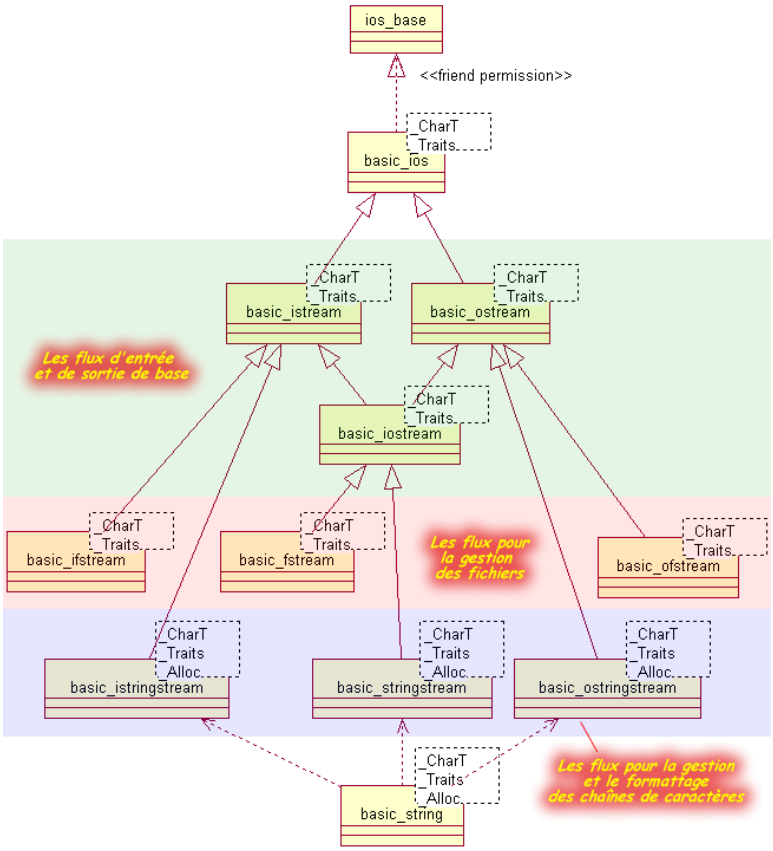
Depuis le temps, nous connaissons bien les objets « *cin* » et « *cout* ». Du moins nous semblons les connaître. Cette étude nous permettra de découvrir un certain nombre de méthodes associées qui peuvent rendre de grands services. Par ailleurs, nous allons étendre plus généralement nos connaissances sur les classes qui représentent l'ensemble des flots. Ainsi, nous pourrons travailler aussi bien sur les entrées/sorties standards que sur des fichiers ou que sur les flots en mémoire pour la gestion et le formatage des chaînes de caractères. Nous pourrons même modifier le comportement standard de ces flots afin de permettre à nos propres classes de pouvoir s'intégrer aux classes représentatives de ces flots.

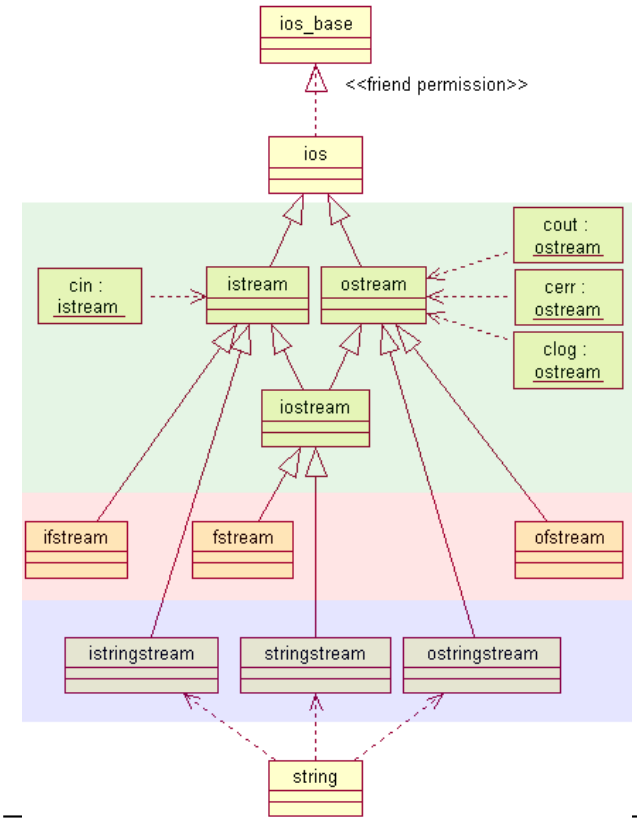
Hiérarchie des classes représentant les flots

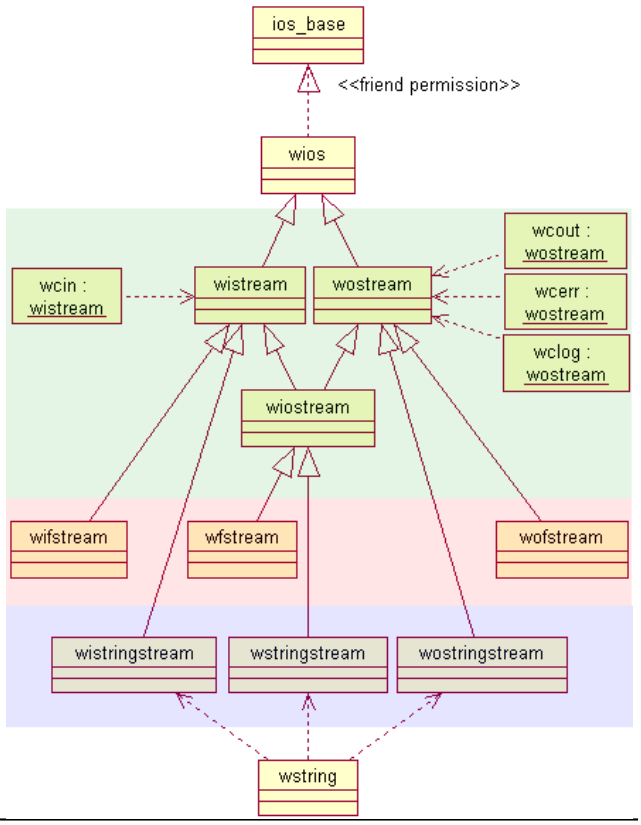
Il existe un certain nombre de classes qui s'occupent de la gestion des flots. De plus, chaque classe est spécialisée afin de répondre parfaitement à l'adéquation recherchée. Ainsi, vous avez des classes réservées pour la gestion des fichiers, d'autres pour le traitement des flots en mémoire sous forme de chaînes de caractères, etc. par ailleurs, ces classes ne sont pas disposées n'importe comment, mais elles font toutes parties d'une même famille et profitent pleinement du polymorphisme dont les conséquences seront pleinement justifiées lorsque nous aborderons la fin de notre étude. Vous en avez une vue dans le diagramme UML de droite.



En réalité, vous remarquez qu'il s'agit de classes génériques. En effet, les flux font maintenant partis de la STL. Maintenant, il est possible de pouvoir travailler sur deux types de caractères prédéfinis : « *char* » et « *wchar_t* ». Nous connaissons bien le premier type. Le deuxième type de caractères concerne les caractères larges (ou étendus) codés sur 16 bits et qui permettent de représenter l'ensemble des langues connues dans le monde. Ainsi, nous pourrons manipuler les caractères Japonais, Hébreux, Arabes, Grecs, etc. Nous pourrions même, en passant par la redéfinition des classes de type « *_Traits* » fabriquer de nouveaux symboles.

A partir de cette hiérarchie de classes génériques, nous aboutissons aux deux véritables hiérarchies (les classes génériques sont là pour fabriquer d'autres classes) suivant le type de caractères que nous choisissons. Ci-dessous, vous avez les deux scenarii possibles.





La première hiérarchie sera celle que nous utiliserons dans la quasi-totalité de nos applications. En fait, il est très facile de s'y retrouver, il suffit d'enlever le préfixe « *basic_* » et vous obtenez automatiquement le nom des classes correspondantes. Remarquez au passage, les objets « *cin* » et « *cout* » associés respectivement aux classes « *istream* » et « *ostream* ». Nous voyons réapparaître la classe « *string* » qui est en fait issue de la classe générique « *basic_string* ». En effet, il existe bien les deux types de chaîne de caractères.

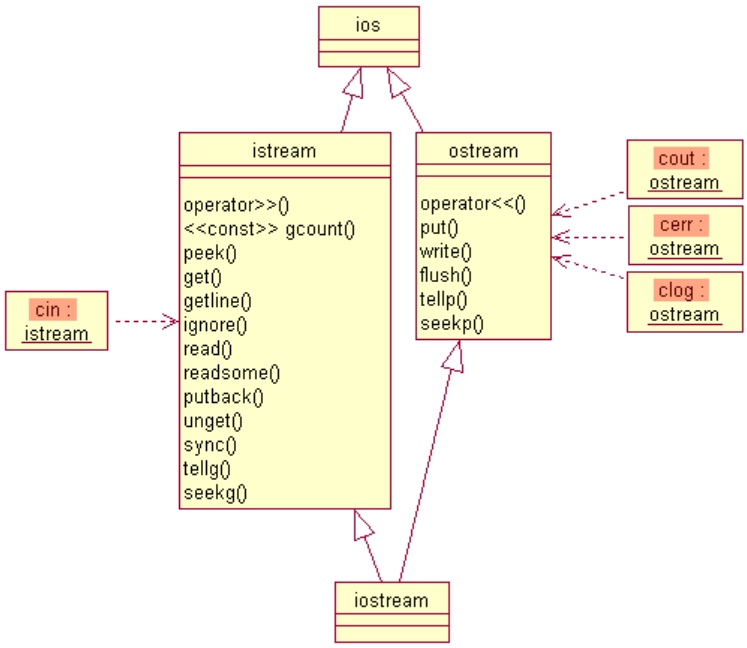
Pour la deuxième hiérarchie, c'est-à-dire celle qui utilise les caractères étendus « *wchar_t* », le nom des classes et des objets se différencie de son équivalent « *char* » par le préfixe « *w* ». Nous retrouvons également les objets associés aux entrées sorties standard qui cette fois-ci se nomment respectivement « *wcin* » et « *wcout* ».

Pour l'instant, toutes ces classes sont visualisées sans leurs attributs et leurs méthodes afin d'éviter une trop grande lourdeur dans le graphisme. Bien entendu, suivant les sujets développés, vous aurez une représentation plus complète des classes concernées.

Les flots d'entrée/sortie – Les flots standard

D'une manière générale, un flot peut être considéré comme un « *canal* » qui permet une communication avec un périphérique ou un fichier ou même une partie de la mémoire (formatée comme une structure de fichier). Il existe trois types de flots :

- Recevant de l'information – *flot de sortie*.
- Fournissant de l'information – *flot d'entrée*.
- Recevant et fournissant de l'information – *flot d'entrée et de sortie*.



Toutes les opérations d'entrées et de sorties sont fournies par les classes « *istream* » (flot d'entrée), « *ostream* » (flot de sortie) et « *iostream* » (classe dérivée des deux premières qui permet donc la bidirectionnalité). Ainsi, l'étude que nous ferons sur ces trois classes particulières sera également utile pour toutes les autres qui dérivent de celles-ci.

Il existe quatre objets prédéfinis qui permet de gérer les flux standard, savoir :

- *cin* : objet de la classe « *istream* » représentant l'entrée standard. En général, *cin* permet de lire les données depuis le terminal de l'ordinateur, c'est-à-dire le clavier.
- *cout* : objet de la classe « *ostream* » représentant la sortie standard. En général, *cout* permet d'écrire des données pour le terminal de l'ordinateur, c'est-à-dire l'écran.
- *cerr* : objet également de la classe « *ostream* » représentant les erreurs standard. *cerr* est l'emplacement vers lequel diriger les messages d'erreur du programme.
- *clog* : objet supplémentaire qui gère les erreurs qui est donc similaire à *cerr*. Il dispose en plus d'un tampon intermédiaire.

Les classes « *istream* » et « *ostream* » sont prépondérantes puisque c'est à partir de ces classes que nous pouvons réellement communiquer. Elles disposent d'ailleurs d'un certain nombre de méthodes que nous allons détailler. Elles ont tellement d'importances qu'il a été décidé d'utiliser les opérateurs de redirection pour permettre une utilisation rapide et pratique. Le sens de redirection n'est pas choisi au hasard. En effet, le sens proposé indique la direction de l'envoi des données. Ainsi, dans le cas général :

- ***>> x ;*** // *insère des données dans x.*
- ***<< x ;*** // *extrait des données de x.*

En prenant l'exemple des flots que nous connaissons :

- ***cin >> x ;*** // *insère les données saisies au clavier dans x.*
- ***cout << x ;*** // *extrait des données de x et l'affiche à l'écran.*

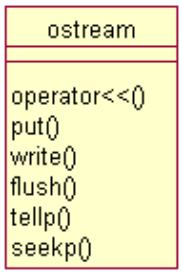
Dans le modèle UML, les deux opérateurs ne sont écrits qu'une seule fois, alors qu'ils sont surdéfinis pour permettre la communication avec différents types de variables. Ainsi, nous serons capable de saisir et d'afficher aussi bien des entiers que des réels, des chaînes de caractères, etc. En fait, tous les types primitifs ainsi que ceux faisant parti de la STL pourront être utilisés directement par les flots. Seuls les types définis par l'utilisateur ne sont pas intégrés. Il sera alors nécessaire de proposer de nouvelles surdéfinitions pour élargir les compétences de ces flots, ce que nous ferons à l'issu du dernier chapitre.

Définition d'un flot (ou flux)

Un flot (ou un flux) est traduit en anglais par « *stream* ». C'est d'ailleurs pour cela qu'une bonne partie des classes de la hiérarchie possède au moins ce mot. Prenons l'exemple de « *cin* » pour bien comprendre.

Lorsque nous écrivons « ***cin>>x ;*** », *x* récupère la valeur issue du clavier, mais pas tout de suite. Il faut attendre que l'opérateur ait fini de tout saisir pour pouvoir récupérer l'ensemble de l'information. Ainsi, lorsque nous saisissons le nombre *1235*, il faut taper successivement chacun des chiffres, ce qui prend un certain temps, et ce n'est qu'au moment ou nous appuyons sur la touche « *Entrée* » que la communication a lieu.

Dans ce principe, nous voyons, qu'il est nécessaire d'avoir une mémoire intermédiaire qui stocke momentanément les chiffres déjà tapés. Cette mémoire est appelée « *mémoire tampon* » ou plus simplement « *tampon* ». Avec cette mémoire, le temps de communication est toujours le plus bref possible afin de supprimer tout aléa de fonctionnement. De plus, les informations du tampon non exploitées lors d'une lecture restent disponibles pour la lecture suivante.



Conclusion

Ainsi, à cette notion de flot (*stream*) est toujours rattachée la notion de tampon, mais également le fait d'envoyer des informations en séquences les unes derrières les autres, un petit peu comme le courrant d'une rivière.

La classe « *ostream* »

Nous connaissons bien l'opérateur de redirection que nous avons utilisé bien souvent. Toutefois, il existe quelques méthodes supplémentaires dans cette classe « *ostream* », et je vous propose de découvrir les plus intéressantes.

1. ***ostream& operator<< (type) ;***

   Transfère une information d'un type prédéfini sur le flot de sortie. Comme l'opérateur renvoie un « *ostream* », il est possible de proposer un enchaînement d'opérations.

   ```
   int x = 5 ;
   double y =-6.3 ;
   cout << x << y ; // envoie la valeur 5 et la valeur -6.3  à l'écran.
   ```

2. ***ostream& put (char) ;***

   Cette méthode transmet un seul caractère dans le flot donné par l'argument. Cette méthode est souvent associée à la méthode *get* de « *istream* ». Comme cette méthode renvoie un « *ostream* », il est possible de proposer un enchaînement d'appels successifs au même titre que l'opérateur « << ».

   ```
   cout.put('A') ;  // envoie le caractère 'A' à l'écran, équivalent à : cout << 'A' ;
   cout.put('A').put('B').put('C') ; // envoie les trois caractères « ABC » à l'écran.
   ```

3. ***ostream& write (const char*, int);***

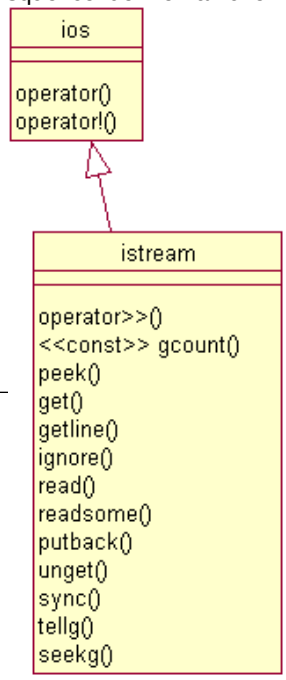
   Cette méthode fournit une alternative à l'opérateur « << » pour transmettre un tableau de caractères. Elle transmet en sortie une certaine longueur de caractères (quels que soient ces caractères). La méthode *write* ne fait donc pas intervenir de caractères de fin de chaîne ; si un tel caractère apparaît dans la longueur prévue, il sera transmis, comme les autres, au flot de sortie. Son comportement est donc totalement différent de l'opérateur « << » puisque ce dernier affiche justement tous les caractères jusqu'à ce qu'il rencontre le caractère de fin de chaîne.

   ```
   char message[] = « Bonjour » ;
   cout.write(message, 4) ; // affiche les quatre premiers caractères de message à l'écran.
   ```

   Cette méthode n'est pas très utile pour un écran, mais elle le deviendra lorsque nous traiterons des informations brutes sans aucun formatage particulier (informations binaires) directement dans un fichier.

4. ***ostream& flush () ;***

   Cette méthode vide explicitement la mémoire tampon.



La classe « *istream* »

Il est possible de tester un flot en le considérant comme une valeur logique (vrai ou faux). Pour ce faire, les opérateur « *()* » et « *!* » ont été redéfinis au niveau de la classe « *ios* ». Ainsi, lorsque nous écrivons :

```
(flot) // ou flot représente n'importe qu'elle classe de la hiérarchie des flots.
```

Le résultat est :

- *true* : si aucun des bits d'erreur n'est activé.
- *false* : dans le cas contraire.

Lorsque nous écrivons :

!flot ou (!flot) // ou flot représente n'importe qu'elle classe de la hiérarchie des flots.

Le résultat est :

- *false* : si aucun des bits d'erreur n'est activé.
- *true* : dans le cas contraire.

Nous allons utiliser cette caractéristique dans la classe « *istream* » puisqu'elle hérite de tout le comportement de la classe « *ios* ».

1. ***istream& operator>> (type) ;***

   Son rôle consiste à extraire du flot concerné les caractères nécessaires pour former une valeur du type voulu en réalisant une opération inverse du formatage opéré par l'opérateur « << »

   ```
   int x ;
   double y ;
   cin >> x >> y ; // saisie à partir du clavier et transfert des valeurs vers les variables x et y.
   ```

   Dans l'exemple ci-dessus, il est nécessaire de se servir de séparateurs pour faire la différence entre la valeur prévue pour *x* et celle prévu pour *y*. Les espaces blancs servent de délimiteurs, savoir : espace « ' ' », tabulation horizontale « \t », tabulation verticale « \v », fin de ligne « \n » et changement de page « \f ».Du coup, les délimiteurs ne peuvent pas être lus en tant que caractères. Ainsi, après la déclaration suivante :

   ```
   char message[50] ;
   ```

   et lorsque que l'opérateur saisie au clavier la chaîne suivante : « *bonjour à tout le monde* », seule la valeur « *bonjour* » est récupérée dans la variable *message* puisque l'espace est considéré comme un délimiteur. Il faudra donc utiliser une autre méthode pour récupérer la totalité de la chaîne saisie, notamment la méthode *getline*. Soit l'écriture suivante :

   ```
   vector<int> ivec ;
   int ival ;
   while (cin >> ival)   ivec.push_back(ival) ;
   ```

   L'expression : *while (cin >> ival)* lit une séquence de valeurs depuis l'entrée standard jusqu'à ce que « *cin* » soit évalué à *false*. Deux conditions générales sont à l'origine de l'évaluation de « *istream* » à *false* :
   a) la lecture d'une fin de fichier (auquel cas, toutes les valeurs contenues dans le fichier ont été correctement lues),
   b) ou la détection d'une valeur invalide – tel *3.14159* (cette valeur est un réel, et c'est un entier qui est attendu). Dans le cas de la lecture d'une valeur invalide, l'objet « *cin* » est placé en état d'erreur et la lecture des valeurs s'interrompt (Ce sujet sera traité ultérieurement).

2. ***istream& ignore (int compteur = 1, int délimiteur = EOF) ;***

   Lorsque nous faisons une saisie à l'aide de l'objet « *cin* », nous validons cette saisie grâce à la touche « *Entrée* » du clavier. Lorsque cette validation est effectuée, la valeur stockée dans la mémoire tampon est envoyée vers la variable correspondant en conservant toutefois dans le tampon le caractère de validation. Du coup, lorsque nous réalisons une nouvelle saisie, le tampon possède toujours ce caractère. Ainsi le système ne s'arrête pas pour demander la nouvelle valeur à l'utilisateur. Finalement, il est impossible de réaliser la saisie proposée.

   La méthode *ignore* permet d'enlever ce délimiteur avant de réaliser une nouvelle saisie.

   ```
   cin >> message ;
   cin.ignore();
   ```

3. ***istream& get (char&) ;***

   Permet d'extraire un caractère d'un flot d'entrée et de le ranger dans la variable passée en argument de la méthode. Contrairement à l'opérateur « >> », la méthode *get* peut lire n'importe quel caractère, délimiteur compris.

   ```
   char car;
   while (cin.get(car))   cout.put(car) ;
   ```

4. ***int get () ;***

   La méthode *get* est surdéfinie et cette deuxième version lit également un caractère unique du flux d'entrée. La différence est qu'elle retourne cette valeur, et non l'objet de la classe « *istream* » auquel elle s'applique. Elle renvoie un type *int* plutôt que *char*, car elle retourne également la représentation de la fin du fichier, souvent représenté par *-1* afin de la distinguer des valeurs du jeu de caractères. Pour savoir si la valeur renvoyée est une fin de fichier, on la compare à la

constante « *EOF* » définie dans l'en-tête *iostream*. La variable déclarée pour stocker la valeur renvoyée par *get* devra déclarée avec un type *int*, de façon à contenir à la fois les valeurs des caractères ou bien la valeur « *EOF* ».

```
int car;
while ((car = cin.get()) != EOF)   cout.put(car) ;
```

5. ***istream& getline (char *chaîne, int taille, char délimiteur = '\n') ;***

Cette méthode facilite la lecture des chaînes de caractères (non « *string* »), ou plus généralement d'une suite de caractères quelconques (espace ou caractères de contrôle compris), terminé par un caractère qui sert de délimiteur et qui n'est donc pas utiliser par la chaîne à récupérer.

Cette méthode doit donc être utiliser à la place de l'opérateur de redirection « **>>** » pour saisir des chaînes comportant plusieurs mots (séparés par des espaces) ou même tout un texte écrit sur plusieurs lignes, auquel cas, il ne faut pas prendre le caractère proposé par défaut « **\n** ».

Cette méthode lit des caractères sur le flot l'ayant appelé et les places dans l'emplacement désigné par chaîne. Elle s'interrompt lorsqu'une des deux conditions suivantes est respectée :

a. le caractère qui sert de délimiteur a été trouvé : dans ce cas ce caractère n'est pas recopié en mémoire ;
b. *taille-1* caractères ont été lus.

Dans tous les cas, cette méthode ajoute le caractère nul de terminaison de chaîne à la suite des caractères lus.

```
char chaine[50];
cin.getline(chaine, 50) ;  cout << chaine ;
```

6. ***int gcount () ;***

Cette méthode fournit le nombre de caractères effectivement lus lors du dernier appel de *getline* (ou de *read*). Ni le caractère délimiteur, ni celui placé à la fin de la chaîne, ne sont comptés ; autrement dit, *gcount* fournit la longueur effective de la chaîne rangée en mémoire par *getline*.

7. ***istream& read (char *chaîne, int taille) ;***

Cette méthode permet de lire sur le flot d'entrée considéré une suite de caractères (octets) en spécifiant la longueur voulue.

```
char chaine[20];
cin.read(chaine, 5) ;  // récupère uniquement 5 caractères du tampon du clavier même si il en possède plus.
```

Ici encore cette méthode peut sembler faire double emploi, soit avec la lecture d'une chaîne avec l'opérateur « **>>** », soit avec la méthode *getline*. Toutefois, *read* ne nécessite ni séparateur de fin de chaîne, ni délimiteur particulier. Cette méthode est plus couramment utilisée, comme la méthode *write*, lorsque nous souhaiterons accéder à des fichiers sous forme binaire, c'est-à-dire en recopiant en mémoire les informations telles qu'elles figurent dans le fichier.

8. ***istream& putback (char) ;***

Cette méthode renvoie à la fin du flot concerné le caractère spécifié.

9. ***istream& unget () ;***

Saute le caractère courant du tampon pour passer au suivant. Cette méthode est utile lorsque nous consultons une information caractère par caractère. Cette méthode permet d'éviter de lire et de prendre un caractère pour passer au suivant.

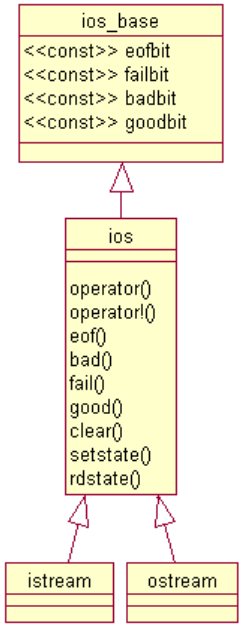
10. ***int peek () ;***

Cette méthode fournit le prochain caractère disponible sur le flot concerné, mais sans l'extraire du flot. Il sera donc de nouveau disponible lors d'une prochaine lecture sur le flot.

Statut d'erreur d'un flot - les classes « *ios* » et « *ios_base* »

A chaque flot d'entrée ou de sortie est associée un ensemble de bits d'un entier (donné par la classe ancêtre « *ios_base* »), formant ce que l'on appelle le statut d'erreur du flot. Il permet de rendre compte du bon ou du mauvais déroulement des opérations sur le flot. Analysons cette situation :

```
int ival ;
cin >> ival ;
```



En reprenant l'exemple ci-dessus, et si nous effectuons la saisie d'une chaîne de caractères tel que « *bonjour* » alors qu'un entier est attendu, « *cin* » est alors placé en état d'erreur. Si nous avions saisi un nombre entier, la lecture aurait donc réussi et « *cin* » aurait conservé son état normal.

Nous allons décrire plus précisément cet ensemble de bit. Cet ensemble est composé de trois bits particuliers codés sur un même entier. Chaque bit est représenté par une constante définie dans la classe « *ios_base* » :

1. ***eofbit*** **:** ce bit est activé si la fin de fichier est atteinte, autrement dit, si le flot correspondant n'a plus aucun caractères disponibles.

2. ***failbit*** **:** ce bit est activé lorsque la prochaine opération d'entrée-sortie ne peut aboutir. Ce bit est validé lorsque nous tentons d'ouvrir un fichier inexistant ou lorsque que le format d'entrée est inadapté (comme pour l'exemple précédent).

3. ***badbit*** **:** ce bit est activé lorsque le flot est dans un état irrécupérable. Cela peut se produire lorsque nous tentons de nous positionner au-delà de la fin du fichier.

La différence entre ***badbit*** et ***failbit*** n'existe que pour les flots d'entrée. Lorsque ***failbit*** est activé, aucune information n'a réellement été perdue sur le flot ; il n'en va plus de même pour ***badbit***.

4. ***goodbit*** **:** il existe une constante qui est la concaténation de l'ensemble des trois premiers bits. Cette constante (valant en fait ***0***) correspond à la valeur que doit avoir le statut d'erreur lorsque aucun de ses bits n'est activé.

Nous pouvons dire qu'une opération d'entrée-sortie a réussi lorsque l'un des bits ***goodbit*** ou ***eofbit*** est activé. De même, nous pouvons dire que la prochaine opération d'entrée-sortie ne pourra aboutir que si ***goodbit*** est activé.

Un objet flux conserve un jeu de signaux conditionnels permettant de surveiller l'état véritable de ce flux. Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que :

- la condition d'erreur n'a pas été corrigée (ce qui va de soi !),
- le bit d'erreur correspondant n'a pas été remis à zéro.

Différentes actions concernant les bits d'erreur

La classe de base des flux « *ios_base* » gère donc l'ensemble des bits d'erreur, tandis que la classe dérivée « *ios* » dispose d'un certain nombre de méthodes qui permettent de consulter ces bits d'erreur ou d'autres pour les modifier explicitement. Les méthodes qui permettent de connaître l'état du statut d'erreur du flot sont les suivantes :

1. ***bool eof()*** **:** cette méthode donne directement l'état du bit ***eofbit***

2. ***bool fail()*** **:** cette méthode donne directement l'état du bit ***failbit***

3. ***bool bad()*** **:** cette méthode donne directement l'état du bit ***badbit***

4. ***bool good()*** **:** cette méthode fournit la valeur *true* si aucune des trois méthodes précédentes n'a la valeur *true*, c'est-à-dire si aucun des bits du statut d'erreur n'est activé.

5. ***iostate rdstate()*** **:** cette méthode retourne directement l'entier (*iostate* correspond en fait à un *typedef*) représentant le statut d'erreur du flot.

Les deux méthodes suivantes permettent de positionner les bits d'erreur :

1. ***void clear(iostate état = goodbit)*** **:** cette méthode positionne un ou plusieurs bit. Par défaut, c'est ***goodbit*** qui est proposé, ce qui permet implicitement de placer tous les autres bits à zéro. Vous pouvez placer plusieurs bits d'état, il faut alors utiliser les opérateurs de traitement binaire, en l'occurrence le ou logique « **|** ».

   cin.clear(ios_base ::badbit | ios_base ::failbit) ;

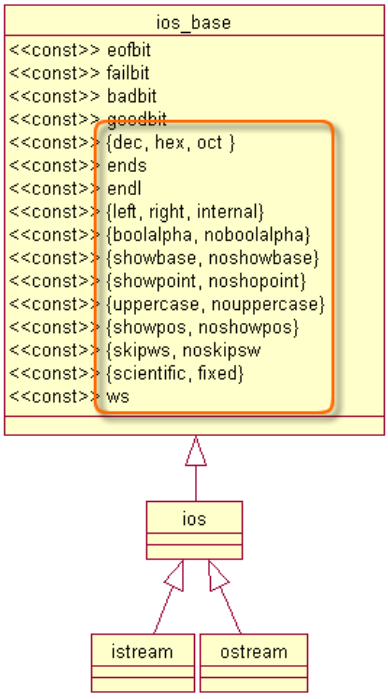
   Les constantes correspondantes au statut d'erreur sont des constantes statiques placées sur la classe ancêtre. Pour y accéder, vous êtes donc obligé de faire référence explicitement à cette classe de base. Ceci dit, puisque tous les flots héritent de tout ce qui se trouve sur la classe de base, vous pouvez prendre d'autres références. Ainsi, l'écriture précédente peut également être proposée sous les formes suivantes :

   cin.clear(ios::badbit | ios::failbit) ; ou *cin.setstate(istream::badbit | istream::failbit) ;*

2. ***void setstate(iostate état)*** **:** cette méthode joue le même rôle que la précédente. Toutefois, avec cette méthode, nous ajoutons, au lieu de rétablir, une condition à la condition existante de l'objet. Il s'agit donc d'un cumul des conditions.

Formatage de l'information sur un flot

Chaque objet flot conserve en permanence un ensemble d'indicateurs spécifiant quel est, à un moment donné, son statut de formatage. Ces indicateurs servent à contrôler, par exemple, l'affichage des valeurs entières suivant la base désirée (décimal, octal, hexadécimal), ou bien encore, permet de contrôler la précision des nombres à virgule flottante. Bien d'autres possibilités de formatage sont offertes.



Ces indicateurs sont positionnés avec des valeurs par défaut, ce qui permet à l'utilisateur d'ignorer totalement cet aspect, tant qu'il se contente de l'affichage par défaut. Un des avantages de ce système est de permettre à celui qui le souhaite, de définir, une fois pour toutes, un format approprié à une application donnée et de plus avoir à s'en soucier par la suite.

Le programmeur dispose de manipulateurs prédéfinis pour modifier l'état de format d'un objet flot. Un manipulateur s'applique à l'objet flux comme s'il s'agissait d'une donnée. Toutefois, au lieu de déclencher la lecture ou l'écriture des données, le manipulateur modifie en fait l'état interne de l'objet flux.

```
4  int main( )
5  {
6    bool binaire = true;
7
8    cout << binaire;
9    return 0;
10 }
```
Affichage -> 1

```
4  int main( )
5  {
6    bool binaire = true;
7    cout << boolalpha;
8    cout << binaire;
9    return 0;
10 }
```
Manipulateur qui modifie l'état de cout pour afficher des valeurs booléennes

Affichage -> true

- *Flot << manipulateur* pour un flot de sortie
- *Flot >> manipulateur* pour un flot d'entrée

#include <iostream> // Cette inclusion est obligatoire pour utiliser les manipulateurs ci-dessous	
***dec* / hex / oct**	Base de numération pour les valeurs entières, respectivement : décimal, hexadécimal, octal.
ends	Insère le caractère de fin de chaîne nul, puis vide le tampon.
endl	Insère une nouvelle ligne, puis vide le tampon.
left	Ajoute des caractères de remplissage à droite de la valeur.
right	Ajoute des caractères de remplissage à gauche de la valeur.
boolalpha / *noboolalpha*	Représente *true* et *false* sous forme de chaînes / Représente *true* et *false* au format *0*, *1*
showbase / *noshowbase*	Génère (ou pas) un préfixe indiquant une base numérique
showpoint / *noshowpoint*	Affichage du point décimal lorsqu'on utilise des réels et qu'il n'y a pas de parties décimales.
uppercase / *nouppercase*	Affichage des caractères hexadécimaux en majuscule.
showpos / *noshowpos*	Affichage des nombres positifs précédés du signe +
***skipws* / noskipws**	Saute (ou pas) l'espace avec les opérateurs d'entrée.
scientific / *fixed*	Notation scientifique des nombres réels : *1.5 e+01*, ou « point fixe » pour les nombres réels : *10.5*
ws	« Mange » l'espace

Avec ce type de manipulateur, nous n'avons pas besoin de repréciser le même traitement pour les variables qui suivent. Une fois que l'on place un manipulateur, il reste actif. Si vous désirez ré-obtenir le comportement par défaut, vous devez appliquez le manipulateur adéquat.

Variables.cpp

```
#include <iostream.h>
//------------------------------------------------------------------------
int main()
{
  int n = 12000;
  double r = 5.643;
  cout << "Par défaut      : "                << n << endl;
  cout << "en hexadécimal  : " << hex         << n << endl;
  cout << "et ensuite      : "                << n << endl;
  cout << "en décimal      : " << dec         << n << endl;
  cout << "Par défaut      : "                << r << endl;
  cout << "scientifique    : " << scientific  << r << endl;
  cout << "et ensuite      : "                << r << endl;
  cout << "en point fixe   : " << fixed       << r << endl;
  cin.get();
  return 0;
}
//------------------------------------------------------------------------
```
Le nombre de caractère espace n'est pas important en langage C++

```
D:\BTS IRIS\Projets\Entrées-so...
Par défaut      : 12000
en hexadécimal  : 2ee0
et ensuite      : 2ee0
en décimal      : 12000
Par défaut      : 5.643
scientifique    : 5.643000e+00
et ensuite      : 5.643000e+00
en point fixe   : 5.643000
```

Il existe également des manipulateurs paramétriques qui représente des valeurs numériques et non plus une information tout ou rien. Le principe est similaire aux manipulateurs précédents, toutefois, ils comportent en plus un paramètre qui récupère la valeur spécifiée. Ces manipulateurs vous sont présentés dans la page suivante.

#include <iomanip.h> // Cette inclusion est obligatoire pour utiliser les manipulateurs paramétrés ci-dessous	
setw (nombre)	Définit le gabarit de la variable à afficher avec une justification à droite par défaut. Si la valeur à afficher est plus importante que le gabarit, cette valeur ne sera pas tronquée et sera donc affichée de façon conventionnelle. Le manipulateur « *setw* » doit être utilisé pour chacune des informations à afficher.
setfill (caractère)	Définit le caractère de remplissage lorsqu'on utilise un affichage avec la gestion de gabarit. Par défaut, le caractère de remplissage est l'espace.
setprecision (nombre)	Permet de définir le nombre de chiffres significatifs pour les nombres réels. C'est uniquement pour l'affichage, la variable garde sa précision, et la valeur affichée est arrondie. Lorsque l'on utilise au préalable le manipulateur « ***fixed*** », le manipulateur « ***setprecision*** » permet d'indiquer le nombre de chiffres significatifs après la virgule.
setbase(base)	Spécifie la base d'affichage sur les nombres entiers.

En ce qui concerne « *setw* », sachez que ce manipulateur définit uniquement le gabarit de la prochaine information à écrire. Si l'on ne fait pas de nouveau appel à « *setw* » pour les informations suivantes, celles-ci seront écrites suivant les conventions habituelles, à savoir en utilisant l'emplacement minimal nécessaire pour les écrire.

Parametres.cpp

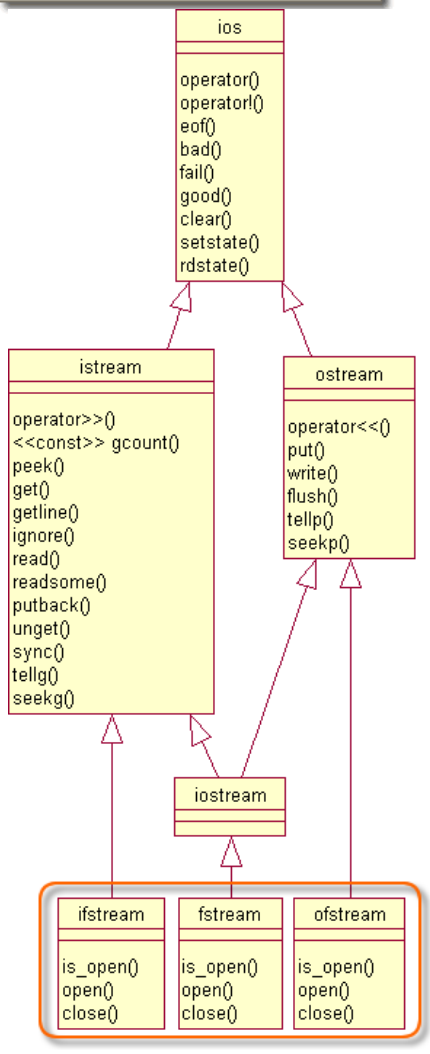
```
#include <iostream.h>
#include <iomanip.h>
//---------------------------------------------------------
int main()
{
  double x = 1023.568935621;
  cout << fixed << setprecision(4) << setfill('.');
  cout << "x       : " << setw(12)   << x << endl;
  cout << "x/1005  : " << setw(12)   << x/1005 << endl;
  cout << "x       : " << x << endl;
  cin.get();
  return 0;
}
//---------------------------------------------------------
```

12 emplacements pour le gabarit

```
D:\BTS IRIS\Projets\Entré...
x       :   ...1023.5689
x/1005  :   ......1.0185
x       : 1023.5689
```

Les fichiers

Nous nous intéressons maintenant à la mise en oeuvre et à la gestion des fichiers. Des classes sont spécialisées dans ce domaine. De plus, elles héritent directement ou indirectement de « *istream* » et « *ostream* », ce qui permet d'utiliser toutes les méthodes que nous venons d'étudier. Ainsi, nous pourrons prendre, par exemple, les opérateurs de redirection pour écrire ou lire directement dans un fichier.



Comme ces classes ont déjà, par héritage, un comportement performant, elles disposent juste de trois méthodes supplémentaires qui s'occupent essentiellement de l'ouverture et de la fermeture d'un fichier. Par ailleurs, ces classes sont spécialisées suivant le mode d'ouverture que nous désirons. Finalement, nous pouvons même nous abstenir d'utiliser ces méthodes. Voici le nom de ces trois classes :

1. ***ofstream*** : flot de sortie associé à un fichier. Permet d'écrire des informations de type quelconque dans un fichier.
2. ***ifstream*** : flot d'entrée associé à un fichier. Permet de lire des informations de type quelconque issues du fichier.
3. ***fstream*** : flot bidirectionnel associé à un fichier. Permet d'écrire ou de lire dans un fichier.

Pour utiliser un fichier, il suffit de créer un objet associé au mode d'ouverture voulu en spécifiant le nom du fichier désiré en argument du constructeur. Attention, tous les constructeurs de ces classes sont des constructeurs explicites. Nous sommes donc obligés d'utiliser impérativement les parenthèses. Dès que l'objet est créé, le fichier s'ouvre automatiquement suivant le mode donné par le type de la classe sélectionné. Enfin, lorsque l'objet est détruit, le fichier est automatiquement fermé, sans spécification particulière.

Entre ces deux phases, vous pouvez, suivant le cas, inscrire ou lire des informations en utilisant simplement toutes les méthodes que nous avons déjà vues, comme par exemple, les opérateurs de redirection. Comme ces opérateurs ont été redéfinis pour tous les types prédéfinis, il est donc possible d'envoyer ou de lire n'importe quelle information, comme des valeurs entières, des valeurs réelles, des chaînes de caractères, etc.

Par ailleurs, toujours grâce à l'héritage, vous pouvez également gérer les bits d'états, formater vos informations à l'aide des manipulateurs, etc. Bref, tout ce qui a déjà été vu, s'applique également aux fichiers.

Dans la page suivante, vous avez un programme qui ouvre un fichier en mode écriture nommé « *test.txt* », et qui écrit à l'intérieur de ce dernier, respectivement, la valeur entière *15*, la chaîne de caractères « *message* », le nombre réel *-5.3* et le caractère *'A'*. Vous avez ensuite un deuxième programme qui ouvre le même fichier, mais cette fois-ci en mode lecture et qui récupère l'ensemble des valeurs stockées.

```
15 message -5.3 A
```

Valeurs stockées dans le fichier « *test.txt* »

```
#include <fstream.h>
//--------------------------------------------------
int main()
{
  ofstream fichier("test.txt");
  fichier << 15 << ' ' << "message" << ' ' << -5.3 << ' ' << 'A';
  return 0;
}
//--------------------------------------------------
```

Inclusion nécessaire pour la gestion des fichiers
Création de l'objet qui représente un fichier nommé "test.txt" ouvert en écriture
Ecriture directe dans le fichier
Délimiteurs pour séparer les valeurs entre elles
Destruction de l'objet et donc fermeture du fichier "test.txt"

```
#include <fstream>
#include <string>
#include <iostream>
using namespace std;
//--------------------------------------------------
int main()
{
  ifstream fichier("test.txt");
  int entier;
  string message;
  double reel;
  char caractere;
  fichier >> entier >> message >> reel >> caractere;
  cout << entier << message << reel << caractere;
  return 0;
}
//--------------------------------------------------
```

Création de l'objet qui représente un fichier nommé "test.txt" ouvert en lecture
Récupération des valeurs stockées dans le fichier
Destruction de l'objet et donc fermeture du fichier "test.txt"

Vous remarquez l'extrême simplicité pour écrire ou lire dans un fichier. En fait, notre façon de procéder est la même que lorsque nous utilisions les objets prédéfinis « *cin* » et « *cout* ». Dans les exemples que nous venons de voir, nous n'avons pas du tout utilisé les méthodes spécifiques de ces trois classes spécialisées. Il existe des cas où ces méthodes peuvent s'avérer utiles, notamment lorsque nous demandons à l'utilisateur de spécifier le nom du fichier à posteriori.

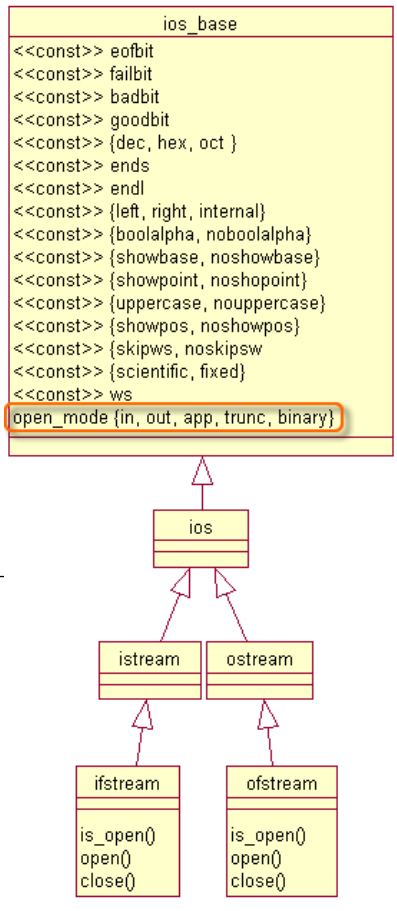
```
#include <fstream>
#include <string>
#include <iostream>
using namespace std;
//--------------------------------------------------
int main()
{
  ofstream fichier;
  string nom;
  cout << "Donnez le nom du fichier ? ";
  cin >> nom;
  fichier.open(nom.c_str());
  fichier << "Texte introduit dans le fichier";
  fichier.close();
  return 0;
}
```

Objet fichier en mode écriture, le fichier n'est pas encore ouvert
Ouverture du fichier avec le nom spécifié par l'utilisateur
Fermeture explicite du fichier

Les différents modes d'ouverture d'un fichier

Le mode d'ouverture est défini par un mot d'état « *open_mode* », dans lequel chaque bit correspond à une signification particulière. La valeur correspondant à chaque bit est définie par des constantes déclarées dans la classe de base « *ios_base* ». Pour activer plusieurs bits, il suffit de faire appel à l'opérateur ou logique « | ».

Bits d'ouverture de fichier dans le mot d'état « *open_mode* ».	
in	Ouverture en lecture. Le fichier doit exister.
out	Ouverture en écriture. Ecrase l'ancien contenu. Si le fichier n'existe pas, il est automatiquement créé.
app	Ouverture en ajout de données (écriture en fin de fichier).
trunc	Si le fichier existe, son contenu est définitivement perdu.
binary	Pour les précédents modes, l'information été systématiquement transformée en une suite de caractère. Dans l'exemple précédent, nous avons sauvegardé un certain nombre de valeurs de type différent. Au moment du transfert vers le fichier, ces valeurs subissent une transformation pour devenir une suite de caractères. Il est alors possible de contrôler le contenu du fichier avec un simple éditeur de texte. Toutefois, vous pouvez désirer conserver le type original et donc demander à avoir un stockage sous forme binaire. C'est ce que permet ce mode d'ouverture.



Signatures des méthodes et choix du mode d'ouverture d'un fichier

ifstream

- *ifstream();* : constructeur par défaut
- *ifstream(const char *nomfichier, ios_base::open_mode mode = ios_base::in) ;* : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert en lecture.
- *bool is_open() const ;* : détermine si le fichier représenté par l'objet est ouvert.
- *void open(const char *nomfichier, ios_base::open_mode mode = ios_base::in) ;* : Ouvre le fichier représenté par l'objet en mode lecture (par défaut).
- *void close() ;* : ferme le fichier représenté par l'objet.

ofstream

- *ofstream();* : constructeur par défaut

- *ofstream(const char *nomfichier, ios_base::open_mode mode = ios_base::out) ;* : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert en écriture.
- *bool is_open() const ;* : détermine si le fichier représenté par l'objet est ouvert.
- *void open(const char *nomfichier, ios_base::open_mode mode = ios_base::out) ;* : Ouvre le fichier représenté par l'objet en mode écriture (par défaut).
- *void close() ;* : ferme le fichier représenté par l'objet.

fstream

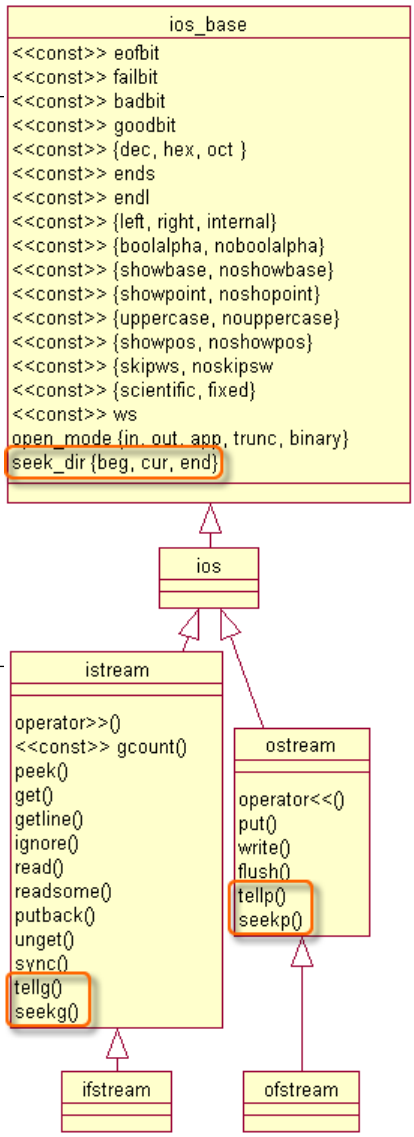
- *fstream();* : constructeur par défaut
- *fstream(const char *nomfichier, ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert à la fois en lecture et en écriture.
- *bool is_open() const ;* : détermine si le fichier représenté par l'objet est ouvert.
- *void open(const char *nomfichier, ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Ouvre le fichier représenté par l'objet en mode lecture et écriture (par défaut).
- *void close() ;* : ferme le fichier représenté par l'objet.

Dans ces classes, les modes d'ouvertures sont positionnés avec des paramètres par défaut, ce qui convient dans la plupart des cas. Il est toutefois possible de proposer un autre comportement. Nous avons, par exemple, souvent besoin d'ouvrir un fichier en mode ajout. Il faut donc prendre la classe « *ofstream* » qui permet d'écrire dans un fichier et changer le mode par défaut. Ainsi :

ofstream fichier("nom du fichier", ios::app) ; ou *ofstream fichier("nom du fichier", ofstream::app) ;*

Et si nous voulons fabriquer un fichier pour écrire des valeurs enregistrées sous forme brute :

ofstream fichier("nom du fichier", ios::out | ios ::binary) ;



Bits d'erreur

```
1  #include <fstream>
2  #include <iostream>
3  using namespace std;
4  //------------------------------------------------------------------
5  int main()
6  {
7    ifstream fichier("test.txt");
8    if (!fichier) { cerr << "Le fichier n'existe pas"; return -1; }
9    char caractere;
10   while (fichier.get(caractere)) cout.put(caractere);
11   return 0;
12 }
```

Puisque ces classes ont récupérées par héritage les différents bits d'erreur, il convient de s'en servir pour contrôler que l'ouverture d'un fichier s'est bien déroulée. D'autre part, cette démarche s'avère utile lorsque nous désirons faire une lecture complète d'un fichier sans spécialement connaître la dimension de ce dernier. Il suffit alors de contrôler uniquement la fin du fichier.

Le programme ci-dessus illustre ces propos et permet de retracer à l'écran le contenu d'un fichier texte.

Accès direct à une position absolue dans le fichier

Le terme flot indique bien que nous soutirons l'information à la volée sous forme séquentielle. Toutefois, il est possible d'accéder à un endroit précis du fichier pour prendre juste la valeur désirée. L'accès direct est implémenté sous la forme d'un pointeur de fichier, c'est-à-dire un nombre précisant le rang du prochain « *octet* » à lire ou à écrire. Après chaque opération de lecture ou d'écriture, ce pointeur est incrémenté du nombre d'octets transférés. Ainsi, lorsque nous n'agissons pas explicitement sur ce pointeur, nous réalisons en fait un accès séquentiel classique; c'est d'ailleurs ce que nous avons fait jusqu'à présent. Attention, l'accès direct n'est possible qu'en considérant le fichier que sous forme d'une suite d'informations binaires.

Finalement, les possibilités d'accès direct se résument donc aux possibilités d'action sur ce pointeur ou à la détermination de sa valeur. Des méthodes héritées respectivement de « *istream* » et de « *ostream* » permettent de se déplacer à une adresse « *absolue* » à l'intérieur du fichier ou à une distance *en octets* depuis une position donnée.

- La méthode « *seek* » permet de positionner le pointeur de fichier à un endroit précis.
- La méthode « *tell* » permet de donner la position actuelle du pointeur de fichier.

En fait, le nom de ces méthodes possède une lettre supplémentaire suivant qu'elles dérivent de « *istream* » ou de « *ostream* ». Dans le premier cas, les méthodes issues de « *istream* » possèdent le suffixe « ***g*** » (pour get). Dans le deuxième cas, les méthodes issues de « *ostream* » possèdent le suffixe « ***p*** » (pour put). Par ailleurs, des constantes ont été spécialement conçues pour indiquer respectivement, le début du fichier, la fin du fichier, ou la position courante du fichier.

Classe	Méthodes associées	
istream	• *istream& seekg (int pos_courante) ;* Le pointeur de fichier pointe sur la position absolue « *pos_courante* ». • *istream& seekg (int pos_relative, ios_base ::seekdir origine) ;* Le pointeur de fichier pointe relativement à la position « *pos_relative* » par rapport à « *l'origine* » fixée par le deuxième argument. • *int tellg ()* renvoie la position courante du pointeur de fichier.	
ostream	• *ostream& seekp (int pos_courante) ;* Le pointeur de fichier pointe sur la position absolue « *pos_courante* ». • *ostream& seekp (int pos_relative, ios_base ::seekdir origine) ;* Le pointeur de fichier pointe relativement à la position « *pos_relative* » par rapport à « *l'origine* » fixée par le deuxième argument. • *int tellp ()* renvoie la position courante du pointeur de fichier.	
ios_base	• *ios_base ::beg* : début du fichier. • *ios_base ::cur* : position courante du fichier. • *ios_base ::end* : fin du fichier.	Ces trois constantes sont à utiliser en corrélation avec « ***ios_base ::seekdir*** »

Pour l'accès direct, puisque nous travaillons octet par octet, il peut être utile de connaître la dimension exacte en octets des variables que nous utilisons pour envoyer ou recevoir des valeurs stockées dans les fichiers. Il existe l'opérateur « ***sizeof*** » qui réalise ce calcul et qui peut être utilisé de trois façons différentes :

- ***sizeof (type)***
- ***sizeof (objet)***
- ***sizeof objet ;***

L'exemple ci-contre permet d'écrire sous forme binaire, grâce à la méthode « ***write*** » une suite de valeur entière dans le fichier « *test.txt* ». Ensuite nous lisons ce fichier et nous nous positionnons sur le troisième entier enregistré que nous récupérons grâce à la méthode « ***read*** » pour l'afficher à l'écran.

```
#include <fstream>
#include <iostream>
using namespace std;
//-----------------------------------------------------
int main( )
{
  ofstream ecriture("test.txt", ios::out | ios::binary);
  int entiers[7] = {15, -3, 25, 12, 48, -33, 99};
  for (int i=0; i<7; i++)
    ecriture.write((char *)&entiers[i], sizeof(int));
  ecriture.close();          écriture dans le fichier de l'ensemble du tableau sous forme binaire et formater en entier

  ifstream lecture("test.txt", ios::in | ios::binary);
  lecture.seekg(sizeof(int)*2);
  int valeur;                Positionnement sur le 3ème entier
  lecture.read((char *)&valeur, sizeof(int));
  cout << valeur;            Récupération du 3ème entier
  return 0;                  La valeur affichée est 15.
}
```

Les méthodes « ***write*** » et « ***read*** » ont spécialement été mise en œuvre pour travailler avec des informations binaires. Comme le premier argument est de type « *char ** » il est nécessaire de provoquer un changement de type explicite.

Choix du type de fichier

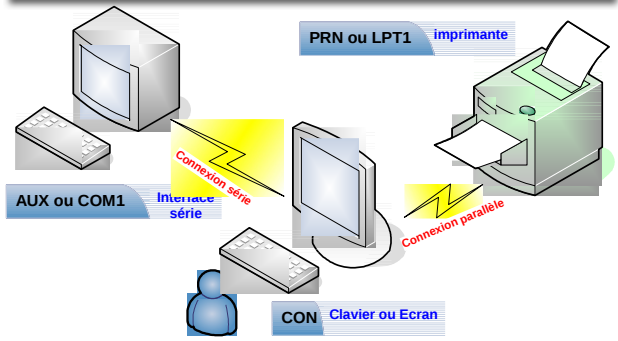
Ce que nous avons proposé comme nom de fichier jusqu'à présent, correspondait à un fichier sur le disque dur. Il est toutefois possible de communiquer avec d'autres ressources.

Si vous tester le programme ci-contre, vous allez vous apercevoir que les objets « *clavier* » et « *ecran* » remplacent « *cin* » et « *cout* ».

```
#include <fstream>
using namespace std;
//-----------------------------------------------------
int main()          Le flux d'entrée est redirigé vers le clavier
{
  ifstream clavier("CON");
  int valeur;                Le flux de sortie est redirigé vers l'écran en mode ajout
  clavier >> valeur;
  ofstream ecran("CON", ofstream::app);
  ecran << valeur << endl;
  return 0;
}
```

En réalité, le nom de fichier « *CON* » est un mot réservé du système d'exploitation qui correspond à la « console » de l'ordinateur, c'est-à-dire, en entrée effectivement le clavier, et en sortie l'écran. (En fait, il s'agit de l'entrée standard et de la sortie standard qui, par défaut, sont réglées sur ces éléments).

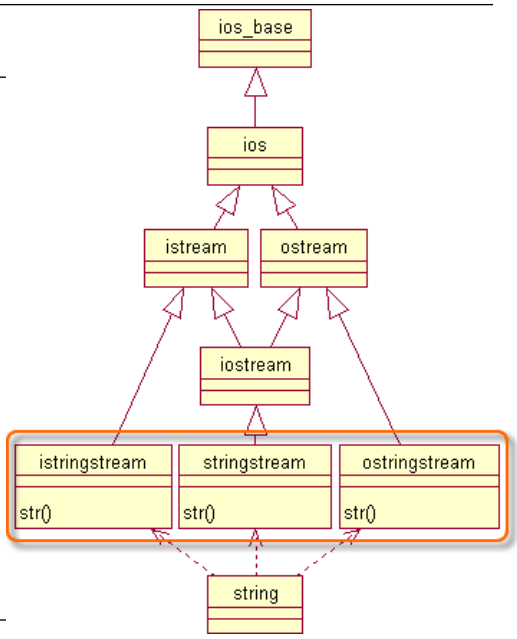
Il existe d'autres mots réservés du système d'exploitation, comme le montre le schéma ci-contre :



Si vous êtes sous UNIX, vous pouvez par exemple proposer un affichage sur une autre fenêtre en mode console (ici la 3ème) en écrivant :

ofstream fenetre("/dev/tty3", ios::app) ;

Les flux de chaîne



Plutôt que d'être en communication avec un périphérique quelconque, il est également possible de gérer les flux directement en mémoire centrale, tout en utilisant les méthodes et les constantes que nous venons de mettre en œuvre. Ce stockage en mémoire se fait sous forme d'une chaîne de caractères. Il sera ensuite possible de récupérer cette chaîne dans un objet de type « *string* ». Nous avons vu que dans les flux, grâce à la redéfinition des opérateurs de redirection, il est possible de stocker et de récupérer des valeurs de type quelconque. Finalement, ce stockage en mémoire sera surtout utiliser pour transformer des valeurs de type quelconque, comme des entiers, des réels, etc. vers une chaîne de caractères ou l'inverse.

Trois classes, à l'instar des fichiers, sont spécialisées dans ce domaine d'intervention :

1. ***ostringstream*** : flot de sortie associé à une chaîne de caractères.
2. ***istringstream*** : flot d'entrée associé à une chaîne de caractères.
3. ***stringstream*** : flot bidirectionnel associé à une chaîne de caractères.

Méthodes associées à ces classes de gestion de chaînes de caractères

En fait, il en existe très peu. Tout le mécanisme interne du traitement de chaîne de caractères est totalement caché (encapsulé) comme d'ailleurs pour la gestion des fichiers. Nous avons juste une méthode pour retourner la valeur de la chaîne de caractères et les constructeurs qui disposent chacun de paramètres par défaut pour permettre une utilisation la plus simple possible et pour correspondre aux cas les plus fréquents. Malgré tout, il existe des constructeurs qui permettent de construire le flot à partir d'une chaîne de caractères afin de proposer un formatage de cette chaîne vers d'autres types par la suite.

- *istringstream(ios_base::open_mode mode = ios_base::in) ;* : Constructeur en mode lecture par défaut.
- *istringstream(string chaîne, ios_base::open_mode mode = ios_base::in) ;* : Constructeur en mode lecture par défaut.
- *ostringstream(ios_base::open_mode mode = ios_base::out) ;* : Constructeur en mode écriture par défaut.
- *ostringstream(string chaîne, ios_base::open_mode mode = ios_base::out) ;* : Constructeur en mode écriture par défaut.
- *stringstream(ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Constructeur en mode lecture et écriture par défaut.
- *stringstream(string chaîne, ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Constructeur en mode lecture et écriture par défaut.
- *string str() ;* : renvoie la chaîne de caractère stockée dans le flux.

Dans l'exemple ci-contre, nous convertissons un nombre réel en son équivalent sous forme de chaîne de caractères.

```
1  #include <sstream>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5  //-------------------------------------------------------------
6  int main()
7  {
8      ostringstream fluxChaine;          // Création du flux de chaîne
9      double x = 15.3;
10     fluxChaine << x;                   // Transformation d'un réel en chaîne
11     string chaine = fluxChaine.str();
12     cout << chaine;                    // Récupération et affichage de la chaîne
13     return 0;
14 }
```

Finalement, nous disposons, grâce à ces classes, de tout un système de formatage pour passer d'un type quelconque vers une chaîne de caractère et vice versa.

Changement du comportement par défaut

Le comportement par défaut de toute cette hiérarchie de la gestion des flux est déjà remarquable. Toutefois, ce serait encore mieux si nous pouvions avoir, par exemple, un affichage automatique sur les classes que nous créons. Il suffirait alors de proposer notre objet directement derrière l'opérateur de redirection pour que, effectivement, il s'affiche suivant notre désir.

En fait, il suffit de redéfinir l'opérateur « **<<** » pour que ce fonctionnement s'applique. Attention toutefois, notre nouvelle classe ne faisant pas partie de la hiérarchie, nous sommes obligés de redéfinir cet opérateur en tant que fonction et non pas en tant que méthode. Il faudra donc proposer une relation d'amitié, à moins que vous ne disposiez de toutes les méthodes requises pour la lecture des attributs. (Revoir l'étude de la redéfinition des opérateurs pour comprendre ces problèmes).

Ce que je viens de dire pour une communication vers l'extérieur s'applique, bien entendu, pour une lecture. Ainsi, il sera également possible de redéfinir l'opérateur « **>>** », et donc de prévoir, par exemple, une saisie clavier adaptée à la classe étudiée.

Cette hiérarchie de classe intègre le polymorphisme. Donc, lorsque nous redéfinirons les différents opérateurs, ils seront alors utiles aussi bien pour la console (le clavier et l'écran) que pour l'écriture ou la lecture dans un fichier (disque dur, imprimante, interface série, etc.) ou pour transformer notre classe en une chaîne de caractères et vice versa.

Entre parenthèse, nous remarquons ici, l'intérêt du polymorphisme, puisqu'une seule étude comportementale se répercute sur l'ensemble de la hiérarchie et donc sur l'ensemble du fonctionnement, c'est-à-dire finalement, sur l'ensemble des périphériques.

Rappelons qu'à droite d'un opérateur de redirection nous avons la classe à traiter, ensuite à gauche le flot concerné, et qu'une fois que l'opération s'est déroulée correctement l'opérateur renvoie également un objet flot, ce qui permet notamment les enchaînement des opérateurs de redirection. Nous aurons donc les gabarits suivants :

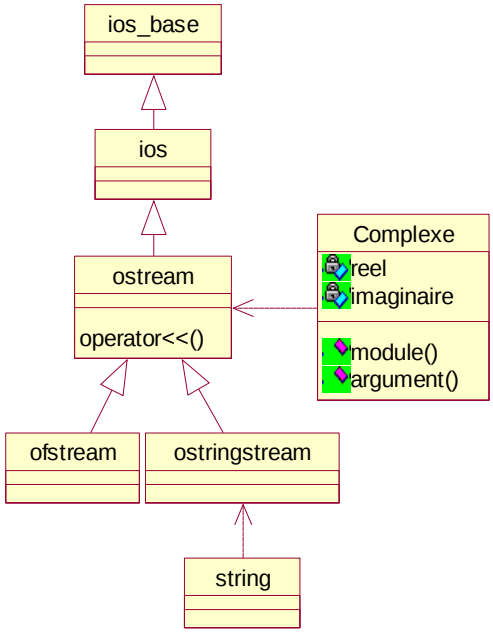
ostream& operator << (ostream&, const NouvelleClasse&) ;
istream& operator >> (istream&, NouvelleClasse&) ;

Pour illustrer ces propos, nous allons redéfinir l'opérateur « **<<** » pour la classe « *Complexe* » que nous connaissons déjà bien.

```
1  #include <iostream>
2  using namespace std;
3  //-----------------------------------------------------------
4  class Complexe
5  {
6    double reel;
7    double imaginaire;
8    friend ostream& operator<< (ostream&, const Complexe&);
9  public:
10   Complexe(double r=0.0, double i=0.0) : reel(r), imaginaire(i) { }
11   double module() const;
12   double argument() const;
13 };
14 //-----------------------------------------------------------
15 ostream& operator<< (ostream& os, const Complexe& c)
16 {
17   return os << '<' << c.reel << ", " << c.imaginaire << '>';
18 }
19 //-----------------------------------------------------------
20 int main()
21 {
22   const Complexe i(0.0, 1.0);
23   cout << i;
24   return 0;
25 }
```

Autorisation à l'opérateur d'atteindre les attributs privés

L'affichage d'un nombre complexe est aussi simple que pour un type primitif, l'affichage est <0, 1>



Une fois que cette redéfinition est faite, et puisque nous sommes au niveau de la classe « *ostream* », nous pouvons stocker dans un fichier un nombre « *Complexe* » dans cette représentation. Ce que je viens de dire pour les fichiers s'applique, bien entendu, pour l'imprimante, pour le formatage sous forme de chaîne de caractères, etc.

Gestion des répertoires - *<dir.h>*

Il peut être utile d'avoir des renseignements sur le contenu d'un répertoire afin de pouvoir contrôler l'existence d'un fichier. Malheureusement, cette possibilité n'a pas été intégrée directement dans les flux standard. Par contre, nous pouvons faire référence à un certain nombre de fonctions qui s'occupent de ce genre de problème et qui existe depuis le début du langage C. Nous allons en recenser quelques unes.

- *int chdir(char * répertoire) ;* Changement du répertoire courant. Retourne *0* si l'opération a réussi.
- *int mkdir(char * répertoire) ;* Création d'un nouveau répertoire. Retourne *0* si l'opération a réussi.
- *int rmdir(char * répertoire) ;* Suppression du répertoire. Retourne *0* si l'opération a réussi.
- *int getcurdir(int unité, char * répertoire) ;* Spécifie le nom du répertoire courant en précisant l'unité de disque (*0 : disque courant, 1 : unité A, 3 : unité C, …*). Retourne *0* si l'opération a réussi.
- *int getcwd(char * répertoire, int taille) ;* Spécifie le nom du répertoire courant avec en plus le nom de l'unité. Il faut, par contre préciser la dimension de la chaîne de caractères. La constante MAXDIR contient le nombre de caractères à réserver en toute sécurité.
- *int getdisk(void) ;* Retourne l'unité par défaut.
- *int setdisk(int unité) ;* Changement de l'unité courante. Retourne *0* si l'opération a réussi.
- *int findfirst(char *répertoire, struct ffblk *info, int attribut) ;* Cherche le premier fichier du répertoire courant en spécifiant les filtres désirés. Toutes les informations relatives à un fichier sont stockées dans la structure de type « *ffblk* ». Il est possible grâce à *attribut* de spécifier le type de fichier attendu (voir plus loin). Retourne *0* si l'opération a réussi.
- *int findnext(struct ffblk *info) ;* Trouve le fichier suivant. Cette fonction s'utilise à la suite de la fonction *findfirst* et se sert de la structure initialisée par cette dernière. Retourne *0* si le fichier a été trouvé, sinon, elle retourne *-1*.

```
60 struct  ffblk   {
61     long            ff_reserved;
62     long            ff_fsize;
63     unsigned long   ff_attrib;
64     unsigned short  ff_ftime;
65     unsigned short  ff_fdate;
66     char            ff_name[MAXPATH];
67 };
```

Structure d'un en-tête de fichier

taille

Date et heure

nom

```
43 struct  ftime   {
44     unsigned    ft_tsec  : 5;
45     unsigned    ft_min   : 6;
46     unsigned    ft_hour  : 5;
47     unsigned    ft_day   : 5;
48     unsigned    ft_month : 4;
49     unsigned    ft_year  : 7;
50 };
```

< io.h >

Attributs d'un fichier - *<dir.h>*

- *FA_NORMAL* : pour obtenir les fichiers normaux,
- *FA_RONLY* : pour obtenir les fichiers à lecture seule,
- *FA_HIDDEN* : les fichiers cachés,
- *FA_LABEL* : l'étiquette de volume (soit le disque, soit la partition),
- *FA_DIREC* : les répertoires,
- *FA_ARCH* : les fichiers marqués à archiver.

Quelques exemples d'utilisation de ces fonctions

```
#include <iostream.h>
#include <dir.h>
#include <io.h>
//------------------------------------------------------------------------------
int main( )
{
  char chaine[MAXDIR];
  ffblk info;
  ftime &temps = (ftime &) info.ff_ftime;

  chdir("..\\..");
  getcurdir(0, chaine);
  cout << "Répertoire courant : " << chaine << endl;
  cout << "Unité de disque : " << char('A'+getdisk()) << endl;
  getcwd(chaine, MAXDIR);
  cout << "Répertoire courant : " << chaine << endl;

  if (findfirst("*.*", &info, FA_NORMAL | FA_DIREC))
    cout << "Aucun fichier dans ce répertoire" << endl;
  else do {
    cout << info.ff_name << '\t';
    if (info.ff_attrib == FA_DIREC) cout << "<REP>\t";
    cout << temps.ft_day << '/' << temps.ft_month << '/' << 1980+temps.ft_year;
    cout << ' ' << temps.ft_hour << ':' << temps.ft_min << endl;
  }
  while (!findnext(&info));

  return 0;
}
```

```
Répertoire courant : Documents and Settings\Emmanuel REMY\cbproject\Test
Unité de disque : C
Répertoire courant : C:\Documents and Settings\Emmanuel REMY\cbproject\Test
.       <REP>   29/12/2004 16:15
..      <REP>   29/12/2004 16:15
bak     <REP>   29/12/2004 16:15
Test.cbx        29/12/2004 9:42
Test.cbx.local  29/12/2004 11:53
Test.cpp        29/12/2004 16:15
windows <REP>   8/12/2004 17:5
```