Langage procédural

Le langage C

Nizar OUARTI

Laboratoire ISIR (email: ouarti@isir.upmc.fr)

2010 2011

A quoi sert une fonction ?

- Découper le code logiquement
- Obtenir une meilleure lisibilité
- Réutilisation des briques créées
- Mettre à jour une fonction sans tout changer
- Trouver les erreurs plus facilement

Fonction

- Une fonction possède un **prototype** et un **corps**
- Le protoype est l'en-tête de la function
- Le corps de la fonction est la partie ou les commandes s'exécutent

```
1 /* Fonction */
2 type_sortie mafonction(type1 parametre1,type2
      parametre2,etc...)
3 // CECI est appelé le PROTOYPE de la function
4 {
5         [Definition de variables locales]//ICI c'est
6         [Instructions]                   //le CORPS
7         return(Valeur_sortie);           //de la
              function
8         /*return fait terminer la fonction*/
9 }
```

Les fonction en C

- Paramètres en entrée
- Sortie de la fonction (unique)
- La sortie peut être vide (void)
- Le compilateur vérifie la cohérence entre le prototype et la façon dont on utilise la fonction
- Tout comme une variable une fonction doit être déclarée avant utilisation
- Les headers souvent servent à déclarer les fonction de son propre code.

Fonction

- exemple concret

```
1  /* Exemple de fonction */
2  double square(double x) // PROTOYPE de la function
3  {
4          double sq=x^2;                    //ICI c'est
5          printf("La valeur est: %f",sq); //le CORPS
6          return(sq);                       //de la
               function
7  }
8
9  /*Déclaration de prototype à mettre dans un header*/
10 double square(double x);
```

Portée des variables

- Les variables déclarées dans une fonction n'existent plus à la fin de la fonction
- Ceci est aussi vrai pour les déclarations dans les blocs
- Il faut bien noter que le main est une fonction comme une autre (point d'entrée du programme)
- On dit de ces variables qu'elles ont une portée **locale**
- On ne peut donc que modifier les variables que l'on qualifie de **globales** dans une fonction
- Un autre moyen d'opérer une *pseudo-modification* dans uns fonction d'utiliser return, la sortie de la fonction

Arguments par valeur ou par adresse

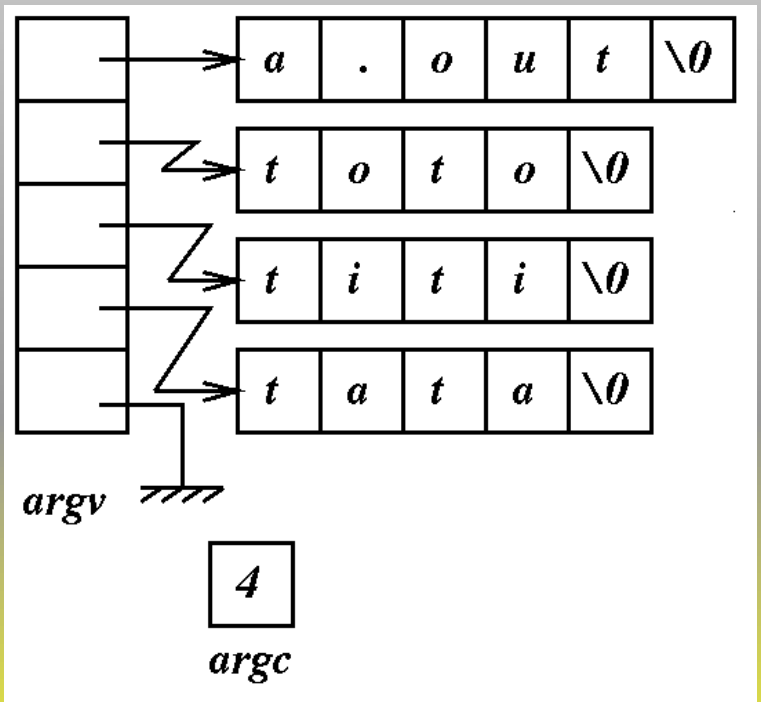
- Il existe un autre moyen de modifier une variable à l'intérieur d'une fonction sans qu'elle soit globale
- Au lieu d'envoyer la valeur de la variable, on envoie son adresse
- L'opérateur pour faire ceci est **&**

Piège à éviter, Usages

- Attention à ne pas retourner l'adresse d'une variable locale ! !
- La fonction main peut intercepter des strings venant de l'environnement
- Un tableau ou une structure aussi peut être un paramètre
- Une fonction peut être envoyée comme paramètre

La fonction **main**

- La fonction **main** est utilisée avec **int argc** et **char *argv[]**
- argc donne le nombre de strings entrés dans le programme
  argv est un tableau de strings



Passage d'un tableau ou structure en argument

- Lorsque l'on passe un tableau en argument d'une fonction, les variables sont modifiables
- Pourquoi ?

Passage d'un tableau ou structure en argument

- Lorsque l'on passe un tableau en argument d'une fonction, les variables sont modifiables
- Pourquoi ?
- En fait lorsque l'on envoie uniquement le nom du tableau, cela revient à envoyer l'adresse
- Lorsque l'on envoie une structure en argument ce n'est pas le cas

Tableau en argument

- exemple concret

```c
#include <stdio.h>

int  somme(int t[], int n)
{
    int i, sum=0;
    for (i=0; i < n; i++)
    {
            sum = sum+t[i];
    }
    return sum;
}
int main(int argc, char* argv[])
{
    int  t[] = { 1, 9, 10, 14, 18};
    int N= sizeof(t)/sizeof(t[0]);
    int  somme(int t[], int n);// facultatif
    printf("%d\n", somme(t, N));
    return 0;
}
```

Structure en argument

- Transmission de paramètres plus compacte

```
1  #include <stdio.h>
2
3  typedef struct
4  {
5     double x;
6     double y;
7     double z;
8  } Point;
9
10 double norme(Point p)
11 {
12    double norm=sqrt(p.x*p.x+p.y*p.y+p.z*p.z);
13    return(double);
14 }
15 main(int argc, char* argv[])
16 {
17    Point p1= { 0.5, 0.8, 0.1};
18    void norme(Point p);// ici facultatif
19    double norme_p=norme(p1);
20    printf("La norme du point est : %f\n", norme_p);
21 }
```

Fonctions comme argument

- On peut aussi avoir des fonctions en arguments
- Il faut pour cela que le prototype possède un pointeur vers la fonction
- C'est à dire qu'il attende son adresse

```
1 /* Exemple fonction comme paramètre */
2 void func ( void (*f)(int) );// prototype
3 //il faut que l'argument soit bien du prototype voulu
4
5 void print ( int x ) {
6   printf("%d",x);
7 }
8
9 func(print);// usage
```

Fonction Récursives

- Une fonction récursive est une fonction qui s'appelle elle-même

```
1 /* Exemple de fonction */
2 int factoriel(int x) // PROTOTYPE de la fonction
3 {
4         if (x==0) return 1;//terminaison
5
6         else return(x*factoriel(x-1));//recursion
7 }
```

Le Préprocesseur

- Le préprocesseur est comme un traitement de texte.
- Il fait l'opération copier coller : include
- Il fait l'opération remplacer tout : define
- Il fait l'opération commenter décommenter avec if endif

Bibliothèques #include

- Cette commande sert à inclure un autre fichier dans votre source
- On peut ajouter des headers créés en standard en C
- On peut ajouter des headers créés par soi même.

```
1 /* Exemple de header */
2 #include <stdio.h>
3 //bibliothèque standard pour les entrées sorties
4
5 #include "monheader.h"
6 // on crée ses propres headers notament on y range les
      prototypes de fonction
```

Constantes #define

- Sert à définir une constante principalement
- Remplacement des occurrences par la valeur définie
- La constante définie peut être vide

```
1  /* Exemple de définition de constantes */
2  #define PI 3.1415
3  //Toutes les occurences de PI seront remplacé par
       3.1415 pas de de point virgule
4  #define DEBUG
5  //On peut créer une variable pour qu'elle existe
6  #undef DEBUG
7  // Peut servir si on veut redéfinir une fonction
8  #define NUMBERS 1, \
9                          2, \
10                         3
11       int x[] = { NUMBERS };
12           // équivalent à int x[] = { 1, 2, 3 };
13 //Pour pourvoir écrire sur plus d'une ligne on utilise
       \
```

ifndef endif

- Sert principalement pour retirer les lignes liées au débugage
- Permet d'avoir une exécution conditionnelle
- Ne pas oublier le endif qui fini (pour remplacer les accolades)

```
1  /* Exemple ifndef */
2
3  #define DEBUG
4  // Il suffit de commenter ou non DEBUG
5  // Pour avoir le bloc du dessous qui est éxécuté ou non
6
7  #ifndef DEBUG
8  printf("La variable x vaut: %f",x);
9  #endif
10 //Des commandes seront exécutées conditionnellement
```

if elif else endif

- Sert par exemple dans le cas de code C qui sert sur différents OS
- Toutes les commandes habituelles des if normaux sont disponibles
- && ¡= == etc...
- Ne pas oublier le endif qui fini

```
1 /* Exemple de if du préprocesseur */
2 #if MAC==1
3     //environnement Mac
4 #elif MSDOS==1
5     //environnement MS DOS
6 #else
7     //environnement Unix
8 #endif
```

Macros

- On peut créer l'équivalent de fonction grâce au préprocesseur

```
1 /* Exemple de macro du préprocesseur */
2 #define MAX(x,y) x > y ? x : y // mauvais résultat
3 // n = 5 * MAX(4,6); /* Sera remplacé par : n = 5 * 4 >
      6 ? 4 : 6; */
4 #define min(X, Y)  ((X) < (Y) ? (X) : (Y))
5 //une autre manière de faire un test en c (x<y) ? (x:y)
6 // equivalent à if (a<b) return(x); else return(y);
7
8         x = min(a, b);          //  equivalent à  x = ((a)
            < (b) ? (a) : (b));
```