Sous-programmes et fonctions

- Les sous-programmes servent à factoriser du code et à améliorer sa lisibilité
- Lorsque le même fragment de code doit être exécuté plusieurs fois, il faut l'isoler, lui donner un nom et y faire référence lorsqu'on veut l'exécuter.
- La plupart des sous-programmes servent à calculer un résultat à partir des valeurs de certaines variables du programme appelant
  - Ils se comportent comme des fonctions mathématiques
- On doit donc spécifier :
  - La liste et le type des paramètres que l'on veut passer à la fonction
  - La nature de la valeur qu'elle renvoie.

Fonctions (1)

- En C, il n'y a pas de différence entre fonction et sous-programme
  - Tous sont définis sous la forme de fonctions censées renvoyer une valeur d'un type spécifié
  - Un sous-programme est une fonction qui ne renvoie rien (type void)

Fonctions (2)

- Syntaxe :

```
type fonction (argum ents){
    déclarations instructions
}
```

- Exemple:

```
int strindex(char s[],char t[]){
    int i,j,k;
    for(i=0;s[i];i++){
      for(j=i,k=0;t[k]&& s[j]==s[k];j++,k++);
      if(k>0 && t[k]==0)
        return i;
    }
    return -1;
}
```

- Exercice : que calcule cette fonction ?

Fonctions (3)

- Les paramètres de la fonction sont vus comme des variables locales au bloc de code de la fonction.
- On peut définir d'autres variables dans le code

```
void                    /*pas de valeur de retour                    */
compte_a_rebours (
int n){                 /*paramètre de la fonction,vu comme local*/
    int i;              /*variable locale à la fonction              */
    printf("Attention,compte à rebours :\n");
    for(i= n;i>= 0;i--)
      printf("% d…\n",i);
}
```


return

- O n spécifie la valeur que renvoie une fonction au moyen de l'instruction return
  - La valeur doit être du même type que le type de retour déclaré de la fonction
- Elle permet la terminaison anticipée de la fonction
  - Exemple:

```
int factorielle(int n){
    int f,i;
    if(n<0)
      return 1;
    for(f=1,i=2;i<=n;i++)
            f *= i;
    return f;
}
```

- Pour les fonctions ne retournant rien, on utilise l'instruction return sans argument.

Appel de fonction

- On appelle une fonction en donnant son nom, suivi de la valeur des paramètres de l'appel, dans l'ordre de leur définition.
- Le nom de la fonction avec ses arguments est une expression typée utilisable normalement.
- Exemple:

```
int main(){
   int i;
   i = factorielle (7);
   printf("La factorielle de 7 est % d\n ",i");
   return 0;
}
```


Prototypage (1)

- Pour générer l'appel à une fonction et vérifier le typage de ses paramètres, le compilateur doit déjà connaître l'existence de la fonction et son typage
  - Il suffit que la fonction appelée soit définie avant la fonction appelante dans le fichier source
  - ☹ pas toujours faisable dans le cas de fonctions s'appelant mutuellement.

Prototypage (2)

- Pour résoudre le problème, il faut pouvoir déclarer une fonction avant de la définir
  - Déclaration d'un prototype de la fonction
- Un prototype est la déclaration d'une fonction sans le bloc d'instructions qui la suit

```
int factorielle(int n); /*Prototype de la fonction */
int main(){
   …
   i= factorielle(7); /*Vérification de l'utilisation */
}
int factorielle(….   /*Définition effective          */
```


Prototypage (3)

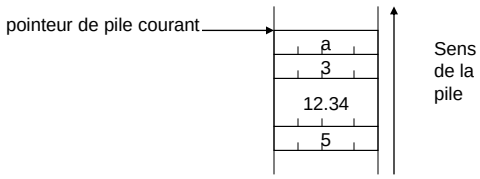
- Avantages du prototypage:
  - Séparation de la spécification d'une fonction de son implémentation :
    - Programmation par contrat
    - Possibilité de compilation séparée
    - Facilité de travail en équipe
  - Double vérification de la cohérence :
    - Entre le prototype et les instances d'appel
    - Entre le prototype et la définition de la fonction

Mécanisme d'appel d'une fonction (1)

- Lorsqu'une fonction est appelée, il se produit les actions suivantes :
  - Empilement des valeurs des paramètres d'appel dans la pile de programme en ordre inverse
  - Empilement de l'adresse de retour (adresse du code situé juste après l'appel)
  - Appel de la fonction

```
int i = 3;
int j = 5;
double d = 12.34;
…
f(i,d,j);
… ←
```

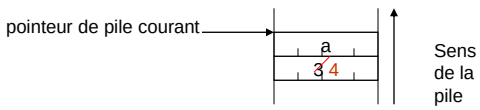
a : adresse de retour après le code d'appel



Mécanisme d'appel d'une fonction (2)

- Le passage de paramètres dans la pile s'effectue par valeur et non par référence
  - C'est la valeur du paramètre qui est passée dans l'appel et non son adresse
  - Si on modifie un paramètre, c'est sa copie qui est modifiée, pas lui-même

```
f(int j) {
  j =4;
}
…
i = 3;
f(i);
```

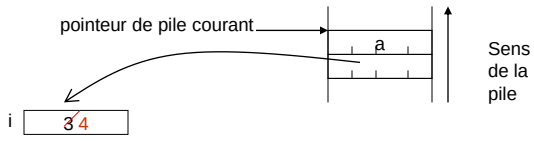


i 3

Mécanisme d'appel d'une fonction (3)

- Si l'on veut modifier la valeur d'un paramètre au-delà de l'exécution de la fonction, il faut passer son adresse :
  - Émulation du passage par référence
  - Exemple scanf

```
f(int *jptr){
  *jptr = 4;
}
…
i = 3;
f(&i);
```



Récursivité

- Une fonction récursive peut s'appeler elle-même.
- En C, la récursivité est réalisée grâce au mécanisme d'empilement
- Exemple:

```
int fact(int n) {
   return (n <= 1) ? 1 :n*fact(n-1);
}
```

La fonction main() (1)

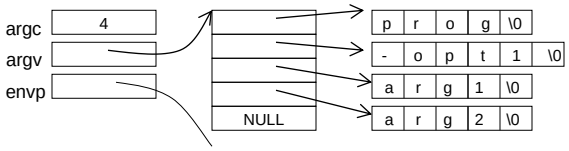
- La fonction main retourne un entier –
  - Code de retour du programme, renvoyé au processus appelant (en général, l'interpréteur de commandes)
- La fonction main possède trois arguments :
  - argc, entier spécifiant le nombre de paramètres passés à la commande,y compris son propre nom
  - argv, tableau de pointeurs de chaînes de caractères listant les arguments de la commande
  - envp, tableau de pointeurs de chaînes de caractères listant les variables d'environnement connues

La fonction main() (2)

- Format complet de la fonction main :

```
int
main(
        int argc,        /*N ombre d'arguments,y compris le nom  */
        char * argv[], /*Tableau des chaînes des arguments        */
        char * envp[]) /*Tableau des chaînes des variables d'env. */        {
            …
        }
```

$ prog –opt1 arg1 arg2



Visibilité des variables (1)

- Une variable déclarée en dehors du corps d'une fonction est visible de toutes les fonctions du fichier dans lequel elle est déclarée (variable globale au fichier)

```
int i;/* Variable visible dans tout le fichier */
f (){
    i= 4;/* On la voit et on l'utilise ici*/
}
int main (){
    i= 3;/* On la voit et on l'utilise ici*/
    f ();
    printf ("% d",i);
}
```


Visibilité des variables (2)

- Une variable déclarée à l'intérieur du corps d'une fonction ou d'un bloc d'instructions n'est visible qu'à l'intérieur du corps du bloc :

```
f (){
   int i;  /* Une première variable locale i*/
   i= 4;
}
g (){
   int i; /* Une autre,différente de la première */
   i= 6;
}
int main (){
   f (); /* Ici,on ne voit ni l'une,ni l'autre */
   g ();
}
```

Visibilité des variables (3)

- Lorsqu'une variable locale a le même nom qu'une variable globale, celle-ci est masquée à l'intérieur du bloc où la variable locale est définie :

```
int i;    /* Variable globale au fichier                    */
f () {
  int i; /* Variable locale de même nom ,qui la masque */
  i= 4; /* Ici,on utilise donc la variable locale          */
}
int main (){
  i= 3; /* Ici,c'est la variable globale que l'on voit     */
  f ();
  printf ("% d\n",i); /* On affiche donc "3"               */
  return 0;
}
```


Classes de variables

- Par défaut, les variables locales sont dynamiques et les variables globales sont statiques.
- On peut préciser la classe de stockage
  - static ,variable dont la durée de vie est celle du programme
  - register,on demande au compilateur d'implémenter la variable dans un registre du processeur
- Par défaut,une variable (dynamique) est créée à l'entrée d'un bloc et libérée à la sortie

```
void incremnter(int i);

int main(){
    static int i= 1;
    incrementer(i);
    incrementer(i);
    printf("% d\n",i); /* affiche 3 */
    return 0;
}
void incrementer(int i){
    i++;
}
```


Initialisations

- Pour les objets statiques, l'initialisation n'est faite qu'une seule fois et les objets doivent être simples.
- Les objets dynamiques sont initialisés à chaque entrée dans le bloc

```
for(i=0;i<1000;i++){
    int pi= 3.141592;
    …
}
```

- Dans cet exemple, pi est initialisée 1000 fois ! !

Exercices

- Écrire, compiler et exécuter les solutions aux exercice suivants :
  1. Écrire un programme qui met en évidence la gain de temps lié à l'utilisation du mot clé «register». On utilisera la commande «time» pour effectuer les mesures
  2. Soit x un réel et n un entier positif. En remarquant que $x^n=(x^{n/2})^2$ si n est pair et que $x^n = x * x^{(n-1)/2}$ si n est impair, écrivez une fonction récursive qui renvoie la puissance $n^{ème}$ d'un nombre réel x.
  3. Écrivez une fonction echanger(int a,int b); qui échange le contenu de la variable a et celle de la variable b puis qui affiche le contenu des deux variables.
     Écrivez un programme qui appelle cette fonction et qui affiche après le contenu des deux variables ainsi échangées. Commentez

Exercices (2)

- Écrire un programme qui affiche la liste des arguments de l'appel.
- Écrire un programme dont le comportement est le suivant :

```
$ a.out -s 23 34 67
  124
$ a.out –k 45 65 7
  a.out : option invalide
  Usage : a.out –(s|m) nbr1 nbr2 …
```