

Séance 2

Programmation C++

Notions de bases. Fonctions

UE Physique numérique. M1 Physique.

M. Ismail, PHITEM, Université Grenoble Alpes

2.1

Table des matières

1	Fonctions	1
1.1	Déclaration	1
1.2	Définition	2
1.3	Transmission d'arguments	2
1.4	Valeurs de retour	4
1.5	Fonctions mathématiques	6
1.6	Fonctions récursives	6
1.7	Variables locales de la classe static	7
1.8	Surdéfinition de fonctions	8
2	Gestion de projets. Compilation séparée	9
2.1	Makefile	9
2.2	Fichiers d'en-tête (<i>headers</i>)	10
3	Mécanismes d'abstraction. Les structures	10

2.2


1 Fonctions

1.1 Déclaration

Déclaration

Prototype

```
type_de_retour identificateur (liste_de_paramètres);
```

 Exemples :

```
int main ();  
int main (int argc, char* argv[]); // main avec arguments de  
                                   // ligne de commande  
double func_1 (double x);  
void func_2 (double x);
```

Prototype complet

```
double func_4 (int nb_iter, double initial, double solution);  
double func_4 (int, double, double);
```

2.3

1.2 Définition

Définition d'une fonction

En-tête (prototype complet)


```
type_de_retour identificateur (liste_de_paramètres)
```

Le corps d'une fonction

```
{  
    ...  
    return expression; // si type_de_retour n'est pas void  
}
```

2.4

Exemple de définition d'une fonction

 Exemple :

```
double abs_som (double u, double v)  
{  
    double result = u + v;  
    if(result > 0) return result;  
    else return -result;  
}
```

Ordre de déclaration

Pour être utilisée, une fonction doit être déclarée avant son utilisation

2.5

1.3 Transmission d'arguments

Passage par valeur

- `fonc` est une fonction qui prend un `double` comme argument et qui retourne un `double` :

```
double fonc(double ); //Déclaration de la fonction fonc
```


- Lors de l'appel de la fonction `fonc`, **seule la valeur de l'argument est transmise**.

```
double x = 1;  
double y = fonc(3*x+2); // utilisation de la fonction fonc
```

- Dans cet exemple c'est la valeur 5 qui est transmise en argument à la fonction `fonc`


2.6

Passage par valeurs. Exemple

 échange de variables

```
void echange (int a, int b)  
{  
    int c;  
    cout << "debut echange " << a << " " << b << endl;  
    c = a;  
    a = b;  
    b = c;  
    cout << "fin echange " << a << " " << b << endl;  
}
```

lechange.C2.cpp

 Utilisation

```
void echange (int a, int b); // declaration of the function  
int n = 1, p = 2;  
  
cout << "avant appel " << n << " " << p << endl;  
echange (n,p);  
cout << "apres appel " << n << " " << p << endl;
```

🕒 Vérification :

```
avant appel 1 2
debut échange 1 2
fin échange 2 1
apres appel 1 2
```

🚩 Pas d'échange à la sortie de la fonction !! Que s'est-il passé?

2.7

Transmission par pointeurs. Exemple

🔗 échange de variables

```
void échange (int* a, int* b)
{
    int c;
    cout << "debut échange " << *a << " " << *b << endl;
    c = *a;
    *a = *b;
    *b = c;
    cout << "fin échange " << *a << " " << *b << endl;
}
```

lechangeAdr.C2.cpp1

🔗 Utilisation

```
void échange (int* a, int* b);
int n = 1, p = 2;

cout << "avant appel " << n << " " << p << endl;
échange (&n, &p);
cout << "apres appel " << n << " " << p << endl;
```

lechangeAdr.C2.cpp1

🕒 Vérification :

```
avant appel 1 2
debut échange 1 2
fin échange 2 1
apres appel 2 1
```

🚩 échange réussi.

2.8

Transmission par Références. Exemple

🔗 échange de variables

```
void échange (int& a, int& b)
{
    int c;
    cout << "debut échange " << a << " " << b << endl;
    c = a;
    a = b;
    b = c;
    cout << "fin échange " << a << " " << b << endl;
}
```

lechangeRef.C2.cpp1

🔗 Utilisation

```
void échange (int& a, int& b); // declaration of the function
int n = 1, p = 2;

cout << "avant appel " << n << " " << p << endl;
échange (n, p);
cout << "apres appel " << n << " " << p << endl;
return 0;
```

🔗 Vérification :

```
avant appel 1 2
debut echange 1 2
fin echange 2 1
apres appel 2 1
```

🔗 échange réussi.

2.9

Tableaux transmis en arguments

🔴 Rappel

L'appel `f(tab);` est équivalent à `f(&tab[0]);`

🔴 Remarque

Seule l'adresse du premier élément d'un tableau est transmis en argument

Définition

Si `tab` est un tableau d'entiers, le prototype d'une fonction compatible peut être `void f(int* t);`

ou `void f(int t[]);` ou `void f(int taille, int t[]);`

2.10

Arguments par défaut

🔗 Exemple de déclaration et d'appels :

```
// Déclaration de la fonction func
void func(int n, int p = 1); //Prototype avec une valeur
                             //par défaut
// Exemples d'appels de la fonction func
func(i, j); // Appel classique
func(i);    // Appel avec un seul argument ⇔ func(i,1)
func();     // Illégal car le premier argument n'a pas
             // de valeur par défaut
```

🔗 Exemple de définition :

```
void func (int n, int p) //en-tête habituelle
{
    //corps de la fonction
}
```

— les valeurs par défaut peuvent dépendre d'autre variables

```
int m;
// Suite d'instructions attribuant une valeur a m
void func (int n, int p = 2*m+1); // La valeur par défaut
                                   // dépend de m
```

2.11

1.4 Valeurs de retour

Valeurs de retour

🔴 Rappel

Une fonction dont le type de retour n'est pas `void` doit contenir une instruction `return`

Remarque

`type_de_retour = T` ↗ `type_de_retour = T*, T&`

🔴 Attention !

Retourner un pointeur ou une référence sur une variable automatique locale à la fonction est une erreur. Pourquoi ?

2.12

Construction et destruction de variables/objets

- ❌ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve
- ❌ Un **objet dynamique** est créé par l'opérateur **new**. Il n'est supprimé que lorsque **delete** est appelé
- ❌ Un objet **statique local** est construit à la première rencontre entre le flux de contrôle et la définition de l'objet (voir exemple fonction factoriel recursive). Il est détruit à la fin du programme.
- ❌ Une **variable globale** est une variable définie en dehors de toute fonction. Elle est initialisée avant l'appel du `main()` et détruite après la fin de son exécution.

2.13

Retour par pointeur. Exemple

Tirage aléatoire de 10 nombres réels et leur normalisation entre 0 et 1.

📄 Fichiers d'en-têtes :

```
1 #include <iostream> // Vous y êtes déjà habitués
2 #include <iomanip>   // pour la fonction setw
3 #include <cstdlib>  // pour la fonction rand
```

|funcRetourPoint.C2.cpp|

📄 Une variable globale :

```
1 const int taille = 10;
```

📄 La fonction `main`. Déclaration des fonctions :

```
1 void hasard (double max, int size, double tab[]);
2 void affiche(int size, const double tab[],
3             const string & titre = "Valeurs ",
4             int largeur = 10, int par_ligne = 5);
5 double * largest (int size, double *tab);
6 double * smallest(int size, double *tab);
```

📄 La fonction `main`. Suite des instructions :

```
1 double vec[taille] = {0};
2
3 double maximum = 0.;
4 cout << "Entrez la valeur d'initialisation maximale " << endl;
5 cin >> maximum;
6
7 hasard(maximum, taille, vec);
8 cout << endl;
9 affiche(taille, vec, "Valeurs initiales", 12);
10
11 double min = *smallest (taille, vec);
12
13 // Décale les valeurs de sorte à annuler la plus petite
14 for(int i=0; i<taille; i++) vec[i] -= min;
15
16 double max = *largest(taille, vec);
17 // Renormalise les valeurs à 1
18 for(int i=0; i<taille; i++) vec[i] /= max;
19
20 affiche(taille, vec, "Valeurs renormalisees");
21 return 0;
```

🚀 Exécution :

```
1      Entrez la valeur d initialisation maximale
2      2
3
4      Valeurs initiales
5      1.68038  0.788766   1.5662   1.59688   1.82329
6      0.395103  0.670446   1.53646   0.555549   1.10794
7
8      Valeurs renormalisees
9      0.89993   0.275637   0.819985   0.841468           1
10     0    0.192791   0.799162   0.112343   0.499119
```

Quelques Détails sur les fonctions utilisées :

Fonction hasard

```
1 void hasard (double max, int size, double * tab)
2 {
3
4     for (int i=0; i<size; i++)
5         tab[i] = double(rand()) * max / RAND_MAX;
6 }
```

Quel est le type de retour ?

Fonction largest

```
1 //trouver l'adresse de l'élément possédant la plus grande valeur
2 double * largest(int size, double * tab)
3 {
4     int indexMax = 0;
5     for(int i=0; i<size; i++)
6         indexMax = tab[indexMax] < tab[i] ? i : indexMax;
7     return &tab[indexMax];
8 }
```

Quelle est la valeur retournée ?

Fonction smallest Idem !

Fonction affiche

```
1 void affiche(int size, const double * tab, const string & titre,
2             int largeur, int par_ligne)
3 {
4     cout << endl << titre;
5     for(int i=0; i<size; i++)
6     {
7         if(!(i%par_ligne)) cout << endl;
8         cout << setw(largeur) << tab[i];
9     }
10    cout << endl;
11 }
```

funcRetourPoint.C2.cpp Identifier les appels de la fonction affiche dans le main. Commenter !

2.14

1.5 Fonctions mathématiques

Bibliothèque standard de fonctions mathématiques

```
double abs(double); //valeur absolue
double ceil(double d); // plus petit entier >= d
double floor(double d); //plus grand entier >= d
double sqrt(double); //racine carrée
double pow(double d, double e); // d à la puissance e
double pow(double d, int n); // d à la puissance n
double exp(double); //exponentielle
double log(double); //logarithme neperien
double cos(double); //cosinus
double sin(double); //sinus
double tan(double); //tangente
double acos(double); //arccosinus
double asin(double); //arcsinus
double atan(double); //arctangente
```

2.15

1.6 Fonctions récursives

Un exemple

— Une fonction peut-elle faire appel à elle-même ?

```

long double factoriel (int n)
{
    if (n == 0) return 1;
    return n*factoriel(n-1);
}

```

|factoriel.C2.cpp|

— Version non réursive :

```

long double factoriel (int n)
{
    if(n == 0) return 1;
    long double result = n;
    for( ; n>1; ) result *= --n;
    return result;
}


```

|factoriel.2.C2.cpp|

2.16

1.7 Variables locales de la classe static

Combien de fois la fonction `factoriel` est-elle appelée ?

 Rappel de la fonction `factoriel` :


```

long double factoriel (int n)
{
    int compteur = 1;
    if (n == 0) return 1;
    cout <<"la fonction factoriel est appele " <<compteur++
        <<" fois"<<endl;

    return n*factoriel(n-1);
}

```

|factoriel.C2.cpp|


 Exemple d'utilisation :

```

int main ()
{
    int n;
    cout <<"Donnez un entier positif n = ";
    cin >> n;
    cout <<n<<"!= " << factoriel(n) <<endl;
    return 0;
}

```


|factoriel.C2.cpp|

 Exécution :

```

Donnez un entier positif n = 3
la fonction factoriel est appele 1 fois
la fonction factoriel est appele 1 fois
la fonction factoriel est appele 1 fois
3!= 6

```

 Ajout du mot-clé `static`

```

long double factoriel (int n)
{
    static int compteur = 1;
    if (n == 0) return 1;
    cout <<"la fonction factoriel est appele " <<compteur++
        <<" fois"<<endl;

    return n*factoriel(n-1);
}

```

|factoriel.C2.cpp|



Donnez un entier positif $n = 3$
 la fonction factoriel est appele 1 fois
 la fonction factoriel est appele 2 fois
 la fonction factoriel est appele 3 fois
 $3! = 6$



La variable `compteur`, de type `static`, n'est initialisée qu'une seule fois! (à la première rencontre entre le flux de contrôle et la définition de la variable)

2.17

1.8 Surdéfinition de fonctions

Surcharge (ou surdéfinition) de fonctions

Peut-on créer des fonctions (qui portent le même nom) et qui agissent différemment en fonction du type des objets passés en arguments?

On considère cette implémentation d'une fonction puissance qui calcule x^a :

```
double puissance (const double x, const int a)
{
    double result = 1;
    if(a>0)
    {
        for(int i=0; i<a; i++)
            result *= x;
        cout << "Appel de la fonction puissance (double, int) : ";
        return result;
    }
    else if(x!=0)
    {
        for(int i=0; i<-a; i++)
            result *= 1./x;
        cout << "Appel de la fonction puissance (double, int) : ";
        return result;
    }
    else
    {
        cout << "Appel de la fonction puissance (double, int) : ";
        cout << "Indetermination!!" << endl;
        exit(1);
    }
}
```

`lsurchargePuissance.C2.cpp`

On teste notre fonction avec ces appels :

```
double x = 2.;
int a1 = -1;
double a2 = 0.5;
cout << x << "^" << a1 << " = " << puissance(x,a1) << endl;
cout << x << "^" << a2 << " = " << puissance(x,a2) << endl;
int y = 2;
cout << y << "^" << a1 << " = " << puissance(x,a1) << endl;
cout << y << "^" << a2 << " = " << puissance(x,a2) << endl;
```

`lsurchargePuissance.C2.cpp`

Résultat :

```
1 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
2 Appel de la fonction puissance (double, int) : 2^0.5 = 1
3 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
4 Appel de la fonction puissance (double, int) : 2^0.5 = 1
```

Commenter les résultats des lignes 2 et 4

On ajoute une autre fonction qui porte le même nom `puissance` et qui calcule $x^a = e^{a \log(x)}$ pour $x \in \mathbb{R}_+^*$:


```
double puissance (const double x, const double a)
{
    if(x>0)
    {
        cout << "Appel de la fonction puissance (double, double) : ";
        return exp(a*log(x));
    }
    else
    {
        cout << "Appel de la fonction puissance (double, double) : ";
        cout << "Indetermination!!" << endl;
        exit(1);
    }
}
```

lsurchargePuissance.C2.cpp1

🔗 Résultat :

```
1 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
2 Appel de la fonction puissance (double, double) : 2^0.5 = 1.41421
3 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
4 Appel de la fonction puissance (double, double) : 2^0.5 = 1.41421
```

🔗 Que va-t-il se passer avec cet appel ?

```
int y1 = 2.; long int a3 = 2;
cout << y1 << "^" << a3 << " = " << puissance(y1,a3) << endl;
```

lsurchargePuissance.C2.cpp1

🔗 Compilation :

```
surchargePuissance.C2.cpp: In function 'int main()':
surchargePuissance.C2.cpp:20: error:
call of overloaded 'puissance(int&, long int&)' is ambiguous
candidates are: double puissance(double, int)
                double puissance(double, double)
```

2.18

Conversion explicite. Mot-clé `static_cast`

🔗 On indique explicitement quel genre de conversion on veut effectuer

```
int y1 = 2.; long int a3 = 2;

cout << y1 << "^" << a3 << " = " <<
    puissance(static_cast<double>(y1),static_cast<int>(a3)) << endl;
```

lsurchargePuissance.C2.cpp1

🔗 Exécution :

```
Appel de la fonction puissance (double, int) : 2^2 = 4
```

2.19

2 Gestion de projets. Compilation séparée

2.1 Makefile

Exemple simple de Makefile

Comment compiler un projet contenant plusieurs fichiers sources et utilisant une (ou plusieurs) bibliothèques externes ?

```
IDIR = /usr/include/qt4
ODIR =.
Cxx = g++
CFLAGS = -I$(IDIR) -Wall
LIBS = -L/usr/lib -lQtCore
OBJ = main.o fic_1.o fic_2.o

# Qques variables internes :
# $$ : nom de la cible
```

```
%o : %.cpp                                # $< : nom de la 1ere dependance
$(Cxx) -c -o $@ $< $(CFLAGS)             # $? : liste des depend.+ recentes
mon_exe : $(OBJ)                          #      que la cible
$(Cxx) -o $@ $^ $(LIBS)                  # $^ : liste des dependances
clean :
    rm -f $(ODIR)/*.o *~ core
```

!Makefile! **Exécution : make**

```
g++ -c -o main.o main.cpp -I/usr/include/qt4 -Wall
g++ -c -o fic_1.o fic_1.cpp -I/usr/include/qt4 -Wall
g++ -c -o fic_2.o fic_2.cpp -I/usr/include/qt4 -Wall
g++ -o mon_exe main.o fic_1.o fic_2.o -L/usr/lib -lQtCore
```

2.20

2.2 Fichiers d'en-tête (*headers*)

Fichiers d'en-tête

Définition

fichiers .h ou .hpp destinés à contenir les déclarations de fonctions

- Ils sont à inclure dans les fichiers sources .c ou .cpp :

```
#include "nomFichier.hpp"
```

- Pour éviter les doublons, on utilise les directives (**#ifndef**, **#define**, **#endif**)

```
#ifndef FIC_HPP
#define FIC_HPP
// contenu du fichier en-tête fic.hpp
#endif // FIC_HPP
```

Remarque

Ces fichiers sont compilés implicitement par le compilateur. On ne les met pas dans le Makefile.

2.21

3 Mécanismes d'abstraction. Les structures

Modéliser une information

Syntaxe générale

```
struct identificateur
{
    type_1 identificateur_1;
    type_j identificateur_j;
    type_n identificateur_n;
};
```

Exemple. Les nombres complexes

- $z \in \mathbb{C} \Leftrightarrow z = (Re, Im) \in \mathbb{R}^2$
- Création d'une Structure Complexe composée de deux **double**
- Complexe=(**double**, **double**)

🔴 ➡ Exemple de conception.

🔵 déclaration de la structure :

```
struct Complexe
{
    double Re;
    double Im;
};
```

!complexe.C2.cpp!

Utilisation :

```
int main()
{
    Complexe z1; // Declaration d'une variable  $z_1 \in \mathbb{C}$ 
    Complexe z2 = {1,1};
    Complexe z3 = {1};
    Complexe z4 = z2; // equivalent a Complexe z4(z2);
    z4.Im = z4.Re + 2.*z3.Im;
    Complexe * ptr = &z1;
    (*ptr).Re = 2.;
    (*ptr).Im = 1.; //  $z_1 = 2+i$ 
    cout << "(*ptr).Re = " << (*ptr).Re << endl;
    ptr->Re = 3.;
    ptr->Im = 4.; //  $z_1 = 3+4i$ 
    cout << "ptr->Im = " << ptr->Im << endl;
    return 0;
}
```

lcomplexe.C2.cpp1

2.22

Structures et fonctions

Objectifs

- Écriture de fonctions manipulant des nombres complexes :
 - une fonction qui conjugue un nombre complexe
 - une fonction qui calcule la somme de deux nombres complexes
 - une fonction qui affiche un nombre complexe sous la forme $a + ib$
- Première approche. Des fonctions “standards” :

fonction addition :

```
Complexe addition(const Complexe & z1, const Complexe & z2)
{
    Complexe result;
    result.Re = z1.Re + z2.Re;
    result.Im = z1.Im + z2.Im;
    return result;
}
```

lstructFunc.C2.cpp1

fonction conjugue :

```
Complexe conjugue(Complexe z)
{
    z.Im = -z.Im;
    return z;
}
```

lstructFunc.C2.cpp1

fonction affiche :

```
void affiche(const Complexe & z)
{
    cout << z.Re;
    if(z.Im > 0) cout << " + i ";
    else if(z.Im < 0) cout << " - i ";
    if(z.Im) cout << abs(z.Im) << endl;
}
```

lstructFunc.C2.cpp1

Utilisation :

```
Complexe conjugue(Complexe z);
Complexe addition(const Complexe & z1, const Complexe & z2);
void affiche(const Complexe & z);

Complexe z1 = {1.,2.};
```

```

Complexe z2 = conjugue(z1);
Complexe z3;
z3 = addition(z1,z2);
affiche(z1); affiche(z2); affiche(z3);

```

|structFunc.C2.cpp|

🔗 Exécution :

```

1 + i 2
1 - i 2
2

```

2.23

Fonctions membres

Définition

Fonctions membres (ou **méthodes**) : “attacher” des fonctions à une structure

🔗 Déclaration de la structure :

```

struct Complexe
{
    // declarations des donnees membres
    double Re;
    double Im;
    //declaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};

```

|structFuncMbre.C2.cpp|

— Définitions des fonctions membres (méthodes) :

🔗 fonction affiche

```

void Complexe::affiche()
{
    cout << Re;
    if(Im > 0) cout << " + i ";
    else if(Im < 0) cout << " - i ";
    if(Im) cout << abs(Im)<< endl;
}

```

|structFuncMbre.C2.cpp|

🔗 fonction conjugue

```

Complexe Complexe::conjugue()
{
    Complexe result;
    result.Re = Re;
    result.Im =-Im;
    return result;
}

```

|structFuncMbre.C2.cpp|

🔗 fonction addition

```

Complexe Complexe::addition(const Complexe & z)
{
    Complexe result = {Re+z.Re, Im+z.Im};
    return result;
}

```

|structFuncMbre.C2.cpp|

🔗 Utilisation :

```
int main ()
{
    void affiche();
    affiche();

    Complexe z1 = {1.,0.};
    z1.affiche();
    cout << endl;
}
void affiche(){ cout << " Voici un exemple " << endl; }
```

lstructFuncMbre.C2.cpp1

🔊 Exécution :

```
Voici un exemple
1
```

— Deux fonctions `affiche()` ??

2.24

Pointeur d'auto-référence : `this`

🛑 Argument implicite des fonctions membres. Pointeur sur l'objet pour lequel la fonction a été appelée.

🔗 Exemple :

```
Complexe Complexe::addition(const Complexe & z)
{
    Complexe result = {this->Re + z.Re, this->Im + z.Im};
    return result;
}
Complexe Complexe::conjugue_moi()
{
    Im = -Im;
    return *this;
}
```

🔗 Utilisation :

```
Complexe z4 = {2., 3.}; // z4 = 2 + 3i
z4.conjugue_moi(); // z4 = 2 - 3i
```

2.25