Langage C : Codeblocks
Intallation et initiation

Lorsque l'on utilise l'ordinateur pour résoudre des problèmes qui nous sont posés, pas besoin d'être virtuose en programmation. Un bagage minimal de connaissances suffit, et comme je vais utiliser le langage C, disons qu'il s'agit plutôt de faire du *C - -* que du *C + +*. Pourquoi le langage *C* ? Parce que c'est le plus œcuménique, et le mieux adapté pour intégrer ensuite des logiciels graphiques. Mais si vous préférez aller plus loin avec d'autres langages que le *C*, poursuivez votre chemin et affirmez votre originalité.

Si vous connaissez déjà des bribes de programmation, ce qui suit vous donnera des éléments sur la syntaxe du langage *C*, étant entendu que la structure interne d'un programme est la même en *C* qu'en Pascal, Basic, Fortran,… Par exemple, ce qui s'écrit *{* en *C* s'écrit *begin* en Pascal. Si vous ne connaissez quasiment rien à la programmation, le rapide survol du langage *C* qui suit vous donnera un premier aperçu des moyens de traitement de problèmes via l'ordinateur. Au fond, tout se résume en quatre termes : boucles, tests, tableaux, fonctions. Il convient de faire avec !

Et maintenant quel langage *C* choisir ? Celui que vous voulez. Dans mon cas j'ai pris *Code Blocks*. La première chose à faire est de télécharger le langage. Je vais vous dire comment installer *Code Blocks* sous *Windows*, et même *Windows 10*,[1] mais si vous préférez le faire sous *Linux*, cela ne sera pas plus compliqué.

1. Télécharger Code Blocks sous Windows en 5 minutes chrono

Allez chez *Google*, et demandez « *télécharger code blocks mingw* ». Prendre la première proposition qui vient, du style *Download binary-Code ::Blocks*. Vous êtes envoyé sur un site où dans la rubrique *Windows* on vous propose *Codeblocks-16.01mingw-setup.exe* à télécharger avec *Sourceforge* ou un autre. Téléchargez et demandez l'exécution du *setup.exe*. Quelques secondes plus tard, le répertoire *CodeBlocks* se trouve installé à l'intérieur de vos *Programmes* (*Program Files(x86)*) mais vous pouvez demander de l'installer ailleurs si vous préférez. A l'intérieur de ce répertoire *C:\Program Files (x86)\CodeBlocks*, vous allez trouver l'exécutable qui s'appelle aussi *codeblocks*, parfaitement repérable grâce à son carré divisé en quatre petits carrés coloriés (rouge, vert, jaune, bleu). Cliquez dessus, à moins que vous ne fassiez un raccourci à partir de votre bureau. Et la page *CodeBlocks* apparaît plein écran. Dans le jargon informatique, cette page constitue l'*IDE* (environnement de développement intégré), le centre de pilotage à partir duquel vous allez écrire votre programme dans l'éditeur de texte, puis lancer la compilation et enfin l'exécution.

2. Premier programme : bienvenue chez les débiles

Dans la page *Code::Blocks* on vous propose, au centre de l'écran : *Create a new project* ; cliquez dessus. Ou bien vous êtes déjà dans une page avec une ligne en haut de l'écran qui commence à gauche par *File*, et vous cliquez sur *File* ; en haut de la colonne apparraît *New* et vous cliquez sur *New* ; dans la fenêtre qui s'ouvre choisissez *Project* et cliquez dessus.

Dans la fenêtre qui apparaît, cliquez sur l'icône *Console application*. Faites *next* pour aller d'une page à la suivante. On va vous demander si vous voulez *C* ou *C++*. Choisissez *C*. Puis on vous

[1] Ce document a été actualisé en 2016, à l'époque de Windows 10. Mais cela vaut aussi bien pour les Windows d'avant.

demande de donner un nom à votre projet. Tapez par exemple dans *Project title* : bonjour. Dans la ligne suivante *Folder to create project in*, vous pouvez indiquer où vous voulez que votre programme soit installé. Vous aurez peut-être avoir intérêt à créer au préalable un dossier *Programmes*. C'est là que vos programmes seront placés, et dans ce cas indiquez le chemin pour y accéder. Puis *next*, et on vous annonce qu'une *configuration release* est créée. C'est fini.

Vous voyez apparaître, dans la colonne *Projets* à gauche de l'écran, le carré *bonjour* en dessous de *Workspace*. Ainsi dans votre espace de travail (*workspace*) se trouve intégré votre projet *bonjour*, encore vierge. Cliquez dessus, et vous voyez apparaître *Sources*. Cliquez sur *Sources*, et vous voyez apparaître *main.c*. Cliquez sur *main.c* et là est déjà édité un petit programme déjà tout prêt. Demandez *build* (un petit rond jaune sur la deuxième ligne en haut de l'écran) puis *run* (un triangle vert). Une fenêtre à fond noir apparaît sur l'écran, avec marqué dessus *hello*, et au-dessous un message en anglais sans aucun intérêt. Cela prouve que votre *codeblocks* marche ! [2]

Maintenant aménagez ce programme. Commencez par supprimer *#include <stdlib.h>* qui ne sert à rien ici. Puis remplacez « Hello World !\n », par « Bonjour\n » ou ce que vous voulez. Rajouter un *getchar()* qui provoque une pause, et seul *Bonjour* reste écrit sur l'écran. Quand cela vous suffit, c'est-à-dire le plus vite possible, appuyez sur une touche, la fenêtre d'exécution se ferme, le message en anglais n'aura plus le temps de se manifester, et vous vous retrouvez dans votre *IDE Code Blocks*. Vous avez fait votre premier programme en C :

```
#include <stdio.h>
int main()
 { printf(« Bonjour\n ») ;
   getchar() ; return 0 ;
 }
```

Pour sauvegarder ce programme, allez dans la première colonne *File* en haut à gauche, et demandez par exemple *Save everything*. Ensuite vous pouvez tout fermer, et quitter *codeblocks* (allez sur votre projet *bonjour*, puis un clic à droite sur la souris donne un menu où l'on trouve *close project*). Votre projet va être pieusement conservé dans le répertoire *c*:*Program Files**CodeBlocks*, ou dans votre dossier *Programmes*, sous le nom *bonjour*. Et en allant visiter ce répertoire *bonjour*, vous allez constater qu'il contient 2 dossiers et 5 fichiers, tout ça pour le prix d'un seul bonjour. Plus tard si vous voulez refaire marcher ce programme *bonjour*, votre *IDE Codeblocks* étant déjà ouverte, vous faites *file* puis *open*, et quand vous arrivez au nom de votre programme *bonjour*, vous cliquez et dans la fenêtre obtenue vous cliquez sur *bonjour.cbp*. Votre programme réapparaîtra dans la colonne des programmes.

Maintenant passons aux explications sur la programmation. Voici comment se présente en général un programme en C :

```
#include <stdio.h>
#include <stdlib.h>  et éventuellement d'autre include de fichiers .h
main()
```

[2] En fait dans les temps anciens j'ai eu un problème avec Windows Vista, avec un refus total de compiler le moindre programme. Faire un programme stupide de deux lignes pour afficher *Bonjour*, et en plus qu'il ne marche pas, c'est plutôt frustrant. En fait si j'ai bien compris, Windows est un centre de haute sécurité. Il peut refuser l'exécution de programmes qu'il ne connaît pas. C'est ce qui se passait avec la laconique réponse de Code Blocks lors de la compilation du programme : *permission denied* (permission refusée). Pour m'en sortir, j'ai dû désactiver le contrôle sur les comptes d'utilisateur. Si cela vous arrive aussi, voici comment faire : aller dans le *panneau de configuration*, puis dans la rubrique *comptes d'utilisateur*. Là, aller à *ajouter ou supprimer des comptes d'utilisateur*. On tombe sur *choisir le compte à modifier*. Aller en bas de la page et cliquer sur *ouvrir la page principale comptes d'utilisateur*. Puis cliquer sur *activer ou désactiver le contrôle des comptes d'utilisateurs*. Et une fois arrivé là, enlever la petite croix qui activait le contrôle.

```
{
  …
}
```

On commence par mettre quelques *#include* de fichiers *.h* où se trouvent certaines fonctions du langage, celles que nous allons utiliser dans le programme, et qui sans cela ne seraient pas reconnues. Il faut passer par là, se dire que pendant des années on va avoir sous les yeux, et taper des milliers de fois ce genre d'idioties. Cela permet au moins de s'aiguiser les doigts en pensant à autre chose. Puis vient le programme principal *int main*(). Nouvelle plaisanterie : le *main* est considéré comme une fonction qui ne prend aucun argument (aucune variable) d'où les parenthèses (), mais qui ramène un nombre entier, d'où le *int* devant *main*. Cela explique la présence du *return* 0 à la fin du programme. La fonction *main* ramène un entier, et cet entier est 0. Vous pouvez mettre si vous préférez *return* 1000, et le programme principal ramènera 1000 ! D'ailleurs vous le verrez apparaître dans le message final sur l'écran, après *Bonjour*. Si cette situation vous exaspère, vous pouvez toujours simplifier, et écrire le programme suivant :

```
#include <stdio.h>
main()
{ printf(« Un petit bonjour\n ») ;
}
```

Cela va marcher, mais vous recevrez un avertissement (*warning*) pour votre mauvaise conduite.

En fait la seule chose intéressante, c'est de remplir l'intérieur du programme principal, entre les deux accolades {…},[3] où vont se trouver une succession d'instructions, **toutes terminées par un point virgule**, comme vous avez déjà pu le constater. C'est là que tout se passe, car c'est dans cette zone que le problème que l'on veut traiter sera écrit et transmis par nos soins à la machine, et se trouvera résolu lors de l'exécution du programme, si tout va bien ! Evidemment avec notre affichage de *Bonjour*, une chose que le moindre traitement de texte ferait beaucoup mieux et plus simplement, on n'a pas l'air malin.[4]

3. Affichage sur l'écran

Comme on l'a vu, on utilise la fonction *printf*(). Premier miracle, lorsqu'on exécute ce programme : l'écran ne reste pas désespérément noir. *Bonjour* apparaît en haut à gauche de l'écran, suivi d'un message en anglais de fin d'exécution du programme, où l'on vous dit aussi d'appuyer sur une touche pour fermer la fenêtre. Si vous ne voulez pas que votre *Bonjour* soit pollué par ce message, vous pouvez retarder l'échéance, en faisant :

```
#include < stdio.h>
main()    { printf(« Bonjour ») ; getchar() ; return 0 ;}
```

[3] Dans ce qui suit, nous omettrons souvent d'écrire les *#include*, les déclarations de variables, et même le *main*() pour nous concentrer sur l'essentiel : le contenu du programme principal. D'ailleurs lorsque l'on fait un nouveau programme, il suffit de faire un copier-coller d'un ancien programme, puis d'enlever le corps principal de l'ancien programme en écrivant le nouveau à sa place.

[4] Pour la petite histoire, signalons qu'il y a une vingtaine d'années et même plus, le langage *Visual C++* de Microsoft coûtait à peu près le prix d'un *Word Office* aujourd'hui, et qu'il était vendu avec une brochure explicative de 250 pages, où le seul programme qui y était expliqué était l'affichage de *Hello world*, sous une forme sophistiquée quand même. Résultats des courses : ventes calamiteuses. Peu de temps plus tard, on retrouvait une disquette *Visual C++* dans des livres à 200 F. Un peu plus tard, on en arrivait à une version gratuite sur Internet.

La fonction *getchar*() attend sagement que vous appuyiez sur une touche avant d'envoyer le message de fin.[5]

Venons-en au symbole *\n*, que nous avions placé initialement dans notre « Bonjour\n ». Après l'écriture de *Bonjour*, il fait descendre d'une ligne. Essayez par exemple *printf*(« *Bonjour\n Et adios* ») et l'écriture se fait sur deux lignes. Si l'on fait plutôt *printf*(« \n\nBonjour »), *Bonjour* apparaît aussi mais deux lignes en-dessous du haut de l'écran. Si l'on préfère *printf*(« Bonjour »), *Bonjour* apparaît en haut de l'écran, mais décalé vers la droite. C'est dans la bibliothèque *stdio.h* que se trouve définie la fonction *printf*(). Si on ne l'incluait pas au début du programme, on aurait un message d'erreur et un refus de l'affichage attendu.

4. Variable et assignation

La notion de variable est ambigüe. Au départ c'est une lettre ou un mot que l'on déclare par exemple en entiers (*int*), comme *int a;* ou en flottants (*float*) quand on a un nombre à virgule, étant entendu que la virgule est représentée par un point ! Pour une plus grande précision, on dispose aussi des *double*.

Il s'agit en fait d'un nom donné à une case mémoire. Ce que l'on met dedans peut être variable ou fixe, cela dépend de ce que l'on veut. Si l'on fait *a*=3 *;* cela veut dire que 3 est placé dans la case qui s'appelle *a*. Le signe = est un symbole d'assignation ou d'affectation : on affecte à *a* la valeur 3. Tout bêtement on met 3 dans *a*, ou encore *a* contient 3. Faisons ce programme :

```
main()  {int a ;      a=3 ;  printf(« %d », a) ;  getchar() ; return 0 ;}
```

On voit s'afficher 3 sur l'écran. A noter dans *printf* le symbole *%d* qui indique que c'est un entier qui va s'afficher, et c'est celui qui est dans *a*.

Si l'on veut mettre dans *a* un nombre à virgule (un nombre dit flottant : *float*), on fait :

```
float a ;  a=3.14159 ; printf(« %f »,a) ;      /* si l'on veut, on peut écrire a = M_PI, qui contient π */
```

et on voit s'afficher 3.141590. Si l'on veut seulement voir s'afficher 3.14 on peut faire *printf*(« %1.2f »,a). Le chiffre 2 de 1.2 indique que l'on veut deux chiffres derrière la virgule. Quant au 1 de 1.2, il n'a pas grande importance ici, essayez avec 10.2, et il se produira un décalage à droite de l'écriture de 3.14 sur l'écran.

On peut aussi changer le contenu de la case *a*, par exemple :

```
int a ; a=3 ; a=2*a ;
```

Cette dernière instruction indique que la nouvelle valeur de *a* est deux fois l'ancienne. L'affichage de *a* donnera 6.

5. Ajouter un, ou incrémentation

Faisons : *i*=0 ; *i*++ ; *afficher i* (pour afficher faites vous-même le *printf*()). On voit 1 sur l'écran. Le fait d'écrire *i*++ ajoute 1 à la valeur qu'avait *i*. C'est seulement un raccourci, on pourrait aussi bien faire *i* = *i*+1 (on met *i*+1 dans *i*). De même si l'on fait *i*-- avec le contenu de *i* qui diminue de 1. Et cela

[5] D'autres langages *C* que *Code Blocks* ont un défaut supplémentaire : la fenêtre où se trouve *bonjour* disparaît aussitôt et on n'a le temps de rien voir. Pour empêcher cela, une solution est de placer un *getchar()*, qui attendra que vous appuyiez sur une touche pour terminer l'exécution et fermer l'écran où se trouvait *Bonjour*.

se généralise : avec *i*+=2 le contenu de la case *i* augmente de 2, ou avec *i**=2 le contenu de la case *i* est multiplié par 2, etc.

6. Boucle et itération

La boucle *for (i=0 ; i<10 ; i++) {…}* indique que la variable *i* va aller de 0 à 9 en augmentant de 1 chaque fois (à cause du *i++*, qui peut s'écrire si l'on préfère *i=i*+1, ce qui indique qu'à chaque étape le nouvel *i* est l'ancien *i* augmenté de 1.

- Imaginons que l'on veuille afficher sur l'écran les 10 premiers nombres entiers, de 0 à 9. On fait, à l'intérieur du *main()* :

```
int i ;
for(i=0 ; i<10 ; i++)
printf(« %d   »,i) ;          /* ou bien for(i=0 ; i<10 ; i++) {printf(« %d   », i) ;}
```

Comme la boucle *for*() ne contient ici qu'une instruction, on n'est pas obligé de placer celle-ci à l'intérieur d'accolades {}.[6] Profitons-en.

- On veut maintenant avoir la somme des *N* premiers nombres entiers, de 1 à *N*. On utilise pour cela une variable de cumul, qui vaut 0 au départ. Puis on lance une boule qui à chaque fois ajoute un nombre entier à *cumul*. A la fin, *cumul* contient la somme demandée. Voici le programme complet :

```
#include <stdio.h>
int main()
{int i, cumul, N ;
  N=10 ;  /* c'est un exemple */       cumul=0 ;
  for(i=1 ;i <=N ; i++) cumul + = 1 ;  (ou si l'on veut cumul= cumul+1 ; )
  printf(« %d », cumul) ;
  return 0 ;
}
```

Première source d'erreurs, si l'on oublie de mettre *cumul* à 0 au départ, on sera surpris des résultats complètement faux que l'on obtiendra, car le langage *C* est très libre : si l'on ne fait rien, *cumul* va quand même contenir quelque chose, pour nous n'importe quoi.

- Calcul de factorielle *N*, soit *N* ! = *N*(*N*-1)(*N*-2)…3.2.1, *N* étant un nombre entier donné.[7] Après la condition initiale *cumul*=1, on a la boucle de cumul :

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

[6] Si l'on fait : *for(i=0 ; i<10 ; i++) printf(« %d   »,i) ;   printf(« %d   »,i) ;* on verra s'afficher les nombres de 0 à 9, car la boucle *for* porte seulement sur le premier *printf*, puis on verra s'afficher 10 à cause du deuxième *printf*, celui-ci étant en dehors de la boucle. A remarquer qu'après la boucle *for* avec *i*<10, *i* prend la valeur finale 10.

[7] La factorielle devient vite très grande. Par exemple 20 ! est de l'ordre de 2 milliards de milliards, soit 2.10^{18}. C'est le moment de voir la limite du bon usage des entiers. Pour cela reprendre le programme de la factorielle, en faisant à la fois un calcul en entiers et un calcul en flottants pour des valeurs croissantes de *N* :

```
int cumul,i,N ; float cumulflottants ;
for(N=1 ; N<=20 ; N++)    { cumul*=i ;   cumulflottants * = i ;}
printf(« Factorielle %3.d = %d  %25.0f \n », cumul, cumulflottants) ;
```

On s'apercevra qu'à partir de factorielle 13, les calculs en entiers deviennent complètement faux (il apparaît même des valeurs négatives). D'où la règle pratique : ne pas dépasser la limite d'un milliard (en gros) pour les entiers (déclarés *int*).

```
#define N 10    /* on donne la valeur 10 à N, car on veut obtenir factorielle 10  */

int main()
{ int cumul,i ;
  cumul=1 ;
  for(i=2 ;i<=N ;i++) cumul*=i ;
  printf(« Factorielle %d = %d »,N,cumul) ;
  return 0 ;
}
```

On voit apparaître la structure habituelle d'un programme : conditions initiales, puis boucle, puis affichage des résultats.

La boucle *for*() est couramment utilisée, quand on sait qu'une variable doit évoluer entre deux valeurs connues, avec un pas connu lui aussi. On peut aussi l'utiliser sous cette forme apparemment plus libre :

```
compteur=0 ;    /* à vous de mettre l'enrobage habituel, les include, le main, etc. */
for( ;;)
{ afficher compteur ;      /* pour afficher faites vous-même un printf ! */
  if (compteur == 10)  break ;    /* l'égalité s'écrit == , voir plus bas. Voir aussi plus bas le
                                     test si… alors, qui s'écrit if()  */
  compteur++ ;
}
```

On voit aussi s'afficher les nombres de 0 à 10. La boucle *for*(;;) n'a aucune contrainte puisqu'elle ne porte sur rien (on peut aussi bien mettre *while(1)*). Elle se poursuivrait à l'infini si l'on ne mettait pas un *break* qui arrête la boucle dès que la variable *compteur* atteint une certaine valeur. On utilise pour cela un test *if* (si… en français).

7. Assignation = et égal ==

Comme on le constate dans le programme précédent, l'égalité dans *if (compteur = 10)* s'écrit avec le double symbole ==.[8] Ainsi, en langage C, l'assignation s'écrit avec un = et l'égalité avec ==. L'argument avancé par les inventeurs du langage en faveur de ce choix a été qu'on utilise beaucoup plus souvent l'assignation que l'égalité, et que dans ces conditions mieux valait que l'assignation s'écrive plus simplement et rapidement que l'égalité. En ce sens c'est bien mieux que le langage Pascal qui écrit l'égalité avec = et l'assignation avec :=. Signalons toutefois qu'en Basic les deux instructions s'écrivent toutes deux avec un =.

8. Les trois types de boucles

On a déjà vu la boucle *for*(). Il existe deux autres types de boucles :

- while(…) {…}
- do {…} while(…) ;

A la différence du *for*() où sous la parenthèse sont en général précisées la zone de variation et l'évolution des variables au coup par coup, la parenthèse du *while*() ne contient que le blocage final, il convient d'ajouter avant le *while* les conditions initiales des variables concernées, et dans le corps {…} de la boucle leur type d'évolution.

[8] Ne mettre aucun espace entre les deux =, sinon la machine ne va pas comprendre.

Exemple : La suite des puissances de 2

```
int compteur, u ;
compteur=0 ; u=1 ; afficher u ;
while(compteur<10)  { compteur++ ; u = 2*u ;  afficher u ; }
```

A chaque pas de la boucle, *compteur* augmente de 1, à partir de *compteur*=0 mis auparavant en conditions initiales, et la variable *u* est multipliée par 2 à partir de *u*=1. La suite des valeurs de *u* affichées donne les puissances successives de 2, de $2^0 = 1$ à $2^9 = 512$. Remarquons qu'au lieu de faire $u = 2*u$ on pourrait écrire $u *=2$.

On peut aussi faire :

```
int compteur, u ;
compteur=0 ; u=1 ; afficher u ;
do {compteur++ ; u= 2*u ; afficher u ; }
while(compteur<10) ;
```

ou ce qui revient au même :

```
int compteur, u ;
compteur=0 ; u=1 ;
do { afficher u ; u= 2*u ; compteur++ ; }
while(compteur<=10) ;
```

Dans ces deux cas, on verra l'affichage des puissances de $2^0= 1$ à $2^{10} = 1024$.

Les trois types de boucles sont interchangeables, mais dans certains cas on peut préférer à juste titre utiliser l'une plutôt qu'une autre, comme on le verra à l'occasion.

Exercice 1

En 2008 une certaine population comptait un million d'individus. Sa croissance annuelle est de 10%. On désire savoir :

1) Quel sera le nombre d'individus en 2060

2) En quelle année elle dépassera les 100 millions d'individus.

D'abord on pose le problème, et cela n'a rien d'informatique. On considère que 2008 est l'année 0 (le point de départ de l'évolution comme si l'on enclenchait un chronomètre), et qu'en cette année 0 la population vaut $u_0= 1$ où l'unité 1 représente un million d'individus. On a la règle d'évolution suivante : en l'année *n*, la population est de u_n et l'année suivante elle vaut $u_{n+1} = u_n + (10/100)\, u_n$ car 10% signifie 10/100, soit $u_{n+1} = 1{,}1\, u_n$. On obtient une suite de nombres définie par la condition initiale $u_0 = 1$ et la relation de récurrence $u_{n+1} = 1{,}1\, u_n$. Maintenant on programme :

1) On veut connaître u_{52} (population en 2060), ce qui permet d'utiliser une boucle *for*() :

```
float u ; int N, annee ;
 u= 1. ; N=52 ;
for(annee=1 ; annee<=N ; annee++)  u = 1.1 * u ;
afficher u.
```

On trouve qu'en 2060 la population atteint 142 millions d'individus.

2) On veut une boucle où l'évolution se fait année par année jusqu'à ce que la population dépasse la valeur 100, ce qui invite à utiliser plutôt un *while* :

```
int annee ; float u ;
u=1. ; annee=0 ;
while( u<=100.) {annee++ ; u=1.1*u ;}
afficher annee
```

On trouve que c'est en 2057 que la population dépasse pour la première fois 100 millions. Vérifiez-le.

9. Boucles imbriquées

Que fait le programme suivant :

```
for(i=1; i<=5 ; i++)   /* la grande boucle */
for(j=0 ; j<i ; j++)    /* la petite boucle imbriquée dans la grande */
afficher i
```

On voit s'afficher sur l'écran 122333444455555. Pour chaque valeur de *i* dans la grande boucle, la petite boucle se déroule complètement. Par exemple lorsque *i* = 3, la petite boucle provoque l'affichage de 3 trois fois de suite (*j* allant de 0 à 2).

On n'a pas eu besoin de mettre des accolades dans le programme précédent, car il n'a qu'une instruction concernée à chaque fois. On aurait pu en mettre :

```
 for(i=1; i<=5 ; i++)
   {   for(j=0 ; j<i ; j++)
         afficher i
   }
```

voire même :

```
for(i=1; i<=5 ; i++)
   {   for(j=0 ; j<i ; j++)
            { afficher i
            }
   }
```

10. Test si … alors, sinon …

Ce test s'écrit *if* (…)
{…}
else {…}
mais selon les problèmes le « sinon » (*else*) n'est pas obligatoire. Au fond si l'on résume, tout problème traité sur ordinateur se résume à des boucles et des tests. Prenons un exemple.

Imaginons que l'on fasse 10000 tirages au hasard de nombres entiers tous compris entre 0 et 9, et que l'on veuille savoir combien parmi eux sont pairs et combien sont impairs. On peut s'attendre à en avoir en gros moitié-moitié, ce qui permet de tester la validité du générateur de nombres aléatoires de notre langage C.

On utilise pour cela la fonction *rand*() qui ramène au hasard un nombre compris entre 0 et *RAND_MAX*, qui est égal à 32767 (éventuellement vérifiez-le, car cela peut changer au fil du temps). Pour obtenir un nombre au hasard compris entre 0 et 9, il suffit de faire *rand*()%10, ce qui se lit *rand*()

modulo 10, ce qui signifie que l'on prend le reste de la division du nombre *rand*() par 10, ce reste étant compris entre 0 et 9. Ensuite, pour savoir si un nombre *n* (≥0) est pair ou impair, il suffit de chercher son reste lorsqu'on le divise par 2. Si le reste est nul, le nombre *n* est pair, et s'il vaut 1, le nombre est impair. Pour avoir ce reste dans la division par 2, on écrit *n*%2, qui se lit *n* modulo 2. Par exemple 13%2 = 1, 18%2 = 0. Cela revient à enlever au nombre un certain nombre de fois 2 jusqu'à arriver à un nombre qui est soit 0 soit 1. Plus généralement *n*%*p* ramène un nombre compris entre 0 et *p* – 1, après avoir enlevé *p* autant de fois qu'il le faut à *n*.

Revenons à notre problème, qui se traite ainsi :

```
#include <stdio.h>
#include <stdlib.h>    /* indispensable pour la fonction rand() */

int main()
{
int i, chiffre,nombrepairs, nombreimpairs ;
nombrepairs=0 ; nombreimpairs=0 ;  /* les deux variables de cumul */
for(i=0; i<10000 ; i++)
  { chiffre = rand()%10 ;  /* chiffre est un nombre au hasard entre 0 et 9 */
    if (chiffre%2 ==0)   nombrepairs ++ ;
    else  nombreimpairs ++ ;
  }
printf(« Nombres des pairs :%d  et des impairs : %d\n », nombrepairs, nombreimpairs) ;
return 0 ;
}
```

Mais exécutons ce programme plusieurs fois de suite. On constate que l'on obtient toujours les mêmes résultats. En fait la série des 1000 nombres aléatoires n'a pas changé d'un *iota*. Pour éviter ce problème où le hasard fait cruellement défaut, on appelle au début du programme la fonction *srand*(*time*(*NULL*)) qui relie le générateur de nombres aléatoires à l'heure qu'il est. Quand on fait plusieurs exécutions du même programme, le temps a changé, et le générateur aléatoire ne démarre pas avec les mêmes nombres, donnant ainsi des séries de nombres différentes.

```
#include <stdio.h>
#include <stdlib.h>    /* pour la fonction rand() */
#include <time.h>   /* pour la fonction srand() */
int main()
{  int i, chiffre,nombrepairs, nombreimpairs ;
   nombrepairs=0 ; nombreimpairs=0 ;   srand(time (NULL)) ;
   for(i=0; i<10000 ; i++)
     { chiffre = rand()%10 ;
       if (chiffre%2 ==0)  nombrepairs ++ ;
       else  nombreimpairs ++ ;
     }
   printf(« Nombres des pairs :%d  et des impairs : %d\n », nombrepairs, nombreimpairs) ;
   return 0 ;
}
```

11. Tableaux

En général, on n'a pas une ou deux variables à traiter dans un programme. On a une collection d'objets, en grand nombre. On est alors amené à les placer l'un après l'autre dans un tableau de cases successives. Si l'on a *N* nombres entiers à mettre, on déclare le tableau *a*[*N*] dans lequel les nombres vont être placés. Chaque case est numérotée de 0 à *N* – 1. Insistons : quand un tableau a pour longueur *N*, sa dernière case est numérotée *N* – 1 et non pas *N*. Un programme consiste alors à parcourir le tableau, avec une boucle *for*(*i*=0; *i*<*N* ; *i*++) en procédant à des transformations ou recherches selon le problème à traiter. Diverses situations peuvent se présenter :

- Soit le tableau nous est imposé au départ, et c'est à nous de le remplir, par exemple :

```
int a[10] ;  /* déclaration du tableau avec son nom et sa longueur */
a[10]={2, -4, 3, 5, -7, 8, -8, 1, 7, -3} ;   /* remplissage des cases successives du tableau */
ou si l'on préfère, on fait au coup par coup :   a[0]=2; a[1]=-4 ; a[2]=3 ; … a[9]=-3.
```

Imaginons alors qu'on nous demande le nombre d'éléments négatifs dans ce tableau. On fera le parcours suivant, dans le programme principal :

```
compteurnegatifs=0 ;
for(i=0 ; i<N ; i++) if (a[i]<0) compteurnegatifs++ ;
afficher compteurnegatifs
```

- Ou bien le tableau est rempli automatiquement par le programme. Voici un exemple :

On nous demande d'afficher les carrés des *N* nombres pairs successifs à partir de 2. Si *N*=5, on veut connaître les valeurs de 2^2, 4^2, 6^2, 8^2, 10^2. On peut enregistrer ces calculs dans un tableau puis on affiche les résultats :

```
for(i=1 ; i<=N ; i++)   { nbpair=2*i ; carre[i]=nbpair*nbpair ;}  /* remplissage */
for(i=1 ; i<=N ; i++)   printf(«%d   », carre[i]) ;  /* affichage */
```

Remarquons qu'en faisant cela le tableau doit être au préalable déclaré sur une longueur *N*+1, en faisant *int a*[*N*+1]. Les carrés qui nous concernent sont dans les cases de 1 à *N*, la case 0 ne nous intéresse pas.

Maintenant, imaginons que l'on veuille connaître les sommes partielles de ces carrés, soit $2^2 = 4$, $2^2 + 4^2 = 20$, $2^2 + 4^2 + 6^2 = 56$, $2^2 + 4^2 + 6^2 + 8^2 = 120$, $2^2 + 4^2 + 6^2 + 8^2 + 10^2 = 220$. On va les enregistrer dans un tableau *int sommepartielle*[*N*+1] :

```
cumul=0 ;
for(i=1 ; i<=N ; i++)
   { nbpair=2*i ; carre=nbpair*nbpair ; cumul+=carre ; sommepartielle[i]=cumul;}
for(i=1 ; i<=N ; i++)   printf(« %d « , sommepartielle[i]) ;
```

Remarquons que si le problème se réduisait à cet affichage, on n'aurait pas besoin d'utiliser un tableau.

Mais imaginons qu'on nous demande le nombre de termes à prendre dans cette somme de carrés pour atteindre ou dépasser le quart de la somme totale des *N* carrés. Par exemple, pour *N* = 5, avec les sommes partielles de carrés obtenues : 4, 20, 56, 120, 220, il faut prendre les trois premiers carrés, dont la somme vaut 56, pour dépasser le quart de la somme totale, soit 220/4= 55. Dans ce cas, l'usage d'un tableau s'avère bien plus utile, car nous pouvons utiliser les données qu'il a enregistrées. Voici le programme complet :

```
#include <stdio.h>
#define N 5   /* ce #define provoque le remplacement de N par 5 partout dans tout le programme */

int main()
  { int i, nbpair, carre, sommepartielle[N+1], cumul, quartdutotal ;
    cumul=0 ;
    for(i=1 ; i<=N ; i++)
       {nbpair=2*i ; carre=nbpair*nbpair ; cumul+=carre ; sommepartielle[i]=cumul;
       }
    quartdutotal= sommepartielle[N] / 4 ;
    i=1 ;  while ( sommepartielle[i] < quartdutotal) i++ ;
```

```
    printf(« Pour N = %d, le nombre de termes pour dépasser le quart du total est %d », N, i) ;
    getchar() ; return 0 ;
  }
```

12. Fonctions

Le langage *C* est à base de fonctions. En ce sens il est unifié et c'est un avantage par rapport à d'autres langages qui mêlent les fonctions et les procédures (des blocs d'instructions) comme le Pascal ou le Basic, ou feu le Logo.

Quant elle est généralisée, la notion de fonction devient très élastique. Il y a les vraies, les vraies-fausses, les fausses… Au sens strict, une fonction prend une variable *x* et ramène une valeur *y*, ce qui s'écrit $y = f(x)$, c'est-à-dire *y* est fonction de *x*. Commençons par voir comment se présente une telle fonction classique en C.

Imaginons que l'on veuille calculer les valeurs du polynôme $P(x) = x^3 - 2x^2 + 4x - 3$ pour les valeurs entières de *x* comprises entre – 5 et 5. Cela peut se programmer ainsi, même si on peut le faire plus simplement :

```
#include <stdio.h>
int polynome(int x) ;  /* on déclare cette fonction avant le main() */

main()
{ int x, y ;
  for(x=-5 ; x<=5 ; x++)
    { y = polynome(x) ;   printf(« x=%d    y=%d \n » ,x, y) ;
    }
  getchar() ; return 0 ;
}

int polynome(int x)
  { int valeur ;
    valeur = x*x*x – 2*x*x + 4*x – 3 ;
    return valeur ;
  }
```

Prenons la fonction *polynome*(). On constate que sa variable *x* doit être déclarée, ici en entiers, par *int x*, et que la valeur entière qu'elle ramène doit être annoncée devant elle, d'où *int polynome*(). D'autre part, à la fin du corps de la fonction, il est obligé de dire qu'elle ramène une valeur entière, d'où *return valeur*. C'est cette valeur retournée qui est mise dans le *y* du programme principal, puis affichée. Enfin il est indispensable de déclarer la fonction au début du programme, avant le *main*().

Mais il existe d'autres types de fonctions, notamment des fonctions qui ne prennent aucune variable, et ramènent quand même une valeur, comme par exemple notre fameux *int main*(), ou encore *rand*() qui ramène un entier entre 0 et 32767.

Il y a encore des fonctions qui prennent des variables mais ne ramènent rien. On les déclare ainsi, par exemple : *void fonction* (*int u, int v*) , une fonction de deux variables, et où *void* indique que rien n'est ramené. Il n'y a alors aucun *return* à mettre à l'intérieur de la fonction. Que fait une telle fonction ? Elle se contente d'afficher des résultats sur l'écran ou d'y dessiner des objets, et c'est déjà beaucoup.

Mieux encore, il existe des fonctions qui ne prennent aucune variable et qui ne ramènent rien. Elle sont déclarées sous la forme *void fonction*(*void*). De telles fonctions ne sont autres que des procédures englobant un bloc d'instructions, elles servent de prête-nom (ou d'étiquette ou de label) pour indiquer ce que l'on veut réaliser.

13. Variables locales et globales

Une variable locale est déclarée à l'intérieur et au début d'une fonction (cette fonction pouvant notamment être le programme principal *main*). Elle n'agit alors qu'à l'intérieur de cette fonction. Par exemple si une variable *int i* est déclarée dans le *main*() et que l'on a aussi une variable *int i* déclarée dans une autre fonction, cela donne deux variables complètement indépendantes et différentes l'une de l'autre, malgré leur nom identique.

Par contre, une variable globale est déclarée au début du programme, avant le *main*() et avant les autres fonctions éventuelles du programme. Alors elle agit dans toutes les fonctions. Si une fonction modifie la valeur d'une variable globale, cette modification est répercutée dans toutes les autres fonctions. Il y a aussi un autre avantage : lorsque l'on déclare un tableau *a*[*N*] en global, il est automatiquement mis à 0 (par contre s'il est déclaré à l'intérieur du *main*(), il contient n'importe quoi, et si on veut l'initialiser à 0, il faut le parcourir en mettant chaque case à 0).

Exercice 2 : coupes d'un paquet de cartes

On a un paquet de N cartes, on va le couper en deux morceaux de longueur L_1 et L_2 ($L_1 + L_2 = N$), aucun n'étant vide, puis on va le reconstituer en intervertissant les deux morceaux.

1) *On fait une coupe au hasard. Programmer cette opération de coupe.*

On va d'abord noter les *N* cartes du paquet 0, 1, 2, …, *N* – 1. Il s'agit là d'une conversion toute simple : si la première carte du paquet est l'as de cœur, on décide de le noter 0, et ainsi de suite. Ainsi simplifié, le paquet est placé dans un tableau *a*[*N*]. Ensuite on va numéroter la coupure avec un nombre *C* compris entre 0 et *N* – 2, et choisi au hasard. Une coupure numérotée *C* signifie que le premier morceau va de l'indice 0 à *C* dans le tableau *a*[*N*], et que le deuxième va de *C* + 1 à *N* – 1. Dans ces conditions chacun des deux paquets a au moins une longueur égale à 1.

```
for(i=0 ; i<N ; i++) a[i]=i ;
C=rand()%(N-1) ; L1=C+1 ; L2=N – C – 1 ;
for(i=0 ;i<N ; i++) if (i<=C) paquet1[i]=a[i] ; else paquet2[i – L1]=a[i] ;
for(i=0 ; i<N ; i++) if (i<L2) a[i]=paquet2[i] ; else a[i]=paquet1[i – L2] ;
afficher le paquet mélangé a[]
```

2) *Pour mélanger les cartes, on se propose de répéter cette opération de coupe un certain nombre de fois. Programmer. Que va-t-il se passer ?*

Evidemment l'essentiel c'est le programme précédent. Il suffit de le répéter un certain nombre de fois (dans le programme qui suit *nbdecoupes* vaut 10). Pour montrer comment on intègre des fonctions au programme principal *main*(), on a placé l'opération de coupe dans la fonction *couper*(), et l'affichage dans la fonction *afficher*(). Voici le programme complet :

```
#include <stdio.h>
#include <stdlib.h>  /* pour le générateur rand() de nombres aléatoires */
#include <time.h>   /* pour la relance  srand() du générateur aléatoire */

#define N 32
#define nbcoupes 10
int a[N], paquet1[N-1], paquet2[N-1], L1, L2;   /* variables globales, qui concernent  toutes
                                                   les fonctions */
void couper(void);   /* deux fonctions ne prenant aucune variable et ne ramenant rien */
void afficher(void);
```

```
int main()
{
  int i,coupe;       /* variables locales */
  srand(time(NULL));
  for(i=0; i<N; i++) a[i]=i;
  afficher();   /* le tableau initial */
  for(coupe=1; coupe<=nbcoupes; coupe++)   couper();  /* les coupes répétées */
  afficher(); /* résultat après les coupes */
}

void couper(void)
{ int i, C;
  C=rand()%(N-1);   L1=C+1;  L2=N-C-1;
  for(i=0; i<N; i++) if (i<=C) paquet1[i]=a[i];  else paquet2[i-L1]=a[i];
  for(i=0; i<N; i++) if (i<L2) a[i]=paquet2[i];  else a[i]= paquet1[i-L2];
}

void afficher(void)
{  int i;
   for(i=0;i<N;i++) printf("%d ",a[i]);
   printf("\n"); getchar();
}
```

Que constate-t-on ? On a beau multiplier les coupes, on a toujours à la fin un décalage cyclique des cartes, par exemple pour *N*=5, avec au départ 01234, on va par exemple obtenir 23401. Tout se passe comme si l'on n'avait fait qu'une seule coupe. Ce n'est pas un véritable mélange. La machine s'est comportée comme un « ordinateur », la machine qui met de l'ordre et qui n'aime pas le désordre. On n'en veut pas.

3) *Modifier l'opération de mélange pour qu'après l'avoir effectuée un certain nombre de fois le paquet soit vraiment mélangé.*

Pour casser le quasi ordre indéfiniment retrouvé de la méthode précédente, et créer un véritable désordre, on décide par exemple de mettre la dernière carte du deuxième morceau en premier, puis on met le reste du deuxième morceau, suivi du premier morceau. Et l'on répète cela un certain nombre de fois, par exemple autant de fois qu'il y a de cartes. On obtient bien à la fin une permutation quelconque des cartes.

Il suffit de reprendre le programme précédent, et après la coupure en deux paquets dans la fonction *couper*(), de modifier à la fin la reconstruction du paquet *a*[*N*] en faisant :

```
a[0] = paquet2[L2 – 1] ;
for(i=1 ; i<N ; i++) if (i < L2)  a[i] = paquet2[i – 1] ; else a[i] = paquet1[i – L2] ;
```

Conclusion provisoire

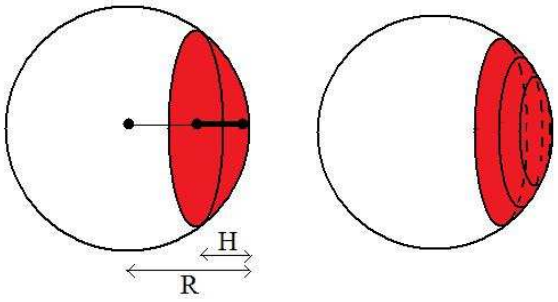
Voilà pour cette petite initiation au langage *C*. Beaucoup de choses n'ont pas été dites, mais avec ce bagage minimal on peut traiter de nombreux problèmes sur ordinateur. D'autres choses non dites sont sous-jacentes à travers les exemples de programmes. Autre limite de cette initiation : nous avons toujours choisi des exemples à base de nombres. Cela peut sembler restrictif, mais l'ordinateur n'est-il pas avant tout un *computer*, c'est-à-dire un computeur, un calculateur ? Et dans de nombreux cas, quand on travaille avec des lettres, on a intérêt à les convertir en nombres.

Pour s'entraîner à la programmation, un bon moyen consiste à prendre le premier chapitre du cours d'algorithmique, dans *enseignements / cours d'algorithmique / Quelques algorithmes et programmes classiques en langage C (sur le site pierreaudibert.fr)*

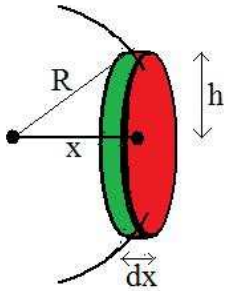
A l'avenir, il reste une chose essentielle à réaliser : ajouter une bibliothèque graphique. La frustration suprême, avec les langages *C* basiques d'aujourd'hui, c'est de ne pas avoir les moyens de dessiner un point sur un écran d'ordinateur. Quand tout n'est qu'image !

Exercice 3 : Calcul d'une calotte sphérique [9]

Sur une sphère de rayon R, on prend une calotte sphérique de hauteur H (avec H entre 0 et R), dont on veut déterminer le volume suivant les valeurs de H.



A gauche, la calotte sphérique de hauteur *H*, et *à droite* son découpage en tranches (3 tranches ici, en fait on en fera des dizaines ou des centaines)



Découpons la calotte en tranches toutes de même épaisseur *dx*. Chaque tranche est assimilée à un cylindre de hauteur *dx*, et dont la base circulaire a un rayon *h*, dépendant de l'abscisse *x*, distance entre le centre de la sphère et celui de la base circulaire du cylindre. Cette abscisse *x* est comprise entre $R - H$ et *R*, et l'on a $h^2 + x^2 = R^2$ (Pythagore), d'où $h^2 = R^2 - x^2$. Le volume d'une tranche cylindrique située à l'abscisse *x* est égal à l'aire de la base circulaire πh^2 multipliée par la hauteur *dx*. Le volume de la calotte est la somme des volumes de toutes les tranches, soit $\sum \pi(R^2 - x^2)dx$, où *x* prend les valeurs comprises entre $R - H$ et *R* en augmentant à chaque fois de *dx*.

D'où le programme, où l'on utilise une variable de cumul *volume* qui vaut 0 au départ, et qui augmente à chaque étape du volume d'une tranche, ce qui donne à la fin le volume de la calotte :

```
#include <stdio.h>
#define pi 3.1416   /* sans define, on peut si l'on préfère utiliser le nombre M_PI intégré au langage C */
float H,R,volume,x,dx;  /* déclaration des variables globales en flottants (nombres à virgule) */

int main()
{   R=1.; H=0.5; dx=0.0001;   /* valeurs prises comme exemple */
    volume=0.;
    for(x=R-H; x<R; x+=dx)        volume + = pi*(R*R-x*x)*dx;
    printf("Volume de la calotte (pour R=%3.1f et H=%3.1f) = %3.3f\n\n",R,H,volume);
    return 0;
}
```

Remarque : Nous venons de réaliser ce qui est la source du calcul intégral, à savoir découper en petits morceaux un volume (ou dans d'autres cas une surface) pour obtenir une valeur approchée du volume global. Cela est connu depuis plus de 2000 ans. Aujourd'hui, le calcul intégral nous permet d'écrire que le volume de la calotte sphérique est

$\int_{R-H}^{R} \pi(R^2 - x^2)dx = \left[\pi(R^2 x - \frac{x^3}{3})\right]_{R-H}^{R}$ par intégration.

[9] Si vous jugez cet exercice trop mathématique à votre goût, passez directement à la suite.

Le calcul donne finalement $\pi H^2 (R - \frac{H}{3})$. On peut comparer les résultats obtenus par le programme précédent et par cette formule théorique, en ajoutant cette ligne au programme précédent :

*volume=pi*H*H*(R-H/3.); printf("Volume théorique = %3.3f\n\n",volume);*

Dans les deux calculs, on trouve pour l'exemple choisi un volume égal à 0,655. C'est plutôt réconfortant.

Tout cela étant fait, il nous reste à apprendre à faire des dessins sur notre fenêtre-écran, grâce à *SDL*, dont nous utiliserons ici la version 1, la plus simple et largement suffisante pour nombre de programmes.