Cours d'introduction au C++

Programme du cours

Contenu théorique: 10 H CM

- Rappels sur le C (CM 2h)
- Ajout du C++ par rappoort au C (CM 2h)
- Programmation objet, les classes (CM 2h)
- Programmation objet, l'héritage (CM 2h)
- Programmation objet, le polymorphisme (CM 2h)

Contenu TD: 16h TD (Moitié Tableau/Moitié Machine)

Contenu Pratique: 21H TP

- 1 projet: Jeu de la vie
- 1 projet: Jeu vidéo RPG

Documents en ligne

- Mes éléments cours
- Des annales
- Des éléments de cours extérieurs et complémentaires
- Les sujets de TP

Cours 1: Rappels sur le C

Ce cours ne se concentre pas sur l'algorithmique. Ici vous n'apprendrez pas à faire de l'algorithmique. Le seul moyen d'apprendre l'algorithmique est de pratiquer l'algorithmique.

Rappel sur les éléments constituant un programme

1 Ecriture d'un programme: Un programme est un ensemble de variables et de fonctions. Une fonction prend en entrée des variables qui sont des paramètres ou attributs de la fonction, réalise une action et renvoie d'autres variables. C'est aussi un langage permettant aux humains de faire faire aux machines ce qu'ils souhaite éxecuter. **On doit connaitre toutes les rigidités du codage (; en fin de ligne,...). On doit pratiquer pour connaitre l'algorithmique.**

2 Interprétation d'un programme par un Compilateur: Le compilateur se charge de transformer le langage "humain" (le code) en langage machine (programme executable). Le compilateur analyse le code et renvoie des messages d'erreurs ou d'avertissement pour determiner si le code est interprétable ou non. Tous les compilateurs (même les différents compilateurs C++) n'interprètent pas la même chose. Certaines ecritures sont possibles avec certains compilateurs et pas avec d'autres. Certaines choses sont possible avec tous les compilateurs. Si erreur il y a, le compilateur renvoie à l'utilisateur des messages d'erreurs afin qu'il puisse comprendre ce qui ne va pas. **La aussi, l'utilisateur doit pratiquer.**

3 Execution du programme: C'est le moment où physiquement, le programme s'execute, le code crée des variables mémoires, et les utilise concrètement. Il peut la aussi apparaitre des erreurs, qui sont le résultat de commandes interprétables mais fausses (souvent erreurs de pointeurs).

Les notions essentielles du C/C++

Les variables

En C, avant d'utiliser une variable, il faut la déclarer. Il existe différents types de base: `int a ;// Entier (1 2 3 4 ...)`
`float b; //réels (1.4 , 1.56754,...)`
`double c; //réels à double précision`
`bool e; //Booléens (TRUE or FALSE, 0 ou 1) char c; //Caractères`

Lorsqu'on déclare une variable, on peut lui assigner directement une valeur (`int a= 2;`). Attention, il faut bien faire attention au type de variable lorsqu'on effectue une opération. Par exemple, la division de deux entiers est un entier:

```
int a=2;
int b=3;
int c;
c=b/a;
Attention ici c=0, car en int, le résultat de 2/3 est 0.
```

Lors de l'execution du programme, lors de la déclaration d'une variable, le code va générer l'allocation de cases mémoires liées à ces variables et les effacer lors de la fin d'exuction du programme.

Les variables constantes

Pour tout type de variable il est possible de définir des variables constantes. Ces variables ne peuvent jamais être modifiées après leur déclaration: `const double PI=3.14159;`

Les tableaux de variables

Il est possible de créer des tableaux contenant plusieurs variables avec une taille donnée.

```
int T[10]; //tableau de 10 entiers
float T2[12];//tableaux de 12 réels
...
```

On peut accéder aux cases d'un tableau en indiquant quelle case lire. Attention ces cases vont de `0` à `taille -1`:

```
int T[10]; //tableau de 10 entiers
T[0]=10; //1ere case vaut 10
T[3]=2; //4eme case vaut 2
T[10]=2;//ERREUR, il n'y a pas 11 cases
```

ATTENTION l'objet `T` n'est pas un entier. `T[i]` est un entier mais pas `T`. `T` est un **pointeur** d'entiers. On va maintenant redéfinir les pointeurs.

Pointeurs

Les pointeurs sont un élément central de la programmation en C/C++. Il faut absolument en comprendre le fonctionnement. Un pointeur est en fait une variable qui est "une adresse d'un type de variable". Conceptuellement, il peut se représenter par une flèche qui pointe vers une variable. On peut déclarer des pointeurs vers n'importe quel type de variable.

```
int* aptr;
crée un pointeur vers un entier. il faut lire "crée une variable aptr de type
pointeur d'entier" float* bptr;
pointeur de float ...
```

Opérations usuelles utilisant des pointeurs:

- Déclarer un pointeur `float* bptr;`
- Faire pointer un pointeur vers l'adresse d'une variable déjà déclarée en utilisant `&`:
  ```
  float *bptr;
  float b;
  bptr= &b;//&b correspond ici à la référence(adresse) de la variable b
  ```
- Modifier la valeur vers laquelle pointe un pointeur en utilisant `*`:
  ```
  float *bptr;
  float b;
  bptr= &b;
  *bptr=3;
  ```
- Déplacer un pointeur:
  ```
  ptr=ptr+1;
  le pointeur pointe maintenant physiquement sur la case mémoire contigüe
  ```

Erreurs de pointeurs

ATTENTION: Déclarer un pointeur d'entier (ou autre) ne veut pas dire "allouer une variable de type entier (ou autre) vers laquelle il pointe. Ceci est la cause de nombreuses erreurs lors de la compilation et de l'execution d'un programme.

En fait à l'execution, le programme va créer des cases mémoires consécutives et contigues, avec le bon type de variables. Ainsi, avant de modifier la valeur d'une case pointée par un pointeur, **il faut toujours faire attention qu'elle soit allouée**. On peut TOUJOURS lire une mémoire pointée vers un pointeur. Cependant, écrire dans une case mémoire non-allouée impliquera une erreur à l'execution du code. Ces Bugs sont en général assez durs à corriger.

Quelques exemples d'erreurs usuelles:

```
float *ptr1;
*ptr1=1; // Erreur: La case mémoire n'est pas allouée

float b;
float *ptr1;
ptr1= &b; //ptr1 pointe vers b *prt1 =2;//C'est OK ptr1=ptr1+1;//Attention cette
ligne n'incrémente pas la valeur de b
//ptr1 pointe maintenant à la case mémoire contigüe à b *ptr1= 3; //Erreur: La
case vers laquelle pointe maintenant ptr1 n'est pas allouée
```

Allocation/Desallocation de pointeurs

En C++, les allocations se font par les mots clefs `new` (allocation) et `delete` (désallocation) (plus efficace que `malloc` et `free`ăen C). Allouer un pointeur c'est allouer au pointeur une case mémoire vers laquelle pointer.
ATTENTION pour faire un programme ne causant pas d'erreur mémoire, il faut toujours désallouer les cases allouées! Sinon, il y aura une surcharge de mémoire et un crash possible de l'ordinateur. Ces erreurs sont très difficile à détecter dans le cadre du débuggage. **Ainsi il faut toujours appliquer la règle suivante: Pour toute allocation utilisateur `new`, il faut une désallocation `delete`.**

- Voici comment se passe une allocation/désallocation:
  ```
  float *ptr1;
  ptr1= new float; //case allouée.
  *ptr1=2;//On peut donc écrire la valeur dans la case
  delete ptr1;
  désallocation
  ```
- Il est possible d'allouer des cases mémoires contigües. Cela correspond à "l'allocation dynamique des tableaux":
  ```
  float *ptr1; //déclaration du pointeur
  int n;
  cin»n; //demande à l'utilisateur d'entrer une valeur de n
  ptr1= new float[n]; //Alloue n cases mémoires pour réel contigües.
  *ptr1=2; //Change la première valeur
  *(ptr1+1)= (*ptr1)+1;//Change la valeur de la case contigües à ptr1
  delete [] ptr1; //Désalloue toutes les cases allouées pour ptr1
  ```

Pointeurs et tableaux

On peut remarquer qu'en C/C++ les tableaux et pointeurs sont le même concept. D'ailleurs, les écritures suivantes sont équivalentes:

- `T+i` et `&T[i]`
- `T[i]` et `*(T+i)`
- Un tableau peut donc être alloué de manière statique ou dynamique. L'allocation statique ne nécessite pas de désallocation, **mais le nombre de cases doit être connu avant l'execution du programme**. L'allocation dynamique nécessite la désallocation, mais peut être codée sans connaitre a priori le nombre de cases:

```
float T[9];//Allocation statique
for(int i=0;i<9;i++)
T[i]=i;
for(int i=0;i<9;i++)
*(T+i)=i;
//La désallocation n'est pas
nécessaire
```

```
float* T;
T=new float[9];//Allocation statique
for(int i=0;i<9;i++)
T[i]=i;
for(int i=0;i<9;i++)
*(T+i)=i;
delete [] T; //La désallocation est
nécessaire
```

Erreurs usuelles de pointeurs

Lorsqu'on désalloue une case pointée par un pointeur il faut faire attention que cette case ait bien été allouée au préalable! Une erreur commune est de créer un pointeur, de l'allouer puis de le déplacer par la suite. On perd ainsi l'adresse de la variable vers laquelle il pointait. `float *ptr1;`

```
float a;
ptr1= new float; //case allouée pour ptr1
ptr1=&a;//pointe vers a
delete ptr1;// Erreur: ptr1 ne pointe plus vers une case allouée avec un new
//La case mémoire que pointait ptr1 est perdue à jamais
```

Les fonctions

Une fonction est comme une variable, mais elle a une fonction , elle effectue quelque chose en fonctions d'arguments (ou paramètres). On lui passer alors des variables comme paramètres pour qu'elle puisse s'effectuer.

De même que pour les variables, une fonction doit avoir une déclaration avant d'être utilisée. Elle peut avoir un type également, comme une variable (`int, float, float*, bool,...`). On utilise le mot-clef `return ()` pour retourner le resultat.
On distinguera:

- La déclaration d'une fonction: déclarer au compilateur que cette fonction va exister:
  `int MultiplieParDeux(int a);`
- La définition d'une fonction: définir ce qu'elle fait (Souvent la définition fait aussi office de déclaration):

```
int MultiplieParDeux(int a)
{
return( a*2);
}
```

- L'utilisation:
```
int a=2;
int b;
b=MultiplieParDeux(a);//Appel de MultiplieParDeux avec en paramètre, la
variable a.
```

De plus il existe un type spécial pour les fonctions qui ne renvoient rien: `void`. Par exemple:
```
void AfficheUnNombre(int a)

cout« "Le nombre est" «a;//comme printf en C++
```

La fonction main

Dans un programme, il existe TOUJOURS une fonction appelée `main`. C'est toujours par cette fonction que le programme commence son execution. Selon les compilateurs, elle peut êtyre de type `int` ou `void`. `int main()`
```
//...on tape ici le noyau de l'execution du programme.
return(0);//utilisé pour retourner d'éventuels codes d'erreur
```

Include et différents fichiers

Il est possible de déclarer variables et fonctions dans des fichiers différents de celui où est située la fonction `main`. Il faut cependant indiquer au compilateur où trouver la déclaration de ces fonctions et variables si on souhaitent les utiliser dans son programme. Ceci se fait grâce au mot-clef `#include`.

- On utilise des cotes pour le fichiers créés par l'utilisateur qui doivent se trouver dans le même dossier que le fichier qui les inclu: `#include "Mabiblio.cpp"`
- On utilise <> pour les fichiers usuels C++. Certaines librairies sont très utiles et le compilateur sait où les trouver: `#include <iostream>`,`#include <math.h>`,`#include <string>`,...

Exemple:
Fichier "mabiblio.h"

```
int MultiplieParDeux(int a)
{return( a*2);}
float retrancheUn(float b)
{return( b-1);}
```

Fichier "mesvariables.h"
```
int toto=2;
float titi=2;
const float pi=3.1415
```

Fichier "main.cpp"
```
#include "Mabiblio.h"
#include "mesvariables.cpp"
#include <stdio.h>
int main() int b; float d;
b=MultiplieParDeux(toto);
d=RetrancheUn(titi); return 0;
```

LARGE Les nouveautés du C++

Nous allons lister ici les différentes nouveautés du C++ par rapport au C:

- L'utilisation de `new` et `delete` en lieu et place de `malloc` et `free`. Nous en avons déjà discuté plus haut

- L'affichage et la saisie utilisateur par les fonctions `cin` et `cout`: Ces fonctions remplacent les fonctions `printf` et sont beaucoup plus facile à utiliser. Elles nécessitent l'inclusion de la bibliothèque `iostream` et du `namespace std`.

```
#include <iostream>;
using namespace std;
int main(){


int a=2;
float b;
bool a=0;
float * ptr=&b;
cin»b; // attend que l'utilisateur rentre une valeur
cin»a;//fonctionne avec n'importe quel type de variable
cout« "je peux afficher ce que je veux entre cotes";
cout« "comme des retours à la ligne"«endl;
cout« "des tabulmations"«"";
cout "des entiers"«a« "ou des reels"«b« "ou des adresses" «ptr;
return(0);}
```

- Le passage par référence dans les fonctions
- La surcharge de fonctions
- La programmation objet par l'utilisation des classes (Cours suivants)

Surcharge de fonctions

En C il n'est pas possible d'écrire deux fonctions ayant le même nom et même type. En C++ c'est possible,tant que ces fonctions n'ont pas les mêmes paramètres et qu'il est possible de déterminer quelle fonction sera appelée lors de l'exécution.
Par exemple, si l'on souhaite écrire 3 fonctions transformant des heures/minutes/secondes en un nombre de secondes mais que l'on souhaite pouvoir l'appeler avec un nombre d'heures seulement, ou d'heures/minutes seulement, c'est possible en déclarant 3 fonctions qui se surchargent (même nom même type):

```
int CalculeSecondes(int heure,int minute, int secondes)
{return( heures*3600 +minutes*60+secondes);}

int CalculeSecondes(int heure,int minute)
{return( heures*3600 +minutes*60);}

int CalculeSecondes(int heure)
{return( heures*3600);}

int main(){
int h=5;
int min=42;
int sec=27;
int res;
//Les trois appels suivants sont possibles:
res= CalculeSecondes(h);
```

```
res= CalculeSecondes(h,min);
res= CalculeSecondes(h,min,sec);
return(0);
}
```

Remarques

On peut remarquer que la même chose en C était possible simplement endéclarant cette fonction comme suit: `int CalculeSecondes(int heure, int minute=0, int seconde=0)`
`{return( heures*3600 +minutes*60+secondes);}`

Cependant le concept de surcharge est plus puissant puisqu'il permet de créer des fonctions qui ont le même nom mais qui ont des fonctions complètement différentes:

```
float perimetre(float rayon)
{return( 2*3.14.rayon);}
float perimetre(float a,float b)
{return( 2*a+2*b);}

int main(){
float=5;
float=3;
float res; res= perimetre(a);//retourne le perimetre d'un cercle
res= perimetre(a,b);//retourne le perimetre d'un rectangle
return(0);
}
```

On peut aussi remarquer que dans l'exemple précedent, je n'aurais pas pu surcharger la fonction :

```
int CalculeSecondes(int heure)
{return( heures*3600);}
```

par la fonction

```
int CalculeSecondes(int minutes)
{return( minutes*60);}
```

Pour s'en rendre compte il suffit de remarquer que je ne saurais pas quelle fonction appeler entre les deux si j'effectue la ligne: `res=CalculeSecondes(b);`

Passage par adresse passage par reference

En C comme en C++, une fonction ne peut avoir qu'un type et ainsi, ne peut renvoyer qu'un résultat. Lorsque plusieurs variables sont à retourner, on utilise alors les variables comme des variables d'entrée/sortie. Cependant, il est important de comprendre que les fonctions, lorsqu'on passe des variables en paramètre utilisent **des copies** de ces variables:

```
void MetsBDansA(int a,int b)
{b=a;}
int main(){
int test=2; int test2;
MultiplieParDeux(test2,test);
return(0);
}
```

Attention `test2` n'est pas modifiée dans la fonction `MultiplieParDeux`. C'est une copie de test2 qui est modifiée. Il existe deux possibilités pour tout de même modifier les variables passées dans les fonctions.

Passage par adresse

La subtilité est ici de passer non pas la variable en paramètre mais **son adresse**. Ceci était déjà poassible en C.

```
void MetsBDansA(int* a,int* b)
{*b=*a;}
int main(){
int test=2; int test2;
int *ptr=&test;
int *ptr2=&test2;
MultiplieParDeux(ptr2,ptr);
return(0);
}
```

Ici, naturellement le programme fera une copie de `ptr2` et `ptr`, mais qui auront la même valeur que `ptr2` et `ptr`. Ainsi les copies pointeront vers les mêmes cases mémoire que les originaux. Et en modifiant la valeur pointée, on modifiera également la valeur pointée originale et dans ce code, `test2` aura bien la valeur de `test`.

Passage par référence

En C++, une nouvelle possibilité apparait. Le passage par **réference**. La motivation pour cette technique est que dans le code en passage par pointeur, on est toujours obligé d'appeler la fonction en appelant un pointeur, ce qui peut être fastidieux. On va donc préférer changer la déclaration de la fonction, en utilisant le symbole `&`. Attention, ici il ne signifie pas l'adresse. Il signifie **la référence vers**. Sans entrer dans les détails, ce signe indique au compilateur **de ne pas faire une copie de la variable, mais d'utiliser directement la variable passée en paramètre.**

```
void MetsBDansA(int& a,int& b)
{b=a;}
int main(){
int test=2; int test2;
MultiplieParDeux(test2,test);
return(0);
}
```

On voit ici que la fonction s'utilise comme lorsqu'on la déclare sans `&`. Cependant ici, il n'y a pas de copie effectuée et `test2` prend la valeur de `test`.

Exercice Corrigé

Attention, cet exercice est corrigé ci-après mais il est conseillé de le faire soit même, sans regarder la solution.
Faire un main et une fonction qui permette de:

- créer une fonction qui construise une matrice MxM qui soit la matrice identité `eye`
- Une fonction qui permette de faire la somme de deux matrices MxM `Somme`.
- Une fonction qui affiche M; (utilisation de cin et cout)
- Une fonction qui renvoie la valeur max et min d'une matrice.

Des bons compléments

Vous trouverez des compléments de cours notemment sur ARCHE avec les documents de Monsieur Soussen. De plus vous trouverez un très bon support pour la programmation objet sous le site web suivant: `https://openclassrooms.com/courses/programmez-avec-le-langage-c`

Solution de l'exercice

```
void Eye(float* Mat, int M)
{ for (int i=0; i<M;i++)
{for (int j=0; j<M;j++)
{
    if (i==j)
    {*(Mat+i*M+j)=1;}
    else
    {*(Mat+i*M+j)=0;}
}
}
}



void somme(float* Mat1, float* Mat2, float* Res, int M)
{ for (int i=0; i<(M*M);i++)
    {* (Res + i)= * (Mat2 + i)+ * (Mat1 + i);
    }
}

float* somme2(float* Mat1, float* Mat2, int M)
{
    float *Res=new float[M*M];
    for (int i=0; i<(M*M);i++)
{* (Res + i)= * (Mat2 + i)+ * (Mat1 + i);
}
    return Res;
}



void Affiche(float* Mat, int M)
{ for (int i=0; i<M;i++)
    {for (int j=0; j<M;j++)
    {
        cout<< *(Mat+i*M+j);
}
        cout<<endl;
}
}

int main(){

    int M;
    cout<<"M?";
    cin>> M;

    float * Mat1=new float[M*M];
    float * Mat2=new float[M*M];
    float * MatRes=new float[M*M];
    float * MatRes2;



    Eye(Mat1,M);
    Eye(Mat2,M);
    somme(Mat1,Mat2,MatRes,M);

    MatRes2=somme2(Mat1,MatRes,M);

    Affiche(MatRes,M);
    Affiche(MatRes2,M);

    delete [] Mat1;
    delete [] Mat2;
    delete [] MatRes;
    delete [] MatRes2;
    return(0);

}
```

```
M?4
2000
0200
0020
0002
3000
0300
0030
0003
Program ended with exit code: 0
```