

Programmation Structurée S6/S7

Cours 3 b Pointeurs de fonctions et fonctions à nombre d'arguments variable

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>
 Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

septembre 2010



- 1 Pointeurs de fonctions
 - Rappels sur les prototypes
 - Pointeurs de fonctions
- 2 Arguments en nombre variable

Rappels sur les prototypes de fonctions

Prototype = déclaration précisant :

- le **nom** de la fonction,
- son **type de retour**, ou `void`,
- le nombre et le **type de ses arguments**.

Exemples :

```
int additionne(int x, int y);
void affiche(char*);
```

Différences avec une définition de fonction :

- le corps est omis (remplacé par ;),
- le nom des arguments est facultatif,
- une fonction peut être déclarée plusieurs fois.

Rappels sur les prototypes de fonctions

Utilité du prototype :

information nécessaire pour compiler un **appel** de fonction.

Si un prototype est donné, la fonction peut être définie

- après l'appel, ou
- dans une autre unité de compilation.

Exemple :

Fichier a.c

```
int fun1(int x);

int fun2(void)
{ return fun1(1); }

int fun1(int x)
{ return x+1; }
```

Fichier b.c

```
int fun2(void);

int main()
{ return fun2(); }
```

Pointeurs de fonctions

Chaque fonction a une adresse en mémoire.

Pointeur de fonction = variable

- contenant l'**adresse** d'une fonction,
- dont le type indique le **prototype de la fonction**.

Déclaration d'un pointeur de fonction ptr :

```
type-ret (*ptr)(type1,...,typeN);
```

ptr peut contenir l'adresse de toute fonction

- prenant N arguments, de types type1 à typeN, et
- retournant une valeur de type type-ret.

Affectation de pointeur de fonction

Affectation : ptr = f; où

- f est le nom d'une fonction de prototype compatible avec ptr,
- f est un pointeur de fonction de même type que ptr.

Exemple :

```
int truc(int x) { ... }
```

```
int bidule(int y);
```

```
void main()
{
    int (*ptr)(int);
    ptr = truc;
    ptr = bidule;
}
```

Appel de fonction par pointeur

Appel par pointeur : identique à un appel de fonction classique

- ptr(arg1,...,argN); ou
- x = ptr(arg1,...,argN);

Exemple d'appel :

```
int bidule(int y);
```

```
int main()
{
    int (*ptr)(int);
    ptr = bidule;
    return ptr(2);
}
```

Application : fonction d'ordre supérieur

Fonction **paramétrée par une fonction**.

► on passe un pointeur de fonction en argument.

```
void affiche_tableau(char* tab[], void (*f)(char*), int n)
{
    int i;
    for (i=0; i<n; i++) f(tab[i]);
}
```

```
void affiche(char* s) { printf("%s\n", s); }
```

```
void main() {
    char* t[] = { "toto", "titi" };
    affiche_tableau(t, affiche, 2);
}
```

Applications

Faire les deux exos suivants :

- Écrire une fonction `fois2` qui réalise la fonction $x \mapsto 2x$.
Écrire une fonction `appliquer_traitement_tab` qui applique une fonction f sur un tableau T (d'entiers) de taille n . Recoller les bouts.
- Réaliser une fonction de tri générique de tableau qui prend en argument une fonction de comparaison de deux entiers.

1 Pointeurs de fonctions

2 Arguments en nombre variable

Koiteske ?

Un exemple : `printf` ...

```
int printf(const char *format, ...);
```

On aimerait faire pareil pour donner la somme d'une énumération de nombres, concaténer plusieurs chaînes, ...

```
somme(2, 3, 12, 7);  
somme(2, 89);
```

Comment ?

On peut le faire, mais il faut au moins un argument qui peut être :

- le nb d'arguments
- la fin de la liste d'arguments

```
somme(3, 2, 3, 5);  
concatener("bonjour", "toto", NULL);
```

Déclaration et définition

Il y a un argument fixe **au moins** et des arguments optionnels :

```
char* concatener(char*, ...);  
int somme (int nargs,...);
```

Les ... sont appelés **ellipses**

Dans stdarg, il y a 4 constructions :

- va_list est le type qui va nous servir à parcourir les arguments
- va_start initialise le parcours sur le paramètre 1
- va_arg donne le param suivant
- va_end termine le parcours.

Exemple : somme à nb de params variable

```
int somme(int nargs)  
{  
    va_list args ; // type  
    int i, total=0; // init  
  
    va_start(args,nargs); // args <- &nargs+1  
  
    for(i=0;i<nargs;++i)  
    {  
        total = total + va_arg(args,int)  
        // val de l'argument et progression d'un entier  
    }  
    va_end(args) // nettoyage (voir la doc)  
  
    return total;  
}
```