Projet de Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

Partie 1: Techniques Avancées de C

- Chapitre 1: Définition de Type
- Chapitre 2: Généricité
- Chapitre 3: Arguments d'un Programme

Agenda

- Chapitre 2: Généricité
  - Principe
  - Duplication de Code
  - Pointeur de Fonction

Agenda

- Chapitre 2: Généricité
  - Principe
  - Duplication de Code
  - Pointeur de Fonction

Principe

- On souhaite pouvoir appliquer le même traitement à plusieurs types de données
  - objectif?
    - ✓ factoriser le code
- **Polymorphisme**
  - mécanisme orienté objet
  - cfr. INFO0062
- Pas de formalisme simple pour la généricité en C
  - il faut "bricoler"
    - ✓ duplication de code
    - ✓ pointeurs de fonctions
    - ✓ liste d'arguments variable

Agenda

- Chapitre 2: Généricité
  - Principe
  - Duplication de Code
    - ✓ Principe
    - ✓ Exemples
    - ✓ Limites
  - Pointeur de Fonction

Principe

- Ecrire autant de routines que de types de données à traiter
  - réservé aux fragments de petites tailles
  - maintenabilité difficile
- Automatisation des routines via les macros
  - passage de code en paramètre textuel des macros
  - utilisation de la direction ##
    - ✓ permet de joindre 2 termes accolés pour faire un seul identifiant

Exemples

- Exemple 1
  - addition pour des `int` et des `float`

```
#define FONCTIONPLUS(type)   \
type                         \
plus_##type(                 \
type a,                      \
type b)                      \
{                            \
  return (a+b);              \
}
```

```
FONCTIONPLUS(int)
```

création de la fonction plus_int()

```
FONCTIONPLUS(float)
```

création de la fonction plus_float()

Exemples (2)

- Exemple 1 (cont.)
  - addition pour des `int` et des `float`

```
int main(){
  int a=3, int b=4;
  float x=3.5, y=4.6;

  printf("%d\n", plus_int(a, b));
  printf("%f\n", plus_float(x, y));

  return 0;
}//fin programme
```


Exemples (3)

- Exemple 1 (cont.)
  - addition pour des `int` et des `float`
- Que se passe-t-il à la compilation?

```
$>gcc -E duplication.c | more


...
int plus_int ( int a, int b) { return (a + b); }
float plus_float ( float a, float b) { return (a + b); }

int main(){
 int a=3, b=4;
 float x=3.5, y=4.6;
 printf("%d\n", plus_int(a,b));
 printf("%f\n", plus_float(x,y));
 return 0;
}//fin programme
```

Exemples (4)

- Quid si l'opérateur n'est pas le même en fonction du type?
  - on peut passer l'opérateur en paramètre
- Exemple 2
  - division

```
#define FONCTIONDIV2(type, operateur) \
type                                  \
div2_##type(                          \
type a)                               \
{                                     \
  return (a operateur);               \
}
```

```
FONCTIONDIV2(int, >>1)
```

div2_int()

```
FONCTIONDIV2(float, /2.0F)
```

div2_float()

Limites

- La duplication de code ne résout pas tous les problèmes
  - *évolutivité*?
    - ✓ quid d'un nouveau type n'utilisant pas des opérateurs "primitifs"?
    - ✓ tous les opérateurs devraient être des fonctions!
  - *modularité*?
    - ✓ un nouveau type implique de toucher au code de la routine principale, puis recompiler
    - ✓ nécessité de disposer du code source
      - pas de livraison du code sous forme de bibliothèque

Limites (2)

- Idéalement
  - passer en paramètre le code que l'on souhaite voir exécuter par la routine
- Problème
  - on ne peut pas passer des fragments de code en paramètre
    - exécution d'instructions sur la pile interdite
      - cfr. INFO0045
- Solution
  - on peut passer la référence à un fragment de code déjà compilé
    - pointeur
    - **pointeur sur fonction**

Agenda

- Chapitre 2: Généricité
  - Principe
  - Duplication de Code
  - Pointeur de Fonction
    - Principe
    - Déclaration
    - Pointeur sur `void`
    - Exemples

Principe

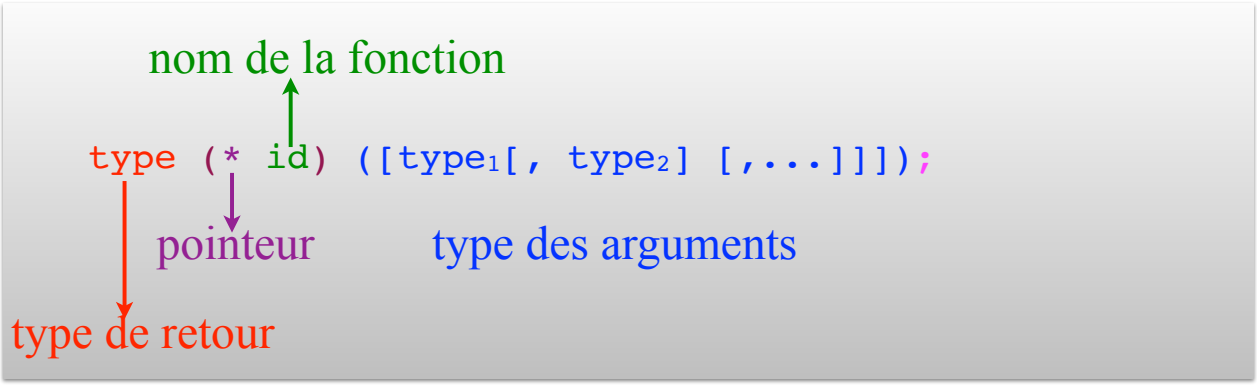
- Une fonction, comme tout élément manipulé par un programme dispose d'une adresse
  - identification univoque
- Adresse d'une fonction?
  - adresse du début de la fonction
  - spécifie l'emplacement mémoire de la 1ère instruction
  - opération sur une fonction
    - ✓ exécution
    - ✓ et non lecture/écriture du contenu

Principe (2)

- Analogie tableaux/fonctions
  - *tableaux*
    - ✓ `t[i]` représente la valeur retournée par la lecture de la (`i`+1)ème case du tableau
    - ✓ l'adresse de début est `t`
  - *fonctions*
    - ✓ `f(i)` représente la valeur retournée par l'exécution de la fonction `f` avec comme paramètre i
    - ✓ l'adresse de début est `f`
- Dans certains langages, on ne fait pas la distinction entre l'un et l'autre
  - les parenthèses servent aux 2

Déclaration

- Comment déclarer un pointeur de fonction?



Déclaration (2)

- Les parenthèses autour du `*` sont obligatoires
  - `int (*fonction) (int, double);`
    - ✓ pointeur sur fonction renvoyant un `int`
  - `int *fonction (int, double);`
    - ✓ fonction renvoyant un pointeur sur un `int`
- On lit toujours une déclaration en partant du nom de l'objet déclaré et en allant vers l'extérieur de la déclaration

Déclaration (3)

- Exemple 1
  - `int (*fonction)(int, int);`
  - pointeur de fonction prenant deux entiers en argument et retournant un entier
- Exemple 2
  - `void (*procedure)(float);`
  - pointeur de procédure prenant un float en argument

Pointeur sur `void`

- Possibilité d'écrire des routines manipulant des données dont le type n'est pas connu à l'avance
- Comment manipuler des données de types non connus?
  - `void *`
  - routines de manipulation mémoire
    - ✓ e.g., `memcpy()`
- Attention
  - la taille des données doit être passée au "client"
  - utilisation (abusive?) des conversions de type
    - ✓ type cast

Pointeur sur `void` (2)

- Exemple

```
#include <stdio.h>

int main(){
  void * tab[3] = {(void *)1, (void *)"1", (void *)'1'};

  printf("%d, %s, %c\n", (int)tab[0], (char *)tab[1],
                         (char)tab[2]);

  return 0;
}//fin programme
```


Exemples

- Exemple 1
  - trouver la pente d'une fonction f de $\mathbb{R} \rightarrow \mathbb{R}$ en un point x

```
#include <stdio.h>
#include <math.h>

//Calcule la pente pour la fonction f en un point x
double pente(double (* f)(double), double x){
  const double EPSILON = 0.00001;

  double res = ((*f)(x + EPSILON) - (*f)(x))/EPSILON;

  return res;
}//fin pente()
```

Exemples (2)

- Exemple 1 (cont.)

```
//calcule le carré d'un nombre
double carre(double x){
 return x*x;
}//fin carre()

int main(){
 printf("%lf\n", pente(carre, 2));
 printf("%lf\n", pente(acos, 0));
 return 0;
}//fin programme()
```


Exemples (3)

- Exemple 2
  - manipulation d'une fonction de tri

```
#include <stdlib.h>

                tableau      taille tableau   taille élément tableau
void qsort(void *base, size_t nmemb, size_t size,

           int (*compar)(const void *, const void *));
                fonction de comparaison d'éléments 2 à 2
```

Exemple (4)

- Exemple 2 (cont.)
  - utilisation de `*compar`

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *){
  //code...

  int x = (*compar)(base[i], base[j]);
  if(x<0){
    //base[i] < base[j]
  }else{
    //etc
  }

  //code...
}//fin qsort()
```


Exemple (5)

- Exemple 2 (cont.)
  - implémentation de la comparaison

```
int compare_int(const void *a, const void *b){
  const int *a_int = (int *)a;
  const int *b_int = (int *)b;

  return (*a_int - *b_int);
}//fin compare_int()
```

Exemple (6)

- Exemple 2 (cont.)
  - utilisation

```
int main(){
  int tab[6] = {40, 10, 100, 90, 20, 25};

  affiche(tab, 6);
  qsort(tab, 6, sizeof(int), compare_int);
  affiche(tab, 6);

  return 0;
}//fin programme
```