Modélisation et programmation

Cours 2

ENSIMAG 2A – MMIS

Agenda

Automatisation de la compilation

Surcharge d'opérateur

Le C++ n'est pas du C

La compilation

- Afin de rendre votre code compilable par n'importe qui, utilisez un Makefile. Exemple de l'ensiwiki pour la compilation en C

```
1  PROG = calc
2  OBJS = calc.o stack.o
3  CXX = g++
4  CXXFLAGS = -Wall -pedantic -g
5
6  $(PROG): $(OBJS)
7          $(CXX) $(CXXFLAGS) -o $@ $(OBJS)
```

  Les espaces au début de la dernière ligne sont une tabulation.

- Les options du Makefile peuvent être modifiées lors de l'appel à la commande `make`

```
1  make CXXFLAGS='-O2 ' calc
```

- Comprenez les options de compilations que vous utilisez : est-ce que `-O3` est utile? Est-ce que je doit effectuer les tests de performances avec l'option `-g` ou `-pg`? Est-ce que le standard donné par l'option `-c99` est complètement défini? Est-ce que j'ai besoin de toutes le bibliothèques données à l'édition de liens?

Etape 1 : compilation simple

- Makefile

```
1  exe : source.o
2          g++ -o exe source.o
3  source.o : source.c
4          g++ -c source.c
```

- Execution
  - `make` : crée les fichier `exe` et `source.o`
  - `make source.o` : crée le fichier `source.o`
- Explication
  - Pour créer le fichier `exe`, le compilateur a besoin du fichier `source.o`. La règle étant donnée plus loin, le fichier est créé.
  - Pour créer seulement le fichier objet, on execute `make source.o`
  - Un makefile est un ensemble de règles qui permettent d'automatiser certaines tâches (compilation, execution)
  - On définit une règle à l'aide d'un label (`exe`). Ce label est suivi de `:`. Ensuite, on spécifie les dépendances de la règle. Enfin, on donne l'instruction à executer.

Etape 2 : Variables

- Makefile

```
1  CXX = g++
2  CXXFLAGS = -02 -Wall
3  exe : source.o
4          $(CXX) source.o -o exe
5  source.o : source.c
6          $(CXX) -c $(CXXFLAGS) source.cxx
```

- Execution
  - `make` : crée les fichier `exe` et `source.o`

- Explication
  - Afin de rendre la compilation flexible et portable, on définit une variable qui va contenir le nom du compilateur. Certains noms de variables correspondent à des langage de compilation particuliers (`CXX` pour le C++, `CC` pour le C).
  - De même, la variable `CXXFLAGS` permet de modifier automatiquement toutes les options de compilations

Etape 3 : Généraliser la compilation

- Makefile

```
1  # Commentaire
2  CXX = g++
3  CXXFLAGS = -02 -Wall
4  LDFLAGS =
5  EXECUTABLE = exe
6  SOURCES = source1.cxx source2.cxx source3.cxx
7  OBJETS = $(SOURCES:.cxx=.o)
8  $(EXECUTABLE) : $(OBJETS)
9          $(CXX) $(CXXFLAGS) -o $(EXECUTABLE) $(OBJECTS) $(LDFLAGS)
10 .cxx.o:
11         $(CXX) $(CXXFLAGS) $< -o $@
```

- Execution
  - **make** : crée les fichier **exe** et **sourcei.o**
- Explication
  - Le raccourci **$<** désigne le nom de la première dépendance.
  - Le raccourci **$@** désigne le nom de la cible.

Etape 4 : Aller plus loin

- Makefile

```
1  # Commentaire
2  CXX = g++
3  CXXFLAGS = -02 -Wall
4  LDFLAGS =
5  EXECUTABLE = exe
6  SOURCES = source1.cxx source2.cxx source3.cxx
7  OBJETS = $(SOURCES:.cxx=.o)
8  $(EXECUTABLE) : $(OBJETS)
9          $(CXX) $(CXXFLAGS) -o $@ $^  $(LDFLAGS)
10 .cxx.o:
11         $(CXX) $(CXXFLAGS) $< -o $@
12 clean:
13         rm -f $(EXECUTABLE) $(OBJETS)
```

- Execution
  - `make` : crée les fichier `exe` et `sourcei.o`
  - `clean` : efface les fichier `exe` et `sourcei.o`

Etape 4 : Aller plus loin

- Explication
  - Le raccourci `$^` désigne la liste des dépendances.
  - On peut ajouter autant de règle que nécessaires : pour executer, pour afficher, pour éditer
  - En utilisant les bonnes variables la ligne de compilation des sources `.cxx` devient superflu.
  - Utilisable pour compiler du latex.

Il s'agit maintenant de générer des **Makefile** qui soient multi plateforment. Nous allons donc utiliser l'outils **cmake**.

- Fichier équivalent

```
cmake_minimum_required (VERSION 2.8.11)

project(Demo)

set(CMAKE_CXX_FLAGS "-g -Wall")

# Commentaire
add_executable(exe source1.cxx source2.cxx source3.cxx)
```

- Invocation `mkdir build; cd build; cmake ..; make`

Agenda

Automatisation de la compilation

Surcharge d'opérateur

Le C++ n'est pas du C

Syntaxe de la surcharge

- On utilise le mot réservé **`operator`** et la syntaxe suivante

  argRetour **`operator`** nomOperateur (argtEnree)

- Le nombre d'arguments d'entrée est lié au type d'opérateur (unaire, binaire, ternaire)

- L'argument de sortie dépend des objectifs fixés.

Exemple de surcharge

```
1   class POINT
2   {
3     public :
4       int dim;
5       double *pCor;
6       POINT(int d, double v=0);    //constructeur (dimension, val)
7       POINT(const POINT &P);       //constructeur par copie
8       POINT operator * (double d); //produit par un double
9   };
10
11  POINT POINT::operator *(double d)
12  {
13    POINT Q(dim);
14    for(int i=0;i<dim;i++)
15    {
16      Q.pCor[i] = pCor[i]*d;
17    }
18    return Q;
19  }
```

Exemple d'opérateur =

- Etape 0 : L'opérateur d'affectation doit faire correspondre une variable membre de la classe cible à une variable membre de la classe source.

```
1    void Point::operator=(Point P)
2    {
3      dim = P.dim;
4      pCor = new double[dim];
5      memcpy(pCor, P.pCor, dim * sizeof(double));
6    }
```

- Etape 1 : L'affectation doit seulement être utilisé pour modifiée la cible. La source doit restée inchangée:

```
1    void operator=(const Point P);
```

- Etape 2 : Afin d'améliorer la vitesse d'accès aux données de la variable source, l'argument peut-être passé par référence.

```
1    void operator=(const Point &P);
```

Exemple d'opérateur =

- Etape 3 : Lors d'une affectation, la valeur de la variable source est donnée à la variable gauche. Ainsi, après l'opération, le type de retour doit être non-**void**. L'opération doit donc retourner une référence sur le type de l'objet.

```
1    Point &operator=(const Point &P);
```

- Etape 4 : Puisque le type de retour n'est plus **void**, l'opérateur doit retourner une valeur. L'opérateur d'affectation est fait entre deux objets de même type. L'opérateur doit donc retourner la même variable que celle qui l'a appelé. On utilise donc l'opérateur **this**. L'implémentation finale est donc

```
1    Point& Point::operator=(const Point &P)
2    {
3      dim = P.dim;
4      pCor = new double[dim];
5      memcpy(pCor, P.pCor, dim * sizeof(double));
6
7      return *this;
8    }
```

- Il est important, dans certaines configuration de
  - Vérifier la mémoire : est-ce que le tableau `pCor` est déjà alloué?
  - Vérifier l'auto-référence

Surcharge externe

- Déclaration dans une class : surcharge interne ne fonctionne pas pour l'opération
  `3*P` (double x POINT)

- Surcharge externe

```
1   POINT operator *(double d, const POINT & P)
2   {
3     POINT Q(dim);
4     for (int i=0;i<dim;i++)
5     {
6       Q.pCor[i] = P.pCor[i]*d;
7     }
8     return Q;
9   }
10
11  int main()
12  {
13    POINT M(3);
14    POINT P=M*2; // surcharge de * interne
15    POINT Q=3*M; // surcharge de * externe
16  }
```

- Pas d'ambigüité : arguments d'entrée distincts entre les deux versions

- Dans l'écriture `M*2` ou `3*M`, 2 et 3 sont des entiers: il y a conversion automatique en double.

Surcharge interne ou externe

- On peut également n'utiliser que des surcharges externes
- Version externe : `POINT * double`

```
1    POINT operator *(double d, const POINT & P)
2    {
3      POINT Q(dim);
4      for (int i=0;i<dim;i++)
5      { Q.pCor[i] = P.pCor[i]*d; }
6      return Q;
7    }
```

- Version externe : `double * POINT`

```
1    POINT operator *(const POINT & P,double d)
2    {
3      POINT Q(dim);
4      for (int i=0;i<dim;i++)
5      { Q.pCor[i] = P.pCor[i]*d; }
6      return Q;
7    }
```

- Mettre en externe si l'opération n'induit pas de modification de l'objet.
- Meilleure visibilité et cohérence.

Surcharge fortement recommandée

- La surcharge de *= doit être interne.

```
1   class POINT
2   {
3     public :
4       ...
5       POINT & operator *= (double d); //produit par un double
6   };
7   POINT & POINT::operator *= (double d)
8   {
9     for(int i=0;i<dim;i++)
10    {
11      pCor[i]*=d;
12      return *this;
13    }
14  }
```

- On retourne une référence à l'objet lui même (on évite la recopie).

- Ne fonctionne pas si `pCor` est protégé (**`private`**)

Liste des opérateurs surchargeables

+	-	*	/	%	^	&	|	~	
+=	-=	*=	/=	%=	^=	&=	|=	++	–
!	=	==	!=	<	>	<=	>=	->	->*
»	«	«=	»=	&&	||	[]	()	,	
new	new[]	delete	delete[]						

- Opérateurs unaires ou binaires
- Opérateurs non surchargables
  - `::` résolution de portée
  - `.` sélection de membre
  - `.*` sélection de membre via un pointeur de fonction.
- Surcharges de **`new`** et **`delete`** pour réécrire les mécanismes d'allocation dynamique.

Surcharge rationnelle

- Utiliser le sens usuel des opérateurs et éviter des surcharges fantaisistes, par exemple la division de 2 points

```
1    POINT operator/(const POINT a, const POINT &);
```

- Eviter toute confusion possible, par exemple `()` et `[]` sont des opérateurs d'accès pour des objets de type vecteur
  - `()` suivant la convention, commence à 0 ou à 1 : logique C++ ou mathématique
  - `[]` commence à 0 : logique du C++

```
1    class vect
2    {
3      public :
4        double *val; //tableau de valeurs
5        ...
6          double & operator()(int i){return val[i+1];}
7        double & operator[](int i){return val[i];}
8    }
```

- Retourne une référence $\Longrightarrow$ modification d'un élément du vecteur.

Opérateur de conversion

Cas particulier de l'opérateur de conversion/transtypage.

```
1    operator POINT()(const vect &);
```

Syntaxe générale

operator type_Sortie ()(type_Entree)

- Pas d'arguments de retour, s'apparente à un constructeur
- Est utilisé implicitement par le compilateur s'il y a un transtypage
  fonction attendant un point et recevant un vect
- Utilisation très délicate, augmente les ambiguïtés, à utiliser avec parcimonie.
- Il est plus simple et fortement recommandée d'utiliser un constructeur de la classe POINT

```
1    POINT(const vect &);
```

Opérateurs de flux

- Généralement les opérateurs << et >> sont réservés à des opérations d'entrée/sortie dans un flux (écran, fichier, buffer, ...).
- Flux standards:
  - `cin` entrée clavier, classe `istream`
  - `cout` sortie écran, classe `ostream`

```
1   class POINT{...};
2   ostream & operator <<(ostream &OPut, const POINT &P)
3   {
4     Out<<"Point : ";
5     for(int i=0;i<P.dim;i++)
6     { Out<<P(i)<<" " ; }
7     Out<<endl;
8     return Out;
9   }
10  istream & operator >>(istream &in, const POINT &P)
11  {
12    for(int i=0;i<P.dim;i++)
13    { In>>P(i); }
14  }
```

- Pour afficher un point:
  `cout<<P;` $\Longrightarrow$ `Point: x y z`

Flux: insertion dans une liste

Insertion d'un point dans une liste de points

```
1   class liste_POINT
2   {
3     public POINT ** liste ;          // tableau de pointeurs
4     unsigned int nb_points ;         // nombre de points
5     liste_POINT(unsigned int i=10); // constructeur
6     ~liste_POINT ( ) ;               // destructeur
7     void allonge ();                 //augmente la taille du tableau de 10
8     POINT & operator(const int &);  //acces au i eme point
9     add(const POINT &);              //ajout d'un point
10  };
11
12  liste_POINT & operator <<(liste_POINT &L,const POINT &P)
13  {
14    L.add(P);
15    return L;
16  }
```

Dans le programme principal

```
1   POINT P();
2   liste_POINT liste();
3   liste<<P;
```

Exemple complet de surcharge : définition

```
1   class POINT
2   {
3     public :
4       ...
5         double& operator()( int i);
6       POINT & operator += (const POINT &);
7       POINT & operator -= (const POINT &);
8       POINT & operator *= (const POINT &);
9       POINT & operator /= (const Point &);
10      bool operator == (const POINT &);
11      bool operator != (const POINT &);
12  };
13
14  POINT operator+(const POINT &,const POINT &);
15  POINT operator-(const POINT &,const POINT &);
16  POINT operator*(const POINT &,const double &);
17  POINT operator/(const POINT &,const double &);
18  POINT operator*(const double &,const POINT &);
19  POINT operator/(const double &,const POINT &);
20  ostream & operator <<(ostream &,const POINT &);
21  istream & operator >>(istream &,const POINT &);
```

Exemple complet de surcharge : implémentation

```
1    POINT & POINT::operator+= (const Point &Q)
2    {
3      if(dim!=Q.dim)
4      { exit(-1); // dimension incompatible }
5      POINT &P=*this; // alias sur this
6      for(int i=0;i<dim;i++)
7      { P(i)+=Q(i); }
8      return P;
9    }
10
11   POINT operator+(const Point P &,const Point &Q)
12   {
13     POINT R(P); // R initialise avec P
14     R+=Q; // addition interne
15     return R; // retourne l'objet R
16   }
17
18   bool POINT::operator == (const Point &Q){...}
19   bool POINT::operator != (const Point &Q){return ! (*this == Q);}
```

Exemple complet de surcharge : commentaires

- l'opération `P+Q` appelle `+=`
- une seule implémentation de `+`.
- plus facile à maintenir.
- plus robuste : diminue le nombre d'erreur possible.

Un exemple plus évolué : définition

- Supposons que l'on dispose d'une classe de matrices (`MATRICE`) et que l'on veuille faire une combinaison linéaire de plusieurs grosses matrices.

- Problème : le calcul de `2*A+4*D-5*C` :

  `T1=5*C` ,`T2=4*D` , `T3=T1-T2` , `T4=2*A` et enfin `T4+T3`

  soit 5 générations de matrices temporaires avec recopie !

- Solution : utiliser une structure intermédiaire (combinaison) décrivant la combinaison linéaire et n'effectuer le calcul qu'au moment du =:

  `5*C=>C1` , `4*D => C2` , `C3=C1-C2` , `2*A => C4` `C4+C3 =>` matrice résultat.

- Les opérations créent ou modifient la structure intermédiaire

- C'est l'opération `MATRICE=COMBINAISON` qui déclenche le calcul en évitant la création de matrices intermédiaires

- Une seule recopie finale !

Un exemple plus évolué : implémentation

```
1  class MATRICE //  classe MATRICE
2  {
3    ...
4  };
5
6  class COMBINAISON // pointeurs matrice et coefficients
7  {
8    public:
9      int nb_matrice;
10     MATRICE **liste_matrice;
11     double *liste_coefficients;
12     ...
13 }
14
15 COMBINAISON & operator*(double, const MATRICE &);
16 COMBINAISON & operator*(const MATRICE &, double);
17 COMBINAISON & operator+(const MATRICE &, const MATRICE &);
18 COMBINAISON & operator-(const MATRICE &, const MATRICE &);
19 COMBINAISON & operator+(const MATRICE &, COMBINAISON &);
20 COMBINAISON & operator-(COMBINAISON &, const MATRICE &);
21 COMBINAISON & operator*(COMBINAISON &, double);
22 COMBINAISON & operator*(double, COMBINAISON &);
23 MATRICE& MATRICE::operator=(COMBINAISON &);
```

Attention au mécanisme de création/destruction des objets intermédiaires combinaison!

- Ne jamais surcharger un opérateur avec un autre sens que le sens commun (+ avec -)
- limiter le nombre d'implémentations pour des opérations similaires (+ et +=)
- ne renvoyer des pointeurs qu'exceptionnellement, résultat inutilisable dans une autre opération :

  si `A+B` retourne un pointeur `(A+B)+C` est impossible

- pour une classe personnelle ne surcharger que l'essentiel
- pour une classe générale, prévoir toutes les surcharges possibles (complétude)
- la surcharge des opérateurs n'est pas obligatoire elle apporte un confort et augmente la lisibilité

Agenda

Automatisation de la compilation

Surcharge d'opérateur

Le C++ n'est pas du C

Le C++ c'est quoi

- C++ est une extension du C : ce n'est pas un langage orienté objet
- Correction de certaines limitations du C
  - Déclarations, commentaires, ...
  - allocation dynamique (**`new`**, **`delete`**)
  - conversion de type via casting
  - références ≠ pointeurs
  - surcharge de fonction et arguments par défaut
  - flux d'entrées sorties
- Extension de la notion de structure **`struct`**

Syntaxe – Déclarations

- En C++ : une variable peut être déclarée n'importe où. Elle est seulement visible dans le bloc de déclaration. On peut les déclarer dans les structures de contrôle.

- En C : une variable doit être déclarée en début de bloc. On ne peut pas les déclarer dans les structures de contrôle.

```
1   {
2           double x;
3           x=0;
4           int N=10;
5           for(int i=1;i<N;i++)
6           {
7                   double lng=log(i); lng=lng*lng ;
8                   double c=cos ( i ) ;
9                   x+=lng*c;
10          }
11  }
```

➤ Structure en C

```c
struct complexe
{
        double x,y
};
/* instantiation d'un complexe */
struct complexe z;
```

➤ Structure en C++

```cpp
struct complexe
{
        double x,y
};
// instantiation d'un complexe
complexe z;
```

➤ Déclaration d'une fonction :

```
1   struct complexe
2   {
3           double x,y;
4           void  print()
5           {printf("valeur du complexe (%f,%f)",x,y);}
6   };
7
8   }
9   int main()
10  {
11          complex Z;
12          Z.x=0;
13          Z.y=1; i
14          Z.print();
15  }
```

- En C, la conversion de type s'effectue de la manière suivante

```
1    float x;
2    int i;
3    i = (int) x;
```

- En C++
  - Pour les types principaux

```
1    float x;
2    int i;
3    i = int (x);
```

  - Pour les types dérivés

```
1    const_cast
2    dynamic_cast
3    reinterpret_cast
4    static_cast
```

- En C, utilisation de **malloc**, **calloc**, **free**, **realloc**
  il est nécessaire de connaître la taille des blocs que l'on souhaite allouer en utilisant **sizeof**

- En C++, utilisation de **new** et **delete**

```
1   {
2     double *pX,*pTab;
3     int lg=10;
4     pX=new double ; // pointeur sur un double
5     pTab=new double [ lg ] ; // pointeur sur un double ,
6     // premiere case du tableau
7
8     delete pX;      // desallocation
9     delete []   pTab;
10  }
```

- Attention à la désallocation des tableaux.

Références

- C++ introduit un nouveau concept : référence = pointeur déguisé
- Syntaxe de déclaration : Type & Identificateur; Obligatoirement initialisée à la création Ne voit que la variable pointée (alias)

```
1  {
2    int i=2;
3    int &j=i;
4    printf ( "i =% i \n" , i ) ;
5    printf("j=%i\n",j); //impression d'un int
6    int &k ;//Erreur de compilation
7  }
```

L'impression fournit 2 et 2; j pointe sur i
Si i est modifiée, j l'est également

Passage d'une référence

- On peut transmettre une référence dans une fonction

```
1    void function ajoute1(double & x) {x=x + 1;};
2    int main()
3    {
4      double x = -1;
5      ajoute1(x);
6    }
```

- Permet de modifier une variable externe à la fonction. Evite de transmettre un pointeur!

```
1    void function ajoute1(double * x) {*x=(*x) + 1;};
2
3    int main()
4    {
5      double x = -1;
6      ajoute1(&x); /*transmission du pointeur de x*/
7    }
```

- Attention : ne pas confondre
  - **`int x;`** &x pointeur sur un **`int`**
  - **`int &x;`** x référence sur un **`int`**

Valeur à gauche

- Une référence est une valeur à gauche. L'objet pointé par la référence est modifiable
- On peut rendre l'objet pointé non modifiable

```
1   void function ajoute1(const double & x) {x = x + 1;};
2
3   int main()
4   {
5     double x = -1;
6     ajoute1(x);
7   }
```

  On obtient une erreur de compilation

- Passage par référence est équivalent au passage par pointeur.
- Intérêt pour les gros objets en ajoutant la sécurité.

Retourner une référence

- Dans certains cas, il est nécessaire de retourner une référence

```
1   double vec[10];
2   double & function element(const int & i);
3   {
4     return vec[-1];
5   }
6
7   int main()
8   {
9     for(int i=1;i<=10;i++) element(i)=1;
10  }
```

- `element(i)` est une référence et permet de modifier `vec[i-1]`.
- `double function element(const int &i)` permet seulement de lire

```
1   double x = element(i);
```

Retour d'une référence

- Il ne faut pas retourner une référence sur une variable locale qui est détruite à la sortie de la fonction

```
1   double & f(double x)
2   {
3     double y;
4     ...
5       y = ...;
6     return(y);
7   }
8
9   int main()
10  {
11    double z=f(1);
12    ...
13  }
```

- Avertissement avec g++

Les nouvelles références/pointeurs

- **nullptr** : remplace NULL.
- unique_ptr : permet d'attribuer une zone mémoire à un objet.
- shared_ptr : permet de partager une zone mémoire entre plusieurs objets.
- **auto** : permet l'inférence automatique de type.

```
1   #include <iostream>
2   int main(){
3
4     int my_array[5] = {1, 2, 3, 4, 5};
5     // Boucle automatique sur le contenu d'un tableau
6     for (auto&& x : my_array) {
7       std::cout<< x << " ";
8     }
9     std::cout<<std::endl;
10    return 0;
11  }
```

- && : sémantique de déplacement mémoire.

Définition des fonctions

C++ offre de nouvelles fonctionnalités pour la définition de fonctions

- Valeur par défaut des arguments
- Surcharge des fonctions
- Nombre variable d'arguments
- Inlining

Valeur par défaut des arguments

- On peut préciser les valeurs par défaut des derniers arguments d'une fonction

```
1   //calcul de la fonction et de sa derivee
2   double fonc(double x, int der =0)
3   {
4     switch(der)
5     {
6       case 0 : ... //calcul de la valeur
7       case 1 : ... //calcul de la derivee
8     }
9   }
10  //appel de la fonction
11  int main()
12  {
13    double f,df;
14    f = fonc(0);    //calcul de la valeur en 0
15    df = fonc(0,1); //calcul de la derivee en 0
16    ...
17  }
```

- Il est possible d'utiliser la technique avec plusieurs arguments en respectant l'ordre

```
1   double fonc(double x, int der=0, double par=0){...}
```

Surcharge des fonctions

- C++ différencie les fonctions suivant leur nom et leurs arguments

```
1   double fonc(double x,double y){...} //reelle
2   int fonc(int x,int y){...} //entiere
3
4   int main(
5       {
6       double r=fonc(0.,0.); //appel de la version reelle
7       r=fonc(0,0) // appel de la version entiere
8       }
```

- Ne différencie pas le type de retour

```
1   double fonc(double x,double y){...}
2   int    fonc(double x,double y){...}
```

$\Longrightarrow$ erreur de compilation

Fonction Inline

- Le corps de la fonction est substitué à son appel dans le code.
- Evite de passer par la pile et de brancher le code.
- Surtout utile pour les petites fonctions
- Remplace la macro du C

```
1    #define carre(y)(y*y)
2    double a = 2;
3    double c = carre(a+1); //retourne la valeur 5
```

- Génère le code `a+1*a+1`, soit `2*a+1`
- En C++

```
1    double fonc(double x,double y){...}
2    int    fonc(double x,double y){...}
```

- Utilisation plus sûre

Flux d'entrées/sorties

- En C, les entrées/sorties standard : `printf()` et `scanf()`
- En C++, toujours possible.
- Opérateurs orientés objet en C++
  - Flux de sortie standard : `cout`
  - Flux d'entrée standard : `cin`

```
1   #include <iostream>
2   #include <string>
3   using namespace std
4   {
5     double pi = 3.1415926;
6     int i = 3;
7     string msg="impression_:\n";
8     cout<<msg<<pi<<","<<i; //ecriture ecran
9     int j;
10    cin>>j;  // lecture clavier
11  }
```

- Les flux sont des objets.

- Impression simple en format libre
- Les opérateurs de flux prennent en charge les types de base
- Les opérateurs de flux ne prennent pas en charge les pointeurs (**char** *)
- Les opérateurs de flux supportent le formatage.
- Extension des opérateurs à des objets quelconques.

```
1   #include <fstream>
2   #include <string>
3   using namespace std;
4   {
5     fstream sortie("fichier_sortie"); //creation
6     double pi = 3.1415926;
7     int i = 3;
8     string msg="impression_:\n";
9     sortie<<msg<<pi<<","<<i; //ecriture dans le fichier
10    sortie.close();          //fermeture du fichier
11  }
```

Utilisation de const

- **const** permet de protéger une variable en écriture
- En C, si un objet **const** ayant comme portée le fichier n'est pas explicitement défini comme **static**, alors il sera visible par l'ensemble du code. Les lignes suivantes sont équivalentes

```
1    const int          i = 1;
2    extern const int   j = 2;
```

- En C++, si un objet **const** ayant comme portée le fichier n'est pas explicitement défini comme **extern**, alors il sera visible uniquement dans le fichier. Les lignes suivantes sont équivalentes

```
1    const int          i = 1;
2    static const int   k = 3;
```

- En C++, **const** a quelques utilisations supplémentaires
  - pointeur sur un double constant **const double** * p
  - pointeur constant sur un double **double** * **const** p
  - référence et objet pointé sont constants **const double** & p