant au  choix
   switch(choix){
         case 'a' :
               printf("affichage de la liste des clients\n");
   /* mettre ici le code pour l'affichage de tous les clients*/
               break;

         case 'b' :
               printf("Afficher les données d'un client\n");
               printf("Nom du client ? :\n);
   /* mettre ici le code pour l'affichage des infos du client*/
               break;

         case 'c' :
               printf("Enter les données du client : \n");
   /* mettre ici le code pour la saisie des infos du client*/
               break;

         case 'q' :   // instruction de sortie
                      // (il faut une boucle)
               break;
         default :   printf("saisie de commande incorrecte\n");
   }
   return 0;
}
```

2. Rupture de séquence : goto avec étiquette

Le goto est une instruction qui permet un branchement direct, sans condition, de quelque part vers une étiquette. L'étiquette de référence doit figurer dans la même fonction que le goto.

Exemple d'utilisation, dans un bloc, dans une fonction on pourrait avoir :

```
{
    (…)
    if (erreur_fatale==1){
          goto erreur;
        (…)
    }


erreur :
    printf( "erreur non récupérable\n");
    exit(EXIT_FAILURE);

}
```

"erreur :" est une étiquette positionnée à un endroit dans le bloc de la fonction. L'instruction goto erreur provoque un saut jusqu'à cet endroit de la fonction et à partir de là les instructions qui s'y trouvent sont exécutées.

Attention !

En principe un programme bien structuré n'a jamais besoin d'utiliser un goto et il est fortement déconseillé de s'en servir parce qu'il affaiblit la structuration et l'organisation du programme. La lecture et l'écriture du programme peuvent même devenir confuses. Il doit toujours être remplacé par une forme plus structurée. Le seul cas où parfois en C il est considéré comme admissible est dans un contrôle d'erreur, lorsqu'il permet de sortir d'une situation désespérée.

3. Expérimentation : Branchement sélectif switch

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*************************************************************
AIGUILLAGE

   La Forme if – else if - else peut être remplacée par :

   Selon (valeur)
      cas 1 : faire bloc d'instructions 1
      cas 2 : faire bloc d'instructions 2
      cas n : faire bloc d'instructions n
      defaut : faire bloc d'instructions par défaut
   FinSelon

*************************************************************/
// Par exemple ce programme :
int main()
{
int d;
   srand(time(NULL));
```

```
   // tirage aux dés
   d=1+rand()%6;

   if (d==1)
      printf("d vaut 1 : %d\n",d);
   else if (d==2)
      printf("d vaut 2 : %d\n",d);
   else if (d==3)
      printf("d vaut 3 : %d\n",d);
   else if (d==4)
      printf("d vaut 4 : %d\n",d);
   else
      printf("d vaut 5 ou 6 : %d\n",d);
   return 0;
}

// peut s'écrire :

/*
int main()
{
int d;
   srand(time(NULL));
   // tirage aux dés
   d=1+rand()%6;

   switch(d){
      case 1  : printf("d vaut 1 : %d\n",d);    // break;
      case 2  : printf("d vaut 2 : %d\n",d);    // break;
      case 3  : printf("d vaut 3 : %d\n",d);    // break;
      case 4  : printf("d vaut 4 : %d\n",d);    // break;
      default :printf("d vaut 5 ou 6 : %d\n",d);
   }
// REMARQUE : l'instruction break; provoque la sortie immédiate du
// bloc switch, sans break les cas suivants sont aussi exécutés

   return 0;
}
*/
```

4. Mise en pratique : l'aiguillage switch

Exercice 1
Qu'imprime la séquence suivante :

```
scanf("%d", &choix);
switch( choix ){
    case 1 :  printf ("bonjour");  break;
    case 2 : printf ( "bonsoir");  break;
    default : printf ( "hello") ;       break;
}
```

Pour choix égal à 0, 1 et 3 ?

Exercice 2
Faire un programme qui affiche une phrase différente à chaque lancement.

Exercice 3

Écrire un programme qui lit deux nombres entiers a et b et donne le choix à l'utilisateur :
1. de savoir si la somme de a+b est paire.
2. de savoir si le produit a*b est pair.
3. de connaître le signe de a-b;
4. de connaître le signe de a*b

Exercice 4
Au lancement du programme le programme demande à l'utilisateur de se situer par rapport à l'apprentissage du langage C : excellent, bien, honnête, mauvais, horrible. En fonction de la réponse le programme propose une solution ou donne un conseil. Programmez.

Exercice 5
Écrire un programme affichant un menu proposant de jouer avec un, deux, trois ou quatre dés. Selon le choix fait le programme lance les dés. Les dés identiques sont relancés et il y a cumul des points. L'utilisateur gagne si : si le total est supérieur au deux tiers du maximum (avec deux dés ça fait 8 : (12 / 3) *2). Le programme indique combien il manque pour gagner ou combien il y a de plus.

Exercice 6
Un personnage arrive devant une porte. Un gardien est devant l'entrée. Il pose une question au personnage. Si le personnage donne la bonne réponse il peut passer, sinon il est détruit.
Faire un programme qui traduise cette situation en langage C.

Exercice 7
Faire un programme qui donne votre horoscope numérologique. L'utilisateur entre le sa date de naissance. Le programme calcul le numéro correspondant. Par exemple soit une date de naissance 09051969 il faut additionner tous les chiffres jusqu'à n'en avoir qu'un : 0+9+0+5+1+9+6+9 donne 39, 3+9 donne 12, 1+2 donne 3, le numéro correspondant à la date de naissance est 3. A partir du numéro obtenu le programme donne un horoscope pour l'année en cours.

D. Les tests multi-conditions (ET/OU)

1. Conjonction ET, opérateur &&

a. ET avec deux expressions membres

soit deux expressions, E1, E2,

```
l'expression  E1 && E2 :
    est vraie (vaut 1) si E1 ET E2 sont vraies
    est fausse sinon (vaut 0)
```

Exemple :

```
int a = rand()%300;
    if (a >= 100 && a<=200)
           printf("a compris entre 100 et 200\n");
```

le test vaut 1 si la valeur de a est comprise dans la fourchette 100-200 bornes comprises, 0 sinon.

Autre exemple à la douane :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int papier, declarer;

   printf("Vous avez vos papiers ? (o/n)\n");
   scanf("%c",&papier);
   // lorsqu'il y a plusieurs appels successifs de scanf() il est
   // nécessaire de réinitialiser le buffer d'entrée (stdin) avec
   // la fonction rewind()
   rewind(stdin);

   printf("Quelque chose à déclarer ? (o/n)\n");
   scanf("%c",&declarer);

   if( papier=='o' && declarer=='n')
          printf ("C'est bon, vous pouvez passer\n");
   else
          printf("Attendez la s'il vous plait\n");

   return 0;
}
```

Si le voyage a ses papiers et n'a rien à déclarer il peut passer.

b. ET avec plus de deux expressions membres

Si il y a plus de 2 expressions membres :

```
E1 && E2 && E3 && E4  est vraie si TOUTES sont vraies, faux sinon
```

Exemple :

```
if (x>0 && x<1024 && y>0 && y<768)
     printf("(x,y) dans ecran en 1024-768\n");
```

Remarque :

Dés qu'elle trouve une expression membre fausse, l'expression entière donne alors faux et les expressions restantes ne sont pas regardées.

Autre exemple liste pour pâte à crêpes :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
char farine, sel, œufs, lait, gruyere;

   printf("Dans le frigo y a t-il des oeufs (o/n), du lait (o/n)"
          "du gruyere (o/n)\n ?");
   scanf("%c%c%c",&oeufs, &lait, &gruyere);
   rewind(stdin);

   printf("Dans le placard y a til de la farine (o/N)"
          "et du sel (o/n) ?\n");
```

```
    scanf("%c%c",&farine, &sel);

    if( oeufs=='o' && lait=='o' && gruyere=='o' &&
        farine=='o' && sel=='o')
        printf ("Top, je fais des crêpes !\n");
    else
        printf("No crêpe\n");
    return 0;
}
```

Si tous les ingrédients sont là nous pouvons faire des crêpes.

2. Disjonction OU, opérateur

a. OU avec deux expressions membres

Soit deux expressions, E1, E2,

```
l'expression  E1 || E2 :
    est vraie (vaut 1) si E1 OU E2 est vraie
    est fausse si les deux sont fausses (vaut 0)
```

Exemple :

```
int a = rand()%300;
   if (a <= 100 || a>=200)
      printf("a non compris entre 100 et 200\n");
```

Le test est vrai si a en dehors du segment 100-200.

Autre exemple à la douane

```
#include <stdio.h>
#include <stdlib.h>

int main()   // A LA DOUANE version OU
{
int papier=0,declarer=0;

   printf("vous avez vos papiers ? (o/n)\n");
   scanf("%c",&papier);
   rewind(stdin);

   printf("quelque chose a declarer ?(o/n)\n");
   scanf("%c",&declarer);
   rewind(stdin);

   if (papier == 'n' || declarer == 'o')
      printf("attendez la SVP\n");
   else
      printf("c'est bon vous pouvez passer\n");

   return 0;
}
```

Si le voyageur n'a pas ses papiers ou s'il a quelque chose à déclarer, il doit attendre.

b. OU avec plus de deux expressions membres

Si il y a plus de 2 expressions membres :

```
E1 || E2 || E3 || E4
    est vraie si au moins une est vraie
    est fausse si toutes sont fausses
```

Exemple :

```
if (x<0 || x>1024 || y<0 || y>768)
    printf("(x,y) n'est pas dans l'ecran en 1024-768\n");
```

Si le test est vrai le point de coordonnées (x,y) n'est pas dans l'écran.

Autre exemple, puis-je faire un sandwich au jambon ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
char pain, beurre, jambon, cornichon, salade;

   printf("Dans le frigo y a t-il :\n"
          "du beurre (o/n), des cornichons (o/n), "
          "de la salade (o/n)\n"
          "ai-je du pain frais (o/n)\n ?");
   scanf("%c%c%c%c%c", &pain, &beurre, &jambon,
                       &cornichon, &salade);

   if(pain=='n'||beurre=='n'||jambon=='n'||
      cornichon=='n'||salade=='n')
          printf ("Pas de sandwich au jambon !\n");
   else
          printf("Miam, un bon sandwich\n");
   return 0;
}
```

3. ET prioritaire sur OU

Si il y a plus de 2 expressions membres avec des ET et des OU :

```
E1 && E2 || E3 && E4 || E5   alors le && est prioritaire sur le ||

ce qui donne     : (E1 && E2) || (E3 && E4) || E5
équivalent à     :     E6     ||     E7     || E5

si E6 ou E7 ou E5 est vraie l'expression est vraie, fausse sinon
```

4. Priorité avec les autres opérateurs

La priorité plutôt faible est juste au-dessus de celle de l'opérateur conditionnel et des affectations (voir annexe 1)

5. Mise en pratique : les opérateurs logiques ET, OU

Exercice 1

Le jeu du bandit à bras multiple. Trois roues sont mises en parallèle elles ont sur leur tranche des numéros ou des lettres. Au départ elles sont lancées et tournent à des vitesses différentes lorsqu'elles s'arrêtent trois numéros ou lettres sont alignés : A5R ou 67T, on gagne lorsque certaines combinaisons sortent, par exemple : AAA, BCD, OIO… Faire un programme pour tenter sa chance un seul coup.

Exercice 2
X,Y, et Z étant des variables numériques, on considère les deux séquences algorithmiques S1 et S2 :

Séquence S1

```
si (X<5 ou Y>2) et Z>3 alors X=1
                            si (Z-Y) >0 alors Z=0
                        finsi
                           Y = Y+Z
sinon
    X = 2
    Z = Y+Z
finsi
```

Séquence S2

```
si  X<5 ou  (Y>2 et Z>3) alors X =1
                                si  (Z-Y) > 0 alors Z = 0
finsi
Y = Y+Z
sinon
    X = 2
    Z = Y+Z
finsi
```

1) Pour chacune des deux séquences, donner les valeurs, après exécution, de X, Y, et Z si l'on suppose qu'à l'état initial ces trois variables ont les valeurs :

```
a) X =4    Y =1          Z =4
b) X =4    Y =5          Z =4
c) X =1    Y =3          Z =1
```

2) Faire un programme pour vérifier vos réponses aux deux séquences.

Exercice 3
Dans un algorithme qui analyse des résultats d'examen, 4 variables permettent de décrire l'environnement :

les variables numériques Nlv, Nf, Nm, Np qui indiquent respectivement, pour un candidat donné, des notes littéraires : langue vivante (Nlv), de français (Nf) , et des notes scientifiques : mathématiques (Nm), et physique (Np). On suppose que les notes sont calculées sur 20 et quelles ont toutes le même coefficient.

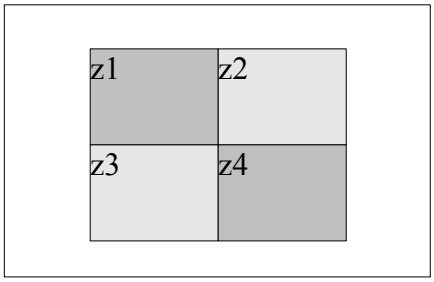
Formez les expressions logiques (et seulement elles) correspondant aux situations suivantes :

```
1) la moyenne des quatre notes est supérieure à 10
2) les notes de mathématiques et de français sont supérieures à la
moyenne des quatre notes
3) il y a au moins une note supérieure à 10
4) toutes les notes sont supérieures à 10
5) la moyenne (10) est obtenue pour l'un des deux types (littéraire
et scientifique)
6) la moyenne des quatre notes est supérieure ou égale à 10 et la
moyenne (10) est obtenue pour l'un des deux types
```

Exercice 4

Donnez l'algorithme qui détermine le nombre de valeurs distinctes parmi trois variables à faire saisir par l'utilisateur. (ex : 8, 8 et 8 saisi par l'utilisateur donne 1 valeur distincte…)
Comment faire avec 4 variables ?

Exercice 5
Nous sommes dans un jeu vidéo avec une résolution d'écran 640x480 pixels. Au centre il y a une zone rectangulaire de 580x420. Cette zone est divisée en quatre parties égales (z1,z2,z3,z4). Chaque partie est en fait un bouton cliquable :



Faire le programme qui :

- □ initialise deux variables x et y avec une position choisie aléatoirement dans l'écran
- □ affiche dans quelle zone se trouve cette position

Exercice 6
Un personnage dispose de plusieurs talents : Bravoure, force, patience, persévérance, résistance à la magie. Chaque talent est une valeur entre 0 et 100. Selon les épreuves, il peut prendre peur, se révéler faible, perdre patience, être tenté de tout laisser tomber. Faire le programme qui répond aux questions et situations suivantes :

- □ si la patience est inférieure à 50 la persévérance baisse de plusieurs degrés.
- □ a t-il autant de bravoure que de patience, et de force que de persévérance ?
- □ a t-il plus de patience que de force, bravoure et persévérance réunies ?
- □ est ce que sa résistance à la magie est égale à la moyenne des autres talents moins 1/10 ?
- □ pour se sortir d'une épreuve magique de niveau 1 il faut soit que sa bravoure et sa force soient comprises entre 30 et 70 soit que sa patience et sa persévérance soient supérieures à 40. Peut-il s'engager dans une telle épreuve ?
- □ il meurt dans une épreuve de survie de niveau 5 si sa force n'est pas au moins supérieure à 50, a t-il des chances de survivre s'il s'y engage maintenant ?
- □ un sort magique d'anéantissement le réduit en ombre si la force du sort est supérieure à la somme de ses talents multipliés par sa résistance à la magie. Tester avec un sort de force aléatoire
- □ lorsqu'il a autant de bravoure que de force, patience et persévérance sa résistance à la magie augmente de 20. Augmente t-il sa sa magie maintenant ?

Exercice 7
Nous sommes dans un monde de champignons microscopiques étranges : lorsque trois entités de la même espèce se touchent elles en produisent une quatrième de la

même espèce puis se dispersent. En revanche s'il n'y en a que deux qui sont de la même espèce la troisième est anéantie et les deux autres restent solidaires. Si elles sont toutes d'espèces différentes rien ne se passe.

Faire un programme simple qui simule le principe de cette interaction.

E. Boucles

1. Boucle TANT QUE : le while

La boucle while a la forme suivante :

```
while (expression vraie){
    instructions;
}
```

Tant que la valeur de l'expression est vraie, c'est-à-dire non nulle et différente de 0, les instructions du bloc lié à la boucle sont répétées. Pour que la boucle puisse s'arrêter il faut que l'expression devienne fausse et pour ce faire une des composantes du test doit être modifiée dans le bloc des instructions. Par exemple :

```
int a=3,i=0;
    while (i<a){
        i++;
        printf("i vaut %d\n",i);
    }
    printf("Fin de la boucle avec i=%d \n",i);
```

Tant que i est inférieur à a, les instructions sont exécutées,

```
au départ a vaut 3 et i vaut 0 :

i < a : le test est vrai, les instructions sont effectuées, i est
augmenté de 1 et i vaut 1.

i < a : le test est vrai, les instructions sont effectuées, i est
augmenté de 1 et i vaut 2.

i < a : le test est vrai, les instructions sont effectuées, i est
augmenté de 1 et i vaut 3.

i < a : le test est faux, fin de la boucle, poursuite de l'exécution
avec les instructions qui suivent le bloc de la boucle.
```

Le test peut être faux dés le départ et dans ce cas les instructions du bloc de la boucle ne sont pas exécutées, par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int a;
   printf("entrer une valeur entre 50 et 100\n");
   scanf("%d",&a);
   while( a < 50 ||  a>100 ){
      printf("le  nombre doit être compris entre 50 et 100\n");
      scanf("%d",&a);
      rewind(stdin);
   }
```

```
   printf("valeur entree : %d\n",a);
   retunr 0;
}
```

Si l'utilisateur entre au départ une valeur entre 50 et 100 le test de la boucle est faux et il n'y a pas de boucle. Il y a boucle tant que la valeur entrée par l'utilisateur est inférieure à 50 ou supérieure à 100.

2. Boucle FAIRE {…}TANT QUE : le do-while

La boucle do-while a la forme suivante :

```
do {
    instructions;
} while (expression) ;        // à noter : le point virgule
                              // après l'instruction de boucle
```

A la différence de la boucle while, les instructions précédent l'évaluation du test d'arrêt et de ce fait les instructions dans la boucle sont exécutées au moins une fois. Qu'imprime le programme suivant ? :

```
int a=0, i=10;
    do{
       i++;
        printf("i vaut %d\n",i);
    }while(i<a);
```

Réponse :

```
i vaut 11
```

La saisie d'un nombre entre 50 et 100 de l'exemple while ci-dessus peut s'écrire avec une boucle do-while justement parce qu'il est nécessaire que l'appel de la fonction scanf() ait lieu au moins une fois :

```
int a;
   do{
      scanf("%d",&a);
      rewind(stdin);
      if (a<50 || a>100 )
         printf("le  nombre doit être compris entre 50 et 100\n");
   }while( a<50 || a>100);
   printf("valeur entree : %d\n",a);
```

3. Boucle comptée POUR : le for

La boucle for a la forme suivante :

```
for (expr1; expr2; expr3 ){
    instructions;
}
```

avec :

```
expr1 : est une affectation qui a lieu une fois pour toute au
commencement de la boucle

expr2 : est le test d'arrêt évalué au début de chaque tour

expr3 : est une modification de variable  en relation avec le test
afin de terminer la boucle.
```

Comme pour toutes les boucles, tant que le test est vrai les instructions du bloc sont exécutées. Par exemple :

```
int i;
    for (i=0; i<3; i++)
       printf("i=%d\n",i);
```

L'ordre d'exécution se décompose en :

```
for ( i=0;                    // 1 initialisation
       i<3;                   // 2 test
       i++)                   // 4 incrémentation
     printf("i=%d\n",i);      // 3 bloc d'instructions
```

Première execution du bloc :

```
1) i est initialisé à 0, ensuite :
2) i est comparé à 3, le test est vrai
3) alors les instructions du bloc sont exécutées
4) et i est incrémenté de 1,  i égal 1,
```

Deuxième execution du bloc :

```
2) i est inférieur à 3,
3) les instructions du bloc sont exécutées,
4) i est incrémenté de 1 et i égale 2,
```

Troisième execution du bloc :

```
2) i est inférieur à 3,
3) les instructions du bloc sont exécutées,
4) i est incrémenté de 1 et i égale 3,
```

Arrêt de la boucle

```
2) i n'est pas inférieur à 3, le test est faux,

la boucle est terminée. Le programme passe aux
instructions qui suivent la boucle
```

Avec une boucle while c'est équivalent à :

```
i=0;
    while( i<3 ){
        printf("i=%d\n",i);
       i++;
    }
```

Chacun des trois champs de la boucle for (initialisation ; test ; incrémentation) peut faire l'objet d'une série d'expressions. Elles sont alors toutes séparées par des virgules. Par exemple :

```
for (      x=0, y=45, z=rand()%100;
           (x<500 && z < 400); x+=10,
           y+=5, z +=9
    )
    printf("x=%d, y=%d, z=%d",x,y,z);
```

Ce for utilise trois variables. La boucle se déroule dans l'ordre suivant :

```
1) elles sont initialisées au départ,

2) le test  porte uniquement sur les variables x et z,

3) il n'y a qu'une instruction dans le bloc : afficher les valeurs
de x, y, z

4) l'incrémentation a lieu pour chacune des trois variables.
```

Tous les champs de la boucle for n'ont pas obligation à être utilisés, mais les deux points virgule sont obligatoires. Par exemple une boucle infinie peut s'écrire :

```
for ( ; ; )
    instructions;
```

Dans ce dernier cas la sortie de boucle devra être forcée avec une instruction de rupture comme le break qui sera prévue dans le bloc d'instructions (voir la section sortie et saut forcés dans une boucle).

4. Boucles imbriquées

Bien entendu il est possible d'avoir plusieurs boucles imbriquées, par exemple :

```
while (expression1 vraie){
    instructions1;
    while expression2 vraie){
        instructions2;
    }
}
```

ou encore :

```
for( expr1 ; expr2 vraie ; expr 3){
    instructions1;
    while expression2 vraie){
        instructions2;
    }
}
```

Lorsque nous aborderons l'utilisation de tableaux à deux dimensions nous trouverons fréquemment des constructions du type :

```
for( expr1 ; expr2 vraie ; expr 3){
    for( expr4 ; expr5 vraie ; expr 6){
        instructions;
    }
}
```

5. Sortie et saut forcés dans une boucle

a. Sortir avec l'instruction break

Notamment il peut être utile de sortir d'une boucle autrement qu'avec un test en début ou en fin de bloc. Nous avons vu que l'instruction break peut provoquer la sortie immédiate du switch englobant le plus proche. Break peut également provoquer la sortie immédiate de la boucle englobante la plus proche. Par exemple dans le cas d'une boucle infinie il est possible de trouver :

```
for ( ; ; ){
    if( …)
       break;
    if( …)
       break;
    if( …)
       break;
}
```

Si l'un des tests est vrai le break provoque la sortie de la boucle for.

Attention dans le cas de boucles imbriquées, la sortie ne concerne que l'instruction de boucle la plus proche :

```
int toto=1000;
    while (1){                  // attention
       for ( ; ; toto=rand())
          if (toto<500)
              break ;   // provoque uniquement la sortie du for
    }
```

Dans cet exemple si toto vaut moins que 500, l'instruction break provoque la sortie de la boucle for mais pas de la boucle while. De ce fait la boucle for recommencera immédiatement juste après son interruption et il n'y a pas de sortie possible de la boucle while.

b. Passer à l'itération suivante avec l'instruction continue

L'instruction "continue" n'est pas souvent utilisée. Elle ne met pas fin à la boucle mais elle relance immédiatement à l'itération suivante la boucle while, do-while ou for dans laquelle elle se trouve. Dans le cas de while et de do-while elle déclenche une réévaluation de la condition d'arrêt. Dans le cas du for elle déclenche le passage à l'étape d'incrémentation.

Si par exemple j'écris :

```
for (i=0 ; i<100 ; i++){
    if (i>45 && i<60)
      continue;
      // instructions suivantes...
       (...)
}
```

Pour i de 46 à 59 les « instructions suivantes » ne sont pas examinées.

c. Sauter à un endroit avec l'instruction goto

Comme nous l'avons déjà mentionné plus haut, en cas de contrôle d'erreur, un goto peut permettre de sortir d'une boucle ou de plusieurs boucles imbriquées :

```
for (...)
    for (...){
      if (catastrophe)
          goto erreur;
    }
…
erreur :
   réparer les dégâts
```

Mais à part pour des contrôles d'erreurs et quelques expérimentations spécifiques précises l'utilisation du goto est déconseillée.

6. Mise en pratique : les boucles while, do-while et for

Exercice 1
Faire un programme qui affiche pour chaque valeur de la table ascii (de 0 à 255) le signe ou la lettre correspondant.

Exercice 2
Faire un programme qui permet de :

- Afficher tous les nombres de 0 à 1000
- Afficher tous les nombres de 0 à 100 sur 10 colonnes

- □ Afficher la table de multiplication
- □ Afficher la table d'addition

Exercice 3

- □ Ecrire le code qui permet d'obtenir : AAAAAAABBBBBZZZZZZZZTTT
- □ avec uniquement quatre appels de la fonction putchar()
- □ Ecrire le code qui permet d'avoir 10 fois la séquence de code ci-dessus.
- □ Combien de boucles faut-il utiliser pour obtenir l'affichage ci-dessous ?

```
00000100000
00001110000
00011111000
00111111100
01111111110
11111111111
```

- □ Ecrire la séquence de code correspondante.
- □ Tester le tout dans un programme.

Exercice 4

Est-il possible avec scanf() de faire un programme qui affiche des zéros en continue et, si une touche est appuyée, un nombre aléatoire de fois la lettre ou le chiffre correspondant ? Faire ce qui est possible.

Exercice 5

Qu'imprime le code suivant :

```
int i;
   for (i=0; i<B ; i++)
      printf("%d", B);

   putchar('\n');

   i--;
   while (i++ < B)
      printf("A");

   do{
         printf("C");
   }while (i < B);
```

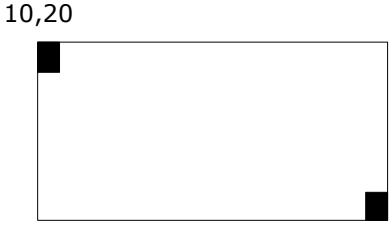
pour B=5, B= 1, B=0 ?

Exercice 6

Soit un segment horizontal [10, 100], faire un programme qui:
affiche toutes les postions par pas de un
affiche toutes les positions par pas de deux
affiche toutes les positions par un pas entré par l'utilisateur

Exercice 7

Soit un rectangle de position :



Afficher toutes les positions du rectangle en en les numérotant à partir de 0 et en augmentant de 1 pour chaque position (afficher avec la fonction printf(), le rectangle est collé à la marge de la console)

Exercice 8

Dans un programme :

- □ afficher un nombre aléatoire de fois le mot "bonjour"
- □ afficher "je repete le bloc" tant que l'utilisateur le demande(s'arrête quand l'utilisateur ne veut plus)
- □ obliger l'utilisateur à entrer un nombre pair entre 100 et 1000.

Exercice 9

Simulation d'une course d'escargots.
Au départ 4 à 5 escargots sur le bord gauche de la console. L'arrivée est à une distance de 60 caractères vers la droite (possibilité de la dessiner avec des 'I'). A l'issue de la course le programme affiche quel est le vainqueur.

Exercice 10

Tirage aux dés. Écrivez un programme qui simule un nombre n, entré par l'utilisateur, de tirages avec 6 dés. Le programme compte et affiche à la fin le nombre de coups où tous les dés ont eu la même valeur.

Exercice 11

Dichotomie. Écrire un programme qui génère aléatoirement un nombre que l'utilisateur devra découvrir. A chaque saisie, le programme dira si le nombre est plus petit ou plus grand. Si le nombre est trouvé par l'utilisateur, le programme lui annoncera qu'il a gagné en indiquant le nombre d'essais qui ont été nécessaires.

Exercice 12

Fabriquer un dé pipé. Par exemple la face 6 sort plus souvent. Tester votre dés pipé afin de mettre en évidence son efficacité. Sophistiquer le dé en privilégiant 3 faces, par exemple 1, 3, 6, tester pour voir si ça marche.

Exercice 13

Faire un programme où c'est l'ordinateur qui pipe un dé à six faces. Une valeur revient plus souvent et vous ne savez pas laquelle. A l'issue de plusieurs tirages vous gagnez si vous découvrez quelle est la face pipée. Éventuellement vous pouvez vous aidez avec un compte des faces sorties. Faire le même programme avec cette fois trois faces pipées.

Exercice 14

Faire un générateur de 100 mots imaginaires prononçables en français et produit par la machine, par exemple : tariduse, ujinoq, elepit, popelepe… Les mots seront affichés à l'écran les uns en dessous des autres.

Exercice 15

Ecrire un programme qui convertit un entier naturel en chiffres romains, en utilisant l'ancienne notation.
Exemple : 4 (IIII), 9 (VIIII), 900 (DCCCC)

Rappelons les éléments de base :
I : 1, V :5, X : 10, L : 50, C : 100, D : 500, M : 1000.

Exercice 16
Inventer un système de comptage inspiré de celui des romains et faites un programme qui convertit des entiers naturels entrés par l'utilisateur.

Exercice 17
Écrire un programme C qui propose à l'utilisateur de dessiner à l'écran certaines figures composées d'étoiles triangle, carré, sablier, des lettres…

Les figures seront proposées par un menu (triangle, carré, sablier…). La hauteur de la figure sera saisie par l'utilisateur. Exemple :

hauteur=4

```
   *          ****          *******
  ***         ****           *****
 *****        ****            ***
*******       ****             *
                               *
                              ***
                             *****
                            *******
```

F. Typiques utilisations de boucles

1. Faire un menu utilisateur

Un programme qui se termine du fait d'une action de l'utilisateur repose toujours sur une boucle, c'est la base de sa dynamique. Pour faire un menu le principe est de donner un choix de commandes à l'utilisateur via une interface et de mettre fin à la boucle si l'utilisateur le demande avec une commande spécifique. L'interface simple que nous proposons propose à l'utilisateur quatre choix :

```
printf ( "1 : Affiche bonjour\n"
         "2 : Affiche il fait beau\n"
         "3 : Entrer un nombre entier\n"
         "0 : Quitter\n");
```

Ensuite le programme capture le choix de l'utilisateur et applique les traitements correspondants. Si l'utilisateur entre d'autres nombres que 0,1,2 ou 3 le programme signale qu'ils ne correspondent pas à des commandes :

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
int choix, res;
   do{
                                                          // 1
       printf ( "1 : Affiche bonjour\n"                   // 2
                "2 : Affiche il fait beau\n"
                "3 : Entrer un nombre entier\n"
                 "0 : Quitter\n");

       scanf("%d",&choix);                                // 3
       rewind(stdin);

       switch(choix){                                     // 4
          case 0 :
             break;
          case 1 :
             printf("bonjour\n");
             break;

          case 2 :
             printf("il fait beau\n");
             break;

          case 3 :
             scanf("%d",&res);
             rewind(stdin);
             printf("le nombre est :  %d\n",res);
             break;

          default :
              printf("Pas de commande %d\n",choix);
             break;
      }

   }while(choix !=0);                                     // 5

   return 0;
}
```

(1) (5) Instruction de boucle do-while. La boucle a lieu tant la variable choix est différente de 0, elle se termine si la variable choix est égale à 0.

(2) Affichage du menu utilisateur.

(3) Première instruction exécutée au moins une fois, l'utilisateur entre une valeur pour la variable choix. L'appel de la fonction scanf() est suivi d'un appel à la fonction rewind() afin de réinitialiser le buffer des entrées clavier stdin et d'éviter une boucle infinie en cas d'erreur.

(4) A partir de la valeur contenue dans la variable choix un switch oriente vers le bloc d'instructions correspondant. Si la variable choix contient autre chose que 0,1,2,3, par défaut le programme affiche qu'il n'y a pas de commande de ce numéro.

Si l'utilisateur entre une valeur erronée ou une lettre c'est le case défaut qui est exécuté avant le réaffichage du menu. Pour quitter il faut explicitement choisir quitter.

2. Boucle d'événements dans un jeu vidéo

Dans la même ligne, voici une autre version de boucle d'évènements adaptée pour un jeu. La différence c'est de pouvoir capturer les entrées utilisateur sans pour autant arrêter le programme, comme c'est le cas avec la fonction scanf() de l'exemple précédent. Dans notre exemple les entrées utilisateur se limitent au clavier, mais en mode grapique avec une librairie adaptée nous pouvons avoir souris, joystick, capteurs etc.

a. Récupérer les entrées clavier (kbhit() et getch())

La fonction scanf() arrête le programme et attend une entrée. Rien ne se passe plus tant que l'utilisateur n'a pas entré quelque chose. Pour éviter cela nous allons utiliser deux autres fonctions :

```
int kbhit()
int getch();
```

La fonction kbhit() (abréviation de keyboard hit) indique si une touche du clavier a été enfoncée et renvoie une valeur vrai ou faux.

La fonction getch() (abréviation de get char) renvoie la valeur numérique ascii de la touche enfoncée.

Ces deux fonctions ne font pas partie des librairies standard du C. En général elles sont associées à une librairie nommée conio.h (abréviation de "console in out").

b. Boucle événements simple

La boucle d'événement ci-dessous permet d'afficher la touche qui est enfoncée en mode lettre (le caractère) et en mode numérique ascii (le nombre qui correspond au codage du caractère) et si aucune touche n'est appuyée des zéros sont affichés automatiquement les uns à la suite des autres :

```
#include <stdio.h>
#include <conio.h> // ne pas oublier d'inclure la librairie conio

int main()
{
   int res;
   while (res!='q'){
      if (kbhit()){
         res=getch();
          printf("touche %c pressee, val ascii : %d\n",res,res);
      }
      putchar('0');
   }
   return 0;
}
```

La fonction kbhit() indique si une touche du clavier a été appuyée. Si oui nous récupérons la valeur ascii de la touche avec la fonction getch() et nous affichons le résultat. Mais pendant qu'aucune touche n'est appuyée les instructions qui suivent le bloc du if sont exécutées : la page est remplie de zéro.

Problème : c'est très rapide et on ne voit presque rien alors nous allons ajouter un petit timer pour ralentir l'affichage des zéros.

c. Toper l'exécution si trop rapide

La fonction clock() renvoie le temps écoulé en millisecondes (ou en tics d'horloge, ça dépend des environnements) depuis le lancement du programme. Voici comment ajouter au programme précédent un petit contrôle du temps.

```
#include <stdio.h>
#include <conio.h>

int main()
{
int res, top=0;

   while (res!='q'){
      if (kbhit()){
          res=getch();
          printf("touche %c pressee, val ascii : %d\n",res,res);
      }
      if(clock()>top+30){  // contrôler le temps
         top=clock();
         putchar('0');
      }
   }
   return 0;
}
```

Maintenant dans la boucle while, si et seulement si le temps passé dépasse le dernier top enregistré de plus de 30 millisecondes, un zéro est affiché et le nouveau top est enregistré, sinon rien ne se passe. Il y a donc affichage d'un zéro uniquement toutes les 30 millisecondes.

```
if(clock()>top+30){     // 1
   top=clock();         // 2
   putchar('0');
}
```

(1) clock() c'est le temps qui passe, top c'est le temps de départ et +30 c'est pour le top prochain à 30 millisecondes.

(2) Quand le temps qui passe arrive à top +30, alors top prend la nouvelle valeur de départ et un 0 est affiché. Un 0 s'affiche toutes les 30 millisecondes.

3. Base création de jeux (mode console non graphique)

Avec une boucle d'événements non bloquante il est possible d'écrire des jeux. Voici un programme qui déplace une lettre dans une zone de jeu. Pour ce faire nous avons besoin d'une fonction supplémentaire :

```
void gotoxy(int x, int y);
```

Cette fonction est écrite au dessus du main(). Elle nécessite l'inclusion du fichier d'entêtes windows.h. Elle déplace le curseur en écriture à la position (x,y) que l'on veut dans la fenêtre console. Nous allons l'utiliser pour déplacer une lettre avec les touches flèche.

Les flèches ont un codage particulier mais obéissent aux valeurs : 72 pour haut, 77 pour droite, 80 pour bas et 75 pour gauche. En fait appuyer sur une flèche revient à appuyer sur deux touches : la touche α (valeur 224) et une des lettres correspondant aux valeurs ascii 72, 77, 80, 75.

```
#include <stdio.h>    // pour utilisation fonctions affichage
#include <stdlib.h>   // pour fonction srand() et rand()
#include <time.h>     // pour initialisation srand() avec time()
#include <conio.h>    // pour fonctions kbhit()et getch()
#include <windows.h>  // pour écriture fonctions gotoxy ci-dessous

void gotoxy(int x, int y)
{
  COORD c;

  c.X = x;
  c.Y = y;
  SetConsoleCursorPosition (GetStdHandle(STD_OUTPUT_HANDLE), c);
}

int main()
{
int fin=0, res, x, y;                              //1
const int TX=20, TY=15;

   srand(time(NULL));
   x=rand()%TX;
   y=rand()%TY;
   gotoxy(x,y);
   putchar('X');

   while (! fin){ // équivalent à fin==0           //2

       if( kbhit()){                               //3

          gotoxy(x,y);                             //4
          putchar(' ');

          res=getch();                             //5
          switch(res){
             case 72 :y--;  break; // haut
             case 77 :x++;  break; // droite
             case 80 :y++;  break; // bas
             case 75 :x--;  break; // gauche
             case 224 :     break; // évacuer la touche combinée
             default : fin=1; break; // autre touche :
                                  //met fin au prg
       }
       if (x<0)                                    //6
          x=TX;
       if (x>TX)
          x=0;
       if (y<0)
          y=TY;
       if (y>TY)
          y=0;

       gotoxy(x,y);                                //7
       putchar('X');
```

```
        }
    }

    return 0;
}
```

(1) Nous avons quatre variables. La variable fin, initialisée à 0, est pour contrôler la boucle. Lorsque fin ne sera plus égale à 0 la boucle sera finie. La variable res sert à récupérer les entrées clavier. Les variable x et y sont pour la position horizontale et verticale de la lettre à déplacer. TX et TY seront des valeurs constantes, c'est à dire non modifiables dans la suite du programme. Cette propriété est donnée par le mot clé const placé avant le type. TX est initialisée à 20 et TY est initialisée à 15. Ces deux valeurs correspondent à la taille de la zone de jeu.

Tout d'abord la fonction srand() initialise la suite de nombres pseudo aléatoires sur l'horloge interne. Ensuite les variables x et y sont initialisées. La position horizontale x est aléatoire entre 0 et TX et pour la verticale y entre 0 et TY. L'appel de la fonction gotoxy() positionne le curseur en écriture à cette position initiale (x,y) et putchar() affiche la lettre X à cette position.

(2) Tant que la variable fin est égale à 0, l'expression est vraie et les instructions du bloc sont répétées.

(3) La première instruction est un test sur la valeur de retour d'un appel à la fonction kbhit(). Au lieu d'écrire if (kbhit()==1), le test est contacté en if (kbhit()) ce qui revient au même parce que c'est la valeur de retour de l'appel à la fonction kbhit() qui donne la réponse du test. Si kbhit() renvoie 0 (aucune touche appuyée) le test est faux. Si kbhit () renvoie 1 le test est vrai et les instructions du bloc sont exécutées.

(4) L'objectif est de bouger une lettre avec les flèches. Pour ce faire il faut :

1 : effacer la lettre à sa position courante

2 : modifier sa position en fonction des touches appuyées

3 : contrôler que la nouvelle position reste dans la zone de jeu

4 : afficher la lettre à sa nouvelle position

Dans le bloc du if (kbhit()) , la première instruction consiste à positionner le curseur sur la position courante de la lettre à bouger. La seconde instruction consiste à effacer cette lettre en affichant à sa place un espace.

(5) Maintenant modifier la position de la lettre en fonction des touches appuyées.

La valeur de la touche enfoncée est renvoyée par la fonction getch(). Cette valeur est affectée à la variable res. Cette valeur détermine un cas dans le switch.

Le repère (0,0) de l'écran d'ordinateur est toujours le coin haut-gauche. Ainsi :

- □ remonter signifie diminuer la valeur de y qui se rapproche du 0.
- □ descendre augmenter la valeur de y.
- □ aller à gauche diminuer la valeur de x,
- □ aller à droite augmenter la valeur de x.

S'il s'agit de la flèche haut, la lettre remonte, y diminue de 1

S'il s'agit de la flèche droite, la lettre avance vers la droite, x augmente de 1

S'il s'agit de la flèche bas, la lettre descend, y augmente de 1

S'il s'agit de la flèche gauche, la lettre avance vers la gauche, x diminue de 1

Tous les autres cas de touches appuyées déclenchent la mise à un de la variable fin, c'est-à-dire la fin de la boucle while au prochain tour, juste après l'achèvement des instructions du bloc.

A cause du codage des flèches la valeur 224 est ajoutée à chaque fois que l'on appuie sur une flèche, exactement comme si l'on appuyait sur les deux touches. De ce fait la valeur 224 déclenche la commande default, fin du programme. Pour éviter cela, le cas 224 est ajouté et il ne fait rien.

(6) Si les positions verticale ou horizontale de la lettre ont été modifiées, il est possible que la lettre sorte de la zone de jeu. Cette zone va de 0 à TX compris pour l'horizontale et de 0 à TY compris pour la verticale.

Pour contrôler les bords il faut prendre une décision : soit la lettre est bloquée en arrivant sur les bords, soit elle passe sur le bord opposé. C'est cette dernière solution que nous avons choisie. Ainsi :

- Si x est inférieur à 0, x sort à gauche et rentre à droite, x prend la valeur TX
- Si x est supérieur à TX, x sort à droite et rentre à gauche avec la valeur 0.
- Si y est inférieur à 0, y sort en haut et rentre en bas, avec la valeur TY
- Si y est supérieur à TY, y rentre en haut avec la valeur 0.

(7) Maintenant que la position est prête le curseur est déplacé à cette nouvelle position avec un appel de gotoxy() et la fonction putchar() affiche la lettre à sa nouvelle position. Le bloc if (kbhit()) est terminé. La boucle continue si la variable fin est toujours égale à 0.

4. Mise en pratique : menus, boucles d'événements

Exercice 1
En reprenant le code du cours :
- ajouter un trèsor, quand l'utilisateur le trouve il marque un point, son score est affiché
- ajouter un ennemi mobile, si l'utilisateur est touché il a perdu.
- ajouter un balon pour le player. Lorsque le player s'en approche il peut shooter et conduire le ballon.

Exercice 2
Faire un petit convertisseur franc-euro et euro-franc. Le programme propose un menu avec les deux possibilités, à l'issue d'une conversion il propose à l'utilisateur de recommencer ou de quitter (1 euro vaut 6,55759 francs)

Exercice 3
Faire le programme qui affiche des zéros en continue et, si une touche est appuyée, un nombre aléatoire de fois la lettre ou le chiffre correspondant.

Exercice 4

Imaginer un automate distributeur de boissons.
Comment fonctionne votre automate : interface, choix, monnaie, approvisionnement ? Quelles sont les variables nécessaires à sa réalisation ? Écrire l'algorithme et programmez une simulation avec les entrées sorties scanf() et printf().

Exercice 5
Le player se déplace à l'écran avec les touches flèche. Dans la zone de mouvement il y a des trésors à découvrir. Le player marque des points en fonction de ce qu'il ramasse sur le terrain. Faire le programme.

Exercice 6
Deux lettres se déplacent dans une zone de jeu de façon plus ou moins aléatoire. Lorsqu'elles atteignent un bord elles rebondissent et partent dans l'autre sens. Faire le programme. Modifier ensuite le programme afin que lorsque les deux lettres se rencontrent une phrase s'affiche.

Exercice 7

Jeu. On dispose d'une dizaine de lettres. A chaque lettre est attribué un facteur d'instabilité. Par exemple la lettre 'M' ; très stable a un facteur d'instabilité de 0, par contre le 'V' a un facteur d'instabilité de 9.

En appuyant sur la touche espace, on empile verticalement des lettres, choisies aléatoirement par le programme. Le but est d'atteindre un certain seuil en hauteur mais si le cumul des facteurs d'instabilité est supérieur ou égal à 50 tout s'écroule. Programmer.

Exercice 8

Un petit bonhomme monte un escalier qui débouche sur rien. Arrivé en haut il avance et tombe dans le vide. Au sol il fait "aie !". Le compteur compte un point. Il revient à l'escalier, le remonte et retombe en faisant "aie" et en marquant un point etc. Programmer.

Exercice 8
Jeu de dés à deux joueurs. Chaque joueur à son tour jette le dé autant de fois qu'il veut. Il accumule les points qu'il obtient sauf s'il fait 1 et s'il fait 1, il ne marque aucun point, il est obligé de s'arrêter et c'est à l'autre joueur de jouer. Le gagnant est le premier qui dépasse 100.
Faire un programme pour jouer contre l'ordinateur. Le dé n'est pas pipé. L'ordinateur joue honnêtement quand c'est son tour et lance le dé quand c'est votre tour. Il affiche toujours ce qui sort et quand c'est à vous de jouer il vous demande à chaque lancé si vous voulez continuer. Lorsque la partie est finie il propose une autre partie ou de quitter.
La programmation de ce jeu suppose de concevoir une stratégie pour l'ordinateur. Quand et comment décide t-il de s'arrêter de lancer le dé ?

Exercice 9
Master_Mind répondeur. Il se joue avec des pions de couleurs. Le joueur doit deviner une combinaison cachée faite avec des pions. Par exemple avec 4 pions ROUGE, VERT, BLEU et JAUNE, voici une combinaison BLEU, JAUNE, JAUNE, ROUGE, l'ordre compte.
Pour jouer le joueur propose des combinaisons. Par exemple : ROUGE, VERT, JAUNE, BLEU.

En retour l'ordinateur indique combien de pions ont la bonne couleur ET sont à la bonne place. Ici 1, JAUNE à la troisième place.
Il indique également le nombre total des bonnes couleurs trouvées, ici 2 couleurs trouvées : rouge et jaune.
Si le joueur à gagné l'ordinateur lui dit en combien de coups et lui propose une autre partie ou d'arrêter.
A chaque partie l'ordinateur crée une nouvelle combinaison cachée et répond aux propositions du joueur. Faire le programme.

Exercice 10
Faire le jeu du poulet qui traverse une route :

```
                p

......v.....................v......>
......v.....v.....v.....v.....v.....>
<.....v.....v.....v.....v.....v.....
<.....v.....v.....v.....v.....v.....
```

P, le poulet doit arriver de l'autre côté de la route sans être touché par une voiture v. Il y a quatre voies avec au maximum deux voitures ensembles par voie.

Décomposez la réalisation du programme en plusieurs étapes. Par exemple : faire le poulet, faire une voie avec une voiture, ajouter un voiture, ajouter une voie et deux voitures, ajouter deux voies en sens inverse et quatre voitures.

G. Fonctions

1. Qu'est ce qu'une fonction ?

- La fonction est une unité de traitement en vue de l'accomplissement d'une action ou d'une étape dans l'action. La conception de programmes à actions multiples va s'appuyer sur un découpage judicieux en multiples fonctions.
- Elle permet de factoriser (généraliser) du code qui se répète. Lorsque dans un programme une séquence d'action se répète en général il y une fonction à écrire pour éclaircir et alléger le code.
- Techniquement la fonction est un bloc d'instruction doté d'un nom et de deux mécanismes :
  - un mécanisme d'entrée de valeur : ce sont les paramètres en entrée
  - un mécanisme de sortie de valeur : la valeur de retour (unique)
- Une fois écrite la fonction est "appelée". L'appel de la fonction permet d'insérer le bloc des instructions de la fonction dans le déroulement du programme à l'endroit où elle est appelée. A ce moment nous donnons pour son exécution des valeurs aux paramètres et s'il y a lieu nous pouvons aussi récupérer la valeur de retour.

Une fonction prend la forme suivante :

```
<type du retour > <nom fonction> < ( liste de paramètres ) >

{  ouverture bloc
```

```
   (...)    // les instructions du bloc

   return  (...)  //instruction de retour si sortie de valeur


}  fermeture bloc
```

La première ligne constitue ce que l'on nomme "l'entête de la fonction". Le bloc d'instruction constitue le corps de la fonction.

L'entête comprend le type de la valeur de retour, le nom de la fonction et une liste de paramètres entre parenthèses. Les paramètres sont simplement des variables locales à la fonction qui ont la particularité de pouvoir recevoir des valeurs au moment de l'appel de la fonction. S'il n'y a pas de paramètre la liste est vide. La valeur de retour n'est pas obligatoire non plus. S'il n'y a pas de valeur de retour le type du retour est void. Une fonction qui ne retourne pas de valeur est appelée aussi une procédure.

2. Écrire sa fonction

a. Où écrire sa fonction ?

La fonction s'écrit sur un fichier C, soit sur celui du main() soit sur un autre.

b. Conditions à remplir

Pour écrire une fonction il faut :

- Définir l'objectif de la fonction : à quoi sert la fonction ? Qu'est ce qu'elle fait ?
- Donner un nom à la fonction. Les noms de fonctions doivent respecter les mêmes contraintes que les noms de variables : pas de mot clé, pas d'opérateur, pas de nombre au début et si le nom est constitué de plusieurs mots pas d'espace entre les mots.
- Indiquer le type de la valeur de retour. A gauche du nom est OBLIGATOIREMENT spécifié le type de la valeur de retour. C'est nécessairement un type de variable simple (char, short, int, long, float, double, pointeur) ou une structure (voir module 3 les variables ensemble). S'il n'y a pas de valeur de retour c'est le type void (rien) qui doit être spécifié.
- Indiquer la liste des paramètres. Les paramètres sont simplement des variables locales à la fonction déclarées entre les parenthèses à droite du nom de la fonction. Ce qui les distingue des autres c'est qu'elles sont initialisées avec l'appel de la fonction.

c. Exemple fonction sans retour ni paramètre

Nous voulons écrire une fonction qui affiche un nombre aléatoire de fois "bonjour".

La fonction s'appelle bonjour1, elle ne retourne pas de valeur et elle n'a pas de paramètre :

```
void bonjour1()
```

```
{
int nb;
   nb=1+rand()%10;     // une valeur entre 1 et 10 compris
   printf("bonjour1() aff %d bonjour :\n",nb);
   while(nb--)
      printf("bonjour\n");
}
```

L'appel de cette fonction est détaillé dans la partie utiliser sa fonction.

d. Exemple fonction avec retour et sans paramètre

Nous voulons écrire une fonction qui affiche un nombre aléatoire de fois le mot "bonjour" et qui retourne combien de fois bonjour a été affiché. La fonction s'appelle bonjour2, elle retourne une valeur de type int et elle n'a pas de paramètre :

```
int bonjour2()
{
int nb,i;

   nb=1+rand()%10;
   printf("bonjour2() :\n");

   // modifier la boucle pour ne pas perdre la valeur nb
   for (i=0; i<nb; i++)
      printf("bonjour\n");
   // instruction pour le retour de la valeur nb
   return nb;
}
```

L'instruction return permet de renvoyer une valeur au contexte d'appel de la fonction. Cette instruction provoque en fait une sortie immédiate de la fonction et elle peut être utilisée dans certains cas pour sortir prématurément d'une fonction.

L'appel de cette fonction est détaillé dans la partie utiliser sa fonction.

e. Exemple fonction sans retour avec un paramètre

Nous voulons écrire une fonction qui affiche un nombre de fois "bonjour" donné cette fois en paramètre. La fonction s'appelle bonjour3, elle ne retourne pas de valeur et elle a un paramètre de type int :

```
void bonjour3(int nb)
{
   printf("bonjour3() aff %d bonjour :\n",nb);
   while (nb--)
      printf("bonjour\n");
}
```

L'appel de cette fonction est détaillé dans la partie utiliser sa fonction.

f. Exemple de fonction avec retour et paramètre

Nous voulons écrire une fonction qui affiche un nombre de fois bonjour et retourne ce nombre. En paramètre c'est un nombre minimum d'affichage qui est passé. Ce

nombre minimum est augmenté d'une valeur aléatoire dans la fonction et le nombre total des affichages est retourné au contexte d'appel.

```
int bonjour4(int min)
{
int max,i;
   printf("bonjour4() :\n");
   max=min+rand()%5;
   for (i=0; i<max; i++)
      printf("bonjour\n");
   return max;
}
```

L'appel de cette fonction est détaillé dans la partie utiliser sa fonction.

g. Conclusion : quatre cas d'écriture de fonction

Une fonction peut être avec ou sans retour, avec ou sans paramètre :

FONCTIONS	RETOUR	PARAMETRE(S)
cas 1	NON	NON
cas 2	OUI	NON
cas 3	NON	OUI
cas 4	OUI	OUI

3. Utiliser sa fonction

a. Appel de la fonction

Une fois la fonction écrite il n'y a plus qu'à l'appeler là où dans le programme nous voulons qu'elle s'exécute. Voici par exemple un programme qui appelle un nombre nb fois entré par l'utilisateur la fonction bonjour1() :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int nb,i;

   srand(time(NULL));
   printf("entrer le nombre de repetition : \n");
   scanf("%d",&nb);

   for(i=0; i<nb; i++){
      printf("appel %d :\n",i);
      bonjour1();               // appel de bonjour1()
   }

   return 0;
}
```

b. Récupération de la valeur de retour

Pour récupérer la valeur de retour il suffit de l'affecter à une variable du contexte d'appel de la façon suivante :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int val;

   srand(time(NULL));

   val = bonjour2();          // affectation du retour à val

   printf ("retour de bonjour2() : %d\n",val);
   return 0;
}
```

La valeur numérique de l'appel de la fonction c'est la valeur de retour et pour juste afficher la valeur de retour sans la stocker nous pouvons écrire aussi :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   srand(time(NULL));
   printf ("retour de bonjour2() : %d\n",bonjour2());
   return 0;
}
```

Remarque :

La récupération de la valeur de retour n'est pas obligatoire. Mais si la valeur retournée n'est pas récupérée elle est perdue.

c. Passage de valeurs aux paramètres

Pour passer des valeurs aux paramètres il suffit de spécifier pour chaque paramètre la valeur que nous souhaitons lui attribuer à l'appel de la fonction, par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int nb;

   srand(time(NULL));

   printf("test 1 ----------------------\n");
   bonjour3( 5);

   printf("test 2 ----------------------\n");
   bonjour3(rand()%2);

   printf("test 3 ----------------------\n");
   printf("entrer le nombre de repetitions :\n");
   scanf("%d",&nb);
```

```
   bonjour3(nb);

   printf("test 4 ----------------------\n");
   nb=bonjour4(nb);
   printf("retour bonjour4 : %d\n",nb);
   return 0;
}
```

Ici nous passons 5 pour le premier appel de bonjour3(), soit 0 soit 1 pour le second et une valeur entrée par l'utilisateur pour le troisième.

L'appel de bonjour4() prend la valeur précédente nb entrée par l'utilisateur. Elle la retourne modifiée à la fin de son exécution et nous la récupérons dans la variable nb du contexte d'appel.

d. Précision sur le passage par valeur

Soit la fonction :

```
void modif (int a, int b)
{
   a=1000;
   b=2000;
}
```

et son appel dans le programme :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=0, b=0;

   modif(a,b);
   printf("a=%d, b=%d\n",a,b);

   return 0;
}
```

A votre avis qu'imprime le programme ?

Au moment de son appel les valeurs 0 et 0 sont affectées aux paramètres a et b de la fonction modif(). Ce sont deux variables locales à la fonction. Elles n'existent que dans le bloc de la fonction. La fonction change leurs valeurs qui passent à 1000 et 2000.

Ensuite nous revenons dans le contexte d'appel que valent maintenant a et b dans le contexte d'appel ? Ces deux variables a et b sont des variables locales au bloc du main(). Elles n'ont pas été touchées. Ce ne sont pas les mêmes variables que celles de la fonction. Printf() imprime donc :

a=0, b=0

Comme nous pouvons le voir sur cet exemple et les précédents les variables déclarées dans un bloc sont locales dans le bloc où elles sont déclarées et disparaissent à la fin de l'exécution du bloc. Nous pouvons faire circuler des valeurs dans le programme grâce aux entrées et sorties de fonction.

e. Visibilité et déclaration de la fonction

Une fonction est globale c'est à dire visible de partout dans un programme quelque soit sa position dans un des fichiers source qui composent le programme à condition de pouvoir être connue avant son premier appel au moment de la compilation.

En effet il y a une erreur de compilation si par exemple une fonction définie après le main() est appelée dans le main(). Pour éviter ce problème deux solutions :

- définir sa fonction avant le main(), au dessus. C'est très bien pour des petits programmes comprenant peu de fonctions et avec un seul fichier source.
- déclarer sa fonction. C'est mieux méthodologiquement même pour un petit programme et c'est indispensable pour des programmes plus importants.

Comment déclarer sa fonction ?

C'est très simple il suffit de prendre l'entête de la fonction, de le copier à l'endroit de la déclaration par exemple au dessus du main() et d'ajouter un point virgule. S'il n'y a pas de paramètre il faut spécifier void entre les parenthèses.

Voici par exemple les déclarations de nos quatre fonctions bonjour() :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// déclarations des fonctions :
// <type retour> <nom> <(paramètres)> <;>
void bonjour1 (void);
int  bonjour2 (void);
void bonjour3 (int nb);
int  bonjour4 (int min);

// le programme
int main()
{
   bonjour1();
   bonjour2();
   bonjour3(5);
   bonjour4(5);
   return 0;
}
// et les fonctions sont définies après le main()
   (..)
```

Pour des programmes importants nous écrirons nos propres librairies associées au programme et les déclarations de fonctions seront en général regroupées dans nos propres librairies (voir chapitre structuration d'un programme).

4. Cas des fonctions avec liste variable de paramètres

Il est également possible en C d'écrire des fonctions qui ont un nombre variable de paramètres, c'est la cas des fonctions printf() et scanf() par exemple. Mais c'est assez rare car peu pratique dans ce langage.

Une telle fonction doit comporter au moins un paramètre fixe qui précède la liste des paramètres variables. Cette liste prend la forme de trois points … dans l'entête, à la place des paramètres. Les paramètres peuvent être de même type ou de types différents. Dans les deux cas il s'agit de récupérer une liste des paramètres et connaître sa fin. Dans le second, en plus, il faut trouver un moyen d'obtenir le type de chaque paramètre.

La liste des paramètre est gérée avec quatre instructions définies dans <stdarg.h>, notons DPF le dernier paramètre fixe, celui qui précède la liste des paramètres variables et donne le point de départ, les quatre instructions sont :

va_list	est le type pour la liste de récupération
va_start (va_list liste, DPF)	est la fonction d'initialisation de la liste
va_arg (va_list liste, type param)	est la fonction pour la récupération de chaque élément de la liste
va_end(va_list liste)	est la fonction qui libère la mémoire de la liste constituée de façon dynamique (l'allocation dynamique est abordée au chapitre pointeurs.

a. Liste variable de paramètres de même type

Pour tester une liste de paramètres de même type voici une fonction qui fait la somme des paramètres entiers qui lui sont donnés :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

int somme(int nb,...)// 1 entête de la fonction avec liste
{
va_list lparam;             // 2 déclarer une liste
int res=0,n;
    va_start(lparam,nb);    // 3 initialiser la liste

    for(n=0; n<nb; n++)     // 4
        res+= va_arg(lparam,int);// 5 récupération de la valeur
                                 // de chaque paramètre

    va_end(lparam);     // 6 libérer la mémoire de la liste
                        // constituée
    return res;

}

int main(int argc, char*argv[])
{
    printf("%d\n",somme(1,1));           // affiche 1
    printf("%d\n",somme(3,1,2,3));       // affiche 6
    printf("%d\n",somme(5,1,2,3,4,5));  // affiche 15
   return 0;
}
```

(1) l'entête de la fonction, après le dernier paramètre fixe, les trois points comme paramètre désigne une liste variable de paramètres.

(2) Première étape, déclarer une variable va_liste

(3) Ensuite appeler la fonction va_start qui initialise la variable va_liste

(4) Le dernier paramètre fixe (DPF) est utilisé pour donner le nombre des paramètres au moment de l'appel. C'est lui qui donne de ce fait la fin de la liste.

(5) l'appel de la fonction va_arg() retourne la valeur de chaque paramètre, en l'occurrence nous les additionnons au fur et à mesure.

(6) à l'issue la mémoire affectée à la liste est libérée.

b. Liste variable de paramètres de types différents

Si les types des paramètres sont différents il faut implémenter un mécanisme pour la reconnaissance des types de paramètres. Par exemple chaque type de paramètre peut être identifié avec un code, une valeur entière, par exemple 0 pour int, 1 pour double, 2 pour chaine de caractère etc. Ensuite au moment de l'appel chaque paramètre est précédé du code qui correspond à son type. C'est ce que fait la fonction test() suivante :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

enum{t_int,t_double, t_string,t_end};  // 1 une énumération

void test(int type,...)                // 2
{
va_list lparam;                        // 3
int fin=0;
int i;
double f;
char*s;

    va_start(lparam,type);             // 4

    while(!fin){
       switch(type){                   // 5
        case t_int :
           i=va_arg(lparam,int);
           printf("int %d\n",i);
           break;

        case t_double :
           f=va_arg(lparam,double);
           printf("float %f\n",f);
           break;

        case t_string :
           s=va_arg(lparam,char*);
           printf("int %s\n",s);
           break;

        case t_end :
           fin=1;
```

```
            break;
        }
        type=va_arg(lparam,int);        // 6
    }
    va_end(lparam);                     //7
}

int main()
{
    test( t_int, 10,                    // 8
          t_int,20,
          t_string, "bonjour",
          t_double, 3.5,
          t_end);

    return 0;
}
```

(1) Tous les codes sont regroupés sous la forme d'une énumération. Ce point est abordé dans le chapitre 3 qui suit mais il n'y a rien de compliqué, l'instruction est suivie d'un bloc dans lequel figure une liste de nom et chaque correspond à une valeur. Par defaut le premier vaut 0 et tous les autres sont incrémentés de un en un, par exemple :

```
enum{ tata, toto, titi, tutu} ;
signifie que :
tata vaut 0
toto vaut 1
titi vaut 2
tutu vaut 3
```

Ainsi dans notre exemple :

```
t_int    vaut 0
t_double vaut 1
t_string vaut 2
t_end    vaut 3
```

(2) L'entête de la fonction test() qui déclare une liste variable de paramètres

(3) Déclaration de la liste lparam

(4) Initialisation de la liste qui commence au paramètre int type

(5) Aiguillage, selon la valeur de type telle ou telle cas est traité, le paramètre correspondant est retiré de la liste.

(6) récupération du type pour le cas suivant. Lorsque t_end est trouvé la boucle prend fin.

(7) à l'issue, lorsque tous les cas sont traités destruction de la liste.

(8) appel de la fonction test () avec 9 paramètres 4 codes type et 4 valeurs, une pour chaque type plus le code spécifique pour la fin de la liste t_end. L'appel produit l'affichage de :

```
10
20
bonjour
3,5
```

c. Transformer printf()

Nous allons transformer la fonction printf() de façon à pouvoir écrire du texte n'importe ou dans la fenêtre console et dans une couleur donnée. Position et couleur sont obtenues avec les fonctions gotoxy() et textcolor()

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <windows.h>   // pour écriture des fonctions gotoxy()
                       // et textcolor()

// déclaration des fonctions
void G_printf(int x, int y, int color, char*format,...);
void gotoxy(int x, int y);
void textcolor(int color);

int main()
{
    G_printf(5,5,12,"num : %d",2341 );     // 0
    return 0;
}
void G_printf(int x, int y, int color, char*format,...)       //1
{
va_list lparam;
char stock[10000];           // 2

    va_start(lparam,format);
    vsprintf(stock,format,lparam);  // 3
    va_end(lparam);

    // afficher à la position et de la bonne couleur    // 4
    gotoxy(x,y);
    textcolor(color);
    printf("%s",stock);
}
void gotoxy(int x, int y)
{
HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
COORD c;
    c.X=x;
    c.Y=y;
    SetConsoleCursorPosition(h,c);
}
void textcolor(int color)
{
HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(h,color);
}.
```

(0) l'appel de la fonction G_printf() affiche "num : 2341" à la position 5,5 de la fenêtre console et de la couleur 12 (rouge)

(1) Entête de la fonction qui a trois paramètres fixes et une liste variable de paramètres

(2) La chaine formatée va être récupérée avec les valeurs associées aux formats intégrées, dans le tableau de caractères stock (les tableaux sont abordés au chapitre suivant).

(3) La fonction standart vsprintf() permet de récupérer dans un tableau de caractères une chaine formatée comme celle de printf () sous la forme d'une va_list.

Après cet appel la chaine est stockée dans le tableau stock (les formats sont remplacés par les valeurs effectives sous forme de caractères) .

(4) Une fois la chaine constituée il n'y a plus qu'à l'afficher à la position passée en x et y et selon la couleur color.

5. Mise en pratique : Fonctions

a. Identifier les composants d'une fonction

Exercice 1
Soit la fonction suivante :

```
void wait(int tmps)
{
int start=clock();
   while (clock()<start+tmps){}
}
```

Donnez le nombre et le type des paramètres, le nom de la fonction, le type de la valeur de retour, les variables utilisées. Que fait la fonction ?

Exercice 2
Donner l'entête d'une fonction destinée à calculer la distance entre deux points (la distance serait calculée avec le théorème de pythagore). Donner l'ensemble des ressources nécessaires pour écrire la fonction.

b. Déclaration de fonctions

Exercice 3
Quels sont les messages d'erreur produits par le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   affiche_bonjour(5);
   return 0;
}

void  affiche_bonjour ( int nb)
{
int i;

   for (i=0; i<nb; i++)
      puts("bonjour");
}
```

Modifier ce programme pour qu'il fonctionne. Donnez deux solutions.

c. Procédures sans paramètre

Exercice 4
Afin de tester la fonction rand() du point de vue statistique, faire une fonction qui tire 100000 fois une valeur comprise entre 0 et 5, compte les occurrences de chaque

résultat et affiche le pourcentage de chaque résultat. Appeler la fonction dans un programme.

Exercice 5
Faire une fonction damier qui affiche un damier de 20 sur 15 à l'écran. Le damier est encadré en vert. Tester la fonction dans un programme.

Exercice 6
Faire une fonction qui affiche l'alphabet en boucle dans un rectangle de 20 lignes sur 40 colonnes. Chaque alphabet aura ses propres couleurs. Tester dans un programme.

Exercice 7
Faire une fonction qui génère automatiquement et affiche une phrase simple différente à chaque appel. Tester dans un programme qui affiche une phrase à chaque fois que l'on appuie sur une touche clavier. Le programme quitte si l'on appuie sur la lettre q

Exercice 8
Dans un programme un menu propose les actions suivantes :
- bombardement d'un nombre aléatoire de lettres multicolores dans la fenêtre console
- bombardement d'un nombre aléatoire de lignes verticales, horizontales ou diagonales. Les tailles sont toujours aléatoires. Il est conseillé de faire une fonction par type de ligne.
- dessin d'un rectangle de taille aléatoire dans la fenêtre console
- bombardement aléatoire de lettres, de lignes ou de rectangles.

Faire le programme. Le programme quitte à la demande de l'utilisateur.

d. Fonctions sans paramètre

Exercice 9
Ecrire une fonction run() qui permet de faire bouger une lettre à l'écran contrôlée par les flèches du clavier. La fonction renvoie une valeur dont l'interprétation permet de mettre fin au programme.

e. Fonctions avec paramètres

Exercice 10
Faire une fonction qui affiche le caractère (table ASCII) correspondant à une valeur numérique décimale entre 0 et 255. A partir de cette fonction faire une deuxième fonction qui affiche toute la table ASCII.

Exercice 11
Faire une fonction qui donne la moyenne de trois nombres quels qu'ils soient (flottants ou non). Tester dans un programme. Le programme quitte uniquement lorsque l'utilisateur le demande.

Exercice 12
Ecrire une première fonction qui indique si un nombre entier est multiple de 2 ou non.
Ecrire une seconde fonction qui indique si un nombre entier est multiple de 3 ou non.

Utiliser ces deux fonctions dans un programme qui lit un nombre entier et précise s'il est pair, multiple de 3 et/ou multiple de 6. Le programme quitte uniquement lorsque l'utilisateur le demande.

```
Exemple d'exécution :          Autre exemple :

   entrer un nombre : 9        entrer un nombre : 12
   il est multiple de 3        il est pair, multiple de 3 et
                               divisible par 6
   Recommencer ?  (o/n)
                               Recommencer ?  (o/n)
```

Exercice 13

Soit un barème de l'impôt défini comme suit : pour un ménage X avec un revenu total R et un nombre n de membres du foyer, l'impôt est donné par :

10% de R si r/n <500 euros
20% de R si R/n >=500 euros

- □ Écrire une fonction qui calcule le montant de l'impôt en fonction de R et de n
- □ Écrire une fonction qui donne le revenu net d'un ménage après paiement de l'impôt en fonction de R et de n. Tester dans un programme, R et n sont entrés par l'utilisateur ensuite le montant de l'impôt et le revenu net du ménage sont affichés.

Le programme quitte uniquement lorsque l'utilisateur le demande.

Exercice 14

Ecrire une fonction qui reçoit en argument deux nombres flottants et un caractère qui correspond à une opération (+, -, /, *, %). La fonction retourne le résultat de l'opération spécifiée par le caractère. Le programme quitte uniquement lorsque l'utilisateur le demande.

Exercice 15

Soit une zone maximum de 24 lignes par 79 colonnes, écrire tout d'abord une fonction qui affiche un caractère à une position donnée et d'une couleur donnée. Ensuite :

- □ Ecrire une fonction qui bombarde nb fois l'écran de lettres de façon aléatoire. Le nombre des répétitions est entré par l'utilisateur.
- □ Ecrire une fonction qui remplit la zone en cible rectangulaire avec une lettre différente pour chaque tour intérieur.
- □ Ecrire une fonction qui peut afficher un rectangle de lettres d'une couleur donnée et d'une taille donnée dans la zone.
- □ Ecrire une fonction qui bombarde nb fois de rectangles colorés la zone (attention à ne pas écrire en dehors de la zone).
- □ Ecrire une fonction qui affiche une spirale rectangulaire d'une taille maximum donnée en paramètre.

Dans un programme de test proposer un menu à l'utilisateur. Le programme quitte lorsque l'utilisateur en donne la commande.

Exercice 16

Soit le programme ci-dessous,

- □ Dites ce qu'il fait, donnez les commentaires
- □ Remplacer les séquences de code qui se répètent par des fonctions
- □ Ecrire une fonction qui permet de tracer un rectangle à la position, de la taille et de la couleur et de la lettre que l'on veut et l'utiliser pour le cadre
- □ A quel endroit appeler cette fonction et avec quelles valeurs dans le code ci-dessous ?
- □ Ecrire une fonction pour ralentir le processus.
- □ Tester le programme

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <conio.h>
#include <time.h>

/************************************************
************************************************/
void    gotoxy          (int x, int y);
void    textcolor       (int color);

/************************************************
************************************************/
int main()
{
const int TX = 40;
const int TY = 20;
int x,y,res,i;

int fin=0;

    srand(time(NULL));

    // commentaire 1
    for(i=0; i<=TX; i++){
        gotoxy(i,0);
        textcolor(12*16);
        putchar(' ');
    }
    // com 2
    for(i=0; i<=TX; i++) {
        gotoxy(i,TY);
        textcolor(12*16);
        putchar(' ');
    }
    // com 3
    for(i=0; i<=TY; i++){
        gotoxy(0,i);
        textcolor(12*16);
        putchar(' ');
    }
    // com 4
    for(i=0; i<=TY; i++){
        gotoxy(TX,i);
        textcolor(12*16);
        putchar(' ');
    }

    // com 5
    x=rand()%TX;
    y=rand()%TY;
```

```
    // com 6
    gotoxy(x,y);
    textcolor(10);
    putchar('P');

    // com 7
    while (!fin){

        // com 8
        if (kbhit()){

            // com 9
            gotoxy(x,y);
            textcolor(0);
            putchar('P');

            // com 10
            res=getch();
            switch(res){
                case 72 : y--; break;  // 11
                case 77 : x++; break;  // 12
                case 80 : y++; break;  // 13
                case 75 : x--; break;  // 14
                case 224 : break;
                default : fin = 1;      // 15
            }

            // com 16
            if (x < 1)
              x = TX-1;
            if (x>=TX)
              x = 1;
            if (y < 1)
              y = TY-1;
            if (y>=TY)
              y = 1;

            // com 17
            gotoxy(x,y);
            textcolor(10);
            putchar('P');
        }
    }
    gotoxy(0,TY+2);
    return 0;
}
/*************************************************
*************************************************/
void gotoxy(int x, int y)
{
COORD c;
    c.X=x;
    c.Y=y;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE),c);
}
/*************************************************
*************************************************/
void textcolor(int color)
{
```

```
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),color)
;
}
/************************************************
************************************************/
```

H. Style, commentaires et indentation

1. Pourquoi le style ?

Le but du style de la programmation est de s'assurer que le code sera facile à lire et partageable au sein d'une équipe. Un bon style est indispensable à toute bonne programmation.

Un programme sans rigueur est peu lisible, peu fiable, difficile à corriger et à comprendre. Il devient vite impossible à développer et se trouve finalement figé dans son incompréhensibilité. Plus un programme est rendu incompréhensible par l'absence de rigueur de sa présentation moins il est possible de continuer son développement, d'apporter des modifications ou des nouvelles fonctionnalités, voir tout simplement de le déboguer. Le style est finalement en relation avec l'efficacité de la conception. Aujourd'hui le style s'est normalisé et il est associé à des conseils et des recommandations qui varient très peu d'une communauté de développement à une autre. Le style n'est pas secondaire, c'est une question de discipline, de rigueur et d'efficacité dans le travail. Montrer votre code et rien qu'à son style tout employeur saura si vous êtes professionnel ou non.

Le créateur du langage C, Brian Kernighan nous dit ceci à propos du style : "le code sera simple et limpide _ logique claire et évidente, expressions naturelles, utilisation d'un langage conventionnel, noms compréhensibles et significatifs, formatage impeccable, commentaires utiles _ et évitera les spécificités sois disant astucieuses ainsi que les constructions inhabituelles. Il est important de rester cohérent car les tierces personnes trouveront ainsi votre code plus lisible _ et vous aurez le même avis pour le leur _ si vous faites tous sur le même modèle." (KERNIGHAN, La programmation en pratique, p.2).

2. Typographie et choix des noms

L'idéal est un nom concis, facile à retenir, si possible prononçable et qui donne la bonne information concernant une variable, une fonction ou tout objet que vous créez dans le programme. De façon cohérente donnez aux objets apparentés des noms similaires avec éventuellement précision sur les relations qui existent entre eux ou leurs différences. Éviter des noms qui peuvent prêter à confusion et être ambigus dans le programme.

Les noms de variables globales sont en majuscules, les variables locales en minuscules. Éventuellement, les noms peuvent être constitués de plusieurs mots chacun commençant par une majuscule

```
int posHorizontalPlayer;
int posVerticalPlayer;
```

C'est recommandé dans la communauté java. mais c'est bien de privilégier clarté et concision :

```
int playerx;
int playery;
```

Les notations du "m_genre" sont considérée comme "_vraiment_mauvaises" et à éviter. L'idéal est une économie de signe sans perte de clarté.

3. Indentation rigoureuse et accolades

Il est essentiel de bien indenter les programmes pour montrer la structure hiérarchique des instructions. Cette répartition graphique du code sur la page est nécessaire à sa lisibilité, sa clarté et sa compréhension notamment lorsqu'il s'agit de trouver une erreur.

Une indentation correspond à un retrait de 3 ou 4 caractères maximum. L'indentation suit la hiérachie des blocs, chaque imbrication suppose une indentation.

L'accolade ouvrante est immédiatement à la droite du test. La fermante est toujours au même niveau que l'instruction d'avant le test (if, while etc.). Il n'y a rien après les les accolades sur la même ligne.

La parenthèse du test est écartée d'un espace de l'instruction if, while etc. :

```
while (!fin){

    if (kbhit()){
        switch(getch()){
            case 1 :
                break;

        }
    }
}
```

Il est également très répandu de mettre l'accolade ouvrante juste en dessous de l'instruction avant le test de la façon suivante (c'est le modèle par défaut sous visual studio) :

```
while (! fin)
{

   if (kbhit())
   {
       switch(getch())
       {
           case 1 :
               break;
       }
   }
}
```

Pour éviter d'oublier une accolade quelque part dans le programme c'est simple : dés que vous ouvrez un bloc, tout de suite vous le refermez correctement en dessous et ensuite seulement vous écrivez votre code dedans, correctement indenté.

Pour les instructions de condition ça donne :

```
if (condition) {
   /*instructions*/
}
else {
   /*instructions*/
}
```

La seule fois où il y a du texte après l'accolade fermante c'est avec un do-while :

```
do {
   /* instructions*/
} while (condition);
```

Le switch a cette allure :

```
switch (expression) {
   case n :
      /*instructions*/
      break;
   default :
      /*instructions*/
      break;
}
```

Pour les espacements ils n'excèdent pas un espace. En général, comme dans un texte normal il y a un espace après une virgule ou un point virgule. L'espace entre l'accolade ouvrante et le test n'est pas obligatoire. Voici quelques exemples d'espacements :

```
int test;
if (condition) {

}
for (i=0; i<10; i++) {

}
res = rand()%20;
```

D'une façon générale, conservez le style graphique que vous choisissez pour les accolades et les espaces tout au long du programme voire de tous vos programmes.

4. Parenthèses pour dissiper les ambiguïtés

Une expression comme :

```
if (x >0 && x <100 || y >0 && y<100)
```

peut prêter à confusion. Même pour un programmeur expérimenté, il est arrive de se tromper parfois avec la priorité des opérateurs et les règles d'associativité. Pour éviter tout ambiguïté le mieux est de spécifier les regroupements voulus :

```
if ( ((x >0) && (x <100)) || ((y >0) && (y<100)) )
```

5. Commentaires pertinents

L'objectif d'un commentaire dans le code est de rendre le code plus facile à comprendre, par exemple en expliquant brièvement un détail ou en donnant une information sur l'algorithme et l'étape dont il est question.

Évitez de répéter avec un commentaire ce qui est évident déjà avec le code, éviter les commentaires inutiles pour des choses évidentes du genre :

```
// défaut
defaut :
       break;

/* retourne zéro */
return 0;

compte++: // incrémente compte

// initialise x avec une valeur aléatoire
x=rand();
```

Ces commentaires sont inutiles, il vaut mieux les supprimer parce qu'ils n'apportent pas d'information sur le code. Ils prennent de la place et parasitent plutôt le code.

Il ne faut pas perdre de vue que le commentaire doit aider à la compréhension du code. Toutefois, si le code est mauvais, il vaut mieux le réécrire plutôt que de le commenter.

Attention aux commentaires qui deviennent contradictoires avec le code : au départ vous écrivez du code et vous faites un commentaire et plus tard vous changez le code et oubliez de changer le commentaire. résultat pour celui qui veut comprendre votre code le commentaire rend les choses plus confuses.

La règle c'est de toujours, dans la mesure du possible, essayer de clarifier les choses et surtout pas de les rendre plus confuses.

6. Mise en pratique : Style, indentation, commentaires

Exercice 1
Que fait le programme suivant ?

```
int main(int argc, char *argv[]){int choix,res;printf ( " 1 :
Affiche bonjour \n"" 2 : Affiche il fait beau\n"" 3 : Entrer un
nombre entier\n"" 0 : Quitter
\n");do{scanf("%d",&choix);rewind(stdin);switch(choix){case
0:break;case 1:printf(" bonjour\n ");break;case 2:printf("il fait
beau\n ") ;break;case 3 :printf("Entrer un nombre
:\n");scanf("%d",&res); printf ("le nombre est :
%d",res);break ;default:printf("Pas de commande
%d\n",choix);break;}}while(choix !=0);return 0;}
```

Il n'y a pas d'erreur de syntaxe. A part quelques allers à la ligne le programme fonctionne. Refaites entièrement la mise en page de ce code., ensuite expliquez ce qu'il fait sur une feuille à part et pour finir testez votre code

Exercice 2
Que fait le programme suivant ?

```
int main(int argc, char *argv[]){int res ;printf("presser des
touches : (q pour quitter) \n");while ( res!='q'){if (kbhit())
```

```
{res=getch() ;printf(" touche %c pressee, val ascii :
%d\n",res,res) ;}printf("la boucle tourne\n");}return 0;}
```

Refaites la mise en page de ce code. Expliquez ce qu'il fait. Testez.

Exercice 3

Que fait le programme suivant ?

```
#include <stdio.h>#include <stdlib.h>#include <time.h>#include
<conio.c>
int main(int argc, char *argv[]){const int arborigene=20;const int
chateau=10;int
pourquoi=0,martial,pomme,level;srand(time(NULL));pomme=1+rand()
%arborigene;level=1+rand()%chateau; gotoxy(pomme,level);
putchar('o');while(!pourquoi){if(kbhit())
{gotoxy(pomme,level);putchar(' ');martial=getch();switch(martial)
{case 72:level--;break;case 77: pomme++;   break;case 80:level+
+;break;case75:pomme--;break;default:pourquoi=1; break ;      }if
(pomme<1)pomme=arborigene-1;if (pomme>arborigene-1)pomme=1;
if(level<1)level=chateau-1;if(level>chateau-
1)level=1;gotoxy(pomme,level) ;putchar('o');}}return 0;}
```

Refaites la mise en page de ce code. Expliquez ce qu'il fait, renommez les variables en conséquence. Testez et trouvez l'erreur.