Licence STS

Université Claude Bernard Lyon I

LIFAP1 : ALGORITHMIQUE ET PROGRAMMATION IMPÉRATIVE, INITIATION

1

COURS 4 : Passage de paramètres
Erreurs fréquentes en C

OBJECTIFS DE LA SÉANCE

- Faire le point sur les paramètres
  - Paramètres formels / effectifs
- Comprendre le mécanisme de passage des paramètres
  - Par valeur (donnée)
  - Par adresse ou référence (donnée / résultat)
- Apprendre à transformer une fonction en procédure
- Faire le tour d'horizon des erreurs fréquentes commises en TP

PLAN

- Paramètres formels / paramètres effectifs
- Passage de paramètres
  - Par valeur ou donnée
  - Par adresse ou donnée / résultat
- Transformation fonction → procédure
- Les erreurs en C

PARAMÈTRE FORMEL / EFFECTIF

- Paramètre formel : variable utilisée dans le corps du sous-programme
  (il fait partie de la description de la fonction)
- Paramètre effectif : variable (ou valeur) fournie lors de l'appel du sous-programme
  (valeurs fournies pour utiliser la fonction et valeurs renvoyées)

- Copie de la valeur du paramètre effectif vers le paramètre formel correspondant lors de l'appel

- Paramètres formel et effectif ont des noms différents

PLAN

- Paramètres formels / paramètres effectifs
- Passage de paramètres
  - Par valeur ou donnée
  - Par adresse ou donnée / résultat
- Transformation fonction → procédure
- Les erreurs en C

PARAMÈTRE FORMEL / EFFECTIF

- Lorsqu'on écrit l'en-tête d'un sous-programme,
  il s'agit des paramètres formels
  - Exemple : int moyenne (int x, int y)

    x et y sont les paramètres formels ;
    ils n'ont pas de valeur particulière dans la définition
    du sous-programme

- Lorsqu'on appelle un sous-programme,
  il s'agit des paramètres réels ou effectifs
  - Exemple resultat = moyenne (a,b)

    a et b sont les paramètres effectifs ;
    ils doivent avoir une valeur du même type
    que les paramètres formels

PASSAGE DE PARAMÈTRES FORMELS

- **Données** (passage par valeur) :
  - Le sous-programme dispose d'une copie de la valeur.
  - Il peut la modifier, mais l'information initiale dans le code appelant n'est pas affectée par ces modifications.

  Syntaxe : type nom ;

- **Résultats** ou **données / résultats** (passage par adresse) :
  - Le sous-programme dispose d'une information lui permettant d'accéder en mémoire à la valeur que le code appelant cherche à lui transmettre.
  - Il peut alors modifier cette valeur, le code appelant aura accès aux modifications faites sur la valeur.

  Syntaxe : type **&** nom ;

PASSAGE PAR VALEUR

- Valeur de l'expression passée en paramètre copiée dans une variable locale
  - Utilisée pour faire les calculs dans la fonction appelée
  - Aucune modification de la variable locale ne modifie la variable passée en paramètre
  - La variable locale ayant servi à effectuer les calculs est ensuite détruite donc sa valeur est perdue

EXEMPLE

```
int carre (int a)
{
  return a*a;
}

int main()
{
  int val=3, car;
  car = carre(val);
  cout << "carré =  »
       << car;
  return EXIT_SUCCESS;
}
```

	main	carre
avant appel	val = 3 car = ?	
dans l'appel de carre		a=3 calcul :9
après appel	val = 3 car = 9	

copie valeur

retour valeur

Et si on modifie a…

```
int carre (int a)
{
  a = a*a;
  return a;
}

int main()
{
  int val=3, car;
  car = carre(val);
  cout << "carré =  »
       << car;
  return EXIT_SUCCESS;
}
```

	main	carre
avant appel	val = 3 car = ?	
dans l'appel de carre		a=3 a=9
après appel	val = 3 car = 9	

copie valeur

retour valeur

PASSAGE DE PARAMÈTRE RÉSULTAT

- Plus de copie des valeurs des paramètres effectifs, plus de variable locale
- On travaille directement sur la variable passée en paramètre
- Toute modification du paramètre dans la fonction entraîne la modification de la variable passée en paramètre
- Matérialisé dans l'entête par le symbole &

EXEMPLE

```
void calculAire(double r,
  double &aire)
{
  aire = 3,14*r*r;
}

int main()
{
  double rayon=1,5, air;
  calculAire(rayon,air);
  cout<<air;
  return EXIT_SUCCESS;
}
```

	main	calculAire
Avant appel	rayon=1,5 air = ?	copie valeur
Dans l'appel de la procédure	air = ? air = 7,06	r = 1,5 copie adresse aire aire = 7,06
Après appel	rayon=1,5 air = 7,06	

PASSAGE DE PARAMÈTRES EFFECTIFS

- Que peut-on mettre dans un paramètre effectif ?

  - Valeurs littérales :

    factorielle(6);

  - Valeur d'une variable :

    factorielle(n);

  - Valeur renvoyée par une fonction :

    factorielle(n_premiers(4));

EXEMPLE

```
#include <iostream>
using namespace std;

void permuter(int & a, int & b)
{
   int t;

   t= a;
   a =b;
   b= t;
}


int main(void)
{
   int u, v;
   cin >> u;
   cin >> v;

   permuter(u, v);

   cout << u << endl;
   cout << v << endl;
}
```

Paramètres formels

passés par adresse
(données / résultats)

Paramètres effectifs :
contenu des variables

Exemple sous Code Blocks

- Que se passe-t-il si on ne met pas le & ??
  - Exemple

- Et maintenant après correction
  - Résultat

PLAN

- Paramètres formels / paramètres effectifs
- Passage de paramètres
  - Par valeur ou donnée
  - Par adresse ou donnée / résultat
- Transformation fonction → procédure
- Les erreurs en C

TRANSFORMER UNE FONCTION EN PROCÉDURE : POURQUOI ?

- Parce qu'en C on ne peut renvoyer qu'une seule valeur dans une fonction
- Parfois on a besoin de retourner deux choses
  - Par exemple le produit et la somme de deux valeurs
- La fonction doit alors céder sa place à une procédure

TRANSFORMER UNE FONCTION EN PROCÉDURE : PRINCIPE

- Rajouter autant de paramètres formels que de résultats à renvoyer
- Passer ces nouveaux paramètres formels en donnée / résultat
- Supprimer l'instruction return

- Exemple : on souhaite renvoyer la **somme** et le **produit** de deux entiers

TRANSFORMER UNE FONCTION EN PROCÉDURE : EXEMPLE

```
int somme(int a, int b)
{
  int som;
  som=a+b;
  return som;
}
```

Fonction qui retourne un entier

```
void somme(int a, int b,
  int & som)
{
  som=a+b;
}
```

Procédure qui contient un nouveau paramètre permettant de stocker la valeur "retournée"

TRANSFORMER UNE FONCTION EN PROCÉDURE : EXEMPLE

- Impossible d'écrire une fonction parfois

➔ on écrit alors une procédure

```
void som_prod(int a, int b, int &s, int &p)
{
   s=a+b;
   p=a*b;
}
```

EXEMPLE : TRADUCTION ALGO → C

- Calculer les racines d'un polynôme

- En algorithmique :

```
Fonction RacinesPolynome (a, b, c : réels, x1, x2 : réels ) :
   entier
données : a, b, c (coefficients du polynôme)
valeur retournée par la fonction : nb_racines : entier
données /résultats : x1, x2 : réels
```

- Traductions possibles en C

```
int racines(float a, float b, float c, float & x1, float & x2)

void racines(float a, float b,float c, int & nb_racines, float
   & x1, float & x2)
```


➢ nb_racines, x1 et x2 sont passés en résultats avec un “&” devant.

EXEMPLE : APPEL DE LA PROCÉDURE

- Les résultats sont des paramètres formels supplémentaires, il faut donc ajouter les paramètres effectifs correspondants.

```
int main(void)
{
   int n;
   float a, b, c, x1, x2;

   cin >> a; cin >> b; cin >> c;
   n= racines(a, b, c, x1, x2);
}
```

- x1 et x2 n'ont pas de valeur avant de rentrer dans la procédure = paramètres données /résultats !!

PLAN

- Paramètres formels / paramètres effectifs
- Passage de paramètres
  - Par valeur ou donnée
  - Par adresse ou donnée / résultat
- Transformation fonction → procédure
- Les erreurs en C

LES ERREURS DANS LES PROGRAMMES

- Il ne faut pas être frustré avec les erreurs de C ;
  c'est comme quand on apprend à parler une autre langue...

- L'ordinateur n'est pas votre ennemi, il se plaint car il ne comprend pas vos intentions, et il n'ose pas prendre des initiatives

- 2 types d'erreurs :
  - Syntaxiques : problème dans l'écriture du code (les plus faciles à corriger : ça ne compile pas)
  - Algorithmiques : Il faut réfléchir, simuler...

Fonction / Procedure

- Une fonction renvoie une valeur que l'on peut utiliser :
  - afficher
  - affecter dans une variable,
  - comparer à une autre valeur.

- Une procédure ne renvoie pas de valeur :
  - on ne peut ni afficher, ni affecter, ni comparer le « résultat » d'une procédure

EXEMPLES : LESQUELS FONCTIONNENT ?

```
cout << factorielle(7);

if(factorielle(factorielle(3))) < 1000)
     cout << "oui";
else
     cout << "non";

cout << mention(12);

mention(factorielle(3));

cout << permuter(x, y);

permuter(x, y);

permuter(factorielle(4), 4);
```

LES ERREURS SYNTAXIQUES FRÉQUENTES

- Le « ; » se met à la fin de chaque instruction
  mais jamais
  - Après l'entête d'une fonction ou procédure
  - Après la condition d'un while, d'un for …
  - Après une « } » (sauf les structures)
- La différenciation majuscules / minuscules
  - int toto ≠ int ToTo
  - Pour les mots clés du langage aussi !!!
    Si pas en gras dans l'interface, pas reconnus
- Parenthèses autour des conditions dans if et while
- Le test d'égalité : « = » en algo, « == » en C
  - if (a = 0) {a++;} else {a--;} => passera toujours dans le sinon !!!

LES ERREURS SYNTAXIQUES FRÉQUENTES : EXEMPLES

```
int puissance(int n);
{

    int i, res;
    res=1
    for (i=1;i<n;i++);
    {
        res = rEs * n;
    }

    RETURN res;
}
```

Pas de ; à la fin de la déclaration

Ici il en faut un !!

Pas là !

variable rEs pas reconnue
car différent de res

pas comme mot clé
car return en majuscules !

Structures de Contrôle : if, while et for

- **if**, **while** et **for** prennent soit
  - une instruction (accolades pas indispensables mais conseillées)
  - un bloc d'instructions obligatoirement délimité par des accolades.
- Les instructions sont par exemple :
  - affectation
  - appels aux autres fonctions
  - ou même autres opérations de contrôle

```
for (i = 0; i < 10; i++)
if ((x=rand()%20-10) && x < 0) cout << endl << x * x;
  else while(x != 0) cout << x-- << " ";
```

RAPPELS : INDENTATION

- Mais il est recommandé de mettre **TOUJOURS** les accolades pour être plus clair et certain du corps des opérations de contrôle
- Prend plus de place
- Mais plus facile à lire !

```
for (i = 0; i < 10; i++)
{
    if ((x=rand()%20-10)
        && (x < 0))
    {
      cout << endl << x * x;
    }
    else {
      while(x != 0)
      {
        cout << x-- << " ";
      }
    }
 }
```

```
for (i = 0; i < 10; i++)
if ((x=rand()%20-10) && x < 0) cout << endl << x * x;        else
 while(x != 0) cout << x-- << " ";
```

STUCTURES DE CONTRÔLE : IF, WHILE ET FOR

- Possible mais à éviter :
  - for (i=0; i< 10; i++);
  - while (a < m);
- Sont des lignes valables en C/C++,
  la première pas trop dangereuse,
  mais la deuxième peut aboutir à une boucle infinie.

ÉQUIVALENCE : WHILE ET FOR

```
for (initialisation ; condition ; itération)
  { instruction1 ; instruction2 ; ...}
```

est équivalent à :

```
initialisation;
while (condition)
{
  instruction1 ; instruction2 ; ...
  itération;
}
```

- Donc on comprend la raison du “;” dans le **for**

SIGNATURE DES FONCTIONS/PROCÉDURES

- La signature des fonctions /procédures avertit le compilateur du type du résultat et des paramètres
- La signature est la fonction sans son corps, le nom des paramètres est facultatif
- Pour l'instant vous écriviez vos fonctions avant de les utiliser.

Ex:

```
int RacinesPoly(int a,int,int,int &r1,int &r2);
void afficherPoly(void)
{
 ... ; nbres = RacinesPoly(1,2,3,res1,res2);....
}
int RacinesPoly(int c0,int c1,int c2,int &r1,int &r2)
{
.... return nb_sol;
}
```

SIGNATURE SUIVI D'UN BLOC

```
int factorielle(int n);
{
   int i; int r;
   r = 1;
   for (i=1; i<=n; i++) {r = r*i;}
   return r;
}
```

Quel est le problème ?

- Le ";" après l'entête : devient une signature et plus l'entête !

MAUVAISE INITIALISATION

```
int i;
int j;
i = 0; j = 0;
while (i < N) {
  while (j < M) {
  ... ;
  j++;
  }
i++;
}
```

```
int i;
int j;
i = 0;
while (i < N) {
  j = 0;
  while (j < M) {
  ... ;
  j++;
  }
i++;
}
```

CONCLUSION

- Approfondissement des notions de fonction et procédure
- Définition des paramètres formels et effectifs
- Compréhension du mécanisme de passage de paramètres
  - Par valeur (donnée) : copie dans une variable locale ; modifications perdues
  - Par référence (donnée résultat) : on travaille directement sur le contenu de la variable ; modifications conservées
- Aperçu des erreurs fréquentes en programmation