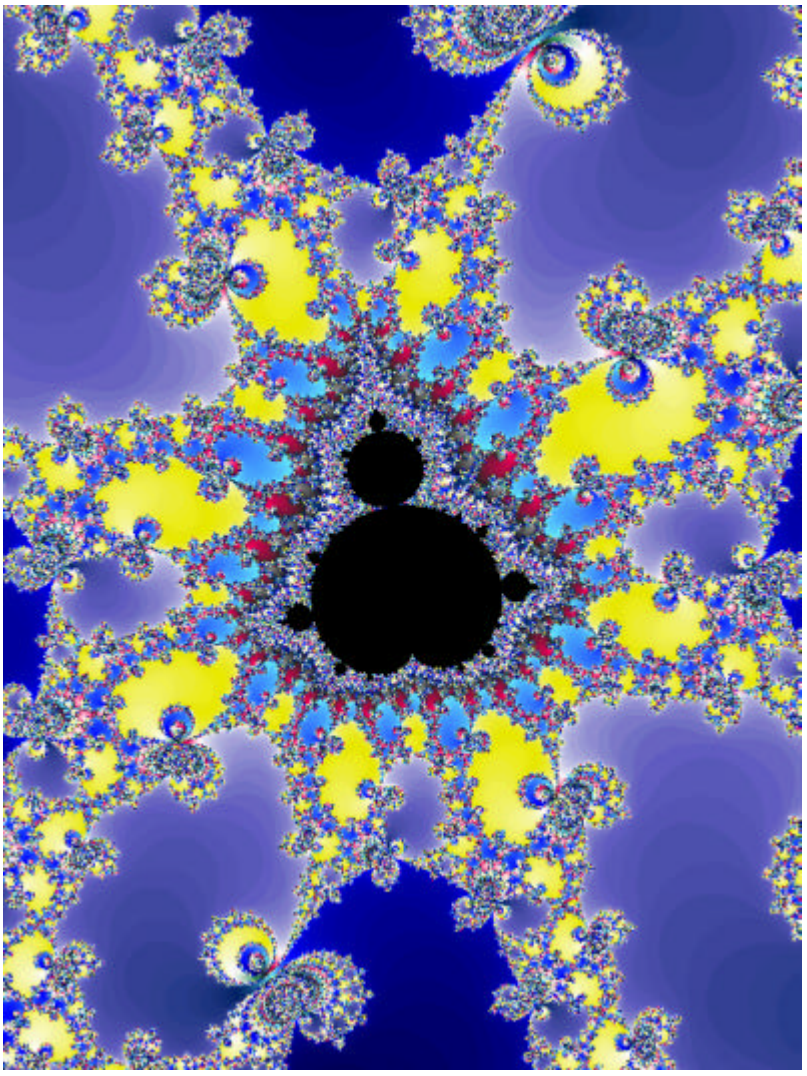
CHAPITRE 19

Librairies



19.1 *L'intérêt des librairies*

La librairie standard de UNIX est actuellement encore largement utilisée. Mais il ne faut pas oublier qu'elle a été prévue pour une utilisation dans un cadre de programmation système. Elle recèle nombre de fonctions de très bas niveau, ce qui, incidemment, en fait un outil très performant dans le cadre de la programmation système ou la programmation de microcontrôleurs. Ainsi des routines couramment utilisées sous UNIX, comme `signal()`, `setjmp()`, `longjmp()`, provoquent une rupture du déroulement normal du programme qui peut poser de sérieux problèmes de récupération de mémoire, en particulier pour des objets crées automatiquement, de manière pas toujours transparente par le compilateur.

En principe, lorsqu'un groupe de travail se met à développer en C, il se met sur pied une librairie de procédures qui va croître au cours du temps et au cours des projets. Cette démarche nécessite une organisation, des règles à respecter absolument. Ces règles sont fonction de la dimension du groupe de travail. Pour de petits groupes largement autonomes (10 personnes), on peut se contenter d'encourager les gens à parler entre eux. Si cette communication ne va pas de soi, souvent en raison de problèmes d'incompatibilité entre les divers développeurs, on peut "organiser" cet échange de vues, sans que cela soit forcément une directive émise par un chef de groupe donné. En fait, une directive fonctionne généralement mieux lorsqu'elle est spontanément appliquée. Une bonne méthode (et de surcroît facile à rendre populaire dans nos contrées) est l'organisation fréquente de verrées ou chacun parle de manière libre de ce qu'il fait.

Pour des groupes de développement plus importants, ou des projets regroupant plusieurs groupes (parfois distribués géographiquement), il n'est plus possible de se contenter d'une communication orale, et il faut recourir à des documents pour échanger les informations concernant les classes produites. La production de documents étant une entreprise longue et fastidieuse, on peut raisonnablement craindre que l'échange d'informations ne se fasse pas aussi bien que l'on pourrait le souhaiter. C'est pourquoi là aussi, il est important de trouver des moyens permettant de favoriser des communications informelles entre les groupes de développement. Ces communications ont de tous temps été précieuses, elles le sont d'autant plus lorsque l'on dispose d'un outil potentiellement efficace pour échanger du code.

La création de modules de librairies, dans ce contexte, doit faire l'objet de soins particuliers par les auteurs de code. Exporter des classes boguées n'apporte que des ennuis, tant à l'auteur qu'aux utilisateurs. Dans le meilleur des cas, on n'utilisera plus les classes développées par un auteur particulier, parce que trop souvent boguées. Dans le pire des cas, l'exportation d'une erreur dans plusieurs projets n'ayant en commun que la réutilisation d'un module particulier peut causer de graves problèmes financiers à une société. Lors du développement d'une classe, il est prudent de partir du principe que l'utilisateur est un ennemi potentiel, contre lequel il convient de se protéger. Si il existe une condition qui peut entraîner une erreur dans le code d'une classe, il est plus que vraisemblable qu'un utilisateur de cette classe déclenchera un jour ou l'autre cette condition. Une classe que l'on désire exporter doit donc impérativement être à l'épreuve de toute manipulation dont on sait qu'elle est dangereuse. Il vaut mieux avorter brutalement l'exécution d'un programme avec un message d'erreur sur la console que de laisser se poursuivre une exécution dans une situation d'erreur.

Le meilleur outil de développement que l'on puisse imaginer, c' est du code fiable, déjà testé

19.2 *Librairie C standard*

La librairie C standard est certainement la plus utilisée des librairies C++, bien qu'elle ne soit pas "orientée objets". Le principal avantage de cette librairie est qu'elle est en partie normalisée par l'ANSI, et de ce fait se retrouve sur toutes les implémentations ANSI du langage. La liste présentée dans le cadre de ce chapitre ne prétend pas être exhaustive, mais présente les principales fonctions de la librairie C telle qu'on la retrouve sur quasiment toutes les implémentations de C et de C++ (à l'exception de certains compilateurs C de bas niveau, utilisés pour la programmation de chips spécialisés, comme des microcontrôleurs et des processeurs de signaux, ou DSP).

Il est important de se souvenir que cette librairie a été conçue indépendemment de C++. Elle peut donc présenter de sérieuses incompatibilités avec certaines constructions plus modernes de C++. En pratique, ces incompatibilités ne sont pas gênantes, dans la mesure où l'on ne mélange pas les fonctions de la librairie C avec les fonctions correspondantes d'une libraire C++, ou pire, avec des instructions réservées de C++. Ainsi, le code suivant produira à coup sûr de sérieuses difficultés à l'utilisation :

```
const     int       nbPoint = 200;
struct    Point
          {
          int  x, y;
          };

Point     *pptr;
pptr = calloc(nbPoint, sizeof(Point));
...
delete pptr;       // suicidaire !
```

19.2.1 *Généralités (#include <stdarg.h>)*

`void va_start(va_list varList, parameter);`
`type va_arg(va_list varList, type);`
`void va_end(va_list varList);`

Ces macros permettent de définir des fonctions avec des listes d'arguments de longueur variable. Elles sont spécialement utilisées pour des fonctions telles que printf() et scanf() entre autres, mais il est également possible de définir des fonctions utilisant un nombre variable d'arguments à des fins personnelles. La fonction suivante effectue la moyenne d'un nombre arbitraire de valeurs réelles **positives**:

```
#include <stdarg.h>
double    mean(float firstArg, ...)

     {
     va_list   argList;
     double    nextArg;
     double    mean;
     int       count;
```

```
          // Indiquer quel est le premier paramètre de la liste,
          // et initialiser les macros :
     va_start(argList, firstArg);
     mean = firstArg;
     next = 0;

          // Une valeur négative sert de terminateur de la liste

     while ((next = va_arg(argList, double)) >= 0)
          {
          mean += next;
          count++;
          }

          // Ne pas oublier va_end !!!

     va_end(argList);
     return (mean / count);
     }

void      main()

     {
     printf("Resultat = %f\n", mean(1.5, 2.3, 4.2, 5, 6, 2.1));
     }
```

Ces macros sont implémentées dans pratiquement toutes les implémentations de C (en tout cas, toutes les implémentations compatibles avec le standard ANSI). Par l'utilisation de classes en C++, il existe toutefois des méthodes autrement puissantes - et autrement plus élégantes- de résoudre ce genre de problèmes, si bien que nous n'insisterons pas plus longtemps sur cette possibilité.

19.2.2 *stdio (`#include <stdio.h>`)*

On connaît déjà `stdio` par le biais de `printf()` et `scanf()`. En principe, on conseille plutôt l'utilisation de iostream pour l'accès à des fichiers en C++. En pratique, `stdio` reste largement utilisé, souvent pour les entrées-sorties sur terminal. De plus, l'utilisation de `stdio` par diverses librairies existantes est intensive, si bien qu'il vaut la peine de s'intéresser quand même à cette librairie.

Dans le cadre de stdio, les conventions suivantes sont utilisées pour définir le modes d'accès aux fichiers :

- "r" : (Read) Le fichier est utilisable en lecture seulement. Pour l'ouvrir, le fichier doit exister au préalable.
- "w" : (Write) Fichier utilisable en écriture seulement. Ouvrir un fichier existant en mode "w" surécrit le contenu précédent de ce fichier. Si le fichier n'existait pas lors de l'ouverture, il sera crée.
- "a" : (Append) Ouverture du fichier en mode ajout. Les données sont appondues à la fin du fichier, mais ce dernier doit exister au moment de l'ouverture.

- “r+” : (Read and Update) Ouverture d’un fichier **existant** en mode lecture / écriture.
- “w+”, “rw” : (Read / write) Ouverture d’un fichier en mode lecture / écriture. Si ce fichier n’existe pas, alors il est crée.
- “a+” : (Append / Create) Comme “a”, mais si le fichier n’existe pas, alors il est crée.

La lettre “b”, ajoutée à une autre indication (comme par exemple “wb+”) indique qu’il s’agit d’un fichier binaire. Utilisée dans le cadre de systèmes comme MVS, VMS, etc.., cette indication est rarement utilisée dans le cadre des systèmes de gestion de fichiers modernes.

Toutes les routines retournent un entier indiquant le résultat -couronné de succès ou non- de l’appel à la fonction concernée. On peut choisir de négliger ce résultat ou non. En règle générale, on testera le résultat de l’ouverture de fichier, et le programmeur choisira fréquemment de négliger les résultats livrés par les autres opérations. Cette manière de faire est courante dans tous les langages, et dangereuse dans tous les langages. Dans le cadre des petits exemples donnés avec les procédures, nous négligerons fréquemment de tester le résultat de l’opération, ceci afin de ne pas surcharger l’exemple. En cas d’erreur, la cause de l’erreur peut être déduite de la variable `errno`. Les valeurs possibles de `errno` sont explicitées dans le fichier `errno.h`.

`FILE *fopen(const char* fileName,`
`    const char* mode);`

Ouverture d’un fichier du nom de **fileName** en mode **mode**. Le résultat de l’appel est un pointeur sur une variable d’un type en principe opaque, qui sera passé ultérieurement aux autres fonctions de manipulation de fichiers. Si l’ouverture échoue, c’est une valeur de pointeur NULL qui sera retournée.

```
FILE* fichier;
fichier = fopen(“porno.gif”, “r”);
if (fichier != NULL)
     {
     ... // Considérations anatomiques ? ...
     }
```

`FILE *freopen(const char* fileName,`
`            const char* mode,`
`            FILE* stream);`

Ferme le fichier `stream` et ouvre en lieu et place le fichier nommé par **fileName** dans le mode **mode**. En cas d’échec de l’ouverture, un pointeur NULL est retourné, mais l’état di fichier ouvert précedemment est indeterminé (généralement, il a été fermé). La valeur de la variable `errno` peut donner une indication quant à ce qu’il s’est effectivement passé.

```
FILE* fichier;
fichier = freopen(“anOtherStdout”, “w”, stdout);
     // redirection de stdout sur un fichier...
if (fichier == NULL)
     {
     printf(“Redirection de stdout manquée pour cause de %d\n”,
               strerror(errno));
```

```
     exit(1);
     }
// else continuer allègrement le travail...
```

int fclose(FILE* stream);

Fermeture du fichier associé au pointeur stream. Rappelons qu'en C, un fichier n'est pas fermé implicitement lors de la destruction du pointeur fichier associé. Retourne 0 si succès, EOF dans le cas contraire. `fclose()` provoque un appel implicite à `fflush()` pour vider le tampon associé à stream.

```
FILE* f = fopen("tmp.tmp", "w");
fprintf(f, "Contenu de fichier bidon\n");
fclose(f);
```

int fflush(FILE *stream)

Provoque l'écriture forcée du tampon associé au fichier stream. En cas de succès, retourne 0, EOF en cas d'échec. Si stream vaut NULL, tous les fichiers ouverts en écriture seront vidés.

```
printf("Ecriture dans le tampon...");
if ((err = fflush(stdout)))
     {
     printf("fflush n'a pas fonctionné pour cause %d\n", errno);
     exit(0);
     }
                    // Provoque l'écriture forcée, malgré
                    // l'absence de \n
```

int remove(const char* name);

Destruction du fichier poratnt le nom **name**. Retourne 0 en cas de succès. Si le fichier portant le nom name était ouvert au moment de l'appel de remove(), le résultat dépend de l'implémentation (en particulier du système d'exploitation), mais généralement, cette situation est signalée comme une erreur.

int rename(const char* oldName, const char* newName);

Permet de renommer le fichier appelé **oldName** en **newName**.Cette opération échoue si le fichier désigné par newName existe déjà (pas vrai pour toutes les implémentations). On trouve parfois également **rename(const char* oldName, int ref, const char* newName**) comme définition. ref indique alors le volume sur lequel doit être effectuée cette opération.

```
FILE* f = fopen("tmp.tmp", "w");
fprintf(f, "Contenu de fichier bidon\n");
fclose(f);
rename("tmp.tmp", "bidon.txt");
f = fopen("bidon.txt", "r");
char buffer[200];
```

```
fscanf(f, “%s”, buffer);
printf(“%s”, buffer);
fclose(f);
remove(“bidon.txt”);
```

int setvbuf(FILE* stream, char *buf, int mode, size_t size);

setvbuf() permet de définir comment les entrées-sorties seront mises en tampon, et quelle sera la taille du tampon. Cet appel doit faire suite immédiatement à l’appel de fopen(), donc avant toute autre utilisation du fichier, sans quoi les résultats peuvent devenir imprévisibles. stream pointe sur le descripteur de fichier pour lequel on définit le tampon, buf, pointe sur un tampon mémoire de taille size, et mode peut prendre l’une des trois valeurs suivantes :

- _IOFBF (Full BuFfering)
- _IOLBF (Line BuFfering)
- _IONBF (No BuFfering)

A noter que beaucoup d’implémentations de librairies offrent la possibilité d’effectuer cet appel implicitement, de manière combinée à l’appel à fopen(), si bien que l’appel explicite n’est plus nécessaire. Cette fonction tend de ce fait à ne plus être utilisée, sauf dans des cas particuliers où l’on désire forcer une dimension de tampon très grande pour des fichiers de très grande taille. Cette fonction retourne 0 en cas de succès. Un appel à cette fonction après une opération sur le fichier (lecture ou écriture) a un résultat indéfini.

void setbuf(FILE* stream, char *buf);

L’appel de cette fonction correspond à appeler setvbuf en mode _IONBF, avec size à 0 si buf est NULL. Si buf n’est pas NULL, il doit avoir au préalable été défini à la librairie par un appel dépendant de l’implémentation. La longueur du buffer, en particulier correspondra à la longueur par défaut du tampon fichier (souvent un multiple d’un bloc disque).

FILE* tmpfile(void);

Crée un fichier temporaire qui sera automatiquement détruit lors de la fermeture de ce fichier, ou à la fin du programme (dans la mesure où le programme peut se terminer normalement, bien sûr). Le fichier sera ouvert en mode “wb+”.

char* tmpnam(char* name);

Génère un nom de fichier correct par rapport au système d’exploitation, et unique dans l’environnement d’exécution. Si **name** est différent de NULL, le nom généré sera écrit dans la chaîne fournie en paramètre. La longueur minimale de cette chaîne dépend de l’implémentation (en particulier du système d’exploitation considéré). Le nombre maximum de noms différents que peut générer `tmpnam()` dépend de l’implémentation, mais devrait dans tous les cas être supérieur à 9999.

Si **name** est NULL, il faut se référer à l’implémentation pour savoir comment gérer le pointeur retourné par `tmpnam()`. Certaines implémentations (la plupart, en fait) utilisent une chaîne de caractères statique qui sera réutilisée à chaque appel de `tmpnam()`, l’ancien conte-

nu se trouvant dés lors surécrit. Plus rarement, une nouvelle chaîne sera générée en mémoire, par un appel à `malloc()`; dans ce cas, il faudra que le programme appelant prenne en charge la destruction de cette chaîne par un appel à `free()`.

`int printf(const char* controlString, ...);`

`printf` écrit une chaîne de caractères formattée sur la sortie standard (`stdout`). La formattage est défini par la chaîne de caractères controlString, alors que les arguments sont contenus dans une liste de longueur indéfinie. Un descripteur de format est indiqué par le caractère % suivi d'un caractère indiquant la conversion à effectuer. Ainsi, la séquence %d indique qu'il faut interpréter l'argument correspondant (défini par sa position dans controlString et dans la liste d'arguments) comme un entier décimal.

En cas de succès, `printf` retourne le nombre de caractères imprimés sur `stdout`, sinon, EOF.

```
printf("La valeur de pi est %7.5f\n", 3.1415926);
```

imprime sur la sortie standard :

```
La valeur de pi est 3.14159
```

Autres références : Voir *Descripteurs de formats de printf()*, page30.

`int scanf(const char* controlString, ...);`

scanf convertit des entrées en provenance de l'entrée standard stdin en une série de valeurs, en se servant de controlString comme modèle de conversion. Les valeurs sont stockées dans les variables sur lesquelles pointe la liste d'arguments indiquée par ... dans le prototype. La chaîne controlString contient des descriptions de format qui indiquent à scanf comment il convient de convertir les entrées de l'utilisateur. Un descripteur de format est indiqué par le caractère % suivi d'un caractère indiquant la conversion à effectuer. Ainsi, la séquence %d indique qu'il faut interpréter les entrées sur stdion comme un entier décimal.

scanf retourne le nombre d'arguments auxquels il a été possible d'assigner une valeur. Ce nombre peut être différent du nombre d'arguments passé à scanf! Si il n'est pas possible de lire sur stdin, EOF sera retourné.

```
int      x;
printf("Entrer une valeur entière: ");
scanf("%d", &x);
printf("Vous avez entré %d\n", x);
```

Autres références : Voir *Descripteurs de formats de printf()*, page30.

`int fprintf(FILE* f, const char* controlString, ...);`

Fonctionnalité équivalente à `printf`(). De fait, `printf`() peut également s'écrire

```
fprintf(stdout, “Valeur de pi = %f\n”, 3.1415926);
```

int fscanf(FILE *f, const char* controlString, ...)

Fonctinnalité équivalente à `scanf`(). De fait, `scanf`() peut également s’écrire

```
float unReel;
print(“Entrer un nombre reel “);
fscanf(stdout, “%f”, unReel);
```

int sprintf(char *buffer, const char* controlString, ...);

Fonctionnalité équivalente à `fprintf`(), mais la sortie se fait dans une chaîne de caractères en lieu et place d’un fichier. L’utilisateur devra veiller lui-même à réserver une place suffisante dans la chaîne référencée par *buffer.

int fgetc(FILE *stream);

Lecture du prochain caractère dans le fichier associé au pointeur stream. Le fait que le résultat soit entier (au lieu de char) est dû à la possibilité de livrer EOF (de type int) en cas d’erreur. Pratiquement, du fait de la conversion implicite int <-> char, ceci n’a pas vraiment d’importance.

Valeurs de retour :

- le prochain caractère si l’opération est couronnée de succès.
- EOF (négatif, en général) si il y a une erreur

char* fgets(char* str, int nbChars, FILE *stream);

Lecture d’une chaîne de caractères du fichier associé au pointeur **stream** dans la zone mémoire pointée par **str**. str doit pointer sur une zone mémoire de dimension au moins égale à nbChars + 1 (+ 1 en raison du caractère \0 qui termine une chaîne de caractères en C). C’est au programmeur de s’assurer que la place qu’il a reservée est suffisante.

La fonction `fgets()` lit des caractères jusqu’à un caractère de fin de ligne, ou jusqu’à la lecture de nbChars caractères.

Valeurs de retour :

- **str** si l’opération est couronnée de succès.
- NULL si il y a une erreur.

int fputc(int c, FILE* stream);

Ecriture du caractère c (donné sous forme d’entier) dans le fichier associé au pointeur stream. Retourne la valeur entière de c en cas de succès, EOF autrement. On ne peut donc pas sans autre écrire EOF au moyen de `fputc()`.

```
// Copie d'un fichier dans un autre :

FILE* source;
FILE* copy;
int x;
if ((source = fopen("source.dat", "r")) == NULL)
     {
     printf("Erreur à l'ouverture de source.dat, %s\n", strerror(errno));
     exit(1);
     }
if ((source = fopen("copy.dat", "w")) == NULL)
     {
     printf("Erreur à l'ouverture de copy.dat, %s\n", strerror(errno));
     exit(1);
     }
while ((x = fgetc(source)) != EOF) fputc(x);
fclose(source); fclose(copy);
```

`int fputs(const char* str, FILE* stream);`

Ecrit la chaîne de caractères pointée par str dans le fichier pointé par stream. Retourne 0 en cas de succès, EOF en cas d'échec.

`int getc(FILE* stream);`
`int putc(int c, FILE* stream);`

Implémentations sous forme de macros de `fgetc()` et `fputc()`. Comportement en principe similaire.

`int puts(const char* str);`

Identique à `fputs()`, mais écriture sur stdout.

`char* gets(char* str);`

Identique à `fgets()`, mais lecture à partir de stdin.

`int getchar(void)`

Retourne la valeur du prochain caractère, sous forme d'un entier, lu sur stdin. Retourne EOF en cas d'erreur.

`int putchar(int car);`

Ecrit le caractère **car** (donné sous forme d'entier) sur stdout. Retourne car en cas de succès, EOF en cas d'erreur.

`int ungetc(int c, FILE* stream);`

Réinsère le caractère c dans le fichier associé au pointeur stream. Retourne c en cas de

succès, EOF en cas d'échec.

size_t fread(void* buffer, size_t sizeOfItem, size_t count, FILE* stream);

Lecture de **count** éléments de taille **sizeOfItem** du fichier associé au pointeur **stream** dans le tampon **buffer**. La fonction retourne le nombre d'éléments lus en cas de succès, ou EOF en cas d'erreur.

size_t fwrite(const void *buffer, size_t sizeOfItem, size_t count, FILE* stream);

Ecriture de **count** éléments de taille **sizeOfItem** du tampon **buffer** dans le fichier associé au pointeur **stream**. La fonction retourne le nombre d'éléments écrits en cas de succès, ou EOF en cas d'erreur.

```
#include <stdio.h>

typedef struct
     {
     int a;
     char b;
     } Foo;

void main ()

     {
     FILE *fp;
     int i;
     Foo inbuf[3];
     Foo outbuf[3] = { { 1, 'a' }, { 2, 'b' }, { 3, 'c' } };

     fp = fopen( "MyFooFile", "wb+");
     fwrite( outbuf, sizeof(Foo), 3, fp );
     rewind( fp );

     fread( inbuf, sizeof(Foo), 3, fp );

     for ( i=0; i<3; i++ )
          printf( "foo[%d] = { %d %c }\n",
                    i, inbuf[i].a, inbuf[i].b );

     fclose( fp );
     }
```

int fgetpos(FILE* stream, fpos_t* position);

Cette fonction livre dans la variable pointée par **position** la position dans le fichier associé au pointeur **stream**. Le type `fpos_t` est un type opaque, qui ne devrait être utilisé qu'en

conjonction avec `fsetpos()`. Cette fonction retourne 0 en cas d'exécution correcte.

`int fsetpos(FILE* stream, fpos_t* position);`

Permet de se repositionner dans le fichier associé au pointeur **stream** à la position **position**. position doit impérativement avoir été livré par `fgetpos()`, sinon le programme peut ne plus être portable. Retourne 0 en cas de succès.

```
#include <stdio.h>

void     main()

     {
     FILE *fp;
     fpos_t pos;
     char s[80];

     fp = fopen( "MyFile", "w+" );

     fgetpos( fp, &pos );

     fputs( "Hello world.", fp );

     fsetpos( fp, &pos );

     fgets( s, 80, fp );
     printf( "You read: %s\n", s );
     fclose( fp );
     }
```

`long ftell(FILE* stream);`

Retourne la position courante dans le fichier associé au pointeur **stream**. Contrairement à `fgetpos`, `ftell` est limité à la taille d'un long pour exprimer la position courante du fichier, et ne convient de ce fait pas aux très grands fichiers. `ftell` retourne -1L en cas d'échec, et ajuste errno à la valeur représentant la cause de l'erreur.

`int fseek(FILE* stream, long offset, int how);`

Permet de se positionner dans le fichier associé au pointeur **stream.**La position est exprimée par la valeur **offset**, qui sera interprétée en fonction du paramètre**how**. how peut prendre les trois valeurs suivantes :

- SEEK_SET : position doit être interprété par rapport au début du fichier.
- SEEK_CUR : position doit être interprété comme un déplacement relatif à la position courante dans le fichier.
- SEEK_END : position doit être interprété comme un déplacement relatif à la fin du fichier.

Le système d'exploitation sous-jacent peut imposer des restrictions à `fseek`, en particulier lorsqu'on l'applique à des fichiers texte. Il est recommandé de jeter un coup d'oeil à la documentation livrée avec l'environnement de développement en cas de doute.

```
#include <stdio.h>

void main()

     {
     FILE *fp;
     long int pos;
     char s[80];

     fp = fopen( "MyFile", "w+" );

     pos = ftell( fp );

     fputs( "Hello world.", fp );

     fseek( fp, pos, SEEK_SET );

     fgets( s, 80, fp );
     printf( "You read: %s\n", s );
     fclose( fp );
     }
```

void rewind(FILE* fstream);

Equivalent à `fseek(stream, 0L, SEEK_SET);`

void clearerr(FILE* stream);

Efface une condition d'erreur préalablement mise sur le fichier associé au pointeur **stream**.

int feof(FILE *stream);

Teste si le fichier associé au pointeur **stream** est positionné en fin de fichier, c'est-à-dire si une opération d'entrée-sortie précédente a causé une erreur. `feof()` retourne 0 si la position actuelle dans le fichier n'est pas la fin du fichier.

```
#include <stdio.h>

void main()
     {
     FILE *fp;
     char c;

     fp = fopen( "MyFile", "w+" );
     c = fgetc( fp );

     if ( feof(fp) )
          printf( "At the end of the file.\n" );
     else
          printf( "You read %c.\n", c );

     fclose( fp );
```

```
}
```

int ferror(FILE* stream);

Retourne le code d'erreur pour le fichier associé au pointeur stream, si une erreur a eu lieu. Le code d'erreur vaut 0 si aucune erreur n'a eu lieu. Le code d'erreur est remis automatiquement à zéro par un appel à `rewind()`, ou explicitement par un appel à `clearerr()`.

void perror(const char* str);

Permet d'afficher sur stderr le message d'erreur en clair associé à la valeur actuelle de errno. On peut ajouter la chaîne de caractères pointée par str à ce message. errno n'est pas utilisé que pour les erreurs d'entrée-sorties, comme le montre l'exemple suivant :

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

void main()
     {
     double i, result;

     i = -4.0;
     result = sqrt(i);

     if ( (result==0) & (errno!=0) )
          perror( "Invalid argument for sqrt()" );
     else
          printf( "The square root of %f is %f\n", i, result );
     }
```

19.2.3 *stdlib (#include <stdlib.h>)*

La plus utilisée (implicitement ou explicitement) des librairies C. Attention! Certaines des entrées de cette librairie peuvent entrer en conflit avec des instructions spécifiques à C++. Il s'agit en particulier des instructions permettant de réserver de la mémoire, comme `malloc(), calloc()`, `realloc()`, `free()`.

void* malloc(size_t size);

malloc() permet l'allocation dynamique de mémoire en cours d'exécution du programme. En cas de succès de l'allocation, il retourne un pointeur sur le bloc mémoire reservé, et en cas d'échec, il retourne la valeur NULL. La dimension du bloc à réserver est donnée en paramètre. Le type `size_t` dépend de l'implémentation: le plus souvent, `size_t` correspond à long.

```
struct Complex     {
                   double real, imag;
                   };
```

```
Complex* cptr;
     // Réserver de la place pour un tableau de complexes
int       size;
printf(“Dimension du tableau ? “);
scanf(“%d”, &size);
cptr = malloc(size*sizeof(Complex));
if (cptr == NULL)
     {
     perror(“Allocation mémoire refusée“);
     exit(1);
     }
```

`malloc()` n’effectue aucune initialisation de la zone mémoire reservée.

En C++, il faut absolument préférer le mot réservé **`new`**. `malloc()` n’utilise pas les constructeurs de classes.

`void* calloc(size_t num, size_t size);`

`calloc()` permet de réserver de la place mémoire pour un tableau de `num` éléments ayant chacun une taille `size`. En cas de succès de l’allocation, il retourne un pointeur sur le bloc mémoire reservé, et en cas d’échec, il retourne la valeur NULL. L’exemple précédent peut se récrire, avec calloc(), de la manière suivante :

```
struct Complex     {
                   double real, imag;
                   };

Complex* cptr;
     // Réserver de la place pour un tableau de complexes
int       size;
printf(“Dimension du tableau ? “);
scanf(“%d”, &size);
cptr = calloc(size, sizeof(Complex));
if (cptr == NULL)
     {
     perror(“Allocation mémoire refusée“);
     exit(1);
     }
```

Contrairement à `malloc()`, `calloc()` effectue une initialisation de la zone mémoire reservée. Cette initialisation consiste à mettre tous les bits de la zone mémoire considérée à zéro.

En C++, il faut absolument préférer le mot réservé **`new`**. `malloc()` n’utilise pas les constructeurs de classes.

`void* realloc(void *ptr, size_t size);`

`realloc()` permet de modifier la taille d’un bloc mémoire réservé préalablement au moyen de `calloc()` ou `malloc()`. Pour ce faire, le bloc considéré, pointé par `ptr`, est

recopié en un autre endroit de la mémoire, dont la taille correspond à `size`. En cas de succès de l'allocation, il retourne un pointeur sur le bloc mémoire reservé, et en cas d'échec, il retourne la valeur NULL.

Si `ptr` vaut NULL lors de l'appel de `realloc()`, cet appel a le même effet que `malloc()`. Si `ptr` n'est pas NULL, et que `size` vaut 0, alors cet appel est équivalent à `free()`. Lors de l'augmentation de la taille d'une zone mémoire, les bits nouvellement alloués ne sont pas initialisés. Lors de la diminution de taille mémoire, les bits excédentaires sont perdus. Le changement de taille du tableau reservé dans l'exemple précédent pourrait s'écrire:

```
int      newSize;
printf("Nouvelle dimension du tableau ? ");
scanf("%d", &newSize);
cptr = realloc(cptr, sizeof(Complex) * newSize);
if (cptr == NULL)
     {
     perror("Allocation mémoire refusée");
     exit(1);
     }
```

Attention! il faut se souvenir que, contrairement à `calloc()`, `realloc()` n'effectue aucune initialisation! En C++, il faut absolument préférer le mot réservé **`new`**. `realloc()` n'utilise pas les constructeurs de classes, ni les opérateurs de copie.

`void free(void* ptr);`

`free()` permet de libérer la place mémoire réservée par `malloc()`, `calloc()` ou `realloc()`. Le pointeur passé à `free()` comme argument doit **impérativement** avoir été réservé par l'une de ces instructions, sans quoi les résultats de l'appel sont imprévisibles (le plus probablement, un "crash" du programme).

Passer un pointeur NULL à `free()` n'a pas d'effet.

```
char*     str = malloc(200 * sizeof(char));
...
free(str);
```

`int abs(int i);`

Valeur absolue de i.

`div_t div(int numerateur, int denominateur);`

Livre la partie entière et le reste de la division de numerateur/denominateur.

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

void main ()
```

```
     {
     int n = 32452, d = 787;
     div_t r;

     r = div(n,d);

     printf( "%d / %d vaut %d\n", n, d, r.quot );
     printf( "et le reste vaut %d.\n", r.rem);
     }
```

`long labs(long j);`

Valeur absolue pour des `long`.

`int rand(void);`

Retourne une valeur aléatoire uniformément distribuée entre 0 et RAND_MAX. Ne convient pas pour des applications nécessitant des nombres à caractéristiques statistiques de bonne qualité.

`void srand(unsigned int seed);`

Permet d'initialiser le générateur de `rand` à une valeur prédeterminée.

`void* bsearch(const void *key,`
`const void* array,`
`size_t count, size_t size,`
`int (*comparison)(const void *key, const void *data));`

Recherche dichotomique (*binary search*) de l'élément **key** dans le tableau **array**. Le tableau comporte **count** éléments, chacun de taille **size**. L'utilisateur fournit la fonction permettant de comparer deux éléments; cette fonction doit retourner un résultat < 0 si key est plus petit que l'élément considéré, 0 si key est égal à l'élément, et > 0 si key est plus grand que l'élément comparé.

Le tableau **array** doit avoir préalablement été trié par ordre ascendant, de manière à permettre la recherche dichotomique. On pourra utiliser `qsort()` pour ce faire.

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

int compare( const void *s1, const void *s2 )
     {
     return strcmp( s1, s2 );
     }

int Findname( char *name )
     {
     char *result;
     static char infoProfs[][15] =
          { "Breguet", "Evequoz", "Guerid", "Molliet", "Pesenti",
```

```
               "Roethlisberger" };

     result = bsearch( name, infoProfs, 6,
                                    sizeof(char[15]), compare );
     return ( result != NULL );
     }

void main()
     {
     char name[80];

     printf( "Votre nom SVP ? " );
     scanf( "%s", name );
     printf("%s ", name);

     if ( Findname(name) )
          printf( "est");
     else
          printf( "n'est pas");
     printf(" un prof d'informatique à l'EINEV\n");
     }
```

void qsort(void* array,
size_t count, size_t size,
int (*comparison)(const void *key, const void *data));

Réalise le tri en place d'un tableau **array** comportant **count** éléments de taille **size**. L'utilisateur fournit lui-même la fonction de comparaison; cette fonction doit retourner un résultat < 0 si key est plus petit que l'élément considéré, 0 si key est égal à l'élément, et > 0 si key est plus grand que l'élément comparé.

L'algorithme de tri utilisé est celui dû à R. Sedgewick, appelé **quicksort**. Il consiste à partitionner le tableau à trier en tableaux partiels que l'on triera individuellement.

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

const int ASIZE = 10;
const int AMAX = 100;

int compare( const void *n1, const void *n2 )
     {
     return ( *((int *) n1) - *((int *) n2) );
     }

void main()
     {
     int a[ASIZE], i;

     srand( (unsigned int) clock() );

     // Initialisation d'un tableau d'entiers
     // avec des nombres aléatoires
     for ( i=0; i<ASIZE; i++ )
```

```
          a[i] = rand() % AMAX;

     printf( "Avant tri par qsort():\n\t" );
     for ( i=0; i<ASIZE; i++ )
          printf( "%2d ", a[i] );
     putchar('\n');

     qsort( a, ASIZE, sizeof(int), compare );

     printf( "Après tri par qsort():\n\t" );
     for ( i=0; i<ASIZE; i++ )
          printf( "%2d ", a[i] );
     putchar('\n');
     }
```

double atof(void char* str);

Conversion d'une chaîne de caractères en une valeur de type double.

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

void main()
     {
     char *s1 = "3.141593";
     char *s2 = "123.45E+676COUCOU!";
     double result;

     result = atof( s1 );
     printf( "s1 vaut %G.\n", result );

     result = atof( s2 );
     printf( "s2 vaut %G.\n", result );
     }
```

Résultat :

```
s1 vaut 3.141593
s2 vaut 1.2345E+676
```

int atoi(const char* str);
long int atol(const char* str);

Conversions de chaînes de caractères en un entier ou en un long entier, respectivement.

double strtod(const char *str, char **end);

Convertit **str** en une valeur en double précision. La conversion saute des blancs initiaux, débute dès qu'un nombre est rencontré, et s'arrête dès qu'un caractère non traduisible est trouvé. C'est l'adresse de ce caractère qui sera retournée dans **end**. end peut être réutilisé pour des appels successifs dans le cas de conversions en chaîne.

`atof(str)` est équivalent à `strtod(str, NULL)`. Une conversion impossible livre 0 comme valeur de retour, et `errno` est mis à `E_RANGE`. Si une conversion dépasse les limites de la machine, la valeur `HUGE_VAL` est retournée (voir <`limits.h`>).

`long int strtol(const char* str, char** end, int base);`
`unsigned long strtoul(const char* str, char** end, int base);`

Conversion d'une chaîne de caractères en une valeur entière longue, respectivement longue non signée. Obéit au mêmes principes que strtod. En plus, il est possible de spécifier une base de conversion comprise entre 2 et 36. Si la base vaut 0, la fonction va chercher à déterminer la base du nombre au moyen du préfixe.

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

main()
{
     char *s1 = "3723682357boo!";
     char *s2 = "0xFACES";
     unsigned long int result;
     char *end;

     result = strtoul( s1, &end, 10 );
     printf( "s1 vaut %lu, avec '%s' laissé non converti.\n", result, end );

     result = strtoul( s2, &end, 0 );
     printf( "s2 vaut %lu, avec '%s' laissé non converti.\n", result, end );
}
```

19.2.4 *math (`#include <math.h>`)*

Cette librairie ne pose pas de problèmes particuliers à l'utilisation. Il faut toutefois se rappeler que la librairie math travaille en double précision, et que l'utilisation en conjonction avec des variables en virgule flottante peut amener certains compilateurs à générer des avertissements, dans le cas où la conversion automatique n'est pas "sûre".

`double sin(double x);`
`double tan(double x);`
`double cos(double x);`

Fonctions trigonométriques sinus, cosinus et tangente.

`double acos(double x);`
`double asin(double x);`
`double atan(double x);`

arc cosinus, arc sinus et arc tangente.

double atan2(double x, double y);

arc tangente de y/x.

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

void main()
     {
     double x, y, result;

     printf( "Enter x: " );
     scanf( "%lf", &x );

     printf( "Enter y: " );
     scanf( "%lf", &y );

     result = atan2(x,y);

     printf( "atan2( %G/%G ) = %G\n",  x, y, result);
     }
```

double sinh(double x);
double cosh(double x);
double tanh(double x);

sinus hyperbolique, cosinus hyperbolique et tangente hyperbolique.

double exp(double x);
double log(double x);
double log10(double x);
double sqrt(double x);

exponentielle, logarithme naturel, logarithme en base 10, et racine carrée. Des erreurs éventuelles (log(-4)) seront signalées par le biais de errno.

double fabs(double x);

Valeur absolue.

double floor(double x);
double ceil(double x);

Arrondi entier par troncature, arrondi entier par excès. Noter que l'entier résultant est retourné sous forme d'un double.

double fmod(double x, double y);

Livre le reste de la division de x par y. errno est mis à EDOM si y == 0.

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

void main()
     {
     double x, y, result;

     printf( "Enter x: " );
     scanf( "%lf", &x );

     printf( "Enter y: " );
     scanf( "%lf", &y );

     result = fmod( x, y );

     printf( "The remainder of %G/%G is %G.\n",
               x, y, result );
     }
```

double ldexp(double x, int exposant);
double frexp(double x, int *exposant);

ldexp() etourne $x \cdot 2^{exposant}$, alors que frexp() décompose un nombre en ses deux parties x et exposant. Ces deux fonctions sont inverses l'une de l'autre.

double modf(double x, double* intpart);

Retourne la partie décimale de x, et dans la variable pointés par **intpart**, la partie entière.

double pow(double x, double y);

Retourne x^y.

19.2.5 *ctype (#include <ctype.h>)*

Dans de nombreuses implémentations de librairis, les fonctions de ctype, permettant la manipulation de caractères, sont implémentées sous forme de macro. Il faut souligner que ces fonctions doivent être préférées à un test direct basé sur un code ASCII, qui pourrait ne pas être utilisé sur une machine utilisant un système d'exploitation différent.

Les caractères sont généralement traités comme des entiers par cette librairie.

int isalnum(int c);

Retourne TRUE (vrai) si c est un caractère alphanumérique (lettre ou chiffre).

int isalpha(int c);

Retourne TRUE (vrai) si c est un caractère alphabétique (lettre majuscule ou minuscule).

int iscntrl(int c);

Retourne TRUE (vrai) si c est un caractère de contrôle (exemple, ESCAPE).

int isdigit(int c);

Retourne TRUE (vrai) si c est un chiffre.

int isgraph(int c);

Retourne TRUE (vrai) si c est affichable, et différent de SPACE.

int islower(int c);

Retourne TRUE (vrai) si c est une lettre minuscule.

int isprint(int c);

Retourne TRUE (vrai) si c est un caractère affichable ou un espace (SPACE).

int ispunct(int c);

Retourne TRUE (vrai) si c est un caractère différent d'une lettre, d'un chiffre, ou d'un caractère de contrôle.

int isspace(int c);

Retourne TRUE (vrai) si c est un espace (SPACE), un tabulateur, une fin de ligne, un tabulateur vertical (interligne), un saut de page.

int isupper(int c);

Retourne TRUE (vrai) si c est une lettre majuscule.

int isxdigit(int c);

Retourne TRUE (vrai) si c est un chiffre héxadécimal.

int tolower(int c);

Retourne le résultat de la conversion de c en minuscule. Si c n'est pas une lettre majuscule, il sera retourné inchangé.

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

```
void main()
     {
     char uc, lc;

     printf("Entrer une majuscule: ");
     uc = getchar();

     lc = tolower( uc );

     if ( uc != lc )
          printf( "Minuscule de '%c' est '%c'.\n", uc, lc );
     else
          printf("Pas de conversion en minuscule possible pour '%c'.\n",
                         uc );
     }
```

`int toupper(int c);`

Retourne le résultat de la conversion de c en majuscule. Si c n'est pas une lettre minuscule, il sera retourné inchangé.

19.2.6 *string (`#include <string.h>`)*

`string` est un module très apprécié des programmeurs en C et C++. Il faut toutefois se méfier de l'utilisation incontrôlée de `string` en C++, puisque ce module fait usage, dans certains cas, de `malloc()` et `free()`. En pratique, tant que le programmeur n'utilise que les primitives de `string`, et ne les mélange pas avec les instrauctions correspondantes de C++, il n'y a pas de problèmes particuliers à utiliser `string`, sinon que certaines librairies C++ implémentent les mêmes fonctionnalités de manière plus confortable.

`void *memcpy(void* dest, const void* source, size_t size);`

Copie de **size** bytes de la zone pointée par **source** vers la zone pointée par **dest**. Attention, si les deux zones mémoires se chevauchent, il peut y avoir destruction de données.

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

typedef struct {
     int a;
     int b;
     } Foo;

void main()
     {
     Foo foo1 = { 123, 456 };
     Foo foo2 = { 987, 321 };

     printf( "Before memcpy():\n"
                    " foo1 = %d, %d   foo2 = %d, %d\n",
                    foo1.a, foo1.b, foo2.a, foo2.b );
```

```
    memcpy( &foo1, &foo2, sizeof(Foo) );

    printf( "After memcpy():\n"
                  " foo1 = %d, %d   foo2 = %d, %d\n",
                  foo1.a, foo1.b, foo2.a, foo2.b );
    }
```

void* memmove(void* dest, const void* source, size_t size);

Comme `memcpy()`, mais la fonction est executée correctement même si les zones mémoire se chevauchent.

char* strcpy(char* dest, const char* source);

Comme `memcpy()`, mais pour le cas particulier où les zones mémoires sont occupées par des chaînes de caractères. Attention à s'assurer que la place reservée dans dest est bien suffisante pour stocker la longueur de source!

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

void main()
    {
    char *source = "Au revoir";
    char dest[25] = "Bonjour";

    printf( "Avant strcpy(): dest = \"%s\"\n", dest );

    strcpy( dest, source );

    printf( "Apres strcpy():  dest = \"%s\"\n", dest );
    }
```

char* strncpy(char* dest, const char* source, size_t size);

Comme `memcpy()`, mais pour le cas particulier où les zones mémoires sont occupées par des chaînes de caractères. La copie s'arrête soit après copie du caractère de fin de chaîne ('\0'), soit après copie de size caractères.

char* strcat(char* dest, const char* source);

Concaténation de **dest** et de **source**, et dépose du résultat dans **dest**. Attention à s'assurer que la place reservée dans dest est bien suffisante pour stocker la longueur de dest + source!

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

void main()
    {
```

```
     char s1[25] = "hello ";
     char *s2 = "world";

     printf( "Avant strcat(): s1 = \"%s\"\n", s1 );
     strcat( s1, s2 );
     printf( "Apres strcat():  s1 = \"%s\"\n", s1 );
     }
```

char* strncat(char* dest, const char* source, size_t size);

Concaténation de **dest** et de **source**, et dépose du résultat dans dest. La concaténation s'arrête au plus tard après la dépose de **size** caractères dans dest.

int memcmp(const void* m1, const void* m2, size_t size);

Compare **size** éléments entre les deux zones mémoire indiquées. Retourne 0 si elles sont égales, > 0 si m1 est plus grand que m2, < 0 si m1 est inférieur à m2. L'opération s'arrête aussi tôt que le résultat est connu.

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

typedef struct {
     int a;
     int b;
     } Foo;

void main()
     {
     Foo foo1, foo2;

     printf( "For each structure, enter  two integers \n"
                  "separated by a space, like: 123 987\n\n" );

     printf( "Enter two numbers for foo1: ");
     scanf( "%d %d", &foo1.a, &foo1.b );
     printf( "Enter two numbers for foo2: ");
     scanf( "%d %d", &foo2.a, &foo2.b );

     if ( memcmp(&foo1, &foo2, sizeof(Foo)) == 0 )
          printf( "foo1 and foo2 are the same.\n" );
     else
          printf( "foo1 and foo2 are different.\n" );
     }
```

int strcmp(const char* s1, const char* s2);
int strcoll(const char* s1, const char* s2);

Comme `memcmp()`, mais pour des chaînes de caractères. `strcoll()` respecte en plus les conditions locales d'utilisation, qui peuvent inclure certaines règles particulières à un langage donné.

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

```
#include <string.h>

main()
{
     char s1[80], s2[80];
     int result;

     printf( "Première chaîne (s1): ");
     scanf( "%s", s1 );

     printf( "Seconde chaîne (s2): ");
     scanf( "%s", s2 );

     result = strcmp( s1, s2 );

     if ( result > 0 )
          printf( "s1 est plus grand que s2.\n" );
     else if ( result < 0 )
          printf( "s1 est inférieur à s2.\n" );
     else
          printf( "s1 et s2 sont identiques.\n" );
}
```

`int strncmp(const char* s1, const char* s2, size_t size);`

Comme `memcmp()`, mais pour des chaînes de caractères. La comparaison s'arrête après un maximum de size caractères, et au plus tôt lorsque le résultat est connu.

`void* memchr(const void* str, int c, size_t size);`

Retourne un pointeur sur le premier caractère identique à **c** trouvé dans les **size** premiers caractères de la chaîne **str**. Si c n'st pas trouvé, `memchr()` retourne NULL.

`char* strchr(const char* str, int c);`

Retourne un pointeur sur le premier caractère identique à **c** trouvé dans la chaîne **str**. Si c n'st pas trouvé, `strchr()` retourne NULL. Identique à `memchr()` pour des chaînes terminées par '\0'.

19.2.7 *signal (`#include <signal.h>`)*

Ces groupes de fonctions permettent de gérer des évènements asynchrones au programme. Il ne faut pas confondre ce mécanisme avec un traitement d'exceptions tel qu'on le trouve en C++ ou en ADA : il s'agit plutôt de communications élémentaires entre processus asynchrones, voire de traitement d'interruptions ou d'erreurs de la librairie.

Dans les implémentations minimales de C, les signaux disponibles reflètent les limitations du système d'exploitation, et se limitent à des évènements internes au programme (erreurs mathématiques, time-outs, etc...). Dans le cadre d'implémentations UNIX, par contre,

les signaux disponibles sont beaucoup plus riches; en revanche, l'utilisation de cette richesse tend à rendre les programmes difficiles à porter.

```
void (*signal(int sig, void(*func)(int)))(int);
```

Détermine la fonction à appeler lorsque le signal **sig** sera émis.

```
int raise(int sig);
```

Provoque l'émission du signal **sig**.

19.3 *iostream (#include <iostream.h>)*

`iostream` est une librairie «standard» de C++. Cette librairie définit les flots d'entrée-sortie, et sert à remplacer la librairie `stdio` de C. Un flot est un canal (recevant, d'entrée ou fournissant, de sortie) que l'on peut associer à un périphérique, à un fichier, ou à une zone mémoire.

Un flot recevant ou d'entrée est un **`istream`**, (ifstream dans le cas d'un fichier) alors qu'un flot de sortie est un **`ostream`** (ofstream dans le cas d'un fichier). Logiquement, un flot d'entrée-sortie se déduit par héritage multiple d'un `istream` et d'un `ostream` pour former un **`iostream`**. Il existe des flots prédéfinis :

- **`cout`** est le flot de sortie standard, connecté à la sortie standard `stdout`.
- **`cin`** est le flot d'entrée standard, qui est connecté à l'entrée standard `stdin`.

19.3.1 *ostream*

ostream redéfinit l'opérateur << sous la forme d'une fonction membre :

```
ostream& operator << (expression)
```

L'expression correspondant au deuxième opérande peut être d'un type standard de base quelconque, y compris `char` ou `char*`. Si on veut utiliser cet opérateur pour des types définis par l'utilisateur, il faudra le redéfinir dans le cadre de la déclaration du type concerné, généralement sous forme d'une fonction `friend`. Il n'est pas possible de redéfinir cet opérateur comme fonction membre, à cause de l'ordre d'interprétation des opérandes.

```
class     X
     {
     friend ostream& operator << (ostream&, const X&);
               // opérandes dans l'ordre correct
     ...
     };
```

L'exemple çi-dessous n'est pas faux, mais est quasi inutilisable, car les opérandes vont se trouver inversés :

```
class     X
     {
     public :
          ostream& operator << (ostream&);
               // les opérandes sont dans le mauvais ordre !!!
     };
```

Parmi les nombreuses fonctions membres, citons :

```
ostream&  put(char c);
ostream&  write(void* adressOfBuffer, long bufferSize);
```

19.3.2 *istream*

istream redéfinit l'opérateur >> de la manière suivante :

```
istream&  operator >> (& base_type);
```

base_type peut être un type de base quelconque, sauf un pointeur (char* est accepté, néanmoins, pour la lecture de chaînes de caractères). >> est compatible avec scanf, en ce sens que les «espaces» (espace, tabulateur, fin de ligne, saut de page, etc...) sont interprétés comme des délimiteurs. Pour redéfinir l'opérateur dans le cas d'un type utilisateur, utiliser la même technique que pour ostream :

```
class     X
     {
     friend istream& operator >> (istream&, const X&);
               // opérandes dans l'ordre correct
     ...
     };

istream&  get(char& c);
int       get(); // returns value of char at input, or EOF
istream&  read(void* bufferAdress, int bufferSize);
     // si moins de bufferSize caractères ont pu être lus, la valeur
     // effectivement lue du nombre de caractères peut être obtenue par :
int       gcount();
```

19.3.3 *iostream*

Dérivée publiquement de istream et ostream, cette classe permet les entrées-sorties conversationnelles, par exemple pour la lecture/écriture sur un fichier.

19.3.4 *Etat d'un flot*

istream et ostream dérivent de la classe ios, qui définit les constantes suivantes :

```
eofbit : fin de fichier
failbit : la prochaine opération sur le flot ne pourra pas aboutir
badbit : le flot est dans un état irrécupérable
goodbit : aucune des erreurs précédentes
```

L'accès aux nits d'erreur peut se faire par l'une des fonctions membres suivantes :

```
int  eof() : valeur de eofbit
int  bad() : valeur de badbit
int  fail() : valeur de failbit
```

```
int  good() : vrai (== 1) si aucun bit d'erreur n'est activé
int  rdstate() : valeur du mot d'état complet
```

Le mot d'état peut être modifié par la méthode suivante :

```
void      clear(int newState = 0);
```

Le mot d'état prend la valeur newState. Pour activer un bit (par exemple badbit sur le flot fl) on utilisera quelque chose comme :

```
fl.clear(ios::badbit | fl.rdstate());
```

Lorsque l'on lit un fichier, il arrive fréquemment que l'on écrive une boucle du type :

```
char      buffer[1001];
int       sze;
while (!f.eof())
     {
     // read the next chunk of data and process it
     f.read(buffer, 1000);
     for (int i = 0; i < f.gcount(); i++) { .... }
     }
```

Il faut se souvenir qu'à la fin de cette boucle, fl.rdstate() est différent de zéro. Aucune opération sur ce flot n'est plus tolérée. En particulier, si il s'agit d'un fichier à accès aléatoire, où l'on peut librement positionner le pointeur de lecture dans le fichier, il ne sera plus possible de le positionner à la sortie de la boucle, à moins de remettre à zéro le mot d'état à l'aide de :

```
f.clear();
```

A chaque flot est associé un état de formattage, constitué d'un mot d'état et de 3 valeurs numériques associées, correspondant au gabarit, à la précision et au caractère de remplissage.

TABLEAU 2. Signification du mot d'état de formatage

Nom du champ	Nom du bit	Signification
ios::basefield	ios::dec	conversion décimale
	ios::oct	conversion octale
	ios::hex	conversion hexadécimale
	ios::showbase	affichage de l'indication de base
	ios::showpoint	affichage du point décimal
ios::floatfield	ios::scientific	notation dite scientifique
	ios::fixed	notation en virgule fixe

Pour influencer le formatage, on peut utiliser soit des «manipulateurs», soit des fonctions membres. Les manipulateurs peuvent être «paramétriques» ou «simples».

Les manipulateurs simples s'utilisent sous la forme :

```
flot << manipulateur
flot >> manipulateur
```

Les principaux manipulateurs simples sont représentés Tableau3, page319.

TABLEAU 3. Principaux manipulateurs simples

MANIPULATEUR	UTILISATION	ACTION
`dec`	Entrée/Sortie	Active le bit de conversion décimale
`hex`	Entrée/Sortie	Active le bit de conversion hexadécimale
`oct`	Entrée/Sortie	Active le bit de conversion octale
`endl`	Sortie	Insère un saut de ligne et vide le tampon de sortie
`ends`	Sortie	Insère un caractère de fin de chaîne (\0)

Les manipulateurs paramétriques s'utilisent sous la forme

```
istream&  manipulateur(argument);
ostream&  manipulateur(argument);
```

Les principaux manipulateurs paramétriques sont représentés Tableau4, page319

TABLEAU 4. Principaux manipulateurs paramétriques

MANIPULATEUR	UTILISATION	ACTION
`setbase(int)`	Entrée/Sortie	Définit la base de conversion
`setprecision(int)`	Entrée/Sortie	Définit la précision de sortie des nombres en virgule flottante
`setw(int)`	Entrée/Sortie	Définit le gabarit. Attention! il retombe à zéro après chaque conversion !

19.3.5 *Association d'un flot à un fichier*

Dérivés de ostream, respectivement de istream, les classes **`ofstream`** et **`ifstream`** permettent la création de flots de sortie ou d'entrée associés à des fichiers :

```
ofstream  of(char *fileName, char openMode=ios:out);
ifstream  if(char *fileName, char openMode=ios:in);
```

Le pointeur d'écriture (respectivement de lecture) peut être manipulé au moyen de

```
seekp(int displacement, int relativeTo);
```

respectivement

```
seekg(int  displacement, int relativeTo);
```

Par défaut, relativeTo prend la valeur ios::begin, c'est-à-dire que le déplacement se fera par rapport au début du fichier. Les autres valeurs possibles sont ios::current (déplacement relatif à la position actuelle) et ios::end (déplacement par rapport à la fin du fichier.

Fermer un ofstream ou un ifstream requiert dans tous les cas l'utilisation de `close()`; noter que détruire une instance de flot d'entrée-sortie (`delete`) ne détruit pas le fichier associé, mais généralement le ferme (au cas où il était resté ouvert). Dans tous les cas, il vaut mieux fermer explicitement un flot associé à un fichier lorsque ce dernier n'est plus utilisé.

L'utilisation des classes ifstream et ofstream requiert l'inclusion de `fstream.h`; fstream.h incluant iostream.h, il n'est en principe pas nécessaire de l'inclure à nouveau explicitement. Le fichier peut être ouvert dans différents modes, selon la valeur du paramètre openMode, comme décrit dans le Tableau5, page320 :

TABLEAU 5. Modes d'ouverture d'un fichier

Bit de mode d'ouverture	Action
`ios::in`	Ouverture en lecture (obligatoire pour ifstream)
`ios::out`	Ouverture en écriture (obligatoire pour ofstream)
`ios::app`	Ouverture en mode ajout (ofstream seulement)
`ios::ate`	Place le pointeur fichier en fin de fichier après ouverture
`ios::trunc`	Si le fichier existe, son contenu est détruit après ouverture
`ios::nocreate`	Le fichier doit exister à l'ouverture (ne doit pas être crée)
`ios::noreplace`	Le fichier ne doit pas exister à l'ouverture (sauf si ios::ate ou ios::app est spécifié)

Dérivant de ifstream et ofstream, un flot de type **`fstream`** peut être associé à un fichier utilisable simultanément en lecture et en écriture.

19.4 *L++ (Standard Components)*

L++ est une tentative de correction de l'erreur commise par les concepteurs de C++, qui n'ont pas défini de librairie standard C++ (à l'exception de iostream). L++ contient des fonctions très générales, universellement applicables. L++ est livré en standard avec les compilateurs conformes à la spécification Cfront de USL, mais n'est actuellement pas prévu pour la standardisation ANSI. L++ se trouve sur la grande majorité des compilateurs UNIX, mais n'est généralement pas inclus dans les versions pour ordinateurs personnels (Symantec, Microsoft ou Borland). C'est fort regrettable, car L++ propose un ensemble de fonctions très puissant et universellement applicable, à l'inverse des librairies proposées sur PC, qui ne concernent souvent que des portione limitées de l'interface utilisateur spécifique à la machine utilisée.

Parmi les classes proposées par L++, citons :

- **Bit :** Permet la définition et la manipulation de chaînes de bits de longueur arbitraire
- **Block :** Gestion de zones mémoires
- **Graph :** Structure de graphes (éléments reliés par des réseaux arbitraires)
- **List :** Listes généralisées.
- **Map :** Gestion de tableaux à adressage multiple, en particulier avec adressage relatif au contenu.
- **Pool :** Gestion de mémoire à usage général, avec réallocation dynamique de bloca devenus libres.
- **Regexp :** Manipulation d'expressions régulières.
- **Set :** Manipulation d'ensembles.
- **Stopwatch :** Chronomètre.
- **String :** Gestion de chaînes de caractères dynamiques. Utilise Pool.
- **Symbol :** Gestion de tables de symboles. Utilise Pool.
- **Time :** Manipulation et contrôle de données associées aux temps.
- **bag :** Container d'information généralisé. Il est possible de faire des opérations d'entrée-sortie sur des bags.
- **g2++ :** Routines d'encodage et décodage de données. g++ permet la transmission d'information entre ordinateurs différents sans précautions relativement à la manière d'encodage propre à chaque machine. Ainsi, pour transférer un entier dans sa représentation binaire entre un PowerPC et un CPU Intel, il est normalement nécessaire de tenir compte de la représentation différente des entiers dans le stack du PowerPC et du chip Intel. L'utilisation de g2++ permet d'éviter cet inconvénient en introduisant la notion de syntaxe de transfert, un peu à la manière de ASN.1 dans le monde OSI, ou IDL / XDR dans le monde RPC.

19.5 *Quelques autres librairies*

Il existe de nombreuses autres librairies, certaines dépendantes du système d'exploitation, d'autres indépendantes. Certaines ne sont que des librairies C avec un interface C++, d'autres sont des librairies natives. Il serait inutile et vain de chercher à faire un listing exhaustif. Nous nous bornerons à citer quelques-unes des plus intéressantes :

Motif++ : Version C++ de Motif 1.2. Domaine public. Devrait être bientôt remplacée par Motif2.0, qui devrait comporter un interface C++ en mode natif.

Interviews : Librairie originellement développée à Stanford, est actuellement implémentée sur plusieurs plate-formes, y compris Motif et Macintosh. Emule Motif sans faire explicitement appel à Motif.

Microsoft Foundation Classes : Librairie d'accès à Windows. Dans sa dernière moutute, avec Visual C++ 4.0, est devenue multi-plateformes. Il est possible de générer du code pour Alpha, Intel, PowerPC et Macintosh, avec une librairie native.

Think C Library : librairie fournie avec le compilateur Think C (Symantec C++) pour le Macintosh. La librairie n'est pas écrite en C++, mais dispose d'un interface C++.

DCE++ : Actuellement en domaine semi-public, DCE++ pourrait être standardisée dans un proche avenir. DCE++ facilite l'utilisation de DCE (*Distributed Computing Environment*) en C++. L'accès à des services distribués comme DTS (*Distributed Time Service*), RPC (*Remote Procedure Call*), NIS (*Network Information Service*) ou DNS (*Domain Name Service*) est encapsulé dans des classes facilement utilisables. DCE++ est distribué par HaL computers (http://hal.com).

GCOOL : Domaine public. Implémente des classes universellement utilisables, un peu à la manière de L++.

Pour les intéressés, il existe sur Internet une rubrique (en fait, il en existe plusieurs qui se référencent les unes les autres) rassemblant les librairies C++, tant du domaine public que commerciales. Cette rubrique se nomme **FAQ : available C++ libraries** et peut être consultée par ftp ou par http (World Wide Web). La mise à jour se fait environ une fois par mois, et est raisonnablement exhaustive. Comme beaucoup de rubriques sur Internet, celle-ci dépend entièrement de la disponibilité de son auteur bénévole. Actuellement (octobre 1995) la rubrique comprend 50 pages et va de librairies mathématiques (LINPACK++, FFT++) à des classes couvrant la biologie et l'astrophysique, en passant par de très intéressantes librairies de classes génériques et de classes utilisables pour la génération d'interfaces utilisateur. La plupart de ces classes sont développées pour UNIX, mais la disponibilité du code source pour beaucoup d'entre elles les rend utilisables aussi dans d'autres environnements (Windows 95 et Windows NT entre autres).

Il est également possible d'utiliser des sites Internet spécialisés (www.apple.com, www.microsoft.com, www.hal.com, www.att.com, www.borland.com, etc...) pour se renseigner sur la disponibilité de librairies. Il est de toutes façons recommnadé, avant d'entreprendre

un nouveau projet, de se renseigner sur les librairies disponibles. Même si, pour des raisons de licences ou de copyright, ou simplement d'erreurs, elles se révèlent inutilisables, il est très utile de connaître d'autres solutions à un même problème : dans l'optique de la réutilisation de logiciels, même les erreurs (?) des autres peuvent être mises à profit.