Cours informatique : Maîtrise de Sciences Cognitives 2003-2004

Session 4 et 5 :

Notion de pointeurs et d'adressage indirect :

Nous avons vu que l'appel de fonctions nécessitait le passage des valeurs d'adresses pour que les résultats des traitements puissent être rangés dans les zones mémoires de la fonction appelante.

Ce principe peut-être utilisé dans une fonction quelconque pour permettre de désigner "indirectement" par son adresse une zone mémoire. La variable destinée à contenir l'adresse d'une autre variable est appelée POINTEUR.

Dans l'exemple suivant, 4 variables sont déclarées :

- 2 entiers : tampon1 et tampon 2

- 1 pointeur sur une variable de type entier p_entier: l'étoile (*) permet d'indiquer au compilateur que le type de cette variable est POINTEUR tandis que la déclaration "int" indique que l'adresse sera celle d'une variable destinée à contenir un entier.

- 1 entier : entier.

Commenter l'exemple en expliquant chaque ligne.

Exemple (version PDF)

```
#include <stdio.h>


int main()

{
int tampon1,tampon2;
int *p_entier;
int entier;


p_entier=&entier; //le pointeur d'entier p_entier prend la valeur de l'adresse de entier
*p_entier=5;//la variable dont l'adresse est dans p_entier prend la valeur 5

printf("l'entier vaut %i \n",entier);
printf("l'entier est à l'adresse %x \n",p_entier);
printf("le pointeur d'entier est à l'adresse %x \n",&p_entier);
tampon1=(int)p_entier;
tampon2=(int)&p_entier;
printf("un entier occupe donc %i octets \n",(tampon2-tampon1));
printf("l'entier tampon1 est à l'adresse %x \n",&tampon1);
printf("l'entier tampon2 est à l'adresse %x \n",&tampon2);
```

```
return 0;
}
```

Il n'est pas toujours possible de savoir à l'avance de quelle place mémoire on aura besoin lors de l'exécution d'un programme. En particulier, lorsqu'il s'agit de gérér des listes, des tableaux dynamiques (dont la taille peut augmenter ou se réduire selon les cas), il est utile de pouvoir définir une variable qui servira de repère au début de la structure (la première case d'un tableau typiquement), puis d'allouer la mémoire nécessaire à cette structure.

La fonction permettant d'allouer de la mémoire est la fonction malloc qui s'appelle de la manière suivante :

pointeur = (type d'unité d'allocation)malloc(nb d'unités d'allocation);

Exemple (version PDF)

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
char *tab;

tab=(char*)malloc(20);
strcpy(tab,"Alain Mille");
printf("la table tab contient la chaîne %s \n",tab);
return 0;
}
```

Autre exemple (version PDF):

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int tampon1;
double *ftab;
int tampon2;
int i;
double f;
f=0;
ftab=(double*)malloc(20);
for (i=0;i<19;i++)
{f=f+2.5;
ftab[i]=f;
printf("ftab [ %i ] = %g \n",i,ftab[i]);
};
printf("adresse de tampon2 = %x \n",&tampon2);
printf("adresse de ftab = %x \n", &ftab);
printf("adresse de tampon1 = %x \n", &tampon1);
```

```
printf("adresse de la première case du tableau = %x \n",ftab);
printf("adresse de la dernière case du tableau = %x \n",&ftab[19]);
tampon1=(int)ftab;
tampon2=(int)&ftab[19];
tampon1=(tampon2-tampon1)/18;
printf("la taille d'allocation est de %i octets \n",tampon1);

return 0;
}
```

Autre exemple pour l'allocation dynamique de variables de type "double" (version PDF)

Type abstrait ? Implantation en langage C :

Exemple de définition d'un type abstrait "Complexe" dans un fichier .h. (version PDF)

```
typedef struct{
double reel;
double imaginaire;
}complexe; // définition d'un type complexe à l'aide de la déclaration typedef

void charger_complexe(complexe *p_complexe,double reel, double imaginaire)
{
p_complexe->reel=reel;
p_complexe->imaginaire=imaginaire;
}

void recuperer_complexe(complexe *p_complexe, double *p_reel, double *p_imaginaire)
{
*p_reel=p_complexe->reel;
*p_imaginaire=p_complexe->imaginaire;
}

void ajouter_complexe(complexe *p_somme, complexe *p_complexe1, complexe *p_complexe2)
{
p_somme->reel=p_complexe1->reel + p_complexe2->reel;
p_somme->imaginaire=p_complexe1->imaginaire+ p_complexe2->imaginaire;
}

void multiplier_complexe(complexe *p_produit, complexe *p_complexe1, complexe
*p_complexe2)
{
p_produit->reel=p_complexe1->reel*p_complexe2->reel - p_complexe1->imaginaire*p_complexe2-
>imaginaire;
p_produit->imaginaire=p_complexe1->reel*p_complexe2->imaginaire + p_complexe1-
>imaginaire*p_complexe2->reel;
}


void afficher_complexe(complexe *p_complexe)

{ printf(" %g +i %g \n ",p_complexe->reel, p_complexe->imaginaire);}
```

Exemple d'utilisation du type abstrait Complexe....

```
#include<stdio.h>
#include "complexe.h"

int main()
{
complexe C1,C2;

charger_complexe(&C1,10.,5.);
printf(" %g +i %g /n",C1.reel, C1.imaginaire);
afficher_complexe(&C1);
charger_complexe(&C2,5.,10.);
afficher_complexe(&C2);
ajouter_complexe(&C1,&C2,&C1);
afficher_complexe(&C1);
multiplier_complexe(&C1,&C2,&C2);
afficher_complexe(&C1);
return 0;

}
```