Modélisation et programmation

Cours 1

ENSIMAG 2A – MMIS

Agenda

Présentation du cours

Les outils d'aide au développement

Introduction à la notion d'objet en C++

Le C++ n'est pas du C

Organisation du cours

Equipe pédagogique

Christophe Picard – christophe.picard@univ-alpes-grenoble.fr
Bureau 174 – Bâtiment– 1er étage

Jean-Louis Roch – jean-louis.roch@imag.fr

Horaires

Lundi 11h15– 12h45 : Séances de cours en amphi.

Vendredi 8h15– 9h45 : Séances de TP par binômes.

Objectifs

- Se familiariser avec les différents concepts.
- Comprendre les concepts présentés dans le cours.
- Développer une maîtrise des concepts nécessaires aux applications.
- Appréhender divers aspects de la programmation scientifique.
- Développer un code scientifique en langage C++.
- Mettre en oeuvre des outils de travail performants.

Consignes– Généralités

- Les TP sont en binômes.
- Les rendus doivent se présenter sous la forme d'une archive au format `.tar.gz` qui créera à l'extraction un répertoire *TPi_nom1_nom2*.
- Les rapports doivent être au format `.pdf` uniquement et ne contiennent pas de code.
- Les codes doivent être réfléchis.

Consignes– Présentation du code

Le code doit :

- être facile à lire.
- s'articuler logiquement.
- être débogable.
- être transmissible.

Vous devez :

- commenter le code abondamment, précisément, clairement et immédiatement.
- indenter proprement et uniformément.
- donner des noms significatifs aux "objets" que vous manipulez.

Pourquoi le C++

- `FORTRAN` : très rapide (transfert pointeur automatique),nombreuses bibliothèques optimisées (blas), nombreux codes
  Dernière norme (F2008) : couche objet moins riche que le `C++`

- `C` : grande liberté, allocation dynamique, plus près du système, pas de variable native de type matrice, programmation complexe

- `C++` : évolution du `C` avec une couche objet complète, difficile à maitriser, un peu plus lent que le Fortran

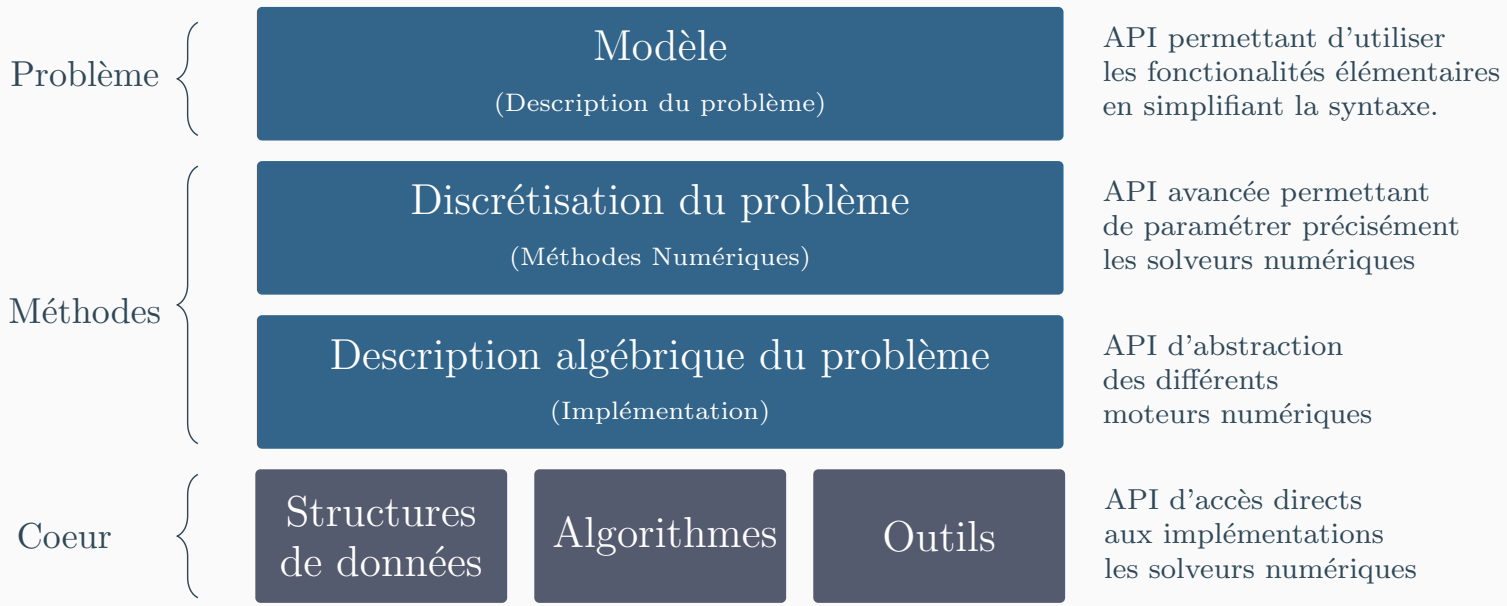
- `java` : objet pur et multi-platforme, pas de surcharge d'opérateurs, généralement plus lent que le `C++`.

- `MATLAB` : script, pas de déclaration de type, objets matriciels natifs, graphique intégré, possibilité d'objet, riche, lent et permissif

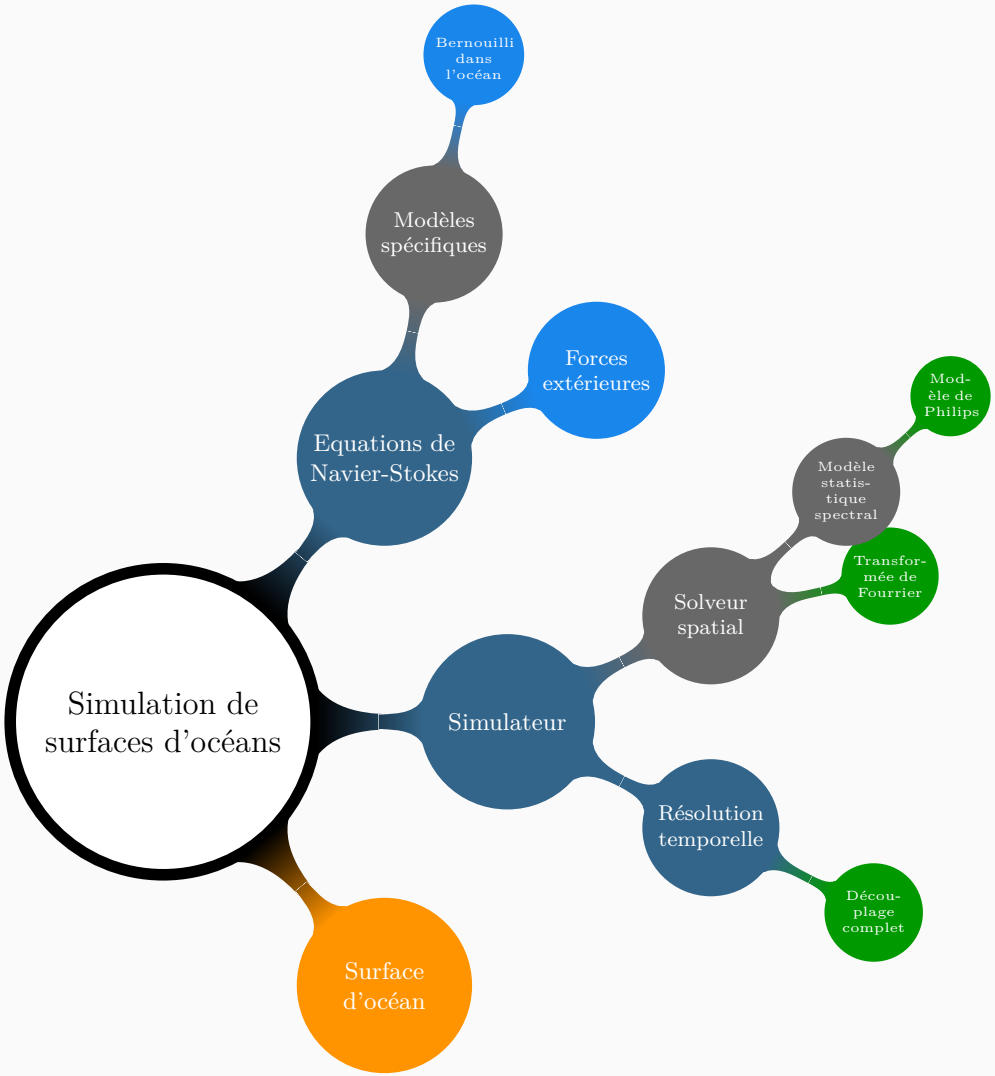
Plan Général

1. Introduction
2. Outils de programmation
3. Notion d'objets en C++
4. Opérateurs
5. Héritage
6. STL

Modélisation en calcul scientifique



Pourquoi en Finance?



Agenda

Présentation du cours

Les outils d'aide au développement

Introduction à la notion d'objet en C++

Le C++ n'est pas du C

Les éditeurs

Vous pouvez utiliser l'éditeur que vous souhaitez: vim, emacs, gedit, sublime ou autre. Mais il y a quelques règles à respecter pour être efficace :

- Une indentation cohérente, éventuellement configurable dans l'éditeur. Attention, vous ne programmez pas le noyau Linux, alors pas d'indentation à 8 caractères
- Des noms de fichiers, de fonctions et de variables significatifs et claires.
- Se limiter dans la longueur des lignes : la ligne doit être lisible d'un seul coup d'oeil.
- Utiliser la coloration syntaxique.
- Nettoyer votre code. Ne laisser pas traîner de code résiduel.
- Quelques règles de base pour rendre votre code lisible `https://google.github.io/styleguide/cppguide.html`
- Pour rendre votre code joli `http://uncrustify.sourceforge.net`
- … et surtout restez cohérents à l'intérieur d'un binôme ou d'un groupe.

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

```
PROG = calc
OBJS = calc.o stack.o
CXX = g++
CXXFLAGS = -Wall -g

$(PROG): $(OBJS)
	$(CXX) $(CXXFLAGS) -o $@ $(OBJS)
```

  Les espaces au début de la dernière ligne sont des tabulations.

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

```
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 lien?

Deboggage

Afin de trouver les erreurs dans votre code, l'utilisation de **gdb** (**ddd** pour son interface graphique) est fortement recommandée.

Quelques commandes pour **gdb**

- Lancer gdb : `gdb ./monProg`.
- Lancer le programme sous gdb avec argument et redirection de l'affichage:
  `(gdb) run arg1 arg2 > sortie`.
- Afficher les données : `(gdb) print i j`.
- Appeler une fonction du programme : `(gdb) call fonction(arg1,arg2,...)`.
- Modifier une variable : `(gdb) set variable nomVariable = expression`.
- Naviguer dans la liste des appels **up** ou**down**.

Le programme doit être compilé avec l'option `-g`.

Gestion de la mémoire

- L'un des aspects critiques de la programmation dans des langages compilés évolués est la gestion de la mémoire.
- `valgrind` est un outil qui permet de vérifier que la mémoire est gérée de façon correcte, et que toute la mémoire utilisée a bien été libérée.
- Quelques options pour `valgrind`
  - Analyser les fuites mémoires
    `valgrind --leak-check=yes myprog arg1 arg2`
  - Montrer les zones encore accessible `--show-reachable`
  - Etre bavard `-v`
  - Enregistrer dans un fichier `--log-file=filename`

Rapport d'exécution valgrind

```
==25832==  Invalid read of size 4
==25832==     at 0x8048724: BandMatrix::ReSize(int,int,int) (bogon.cpp:45)
==25832==     by 0x80487AF: main (bogon.cpp:66)
==25832==   Address 0xBFFFF74C is not stack'd, malloc'd or free'd
==25832==
==25832==  ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==25832==  malloc/free: in use at exit: 0 bytes in 0 blocks.
==25832==  malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==25832==  For a detailed leak analysis, rerun with: --leak-check=yes
```

- Le programme doit être compilé avec l'option `-g`.
- Un rapport avec erreur est la garantie d'un problème dans le code.

La documentation

- Il existe plusieurs types de documentations : besoin, conception, technique, utilisateur.
- La documentation technique peut-être générée en partie automatiquement, mais il est préférable d'être le plus exhaustif possible.
- L'un des outils pour générer la documentation technique est `doxygen`.
- Quelques commandes pour `doxygen`
  - `@param` En instrumentant votre code, vous pouvez, par exemple, spécifier les arguments retour de vos fonctions.
  - `@brief` : Résumé de description.
  - `@fn` : Documentation de fonction.
  - `@todo` : Un bon moyen de communiquer sur l'état d'un morceaux de code.
  - `@bug` : Un bug a été identifié. Donner les informations nécessaire.
- Le logiciel `doxywizard` permet de générer le fichier de configuration pour `doxygen`
- La documentation est générée par la commande `doxygen projet.cfg`.

Si vous n'instrumentez pas votre code, la documentation ne sera pas faîte pour vous.

Agenda

Présentation du cours

Les outils d'aide au développement

Introduction à la notion d'objet en C++

Le C++ n'est pas du C

- C n'est pas un sous-ensemble de C++ : certains programmes C ne sont pas valides en C++.
- Certains programmes ont un sens différent en C et C++.
- Toutes les techniques de programmation valables en C sont également valables en C++.
- Les programmes biens écrit en C sont valables en C++.
- Un programme C convertit en C++ s'exécutera aussi rapidement dans les 2 cas, et auront la même empreinte mémoire.
- C++ supporte des paradigmes multiples : programmation procédurale, programmation objet et programmation générique.
- Il est possible de n'utiliser qu'un seul paradigme pour écrire un code C++: ce choix se fait souvent au détriment de la maintenabilité et de l'élégance.

- Le C++ supporte tous les éléments de la programmation objet :
  - Encapsulation des données : les données peuvent avoir des données de visibilités différentes (publique, privée ou protégée). Avec le C++, l'encapsulation est une possibilité, elle n'est jamais forcée.
  - Abstraction : mécanisme permettant de réduire le niveau de détail d'un code. On regroupe les classes selon des caractéristiques communes. Une abstraction bien conçue est généralement simple et s'utilise facilement.
  - Héritage : les propriétés et les fonctionnalités d'une classe existante peuvent être transmise par définition à une autre classe. Les principaux concepts sont l'extensibilité et la reutilisabilité.
  - Polymorphisme : capacité de manipuler à l'exécution des objets en fonction de leur type et de leur utilisation. Le polymorphisme peut-être obtenu soit à la compilation en utilisant les surcharges d'opérateurs et de fonctions, soit à l'exécution en utilisant des fonctions virtuelles.

Afin de mettre en oeuvre les techniques de la programmation orientée objet, le C++ utilise les techniques suivantes:

- Les classes et la surcharge d'opérateur et de fonctions pour le polymorphisme
- Les espaces de noms.
- Les exceptions qui permettent de transférer le contrôle du programme à des fonctions spécifiques.
- La conversion de types complexes.
- Les signaux.
- Les flux.
- La gestion de la mémoire dynamique.
- Les patrons.

Notion d'objet

Une classe d'objet est une nouvelle structure définie par l'utilisateur et qui encapsule des membres

- des données
- des fonctions

Le concepteur doit assurer

- l'autoconsistance
- la robustesse
- la généricité

La conception nécessite une réflexion.

Déclaration

- Un objet est un nouveau type de variable C++ défini par l'utilisateur.
- Déclaration avec **`struct`** ou **`class`**

```
1   class POINT
2   {
3     public :
4       int dim;
5       double x,y,z;
6       void print()
7       {cout<<"coordonnees : "<<x<<" "<<y<<" "<<z;}
8   };
9   int main()
10  {
11    POINT P; //instanciation d'un objet POINT
12    P.dim = 3;
13    P.x = 0; P.y=0; P.z=0;
14    P.print();
15    POINT *pQ = &P; //pointeur sur le POINT P
16    pQ->x=1.;
17  }
```

- Opérateur d'accès à un membre : `Objet.membre`.
- Pointeur sur un objet : `pObjet->membre`.

Allocation

- Lors de l'instanciation d'un objet, il y a allocation automatique de l'espace nécessaire pour stocker les membres. Par exemple, pour la classe `POINT`, il y a une allocation de 8 octets pour `dim` et 3×16 octets pour `x, y, z`.
- Lorsque l'objet est détruit, il y a libération automatique de l'espace alloué.
- Attention, la destruction ne libère pas l'espace alloué par un pointeur.

```
1   class POINT
2   {
3     public :
4       int dim;
5       double *pCor;
6   };
7
8   int main()
9   {
10    POINT P; // instanciation d'un objet POINT
11    P.dim=3;
12    P.pCor = new double[P.dim];
13  }
```

Protection des attributs

Un membre d'une classe peut-être déclaré

- **public** accessible partout, par tout le monde
- **private** accessible seulement par la classe et dans la classe
- **protected** accessible dans la classe et par des relations d'héritage.

```
1   class POINT
2   {
3     public :
4       int dim;
5     private :
6       double *pCor;
7   };
8
9   int main()
10  {
11    POINT P; // instanciation d'un objet POINT
12    P.dim=3;
13    P.pCor = new double[P.dim]; // Erreur
14  }
```

- L'allocation doit être gérée par la classe et non par l'utilisateur.

Accesseur

- La protection **private** sert à renforcer la sécurité du code en limitant l'accès à des données fondamentales de la classe

```
1   class POINT
2   {
3     private :
4       double *pCor;
5     public :
6       int dim;
7       void alloc(int d)      // allocation
8       {
9         dim = d;
10        pCor = new double[P.dim];
11      }
12      double & val(int i)
13      {
14        if(i<1 || i>dim) exit(-1); // ERREUR indice hors limite
15        return pCor[i];
16      } };
17
18  int main()
19  {
20    POINT P; // instanciation d'un objet POINT
21    P.alloc(3);
22    cout<<P.val(1);
23  }
```

Constructeur

- Il est fortement conseillé d'initialiser un objet à l'aide d'un constructeur **class** (arguments)

```
1   class POINT
2   {
3     public :
4       int dim;
5       POINT(int d); // declaration du constructeur
6     private :
7       double *pCor;
8   };
9
10  POINT::POINT(int d) //implementation externe
11  {
12    dim = d;
13    pCor = new double[dim]; //allocation
14    for(int i=0;i<dim;i++) pCor[i] = 0;
15  }
16  int main()
17  {
18    POINT P(3); // creation d'un POINT  3D
19    POINT M=POINT(3);  //creation d'un second POINT 3D
20    POINT *pQ = new POINT(3); // pointeur sur un POINT
21  }
```

- Un constructeur est généralement déclaré **public**

Surcharge de constructeur

- Un constructeur peut-être surchargé.

```
1   class POINT
2   {
3     double *pCor;                // private par defaut
4     public :
5     unsigned int dim;
6     POINT(int d, double v=0); //constructeur (dimension, val)
7     POINT(const POINT & P);   // constructeur par copie
8   };
9   POINT::POINT(unsigned int d, double val) // dimension, valeur
10  {
11    dim=d ;
12    if (dim == 0) return ;
13    pCor=new double[dim];
14    for(int i=0;i<dim; i++)
15      pCor[i]=val;
16  }
17  POINT::POINT(const POINT & P) // par copie
18  {
19    dim=P.dim;
20    if (dim == 0) return ;
21    pCor=new double[dim];
22    for(int i=0;i<dim;i++)
23      pCor[i]=P.pCOR[i]; //recopie
24  }
```

Constructeur par copie

- Par défaut, un constructeur par recopie est toujours créer. Mais ce constructeur se contente de recopier les membres bit à bit.
- Il ne se préoccupe pas des zones mémoires allouées par l'utilisateur.
- Le constructeur par recopie est invoqué implicitement lors du transfert d'objet comme argument d'entrée ou de retour.

```
1   double distance(POINT P)
2   {
3     double d=0;
4     for(int i=1; i<=P.dim; i++)
5       d+=P.pCor[i]*P.pCor[i];
6     return sqrt(d);
7   }
8
9   int main()
10  {
11    POINT P(3,1.);
12    POINT M(P);
13    POINT Q=M;
14    double d = distance(Q)
15  }
```

- `POINT P=Q` appelle le constructeur par copie.

Destructeur

- Lorsqu'un objet est détruit, il libère l'espace qu'il a crée pour stocker des membres mais pas l'espace alloué par l'utilisateur. On peut à l'aide d'un destructeur libérer cette espace mémoire.

```
1   class POINT
2   {
3     double *pCor;
4     public:
5     int dim;
6
7     POINT(int d);
8     ~POINT();
9   }
10
11  POINT::~POINT()
12  {
13    delete [] pCor;
14  }
```

- Un destructeur n'a jamais d'arguments et est public.

Fonctions membres

- On peut déclarer dans une classe autant de fonctions que l'on souhaite.
- S'il n'y a pas de protection spéciale, ces fonctions ont accès à toutes les données de la classe
- On peut renvoyer une autoréférence

```
1   class POINT
2   {
3     double *pCor;
4     public:
5     int dim;
6
7     POINT(int d);
8     ~POINT();
9
10    POINT & rotate_xy(double theta)
11    {
12      double ct = cos(theta), st = sin(theta);
13      double x = pCor[1], double y = pCor[2];
14      pCor[1] = x*ct-y*st; pCor[2] = x*st+y*ct;
15      return *this;
16    }
17  };
```

- **this** est le pointeur sur l'objet en cours. ***this** est l'objet lui même.

On peut, dans certains cas, utiliser **`struct`** au lieu de **`class`**

- Par défaut les membres de **`struct`** sont publics
- Il est obligatoire de définir en premier les données membres dans une **`struct`**
- **`struct`** est conseillé pour les types simples.

```
class Complex {
  public:
    double & reel(){return x_;}
    double & imag(){return y_;}
    double module()
    {
      return sqrt(x_*x_+y_*y_);
    }
  private :
    double x_, y_;
}
```

```
struct complex {
  double x,y;
  double module()
  {
    return sqrt(x*x+y*y);
  }
}
```

- Afin de rendre le code plus lisible et modulable, il est souvent préférable de séparer la définition de la classe de son implémentation

```
1   #ifndef POINTH
2   #define POINTH
3
4   class POINT {
5     double *pCor;
6     public:
7     int dim;
8     POINT(int d, double v=0);
9     POINT (const POINT &P);
10  };
11  #endif
```

```
1   #include "point.h"
2   POINT::POINT(int d,double v)
3   {
4     dim = d;
5     pCor = new double[dim];
6     for(int i=0; i<dim; i++)
7       pCor[i] = val;
8   }
9
10  POINT::POINT(const POINT &P)
11  {
12    dim = P.dim;
13    if (dim=0) return;
14    pCor = new double[dim]
15      for (int i=0;i<dim; i++)
16        pCor[i] = P.pCor[i];
17  }
```

L'implémentation dans l'entête correspond à un **`inline`** implicite.

- Cette séparation permet également d'améliorer l'abstraction.

Agenda

Présentation du cours

Les outils d'aide au développement

Introduction à la notion d'objet en C++

Le C++ n'est pas du C

Le C++ c'est quoi

- C++ n'est pas une extension du C, mais ce n'est pas un langage orienté objet
- Correction de certaines limitations du C
  - déclarations, commentaires, ...
  - allocation dynamique (**`new`**, **`delete`**) grâce à la vérification de type
  - 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`**

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.

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};
/* instanciation d'un complexe */
struct complexe z;
```

- Structure en C++

```cpp
struct complexe
{double x,y};
// instanciation 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;Z.y=1; Z.print();
13  }
```

- 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
```

Allocation Dynamique

- En C, pour l'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 libération 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); // Affichage d'un int
6    int &k; // Erreur de compilation
7  }
```

L'affichage produit 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
3  int main()
4  {
5    double x = -1;
6    ajoute1(x);
7  }
```

- 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 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++

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   inline double carre(double y){return y*y;}
```

- Utilisation plus sûre.

- 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  }
```

Protection en écriture

- **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