Programmation avancée en C++: rappel et compléments sur les fonctions

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

1. Complements Tableaux

Parcours de tableau en C++ 11

- Le C++11 nous offre une nouvelle façon de parcourir les tableaux proche de ce qui se fait en python.

```
1   int main(){
2     double notes[6]={10.0,11.0,16.5,9.0,7.0,14.5};
3
4     for (double note:notes){
5         cout<<" note "<<note<<endl;
6     }
7     for (auto note:notes){
8         cout<<" note "<<note<<endl;
9     }
10    return 0;
11 }
```

- Cela permet de parcourir la liste en mettant dans ”note” a chaque iteration un élément du tableau ”notes”.
- Puisque le tableau contient des doubles, ”note” sera du type double. Mais la boucle peut être général en utilisant ”auto”.

2. Fonctions

Fonctions

- Dès qu'un code devient gros il est **essentiel** de le décomposer en parties indépendantes et simples: (les fonctions).
- **Une fonction** effectue une tâche sur des données d'entrée et renvoie/modifie des données en sortie.

```
1  double polynome_simple(double x){
2    double res= 0.0;
3    res = x*x +2*x +1.0;
4    return res;
5  }
6  int main() {
7    double x= 0.0;
8    cout<<" >>>> Donnez un nombre "<<endl;
9    cin >> x;
10   cout<<" >>>> Polynome: "<<polynome_simple(x)<<endl;
11   return 0;
12 }
```

- Définition de fonction: "type renvoyé - nom - (paramètres)".
- Instruction **Return**: permet de définir le résultat renvoyé par la fonction et interrompt la fonction.

Passage d'arguments

- **Question essentiel**: transmission des arguments.

```
1  int modify(double x){
2    x = x+ 3;
3    return 0;}
4  int main() {
5    double x= 0.0;
6    cout<<" >>>> Donnez un nombre "<<endl;
7    cin >> x;
8    modify(x);
9    cout<<" >>>> x: "<<x<<endl;
10 }
```

```
user $  >>>> Donnez un nombre
user $ 2
user $  >>>> x: 2
```

- "x" n'est pas modifié par la fonction: passage par copie.
- La fonction reçoit une copie du paramètre et modifie la copie, pas l'original.
- Passage par adresse pour solutionner le problème. **Outil fondamental**:

Passage argument II

- Pointeur: variable contenant l'adresse mémoire d'une variable. Exemple:

```
1 int * p;  int n =10;
2 p= &n;  *p = 20;
3 cout<<">> p "<<p<<" *p "<<*p<<endl;
```

```
user $ >> p
       :0x7ffeea6a0944
       *p :20
```

- " & " permet d'accéder à l'adresse de "n", " * " permet d'accéder à la valeur de sur lequel pointe *p*. Utilisation:

```
1 int modify2(double *p ){
2   *p = *p+ 3;
3   return 0;
4 }
5 x = 10.0;  double * q= &x;
6 modify2(q);
7 cout<<" >>>> x: "<<x<<endl;
```

```
user $  >>>> x: 13
```

Passage d'argument par référence

- **Spécificité du C++** (par rapport au C): passage par référence: permet de passer une variable modifiable sans pointeur.

```
1 int modify3(double & x){
2   x = x+ 3;
3   return 0;
4 }
5 x = 10.0;
6 modify3(x);
7 cout<<" >>>> x: "<<x<<endl;
```

```
user $  >>>> x: 13
```

- Très pratique mais il faut faire attention si une variable ne doit pas être modifiée. Source classique d'erreur.
- **Utilisation intéressante**: référence et pointeur sur une fonction.

```
1 int (*pf)(double &,double);
2 pf = modify3;
```

Variables locales et globales I

- Il est important de comprendre la portée des variables.
- Deux cas: allocation statique et dynamique.
- Alloction statique:
  - Il y a les variables globales potentiellement accessible dans tous le programme, toutes les fonctions.

```
1 int global_var = 123;
2
3 int main() {
4    ....
5 }
```

  - Plus précisément, la portée (ou espace de validité) est l'ensemble du code qui suit la déclaration (y compris à l'intérieur des fonctions).
  - Il y a les variables locales dont la portée est la fonction dans laquelle elles sont définies.
  - A chaque utilisation de la fonction la mémoire pour la variable est allouée et de-allouée à la fin automatiquement.
  - On peut faire des variables locales à un bloc d'instruction (délimitée par des accolades).

Variables locales et globales II

- Dans le cas dynamique la situation est différente

```
1  double * test(){
2      double * p=new double ;
3      *p= 10.0;
4      return p;
5  }
6  int main()
7  {
8      double *q;
9      q=test();
10     cout<<">>>>"<<*q<<endl;
11     return 0;}
```

```
user $   >>>>10
```

- La variable qui est locale et détruite à la fin de la fonction c'est le pointeur. La case mémoire contenant 10 reste allouée.
- Ici le pointeur *p* est copié dans *q* par le "return" puis dé-alloué à la fin de la fonction mais *q* contient toujours l'adresse de la case contenant 10.

Fonctions: Pointeur de fonction

- Exemple: éviter un ”if” dans une boucle.
- Code naif:

```
1  double locale1(double x, double y){
2   .....
3  }
4  int id = 1;
5  double z = 0.0;
6  for (int  i=1;i<5000000;i++){
7     if( id == 1){
8       z = z + locale1(2.0,1.5);
9     }
10    if( id == 2){
11      z = z + locale2(2.0,1.5);
12    }
13    if( id == 3){
14      z = z + locale3(2.0,1.5);
15    }
16 }
```

- **CPU time**: 2.9 sec (naïf), 2.8 sec (autre code).
- Les ”if” dans la boucle empêchent des optimisations du compilateur. La différence est plus grande avec des fonctions lourdes en calcul.

Fonctions: Pointeur de fonction

- Exemple: éviter un ”if” dans une boucle.
- Code avec pointeurs de fonction:

```
1  double locale1(double x, double y){
2   ..... }
3   double (*pf)(double,double) = NULL;
4   if( id == 1){
5       pf = locale1;
6   }
7   if( id == 2){
8       pf = locale2;
9   }
10  if( id == 3){
11      pf = locale3;
12  }
13  for (int  i=1;i<50000000;i++){
14      z = z + pf(2.0,1.5);
15  }
```

- **CPU time**: 2.9 sec (naïf), 2.8 sec (autre code).
- Les ”if” dans la boucle empêchent des optimisations du compilateur. La différence est plus grande avec des fonctions lourdes en calcul.

Arguments par défaut

- On peut donner une valeur par défaut à des paramètres d'une fonction.

```
1 double somme(double a, double b, double c=0.0){
2     return a+b+c;}
3 int main(){
4     double res1,res2;
5     res1 = somme(1.0,2.5,3.0); res2 = somme(1.0,2.5);
6     cout<<" res 1: "<<res1<<" res 2: "<<res2<<endl;
7     return 0;
8 }
```

```
user $  res 1: 6.5 res 2:3.5
```

- **Important**: les paramètres avec valeurs par défauts doivent être les derniers.

```
1 double somme(double a, double b=0.0, double c){return a+b+c;}
```

```
user $     13 | double somme(double a, double b=0.0, double c){
user $    main.cpp:13:31: note: ...following parameter 2 which .....
```

Fonctions: Sur-définition

- On peut surdéfinir une fonction. C'est-à-dire avoir le même nom avec des déclarations/définitions différentes.

```
1  int maxi (int a, int b){
2    int res=b;
3    if( a > b) { res = a; }
4    return res;
5  }
6  int maxi (int a, int b,int c){
7    int res=b;
8    if( a > b) { res = a; }   if( c > res) { res = c; }
9    return res;
10 }
11 double maxi (double a, double b){
12   double res=b;
13   if( a > b) { res = a; }
14   return res;
15 }
```

Fonctions: Sur-définition

- On peut surdéfinir une fonction. C'est-à-dire avoir le même nom avec des déclarations/définitions différentes.

```
1  int maxi (int a, int b){
2    int res=b;
3    if( a > b) { res = a; }
4    return res;
5  }
6  double maxi (double a, double b){
7    double res=b;
8    if( a > b) { res = a; }
9    return res;
10 }
```

```
1  cout << "m1: "<<maxi(1,2) << " m2: "<<maxi(1,2,5)<<
   " m3: "<<maxi(1.2,1.0)<<endl;
```

```
user $ m1: 2 m2: 5 m3: 1.2
```

- Tout va bien.

Fonctions: Sur-définition

- On peut surdéfinir une fonction. C'est-à-dire avoir le même nom avec des déclarations/définitions différentes.

```
1  int maxi (int a, int b){
2    int res=b;
3    if( a > b) { res = a; }
4    return res;
5  }
6  double maxi (double a, double b){
7    double res=b;
8    if( a > b) { res = a; }
9    return res;
10 }
```

```
1  cout << "m1: "<<maxi(1,2) << " m2: "<<maxi(1,2,5)<<
   " m3: "<<maxi(1.2,1)<<endl;
```

```
functions.cpp:185:65: error: call to 'maxi' is ambiguous
```

- Il ne sait pas s'il doit convertir le "double" en "int" ou l'inverse.

Fonction et déclaration Auto I

- On avait dans le cours précédemment qu'on pouvait déclarer une variable sans type avec "auto". En effet, il interprète le type en fonction de la valeur de la variable.
- Possible d'utiliser cela avec les fonctions.

```
1 auto test_auto(int a){
2     return a*a;}
3 auto test_auto2(int a){
4     return 5.25*a;}
5 int main(){
6     cout<<" res : "<<typeid(test_auto(5)).name()<<endl;
7     cout<<" res 2: "<<typeid(test_auto2(5)).name()<<endl;
8     return 0;
9 }
```

```
user $ res : i
user $ res 2: d
```

- on voit que le type de retour est adapté en fonction de la valeur.

Fonction et déclaration Auto II

- on voit que le type de retour est adapté en fonction de la valeur.

```
1 auto test_auto(int a){
2 if a>0 {return a*a;}
3 else return 5.25*a;
4 }
```

- Dernier code interdit car le ”if” génère deux types potentiellement différents.
- **Les arguments aussi peuvent être Auto**.

```
1 auto test_auto(auto a){
2     return a*a;}
3 int main()
4 {
5     cout<<" res : "<<typeid(test_auto(5)).name()<<endl;
6     cout<<" res : "<<typeid(test_auto(5.5)).name()<<endl;
7     return 0;}
```

```
user $ res : i
user $ res 2: d
```

Fonctions: fonctions ”constexpr”

- ”constexpr” permet de définir des expressions calculées à la compilation.
- **Pourquoi ?** accélération des codes en effectuant certains calculs à la compilation.
- On peut faire la même chose avec des fonctions simple.

```
1  constexpr int pol( int x, int n){
2   int res=0.0;
3   for (int i=0;i<n;i++){ res= res + n*x; }
4   return res;
5 }
6 #include <iostream>
7 ...
8 const int xc =4; const int nc=10;
9 constexpr int yc = 1 +pol(xc,nc);
10 cout<<" >> y >> "<<yc<<endl;
```

```
user $  >> y >> 401
```

- Utilisation avec des arguments non constant, cela devient une fonction classique.

Fonction à argument variables I

- **Possibilité du C++ 11**: donner un nombre variable d'arguments
- On utilise un type générique pour cela "initializer_list< T >" avec T un type.
- Cela va permettre de construire une liste non modifiable de valeurs compatible avec T.
- On commence par manipuler cet objet:

```
1  int n=10;
2  initialize_list<int> li_n={1, 3, 5, n, n*n, n+12}
3  initialize_list<double> li_d={1.1, 1.5, 2.7, n*2.2}
4  int main(){
5     for(int i : li_n){
6         cout<<" >> "<<i<<endl;}
7     for(double d : li_d){
8         cout<<" >> "<<d<<endl;}
9     return 0;}
```

Boucle très particulière pour parcourir cette structure. Ce type d'outils plus spécifique sera introduit en fin de semestre.

Fonction à argument variables II

- Exemple d'utilisation:

```
1  double sum(initializer_list<double> li){
2    double res;
3    for(double d : li){
4        res=res+d;
5     }
6     return res;}
7  int main()
8  {
9     cout<<" >>> "<<sum({1.1, 2.7, 2.7, 1.7, 5.2})<<endl;
10    return 0;
11    }
```

```
user $  >>> 13.4
```

3. Tableaux et fonctions

Tableaux et fonction I

- Passage en argument de fonction:

```
1 void print_tab(double tab[10]){ cout<<"print 1: "<<tab[4]<<endl; }
2 void print_tab2(double tab[]){ cout<<"print 2: "<<tab[4]<<endl; }
3 void print_tab3(double * tab){ cout<<"print 3: "<<tab[4]<<endl; }
```

- Les trois possibilités marchent. La première nécessite de connaître la taille du tableau.
- **Important**: Dans tous les cas le tableau est passé en référence. Il peut être modifier par la fonction.
- Si on veut éviter la modification du tableau:

```
1 void print_tab(const double tab[])
```

Tableaux et fonction II

- Exemple:

```
1  void tri_1(double  T[], int N){
2      double c;
3      int i,j;
4      for(i=0;i<N-1;i++){
5          for(j=i+1;j<N;j++){
6              if ( T[i] > T[j] ) {
7                  c = T[i];
8                  T[i] = T[j];
9                  T[j] = c;
10                 }
11         }
12     }
13 }
```

- Variante (même code):

```
1  void tri_2(double * T, int N){
```

Tableaux et fonction III

- Exemple suite:

```
1 int main(){
2     double T[4]={1.2,2.3,1.7,0.2};
3     cout<<"before >>"<<T[0]<<", "<<T[1]<<", "<<T[2]<<", "<<T[3]<<endl;
4     tri_1(T,4);
5     cout<<"1er >>"<<T[0]<<", "<<T[1]<<", "<<T[2]<<", "<<T[3]<<endl;
6     tri_2(T,4);
7     cout<<"2eme >>"<<T[0]<<", "<<T[1]<<", "<<T[2]<<", "<<T[3]<<endl;
8     return 0;
9 }
```

```
user $ before >>1.2, 2.3, 1.7, 0.2
user $ 1er >>0.2, 1.2, 1.7, 2.3
user $ 2eme >>0.2, 1.2, 1.7, 2.3
```

- On voit bien que les deux approches permettent de modifier le tableau.

Tableaux et fonction IV

- Le passage par référence concerne aussi les pointeurs.
- **Ex**: allouer dynamiquement de la mémoire dans une fonction. **Cas 1**:

```
1 void initp1 (double *p){
2   p = new double[10];
3   p[0] =5.0; }
4   double * p1=NULL;
5   cout<<" adresse b>>"<<p1<<endl;
6   initp1(p1);
7   cout<<" adresse f>>"<<p1<<endl;
8   cout<<" first case p1>>>"<<p1[0]<<endl;
```

```
user $ adresse b>>0x0
user $ adresse f>>0x0
user $ Segmentation fault: 11
```

- **Remarque**: en utilisant un pointeur, la valeur vers laquelle pointe le pointeur peut être modifiée.
- Avant la fonction il pointe vers NULL, après aussi. Il ne pointe pas vers le tableau alloué dans la fonction.

Tableaux et fonction V

- **Problème**: le pointeur est passé en copie donc on peut pas modifier sa valeur à lui, c'est-à-dire l'adresse vers quoi il pointe. **Cas 2**:

```
1 void initp2 (double *&p){
2   p = new double[10];
3   p[0] =5.0; }
4   double * p2=NULL;
5   cout<<" adresse b>>"<<p2<<endl;
6   initp1(p1);
7   cout<<" adresse f>>"<<p2<<endl;
8   cout<<" first case p2>>>"<<p2[0]<<endl;
```

```
user $ adresse b>>0x0
user $ adresse f>>0x7f7f3ed000c0
user $ first case p2>>>5
```

- Pointeur est passé par réf.., on peut modifier sa valeur (adresse du tableau).
- **Important**: déclaration d'un tableau statique locale à une fonction. Une allocation est globale (elle subsiste à la sortie de la fonction).

4. Applications

Exemple: Résolution d'équation non-linéaire I

- On souhaite résoudre

$$F(\mathbf{X}) = 0, \quad \mathbf{X} \in \mathbf{R}^d$$

- Algorithme 1: **Point fixe**

$$\mathbf{X}^{n+1} = G(\mathbf{X}^n)$$

- avec $G(\mathbf{X}) - \mathbf{X} = F(\mathbf{X})$
- Convergence si $\| \mathbf{X}^{n+1} - \mathbf{X}^n \|_2 < \epsilon$

```
1 double g1(double x){
2   double res= 0.0;
3   res = cos(x);
4   return res;
5 }
```

Exemple: Résolution équation nonlinéaire II

- Solveur de Picard:

```
1  int picard_solver(double (*f)(double),double eps,double xinit){
2    double norm2 = 1.0;
3    double x0 = xinit, x = 0.0;
4    int n=0, nmax = 100;
5    while((abs(norm2) >eps) && (n <nmax)) {
6      x = g(x0);
7      norm2 = sqrt((x-x0)*(x-x0));
8      x0 = x;      n++;
9    }
10   cout<<" solution: "<<x<<endl;
11   cout<<"nb iteration: "<<n<<" residu: "<<g(x)-x<<endl;
12   return 0;
13 }
14 int main() {
15   picard_solver(f1,g1,0.00001,1.2);
16   return 0;
17 }
```

```
user $ Solution: 0.739088
user $ Nb iteration: 30 residu: 5.31218e-06
```

Exemple: Résolution équation nonlinéaire III

- Solveur de Picard plus général:

```
1 double g(initializer_list<double> li,double x){
2     double res=0.0;
3     int k =0;
4     for (double d: li){
5         res = res+ d*pow(x,k);
6         k++;
7     }
8     return res;
9 }
```

Exemple: Résolution équation nonlinéaire IV

- Solveur de Picard plus général:

```
1
2  int main() {
3    double norm2 = 1.0, x0 = 0.1, x = 0.0;
4    int n=0;
5    initializer_list<double> li_d={0.1, -0.5, 0.7, 0.2};
6
7    while((abs(norm2) >0.0001) && (n <100)) {
8      cout<<" >> "<<x0<<endl;
9      x = g(li_d,x0);
10     cout<<" >> "<<x<<endl;
11     norm2 = sqrt((x-x0)*(x-x0));
12     x0 = x;      n++;
13   }
14   cout<<"nb iteration: "<<n<<" residu: "<<g(li_d,x)-x<<endl;
15   return 0;
16 }
```