1 le langage C/LabWindows

Un programme informatique est, en toute généralité, une suite d'instructions intermédiaires entre le langage courant (les mots et phrases de tous les jours permettant de décrire un algorithme, une « marche à suivre ») et le langage machine (une suite de 0 et de 1).

Des milliers de langages de programmation existent, généralistes (comme le C, le pascal, ...) ou spécialisés (par exemple maple pour le calcul formel, php pour la programmation de sites web dynamiques, perl pour les manipulations d'expressions, R pour les statistiques, ...)

LabWindows propose un environnement complet de programmation en langage C, l'un des plus répandus.

2 variables

Une variable est une « case mémoire » permettant de stocker une donnée (un entier, un réel, un caractère, ...).

Le langage impose, avant utilisation, de **déclarer** les variables, c'est-à-dire de préciser son nom et le type de données qu'elle pourra contenir.

Il est de plus fortement conseillé d'**initialiser** chaque variable avant usage, c'est-à-dire de lui affecter une valeur initiale. En effet, une variable déclarée mais non initialisée aura une valeur aléatoire, dépendant de l'état de la mémoire de l'ordinateur, ce qui peut créer lors de l'éxécution des dysfonctionnements difficiles détecter et à corriger.

exemple 1 :

```
int i;
i=2;
```

déclare une variable entière nommée `i` puis lui affecte la valeur 2. On peut aussi effectuer ces deux opérations en une seule instruction :

```
int i=2;
```

Le nom de la variable doit commencer par une lettre, et peut ensuite être composé librement de lettres, de chiffres et du tiret de soulignement _.

Le C distingue lettres majuscules et minuscules : `A` et `a` sont deux variables distinctes. De plus, on ne peut définir en C deux variables de type différent ayant le même nom : les déclarations `int a; double a;` ne peuvent se suivre.

Les principaux types de variables sont :

- **les entiers,** déclarés par `int`,
  ce sont des entiers signés codés sur 4 octets, et donc compris entre -2 147 483 648 et +2 147 483 647.

- **les réels** simple précision déclarés par `float`,
  ce sont des nombres à virgule codés sur 4 octets, donc compris entre -3.4e-38 et +3.4e+38. Leur précision est de 7 chiffres significatifs.

  exemple : `float pi=3.1415927;`

  Compte-tenu de la puissance et de la mémoire des ordinateurs dont vous disposez, utiliser les réels simple précision ne comporte plus grand intérêt par rapport aux réels double-précision :

- **les réels double-précision,** déclarés par `double`,
  ce sont des nombres à virgule codés sur 8 octets, donc compris entre -1.7e-308 et +1.7e+308. Leur précision est de 15 chiffres significatifs.

  exemple : `double pi=3.14159265358979;`

  exemple : `double x=1.265443929243E+222;`

- **les caractères,** déclarés par `char`,
  permettant de coder l'un des 256 caractères alphanumériques de la table des codes ASCII.

  exemple : `char c='a';`

– **les chaînes de caractères,** déclarées par `char nomvariable[n]` où n est un entier qui représente le nombre maximal de caractères dans la chaîne.

exemple : `char c[30];`

déclare une chaîne nommée `c` et pouvant contenir au plus 30 caractères. Il est cependant difficile de lui affecter directement une valeur dans le corps du programme : nous verrons plus loin comment l'utilisateur pourra saisir au clavier une chaîne.

remarque : une chaîne de caractère est en fait un tableau de caractères, de construction analogue aux tableaux de nombres que l'on décrira plus loin (voir 10).

3 constantes

Si un programme utilise plusieurs fois une valeur approchée du nombre π, on peut pour éviter de re-taper cette valeur définir une variable réelle par l'instruction `double pi=3.1415926;`.

Mais la valeur de la variable ne changeant pas tout au long de l'exécution du programme, utiliser une variable n'est pas le meilleur choix : on préfèrera utiliser une **constante.**

Il s'agit d'un simple raccourci : un

`#define PI 3.1415926`

placé en tout début de programme, et qui indique au compilateur de remplacer chaque occurrence de `PI` dans le code-source par la valeur associée `3.1415926`. Ainsi, dans le programme compilé aucune variable inutile n'apparaît, ce qui assure un (petit) gain de place mémoire et de temps d'exécution.

Attention à la syntaxe : pas de signe `=`, pas de `;` à la fin de la ligne `#define PI 3.1415926`, qui n'est pas à proprement parler une instruction mais une directive de compilation.

4 affichage : `printf`

Pour afficher le contenu d'une variable, on utilise la fonction `printf` qui réalise un affichage formaté, en permettant de mélanger texte et contenu de variables.

exemple : `printf("message");`

affiche simplement le texte `message`

Une chaîne de caractères explicite est écrite, dans un `printf`, en indiquant les caractères entre `"`, pour la distinger d'un nom de variable.

Si l'on souhaite afficher le contenu d'une variable, il faut préciser son type dans la chaîne de formatage, et ensuite indiquer son nom :

exemple : `printf("%d",message);`

affiche le contenu de la variable entière `message` (sous réserve qu'elle ait été déclarée préalablement bien sûr).

On peut bien entendu mélanger texte et contenu de variables, en donnant une chaîne de caractère composée des caractères à afficher et de symboles spéciaux débutant par `%` qui indiquent la place et le type des variables. Ainsi :

`%d` indique un entier écrit en base 10 (**d**écimal), `%x` indique un entier écrit en base 16 (he**x**adécimal), `%o` indique un entier écrit en base 8 (**o**ctal),

`%f` indique un réel (**f**loat),

`%e` indique un réel en écriture scientifique (avec un **e**xposant),

`%lf` indique un réel double-précision (**l**ong **f**loat),

`%le` indique un réel double-précision en écriture scientifique,

`%c` indique un caractère (**c**har),

`%s` indique une chaîne de caractères (**s**tring).

exemple :

```
main()
  {
  int i=2;
  double x=2.256;
  char c='a';

  printf("L'entier %d, le réel %lf, le caractère %c", i,x
  }
```

affiche le texte : `L'entier 2, le réel 2.256 et le caractère a`.

Les différentes options d'affichage de réels (`%f, %e, %lf, %le`) permettent, en option, de préciser à la fois le nombre de caractères total à afficher et le nombre de chiffres après la virgule.

On indique ces nombres, séparés par un point, juste après le % :

exemple : le programme

```
main()
{
  double pi=3.14159265358979;
  double x=3009444.34;

  Cls();

  printf("%6.2lf", pi);

  printf("\n%20.14le", x);

  printf("\n%10.6le", x);
}
```

affiche successivement :

- la valeur de π sur 6 caractères (point de séparation décimale inclus) avec deux chiffres après la virgule. Comme 3.14 n'utilise que 4 caractères, la fonction insère deux espaces avant les chiffres.
- la valeur de la variable `x` en notation scientifique, avec 14 chiffres après la virgule, soit `3.00944434000000e+06`, sur 20 caractères en tout.
- la valeur de la variable `x` en notation scientifique, avec 6 chiffres après la virgule, sur 14 caractères en tout, `3.009444e+06`

Outre ces symboles précédés de % qui permettent d'afficher le contenu des variables, on note aussi l'existence de certains "caractères spéciaux", précédés d'un « backslash » : `\n` qui insère un saut de ligne, `\s` qui insère une espace, `\t` qui insère une tabulation.

Les caractères \ et % n'apparaissent donc pas à l'écran à l'exécution du programme : si l'on désire réellement les afficher, il faut les doubler :
`printf("\\");` affichera \, `printf("%%");` affichera %.

Notons enfin la fonction `Cls()` (spécifique à LabWindows) qui efface le contenu de la fenêtre d'affichage.

5 saisie de valeurs au clavier : `scanf`

La fonction `scanf` est symétrique de la fonction `printf` : elle permet de saisir au clavier une valeur à ranger dans une variable.

exemples :
`scanf("%d",&i);` range dans la variable `i` un entier saisi au clavier.
`scanf("%lf",&x);` range dans la variable `x` un réel saisi au clavier.
`scanf("%c",&c);` range dans la variable `c` un caractère.

Notons une différence de syntaxe importante avec `printf` : le `&` devant le nom de la variable, qui indique qu'on s'intéresse ici à l'**adresse** de la variable, l'endroit où ranger la valeur saisie au clavier, et non pas à son ancienne valeur.

A noter une exception de taille à cette règle :
`scanf("%s", chaine);` range dans la variable `chaine` une chaîne de caractères.

Il n'y a pas ici de & devant le nom de la variable, car le nom d'une variable de type « chaîne de caractères », ou plus généralement le nom de tout tableau de valeurs, correspond déjà à une adresse.

Ainsi, `scanf("%s", chaine);` affecte la chaîne saisie au clavier à la variable `chaine`. Attention, si la chaîne rentrée au clavier par l'utilisateur comporte des espaces, seule la partie précédant la première espace sera prise en compte.

remarque 1 : LabWindows est très peu tolérant avec les mélanges de type, et aura une tendance certaine à planter si l'utilisateur rentre une lettre alors qu'un nombre est attendu, ou un séparateur décimal . si un nombre entier est attendu.

remarque 2 : quand on utilise deux instructions `scanf` successives pour récupérer un nombre, puis un caractère ou une chaîne de caractères, on voit apparaître un petit problème.

En effet, quand l'utilisateur rentre un nombre, il tape la suite de chiffres correspondante puis appuye sur la touche entrée du clavier. La suite de

chiffres est transformée en un nombre, rangé dans la variable indiquée. Mais le caractère `\n` qui correspond à l'appui sur la touche entrée n'était pas un chiffre, il est « gardé en mémoire » par le clavier. Mais ce caractère convient au second `scanf`, qui ne laisse donc pas le temps à l'utilisateur de taper sa réponse au clavier.

Pour éviter tout problème, on fera précéder toutes les instructions `scanf` par l'instruction `fflush(stdin)` qui signifier « vider l'entrée-standard (stdin) », autrement dit vider la mémoire du clavier (flush).

exemple : une séquence-type d'instruction pour saisir une valeur au clavier ressemblera donc à cela :

```
main()
{
  char reponse;

  printf("Répondez par o ou n : ");
  fflush(stdin);
  scanf("%c",&reponse);
}
```

6 opérations mathématiques

Beaucoup d'opérations ou fonctions mathématiques sont connues du C ou de la bibliothèque de fonctions mathématiques du C et de LabWindows. Citons quelques exemples :

- les quatres opérations `+, -, *, /`,
- le modulo `%` qui renvoie le reste de la division entre deux entiers,
- la fonction racine carrée `sqrt`,
- les fonctions trigonométriques `sin, cos, tan, atan`,
- les exponentielle et logarithme `log, exp`, ...

opérandes entières : On fera attention aux résultats de la division entre deux entiers : `a/b` renvoie le quotient entier de la division, ainsi `5/2` renvoie `2`. Si l'on souhaite obtenir un résultat correspondant à la division réelle, il faut forcer l'une des deux opérandes à être un réel, par `5.0/2` ou par `(double)5/2`, la deuxième syntaxe plus longue ayant l'avantage de fonctionner même si l'opérande est une variable.

```
main()
{
  int a=3;

  printf("%d",3/2);
  printf("\n%lf",3.0/2);
  printf("\n%lf",(double)3/2);
  printf("\n%lf",(double)a/2);
}
```

affiche successivement `1`, `1.5`, `1.5`, `1.5`.

raccourcis : le C propose quelques raccourcis pour les additions simples :
`i++` est équivalent à l'instruction `i=i+1`,
`i--` est équivalent à l'instruction `i=i-1`,
`i+=5` est équivalent à l'instruction `i=i+5`.

7 instructions conditionnelles

Tout programme informatique dépassant quelques lignes de code doit pouvoir adapter son comportement aux résultats de calculs, aux réponses de l'utilisateur...Le langage C dispose de deux types d'instructions pour cela.

7.1 l'instruction `if`

L'instruction de base est le « si » : `if` :

exemple : `if (x>0) { printf("x est positif"); }`
Le fonctionnement est simple à comprendre : si la condition entre parenthèse est vérifiée, on exécute le bloc d'instruction qui suit (qui peut être une instruction unique ou bien un ensemble d'instructions compris entre accolades).

Dans l'exemple si dessus, si la valeur de la variable `x` est strictement positive, le programme affiche le message `x est positif`, sinon, rien ne se passe et l'exécution du programme continue à l'instruction suivante.

7.2 le test `if ... else`

Un raffinement est un test du type « si . . .alors . . .sinon . . . » :

exemple :

```
main()
{
     double x;

     printf("Rentrez un nombre : ");
     scanf("%lf",&x);

     printf("Vous avez rentré le nombre %lf, qui est ");

     if (x>0){ printf("positif."); }
       else  { printf("négatif ou nul."); }
}
```

Ici, on peut donc afficher deux messages différents, selon la valeur (et en particulier ici, le signe) de la variable.

Notons au passage que l'obligation d'indiquer à l'instruction `printf` les passages à la ligne est ici un avantage : on peut écrire une phrase unique en combinant deux `printf` successifs : un affichage commun complété d'une partie variable en fonction du résultat du test.

7.3 le test `switch ... case`

Pour tester successivement plusieurs valeurs d'une variable, on peut imbriquer les tests.

Par exemple, dans un menu permettant à l'utilisateur de choisir entre trois actions, on peut stocker le choix 1, 2 ou 3 dans une variable et utiliser ensuite des test `if ... else ...` imbriqués pour agir en fonction de ce choix :

```
if (c==1){ /* instructions en cas de choix 1 */ }
else {
     if (c==2) { /* instructions en cas de choix 2 */ }
     else {
          if (c==3) { /* instructions en cas de choix 3 */ }
          else { /* instructions dans les autres cas */ }
          }
     }
```

Mais cette méthode, si elle fonctionne, donne un code lourd et peu lisible.

On peut lui préférer l'instruction `switch ... case ...` :

```
switch(c)
     {
     case 1: /* instructions en cas de choix 1 */ break;
     case 2: /* instructions en cas de choix 2 */ break;
     case 3: /* instructions en cas de choix 3 */ break;
     default: /* instructions dans les autres cas */;
     }
```

Ainsi, le programme regarde (`switch`) le contenu de la variable, et dans chaque cas (`case`) exécute les instructions prévues. Le `break` est nécessaire pour reprendre l'exécution après le `switch ... case ...`, sans examiner les cas suivants et en particulier le `default`, qui correspond aux instructions à exécuter quand aucun des cas énumérés par `case` n'est apparu.

Attention, il est impossible de tester avec une instruction `switch ... case` autre chose que l'égalité du contenu d'une variable à des constantes : pas d'inégalité, pas de tests composés à l'aide d'opérateurs « et » (&&), « ou » (||), ...

8 boucles

Il est souvent utile de répéter une opération, un grand nombre de fois : par exemple, 10 fois, ou bien tant qu'une condition est vérifiée, en changeant à chaque exécution seulement quelques paramètres. Pour cela on distingue deux grandes familles de boucles : la boucle `for` et la boucle `while`.

Si, théoriquement, les deux sont équivalentes, leur usage en pratique diffère légèrement :

– on utilisera souvent la boucle `for` pour exécuter un nombre prédéfini de fois un bloc d'intruction dans lequel on aura besoin d'une variable modifiée (incrémentée de 1, le plus souvent) à chaque exécution,
– on utilisera souvent la boucle `while` pour une boucle dont l'exécution est controlée par une condition booléenne.

8.1 la boucle `for`

C'est la boucle qui correspond à la phrase française « pour i allant de 0 à 9 par pas de 1, ... ».

exemple 1 :

```
for ( i=0 ; i<10 ; i++ )
     { printf("Le compteur vaut maintenant %d", i); }
```

On voit qu'une boucle `for` comporte trois parties :
– le mot-clé `for`,
– entre parenthèses, les conditions d'éxécution de la boucle, elles-mêmes composées de trois éléments séparés par des points-virgules :
  – **l'initialisation** `i=0` : avant la première exécution de la boucle, on affecte à une variable préalablement déclarée `i` la valeur 0.
  – **le test d'exécution** `i<10` : la boucle sera exécutée tant que `i` gardera une valeur strictement inférieure à 10. Cette condition est donc vérifiée avant chaque exécution de la boucle.
  – **la modification de la variable** `i++` : après chaque exécution de la boucle, on modifie la valeur de `i`, ici en l'incrémentant de 1.
– entre accolades, les instructions proprement dites :
  ici, à chaque itération (répétition) de la boucle, on affiche un message comportant la valeur de la variable.

On peut imaginer des variantes :

exemple 2 :

```
for ( i=10 ; i<20 ; i=i+2 )
     { printf("Le compteur vaut maintenant %d", i); }
```

exécute 5 fois la boucle avec une variable qui prend successivement les valeurs 10, 12, 14, 16, 18.

exemple 3 :

```
for ( i=100 ; i>=0 ; i-- )
     {
       j = i/3;
       if (3*j==i) { printf("%d est divisible par 3", i);
     }
```

exécute 101 fois la boucle, la variable prenant ses valeurs par ordre décroissant 100, 99, 98, …, 1, 0, mais seules les valeurs divisibles par trois sont affichées.

Comment écrire la même chose en enlevant le test `if` entre accolades et en modifiant les instructions de contrôle entre parenthèses ?

Que font les boucles suivantes :

exemple 4 :

```
for ( i=1 ; i>0 ; i++ )
     { printf("Le compteur vaut maintenant %d", i); }
```

et

exemple 5 :

```
for ( i=100 ; i>=0 ; i=i-2 )
     { printf("Le compteur vaut maintenant %d", i); }
```

?

8.2 la boucle `while`

Cette structure correspond à la phrase « tant que » : la boucle s'exécute tant qu'une certaine condition est vérifiée.

exemple 1 :

```
while ( reponse != 'n' )
     { printf("Voulez-vous continuer (o/n) ?");
       scanf("%c",&reponse); }
```

Ici, tant que l'utilisateur ne répond pas "n", la boucle continue de s'exécuter.

Il est très simple d'obtenir une boucle s'exécutant indéfiniment :

exemple 2 :

```
x=7;
while ( 1 )
     {
       x=x/2;
       printf("%lf", x);
     }
```

La valeur numérique 1 correspond à la valeur de vérité « vrai », cette boucle ne s'arrêtera jamais (ici, en affichant successivement 3.5, 1.75, 0.875, ... puis une fois atteinte la limite de précision de la machine, une infinité de 0).

exercice : comment fabriquer à l'aide d'une boucle `while` l'équivalent d'une boucle `for` ?

9 fonctions

9.1 notion de fonction

Un programme en C est constitué d'une ou plusieurs fonctions, permettant de distinguer des tâches élémentaires. Ainsi, la fonction principale `main` présente dans tous les programmes va à son tour appeler d'autres fonctions pour réaliser le travail complexe demandé au programme.

Beaucoup de fonctions sont déjà fournies par l'environnement LabWindows (via les lignes `#include` présentes en début de programme), comme `printf` pour réaliser des affichages, `Cls` pour effacer l'écran (la fenêtre d'entrée-sortie plus précisément), etc.

Mais il est bien sûr possible d'en définir de nouvelles...

9.2 définition d'une fonction

Schématiquement, une fonction est un petit programme qui prend en entrée zéro, un ou plusieurs paramètres, exécute une action (affichage, écriture dans un fichier, calculs, etc.) et éventuellement retourne un résultat qui peut être utilisé par la fonction appelante.

Pour définir une fonction, on doit indiquer quatre éléments :

- le type de la valeur renvoyée ;
- le nom de la fonction ;
- la liste des paramètres utilisés (nom et type) ;
- le corps de la fonction constitué des instructions à exécuter.

exemple : un exemple complet simple est la définition d'une fonction `carre` qui calcule le carré d'un nombre :

```
double carre(double x)
{
       return x*x;
}

main()
{
       printf("%lf", carre(2.5));
}
```

Ici donc on définit successivement une fonction carre en indiquant :

- qu'elle renvoie une valeur réelle (`double`),
- son nom (`carre`),
- qu'elle attend un paramètre réel, qui sera appelé `x`,
- la seule instruction exécutée : le carré de `x` est calculé par l'opération `x*x` et renvoyé (retourné).

A l'intérieur de la fonction `main`, la fonction `carre` est appelée avec le paramètre 2.5 pendant l'exécution de l'instruction `printf`, et le programme va donc afficher le résultat renvoyé par la fonction `carre`, soit 6.25.

9.3 compléments

- comme pour les noms de variables, un nom de fonction commence par une lettre et sera constitué de lettres (majuscule ou minuscule), chiffres et tirets de soulignement.
- si plusieurs paramètres sont attendus, ils sont précisés successivement, séparés par des virgules : `(double x, double y, int a)`.
  On ne peut pas, dans les paramètres d'une fonction, utiliser d'abréviation du type `double x,y`.
- par défaut, si aucun type n'est précisé devant le nom de la fonction, le compilateur estimera qu'il s'agit d'un entier.
  Pour écrire une fonction qui ne renvoie aucune valeur, il est donc nécessaire de préciser qu'elle est de type vide (`void`) pour éviter un avertissement : si la fonction n'est pas de type vide, l'instruction `return` est attendue par le compilateur.
- si une fonction peut demander plusieurs paramètres, il n'est pas possible simplement de faire en sorte qu'elle renvoie plusieurs valeurs.
- outre les paramètres, on peut définir au début de chaque fonction des variables par des instruction du type `double x;`, exactement comme pour la fonction `main`.
  Par défaut, en C, les variables sont locales : elles n'existent qu'à l'intérieur de la fonction où elles sont définies, et ne sont pas connues dans les fonctions appelantes ou appelées.

exemple : on souhaite définir une fonction `sinc` qui renvoie le sinus cardinal d'un nombre.

Une manière de procéder est la suivante :

```
double sinc(double x)
{
 double y;

 if (x==0)
     { y=0;}
 else
     { y=sin(x)/x; }

 return y;
}


main()
{
 double a;

 Cls;
 printf("Rentrez un réel :");
 scanf("%lf", &a);
 printf("\n Le sinus cardinal de %lf est %lf", a, sinc(a)
}
```

Ainsi, la fonction prend un paramètre `x` (fourni lors de l'appel de la fonction) et utilise une variable `y` pour stocker le résultat, calculé par les deux formules possibles, l'une si `x` est nul, l'autre si `x` est nul. Le résultat calculé est ensuite retourné par l'instruction `return y;`.

Il est à noter que la variable `a` n'existe qu'à l'intérieur de la fonction `main()` ; si l'on souhaite utiliser sa valeur, il faut comme ici la passer en paramètre à la fonction `sinc`. De même, une fois revenu dans la fonction `main`, après calcul du sinus cardinal, les variables `x` et `y` n'existent plus.

10 tableaux

Pour stocker plusieurs valeurs numériques, on peut bien sûr utiliser plusieurs variables, que l'on peut nommer `a, b, c, ...`, voire `a1, a2, a3, ...`

Mais si l'on effectue des opérations concernant l'ensemble de ces variables (par exemple, affecter une valeur à chacune, calculer leur somme, etc.) cette solution n'est absolument pas souple car elle oblige à prévoir, dès l'écriture du programme, le nombre précis de valeurs qui seront à traiter, et surtout d'écrire explicitement les expressions de type saisie de valeurs ou somme : c'est encore possible si l'on a une poignée de valeurs, difficilement praticables si le programme doit en traiter quelques dizaines, centaines, milliers, ...

Le C, comme tous les langages de programmation, propose une structure

de données plus adaptées : les tableaux, autrement dit des variables indicées par un nombre entier.

On peut déclarer un tableau ainsi :

```
double a[10];
```

Ainsi, dix variables sont créées, `a[0], a[1], ..., a[9]`. On peut les manipuler, exactement comme les variables simples habituelles.

Bien sûr, les tableaux peuvent aussi être de type `int`, `char`, ...Et le nombre de valeurs n'est pas nécessairement 10...

remarque : attention, si on déclare un tableau de 10 valeurs par l'instruction `double a[10];` il n'existe pas de variable `a[10]` !

exemple 1 : Un avantage important de cette manière de procéder est de permettre de traiter toutes les valeurs à l'aide d'une boucle. L'instruction :

```
for (i=0; i<10; i++){ a[i]=0; }
```

affecte ainsi à chaque variable -à chaque case du tableau- la valeur nulle. On évite ainsi de devoir écrire les dix instructions `a[0]=0;`, ..., `a[9]=0;`

exemple 2 : une pratique utile, en particulier pour tracer des courbes, est de placer dans un tableau `n` valeurs régulièrement réparties entre deux réels `a` et `b`. On utilisera pour cela l'instruction :

```
for (i=0; i<n; i++) { X[i]=a+ i*(b-a)/(n-1); }
```

exemple 3 : l'utilisation d'une boucle permet aussi de calculer facilement la somme des valeurs d'un tableau : si l'on dispose d'un tableau `X` avec `n` valeurs, et d'une variable `s`, on peut procéder ainsi :

```
s=0;
for (i=0; i<n; i++){ s = s + X[i]; }
```

A la fin de l'exécution de ce bout de programme, la variable `s` contient bien la somme des éléments du tableau.

11 graphiques

Etant donnée une fonction - qui peut être l'une des fonctions déjà connues du langage (comme `sin`, `sqrt`, `exp`, ...) ou bien une fonction écrite pour l'occasion (par exemple `sinc`)- on souhaite en tracer le graphe sur un intervalle donné.

La méthode informatique correspond à ce qui aurait été fait à la main : on détermine un certain nombre d'abscisses régulièrement réparties sur l'intervalle, que l'on place dans un tableau `X`. Puis on calcule les ordonnées correspondantes, placées dans un tableau `Y`.

Il ne reste alors plus qu'à faire appel à une ou plusieurs instructions qui vont placer les points correspondants dans une fenêtre graphique et les relier entre eux.

11.1 solution simple...

Le programme suivant présente une solution simple pour réaliser le tracé graphique de `exp` sur `[-1,1]` :

```
main()
{
 double X[200], Y[200];
 int i;

 for (i=0; i<200; i++)
       {
       X[i] = -1 + 2.0*i/199;
  /* Attention, le 2.0 est obligatoire ici : sinon,
     2 et i étant entiers, le calcul de 2*i/199 renverrai
     une valeur entière (troncature, sans décimale, de la
     valeur exacte)
  */
       Y[i] = exp(X[i]);
       }

 XYGraphPopup("premier tracé", X, Y, 200, VAL_DOUBLE,
              VAL_DOUBLE);
}
```

La fonction `XYGraphPopup` s'occupe de tout...il suffit de lui fournir deux tableaux de valeurs et de lui indiquer le nombre de points à tracer.

11.2 solution souple...

La solution précédente a l'avantage d'être très simple à mettre en oeuvre, mais l'inconvénient que l'on ne maîtrise par du tout l'aspect de l'affichage (couleurs, dimensions, échelles, ...).

Il est possible de remplacer l'instruction unique `XYGraphPopup` par une série d'instructions qui vont successivement :

- définir une fenêtre,
- l'afficher,
- définir dans cette fenêtre une (ou plusieurs) zones graphiques,
- régler la taille de cette zone graphique
- effectuer un tracé dans cette zone graphique.

Le bout de code suivant illustre cela :

```
main()
{
 double X[200], Y[200];
 int i;
 int fenetre, zone;

 for (i=0; i<200; i++)
       {
       X[i] = -1 + 2.0*i/199;
       Y[i] = exp(X[i]);
       }

 fenetre = NewPanel(0, "fenetre de tracé", 10, 20, 600, 800);
 DisplayPanel(fenetre);

 zone = NewCtrl(fenetre, CTRL_GRAPH,
                   "zone de tracé", 10, 20);
 SetCtrlAttribute(fenetre, zone, ATTR_HEIGHT, 550);
 SetCtrlAttribute(fenetre, zone, ATTR_WIDTH, 750);

 PlotXY(fenetre, zone, X, Y, 200,
         VAL_DOUBLE, VAL_DOUBLE, VAL_THIN_LINE,
         VAL_EMPTY_SQUARE, VAL_SOLID, 1, VAL_RED);
 GetKey();
}
```

Ici, la fonction `NewPanel` crée une fenêtre dont on détermine le titre, la position initiale (à 10 pixels du haut de l'écran et 20 du bord gauche) et les dimensions (600 pixels de haut et 800 de large). De plus la fonction renvoie un identifiant (un nombre entier) pour cette fenêtre, que l'on range dans une variable. Puis cette fenêtre est affichée par la fonction `DisplayPanel` dont le paramètre est justement l'identifiant précedemment attribué à la fenêtre.

A l'intérieur de la fenêtre, une zone graphique est affichée : à 10 pixels du haut de la fenêtre, 20 du bord gauche, avec une hauteur de 550 pixels et une largeur de 750, par les instructions `NewCtrl` et `SetCtrlAttribute`.

Puis l'instruction `PlotXY` trace la courbe. Mais on voit que le nombre de paramètres est plus important que pour la fonction `XYGraphPopup` : on peut choisir l'aspect des traits (lignes continue, pointillée, ...), des points (marqués ou non), la couleur...

12 fichiers

12.1 généralités

Les données calculées par un programme peuvent être archivées, ou bien traitées des heures ou des jours plus tard par un autre programme...Il sera pour cela nécessaire de savoir les écrire ou les relire dans un fichier.

12.2 écriture

L'opération la plus simple est l'écriture dans un fichier.

exemple 1 : dans cet exemple, on se contente d'écrire un message dans un fichier.

```
main()
{
 FILE *fichier;

 fichier = fopen("Z:\\Programmation\\essai.txt", "w");
```

```
 fprintf(fichier, "Test de la fonction fprintf");
 fclose(fichier);
}
```

Les trois étapes du programme sont donc :

- l'ouverture par la fonction `fopen` du fichier dont le chemin d'accés est passé en paramètre. Le chemin d'accés réel est en fait ici `Z:\Programmation\essai.txt`
  mais il est nécessaire de doubler les \ car ceux-ci sont des caractères spéciaux.
  Le `"w"`, pour « **w**rite » indique que le fichier est ouvert (créé s'il n'existe pas, et préalablement vidé s'il existe déjà) pour écriture.
  Un identifiant est attribué et rangé dans une variable `fichier` (d'un type spécifique `FILE`). Dorénavant, le programme ne fera plus référence au chemin d'accés du fichier sur le disque, mais uniquement à cet identifiant.
- L'écriture dans le fichier, par la fonction `fprintf`, très analogue à la fonction `printf`, qui exige comme argument supplémentaire l'identifiant du fichier dans lequel écrire.
- Puis, une fois l'écriture terminée, on signale au système que le fichier est traité, par l'instruction `fclose`.

Quand, comme souvent, les données à sauvegarder sont des valeurs numériques rangées dans un tableau (typiquement, une série de mesures, ou bien les coordonnées d'un grand nombre de points), on combine le programme précédent avec une boucle :

```
main()
{
 FILE *fichier;
 double X[200];
 int i;

 fichier = fopen("Z:\\Programmation\\essai.txt", "w");
 for (i=0; i<200; i++){  fprintf(fichier, "%lf\n", X[i]); }
 fclose(fichier);
}
```

Ici les valeurs contenues dans le tableau sont écrites les unes à la suite des autres dans le fichier (avec ce programme, les valeurs de `X` n'étant pas initialisées, le contenu du fichier sera sans grand intérêt : mélange « aléatoire » de 0 et de valeurs du type 1.2994944e+261...).

12.3 lecture, version 1

On peut avoir besoin de relire des données écrites lors d'une séance de mesures quelques jours avant, ou fournies par un autre technicien.

La méthode s'inspire naturellement de ce qui précède :

```
main()
{
 FILE *fichier;
 double X[200];
 int i;

 fichier = fopen("Z:\\Programmation\\essai.txt", "r");
 for (i=0; i<200; i++){  fscanf(fichier, "%lf\n", &X[i]); }
 fclose(fichier);
}
```

Il s'agit de relire des données, donc le `w` de « **w**rite » est devenu un `r` pour « **r**ead », et le `fprintf` est devenu un `fscanf`.

Bien entendu, pour fonctionner ce programme nécessite la présence d'un fichier `essai.txt` au bon emplacement et contenant 200 valeurs numériques séparées par des retours-chariot.

12.4 lecture, version 2

Le principal défaut du programme précédent est qu'il nécessite de connaître à l'avance le nombre de valeurs présentes dans le fichier.

Si tel n'est pas le cas, et que l'on connaît seulement une borne supérieure à ce nombre de données, on peut procéder un peu différemment : la boucle `for`, exécutée autant de fois que le nombre de données, peut être remplacée par une boucle `while` qui sera exécutée « tant que » le fichier contient encore des données.

```
main()
{
 FILE *fichier;
 double X[200];
 int i;

 fichier = fopen("Z:\\Programmation\\essai.txt", "r");

 i=0;
 while (!feof(fichier))
       {
        fscanf(fichier, "%lf\n", &X[i]);
        i++;
       }
 fclose(fichier);
}
```

`feof` est la fonction qui teste si l'on est arrivés à la fin du fichier. Et le ! est la négation : l'instruction signifie donc « tant que l'on n'est pas à la fin du fichier », on lit un nombre supplémentaire dans le fichier, que l'on range dans la variable `X[i]`, et on augmente l'indice `i` de 1.

Ainsi, à la fin de cette boucle, la variable `i` contient le nombre de données lues, et ces données sont rangées dans le tableau `X`.

Bien entendu, si le nombre de données est strictement supérieur à 200 dans le fichier, le programme ne fonctionnera pas correctement. Mais en revanche il est capable de lire correctement tout fichier contenant entre 0 et 200 nombres réels.

13 traitement d'images

L'utilité des outils précédents peuvent être illustrés de manière visuelle par quelques algorithmes simples de traitement d'images.

Une image (noir et blanc) sera représentée par une matrice (un tableau de lignes et colonnes). Chaque case, correspondant à un point de l'image, sera d'un type de données nouveau : des `unsigned char`, soit des entiers compris entre 0 et 255. La valeur 0 correspond à un pixel noir, la valeur 255 à un pixel blanc, et les différentes opérations sur les images vont consister à modifier ou ranger différemment ces valeurs.

On précisera en début de programme la taille des images (fixée une fois pour toute dans le programme) :

```
#define LARGEUR 400
#define HAUTEUR 266
```

et on déclarera deux variables pour disposer de deux images (typiquement, un original et une image modifiée) :

```
unsigned char Image[HAUTEUR][LARGEUR];
unsigned char ImageTrans[HAUTEUR][LARGEUR];
```

Une bibliothèque, utilisable grâce aux directives :

```
#include <ansi_c.h>
#include <X:\Info_S2\ImagesMph.h>
```

offre les deux fonctions :

```
ChargeImage("image.bmp", Image)
```

qui place dans la matrice `Image` contenue dans le fichier `image.bmp`, et

```
AfficheImage(Image,"mon titre", T, G)
```

qui affiche une image dans une fenêtre dont le bord supérieur gauche est à `T` pixels du haut et `G` pixels de la gauche du coin supérieur gauche de l'écran.

Il est donc possible maintenant de créer une image blanche, noire, de passer en négatif une image, de faire des inversions droite-gauche ou haut-bas, d'augmenter la luminosité, le contraste d'une image, de l'accentuer (rendre plus nette), ...