Programmation en langage C

Plan

1. Rappels (Types de bases, variables, opérateurs, structures de contrôles,..)
2. Les fonctions
3. Les pointeurs
4. L'allocation dynamique
5. Les chaines de caractères
6. Les types composés (structures, énumérations,…)
7. Les fichiers
8. Les préprocesseurs

3 Les Pointeurs

Variables

- Les variables servent à stocker les données manipulées par le programme.
- Lorsque l'on déclare une variable, par exemple un entier « i », l'ordinateur réserve un espace mémoire pour y stocker les valeurs de i.
- L'emplacement de cet espace dans la mémoire est nommé adresse.

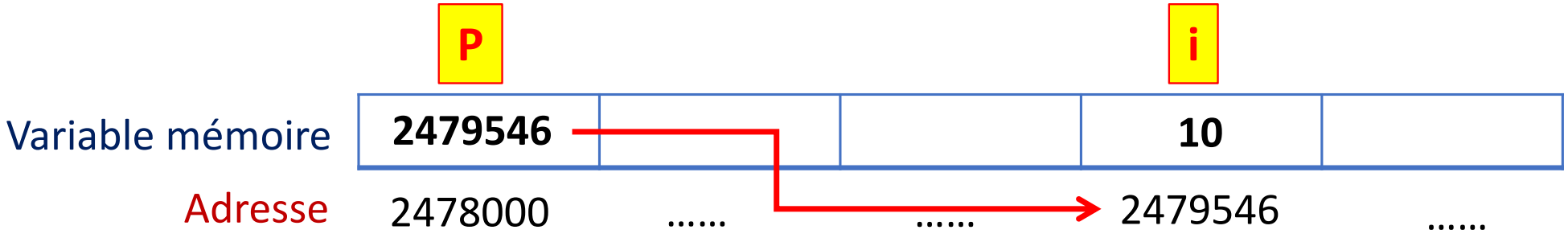
Pour connaître l'adresse d'une variable donnée, on utilise l'opérateur &.
Ainsi, &i vaut 2479546

Le nom d'une variable permet d'accéder directement à sa valeur (**adressage direct**).

3 Les Pointeurs

Variable Pointeur

- Un pointeur est une variable spéciale dont le contenu est une adresse.
- Le nom d'une variable est lié à la même adresse, alors qu'un pointeur peut pointer sur différentes adresses.
- Par exemple, si on déclare une variable entière « i » initialisée à 10 et que l'on déclare un pointeur p dans lequel on range l'adresse de « i », on a un schéma qui ressemble à :



On dit que p pointe sur l'adresse mémoire occupée par la variable i

Un pointeur qui contient l'adresse de la variable, permet d'accéder indirectement à sa valeur (**adressage indirect**).

Intérêts des pointeurs

Les pointeurs présentent de nombreux avantages :

- Ils sont indispensables pour permettre le passage par référence pour les paramètres des fonctions

- Ils permettent de créer des structures de données (listes et arbres) dont le nombre d'éléments peut évoluer dynamiquement. Ces structures sont très utilisées en programmation.

- Ils permettent d'écrire des programmes plus compacts et efficaces

Déclaration d'un pointeur

- Le type d'un pointeur dépend du type de la variable pointée. Ceci est important pour connaître la taille de la valeur pointée.

- On déclare un pointeur par l'opération d'indirection «*» :

type *nom-du-pointeur ;

- type est le type de la variable pointée
- * est l'opérateur qui indiquera au compilateur que c'est un pointeur

- **Remarque:** la valeur d'un pointeur donne l'adresse du premier octet parmi les n octets où la variable est stockée

Déclaration d'un pointeur

Exemple :

- **int *p; // déclaration d'un pointeur sur une variable entière**
- **float *ptr; // déclaration d'un pointeur sur une variable float**
- **char *ptr; // déclaration d'un pointeur sur char.**

3 Les Pointeurs

Opérateurs de manipulation

Lors du travail avec des pointeurs, nous utilisons :

- un opérateur **'adresse de': &** pour obtenir l'adresse d'une variable
- un opérateur **'contenu de': *** pour accéder au contenu d'une adresse

Exemple :

- **int *p; //on déclare un pointeur vers une variable de type int**
- **int i=10, j=30; // deux variables de type int**
- **p=&i; // on met dans p, l'adresse de i (p pointe sur i)**
- **printf("*p = %d \n",*p); //affiche : *p = 10**
- ***p=20; // met la valeur 20 dans la case mémoire pointée par p (i vaut 20 après cette instruction)**
- **p=&j; // p pointe sur j**
- **i=*p; // on affecte le contenu de p à i (i vaut 30 après cette instruction)**

3

Les Pointeurs

Initialisation d'un pointeur

A la déclaration d'un pointeur p, on ne sait pas sur quel zone mémoire il pointe. Ceci peut générer des problèmes :

int *p;

***p = 10;** //provoque un problème mémoire car le pointeur p n'a pas été initialisé

Conseil : Toute utilisation d'un pointeur doit être précédée par une initialisation.

On peut initialiser un pointeur en lui affectant :

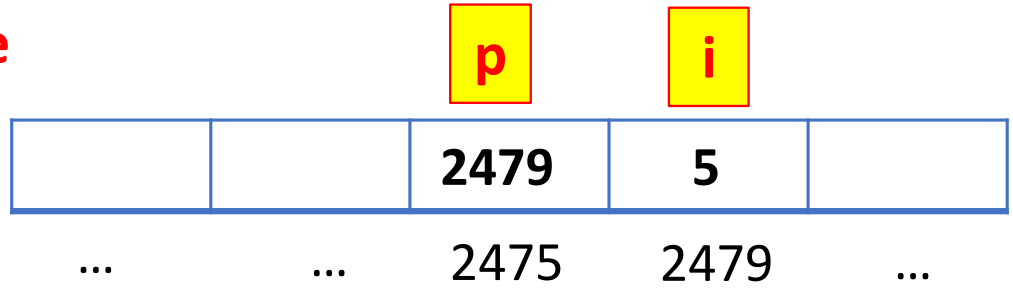
- l'adresse d'une variable
- un autre pointeur déjà initialisé
- la valeur 0 désignée par le symbole NULL, défini dans <stddef.h> (on dit que p pointe 'nulle part': aucune adresse mémoire ne lui est associé)

Initialisation d'un pointeur

Exemples :

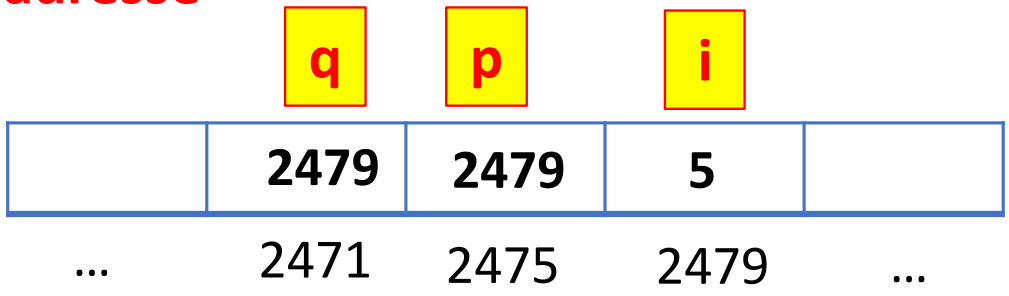
- **Pointeur qui pointe sur une variable**

```
int i=5;
int *p=&i;
```



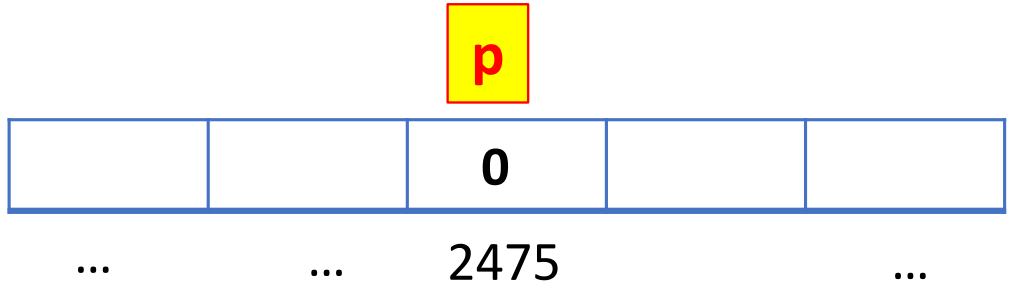
- **Pointeurs qui pointent sur la même adresse**

```
int i=5;
int *p=&i;
int *q=p;
```



- **Pointeur qui ne pointe sur rien**

```
int *ptr = NULL;
```



Opérateurs de manipulation

Remarque : si un pointeur **P pointe sur une variable X**, alors ***P** peut être utilisé partout où on peut écrire X

- **X+=2 équivaut à *P+=2**
- **++X équivaut à ++ *P**
- **X++ équivaut à (*P)++** // les parenthèses ici sont obligatoires car l'associativité des opérateurs unaires * et ++ est de droite à gauche

Opérations arithmétiques

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instructions

P1 = P2; fait pointeur P1 sur le même objet que P2

Remarque:

- on peut également utiliser les opérateurs **++** et **--** avec les pointeurs
- **la somme de deux pointeurs n'est pas autorisée**

3 Les Pointeurs

Opérations arithmétiques

Manipulations :

- **int i=5;**
- **int *p=&i;**
- **&i: l'adresse mémoire de la variable i**
  - **printf("%d\n",&i); //affiche par exemple 2479**
- **p: l'adresse mémoire sur laquelle pointe p**
  - **printf("%d\n",p); //affiche par exemple 2479**
- ***p: représente le contenu de la case mémoire sur laquelle p pointe**
  - **printf("%d\n",*p); //affiche par exemple 5**
- **&p: l'adresse mémoire de la variable p:**
- **printf("%d\n",&p); //affiche par exemple 2475**

3 Les Pointeurs

Pointeurs et tableaux

int tab[5]={1,58, 7};

- Le nom d'un tableau **tab** représente l'adresse de son premier élément (**tab=&tab[0]**)

→ on peut donc dire que **tab** (le nom d'un tableau) est un pointeur constant sur le premier élément du tableau.

- En déclarant un tableau T et un pointeur P du même type, l'instruction **P=T** fait pointer P sur le premier élément de T et crée une liaison entre P et le tableau T.

→ **P=T; est équivalente à P=&T[0];**

Pointeurs et tableaux

- A partir de là, on peut manipuler un tableau T en utilisant un pointeur P :
  - **P pointe sur T[0] et *P désigne T[0]**
  - **P+1 pointe sur T[1] et *(P+1) désigne T[1]**
  - **….**
  - **P+i pointe sur T[i] et *(P+i) désigne T[i]**

Exemple:

```
short x, A[7]={5,0,9,2,1,3,8};
short *P;
P=A;
x=*(P+5);
```

x est égale à la valeur **de A[5]**

Pointeurs : saisie et affichage d'un tableau

Version 1 :

```
main()
    {
        float T[100] , *pt;
        int i,n;

        do {printf("Entrez n \n " );
            scanf(" %d" ,&n);
            }while(n<0 ||n>100);
    pt=T;

    for(i=0;i<n;i++)
            { printf ("Entrez T[%d] \n ",i );
              scanf(" %f" , pt+i);
            }

     for(i=0;i<n;i++)
            printf (" %f \t",*(pt+i));
    }
```

Pointeurs : saisie et affichage d'un tableau

Version 2 :

```
main()
    {
        float T[100] , *pt;
        int n;

        do {printf("Entrez n \n " );
            scanf(" %d" ,&n);
            }while(n<0 ||n>100);

        for(pt=T;pt<T+n;pt++)
            { printf ("Entrez T[%d] \n ",pt-T );
            scanf(" %f" , pt);
        }

        for(pt=T;pt<T+n;pt++)
            printf (" %f \t",*pt);
    }
```

3 Les Pointeurs

Exercice 1

Soit P **un pointeur** qui **pointe** sur un **tableau** A tel que :

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou **adresses** fournissent ces expressions:

a) ***P+2** **= 14**

b) ***(P+2)** **=34**

c) **&A[4]-3** **=&A[1]**

d) **A+3** **=&A[3]**

e) **&A[7]-P** **= A + 7 – P = 7**

f) **P+(*P-10)** **=&A[2]**

g) ***(P+*(P+8)-A[7])** **= *(P + A[8] – A[7]) = *(P + 90 - 89) = *(P + 1) = A[1] = 23**

Pointeurs et tableaux à deux dimensions

- Le nom d'un tableau A à deux dimensions est un pointeur constant sur le premier élément du tableau càd **A[0][0].**
- En déclarant un tableau **A[n][m]** et un pointeur P du même type, on peut manipuler le tableau A en utilisant le pointeur P en faisant pointer P sur le premier élément de A (**P=&A[0][0]**)

Pointeurs et tableaux à deux dimensions

Ainsi :

- **P pointe sur A[0][0] et *P désigne A[0][0]**
- **P+1 pointe sur A[0][1] et *(P+1) désigne A[0][1]**
- **….**
- **P+M pointe sur A[1][0] et *(P+M) désigne A[1][0]**
- **….**
- **P+i*M pointe sur A[i][0] et *(P+i*M) désigne A[i][0]**
- **….**
- **P+i*M+j pointe sur A[i][j] et *(P+i*M+j) désigne A[i][j]**

Pointeurs : saisie et affichage d'une matrice

```
#define N 10
#define M 20
main( )
   { int i, j, A[N][M], *pt;
     pt=&A[0][0];
     for(i=0;i<N;i++)
         for(j=0;j<M;j++)
         {
            printf ("Entrez A[%d][%d]\n ",i,j );
            scanf(" %d" , pt+i*M+j);
         }

     for(i=0;i<N;i++)
         { for(j=0;j<M;j++)
             printf (" %d \t",*(pt+i*M+j));
             printf ("\n");
         }
}
```

Pointeurs et tableaux : remarques

En C, on peut définir :

- **Un tableau de pointeurs :**

  Ex : **int *T[10];** //déclaration d'un tableau de 10 pointeurs d'entiers

- **Un pointeur de tableaux :**

  Ex : **int (*pt)[20];** //déclaration d'un pointeur sur des tableaux de 20 éléments

- **Un pointeur de pointeurs :**

  Ex : **int **pt;** //déclaration d'un pointeur pt qui pointe sur des pointeurs d'entiers

Exercice 2

Ecrire **un programme** qui lit deux **tableaux A et B** et leurs dimensions N et M au clavier et qui **ajoute les éléments de B à la fin de A.** Utiliser le formalisme **pointeur** à chaque fois que cela est possible.

Exercice 2 - Correction

```c
#include <stdio.h>
main()
{
 /* Déclarations */
 int A[50], B[50]; /* tableaux */
 int N, M;  /* dimensions des tableaux */
 int I;     /* indice courant */

 /* Saisie des données */
 printf("Dimension du tableau A (max.50) : ");
 scanf("%d", &N );
 for (I=0; I<N; I++)
     {
      printf("Elément %d : ", I);
      scanf("%d", A+I);
     }
 printf("Dimension du tableau B (max.50) : ");
 scanf("%d", &M );
 for (I=0; I<M; I++)
     {
      printf("Elément %d : ", I);
      scanf("%d", B+I);
     }
```

```c
 /* Affichage des tableaux */
 printf("Tableau donné A :\n");
 for (I=0; I<N; I++)
     printf("%d ", *(A+I));
 printf("\n");
 printf("Tableau donné B :\n");
 for (I=0; I<M; I++)
     printf("%d ", *(B+I));
 printf("\n");

 /* Copie de B à la fin de A */
 for (I=0; I<M; I++)
     *(A+N+I) = *(B+I);

 /* Nouvelle dimension de A */
 N += M;
  /* Edition du résultat */
 printf("Tableau résultat A :\n");
 for (I=0; I<N; I++)
     printf("%d ", *(A+I));
 printf("\n");
 return 0;
}
```

3

Les Pointeurs

Exercice 3

Soient deux tableaux d'entiers. Ecrire **un programme en C** qui permet de tester **l'égalité entre les deux tableaux** : il rend **VRAI** si les composants des deux tableaux correspondent position par position, et **FAUX** sinon.

Exercice 3 - Correction

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
int t1[10],t2[10],*p,*k,a;
char e[10];
p=t1;

printf("donnez la dimension du tab");
scanf("%d",&a);

for(p=t1;p<t1+a;p++)
{printf("donnez les valeurs t1[%d]=",p-t1+1);
scanf("%d",p);
}

p=t2;
for(p=t2;p<t2+a;p++)
{printf("donnez les valeurs t2[%d]=",p-t2+1);
scanf("%d",p);
}
```

Exercice 3 - Correction

```
for(p=t1,k=t2;p<t1+a,k<t2+a;p++,k++)
{
    if(*p!=*k)
    {
        strcpy(e,"faux");
        break;
    }
    else
    {
        strcpy(e,"vrai");}
    }

    printf("%s",e);
    getch();
}
```

Pointeurs & Fonctions

1. Passage d’un pointeur à une fonction
2. Passage d’un tableau à un indice à une fonction
3. Passage d’un tableau à deux indices à une fonction

3 Les Pointeurs

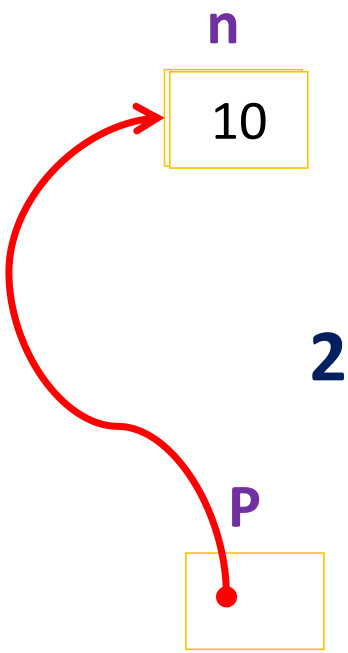
Pointeurs & Fonctions

```
void f(void)
{
      int n = 3;
      h(&n);
}

void h(int *P)
{
      *P = 10;
}
```

adresse de n

1. Appel de la fonction f



2. Appel de la fonction h

P pointe sur n => *P est équivalente à n

à l'aide de l'adresse contenue dans P, l'opérateur d'indirection peut affecter une nouvelle valeur à cet endroit de la mémoire

Pointeurs & Fonctions

1. Passage d’un pointeur à une fonction
2. **Passage d’un tableau à un indice à une fonction**
3. Passage d’un tableau à deux indices à une fonction

Pointeurs & Fonctions

On ne peut pas passer un tableau complet à une fonction qui travaillerait sur une copie locale de ce tableau dans la fonction

Alternative : passage d'un paramètre par élément du tableau

Solution générale : repose sur l'équivalence entre le nom du tableau et un pointeur constant

- lorsqu'on transmet le nom d'un tableau, on transmet l'adresse de son premier élément
- cette adresse initialise un paramètre formel de type pointeur

Conséquence : un tableau passé à une fonction est modifiable dans cette fonction

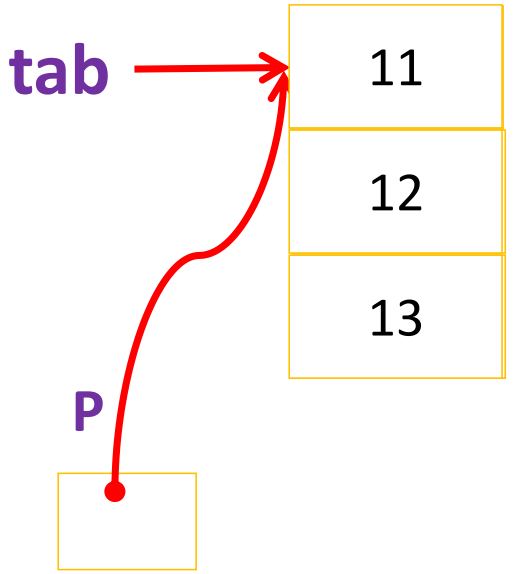
Pointeurs & Fonctions

Exemple :

```
void f(void)
{
    int tab[3] = {10, 20, 30};
    h(tab);
}

void h(int *P)
{
    *P = 11; // P[0] = 11;
    *(P + 1) = 12; // P[1] = 12;
    *(P + 2) = 13; // P[2] = 13;
}
```

adresse du premier élément



Exemple : remplissage et affichage d'un tableau passé en paramètre d'une fonction

```c
#include <stdio.h>
#define N 4
void remplir(int *tableau, int tailleTableau);
void affiche(int *tableau, int tailleTableau);

int main(int argc, char *argv[])
{
    int tab[N];
    remplir(tab, N);
    affiche(tab, N);
    return 0;
}

void remplir(int *tableau, int tailleTableau)
{
    int i;
    for (i = 0 ; i < tailleTableau ; i++)
    {
        printf("Element %d = ", i);
        scanf("%d", &tableau[i]);
    }
}
```

```c
void affiche(int *tableau, int tailleTableau)
{
    int i;
    for (i = 0 ; i < tailleTableau ; i++)
    {
        printf("%d\n", tableau[i]);
    }
}
```

Pointeurs & Fonctions

Remarque : il existe une autre façon d'indiquer que la fonction reçoit un tableau. Plutôt que d'indiquer que la fonction attend un int *tableau

```
void affiche(int tableau[], int tailleTableau)
```

Cela revient exactement au même, mais la présence des crochets permet au programmeur de bien voir que c'est un tableau que la fonction prend, et non un simple pointeur. Cela permet d'éviter des confusions.

Pointeurs & Fonctions

1. Passage d’un pointeur à une fonction
2. Passage d’un tableau à un indice à une fonction
3. Passage d’un tableau à deux indices à une fonction

Pointeurs & Fonctions

Même principe que pour un tableau à un seul indice : transmission d'une adresse, mais il faut en plus indique la taille d'une ligne (c.à.d. le nombre de colonnes)

```
void saisieMatrice(int (*mat)[M])
{
    int i,j;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
        {
            .....
         }
}
```

Exemple : remplissage et affichage d'une matrice passé en paramètre d'une fonction

```c
#include <stdio.h>
#define N 2
#define M 3
void remplir(int (*mat)[M]);
void affichage(int (*mat)[M]);
int main()
{
    int matrice[N][M];
    remplir(matrice);
    affichage(matrice);
    return 0;
}
void remplir(int (*mat)[M])
{
    int i,j;
    for(i=0; i<N; i++)

        for(j=0; j<M; j++)
        {
            printf("Element [%d][%d] = ",i,j);
            scanf("%d", &mat[i][j]);
        }

}
```

```c
void affichage(int (*mat)[M])
{
    int i,j;
    for(i=0; i<N; i++)
    {
        for(j=0; j<M; j++)
        {
            printf("%d\t",mat[i][j]);
        }
        printf("\n");
    }
}
```

Les pointeurs sur les fonctions

Les pointeurs sur les fonctions

objectif : **arriver à appeler une fonction par son adresse.**

3 Les Pointeurs

Les pointeurs sur les fonctions

Rappel :

- Pour chaque **variable déclarée**, **une place mémoire lui est attribuée**. Cette place mémoire va servir pour contenir les valeurs qu'on affectera à notre variable. Et cet espace sera connu par **une adresse**.
- Chaque objet créé en mémoire, a une adresse propre à lui. Cette adresse peut être stockée dans un type qu'on appelle ***pointeur***.

- Un pointeur doit avoir le même type que l'objet pointé.

  Exemple : un pointeur sur int, doit contenir une adresse d'une variable int

```
int * pointeur; //déclare un pointeur de type int
int var; //déclare une variable de type int
pointeur = &var; //on met dans pointeur l'adresse de la variable var
```

Les pointeurs sur les fonctions

Ce typage des pointeurs peut s'avérer gênant dans certaines circonstances telles que celles où une fonction doit manipuler les adresses d'objets de type non connu (ou, plutôt susceptible de varier d'un appel à un autre).

- **Utilisation d'un pointeur sur un objet quelconque : pointeur générique**

Les pointeurs sur les fonctions

Le pointeurs générique :

- C'est un pointeur vers un objet de n'importe quel type.
- Il peut contenir, soit l'adresse d'un objet de n'importe quel type, soit un autre pointeur de n'importe quel type.
- Il est désigné par **void ***.

Exemple : void * pointeurGenerique;

Les pointeurs sur les fonctions

Le pointeurs générique :

C'est un pointeur sans type → on ne peut pas utiliser des opérations arithmétiques :

- Si p est de type void *:
  - on ne peut pas parler de **p+i** ou **p-i** (i étant un entier)
  - on ne peut pas utiliser l'expression **p++** (on ne connaît pas la taille des objets pointés)
  - on ne peut pas utiliser l'opérateur d'indirection ***

Les pointeurs sur les fonctions

Le pointeurs générique :

- Un objet pointé par un pointeur générique ne peut contenir une valeur qu'après avoir spécifié le type de l'objet pointé par le pointeur générique par le type de la valeur.

- Un pointeur ne peut contenir un pointeur générique qu'après avoir spécifié le type du pointeur générique par le type du pointeur.

Les pointeurs sur les fonctions

Le pointeurs générique :

Exemple :

```
INT I ;
FLOAT F ;
VOID *P ;
INT *Q ;
P = &I ;                  // permit
P = &F ;                  // permit
*P = 2.5 ;                // interdit
*(FLOAT *)P = 2.5 ;       // permit
P = Q ;                   // permit
Q = P ;                   // interdit
Q = (INT *)P ;            // permit
```

Les pointeurs sur les fonctions

Les pointeurs et les fonctions :

- **Une fonction a également une adresse mémoire** et donc un pointeur capable de sauvegarder cette adresse mémoire, pour qu'elle soit utilisée par la suite.
- Identiquement aux variables on peut utiliser **l'opérateur '&', suivi du nom de la fonction pour avoir son adresse.**

Les pointeurs sur les fonctions

Pointeur de fonction = **variable dont la valeur correspond à à l'adresse du début du code d'une fonction**

Les pointeurs sur les fonctions

Les pointeurs et les fonctions :

Exemple :

```
void ma_fonction(void)
{
  //instructions...
}

pointeurSurFonction = &(ma_fonction); /* Cela stockera l'adresse de 'ma_fonction' dans
                                         la variable 'pointeurSurFonction'*/
```

Les pointeurs sur les fonctions

Les pointeurs et les fonctions :

- Le nom d'une fonction est un pointeur dit statique. En conséquence, **l'utilisation de l'opérateur '&' est facultative**, ainsi **seulement par l'utilisation du nom de la fonction on récupère son adresse.**

```
pointeurSurFonction = ma_fonction;
```

Les pointeurs sur les fonctions

Les pointeurs et les fonctions :

- Si on combine les deux notions : la déclaration d'un **pointeur générique**, et la lecture de **l'adresse d'une fonction**, on aura alors un code comme ceci :

```
void ma_fonction (void)
{
  printf("Hello world!\n");
}

void * pointeurGenerique = ma_fonction ;
/* on a gardé l'adresse de notre fonction dans le pointeur générique. Ce pointeur ne
      permet pas d'appeler la fonction/*
```

Les pointeurs sur les fonctions

- Généralement, un pointeur sur fonction est déclaré suivant la syntaxe :

type de retour (*nom du pointeur) (types des paramètres);

- Ainsi ces pointeurs sont déclarés en respectant le prototype de la fonction sur laquelle ils vont pointer.

Exemples :

```
int  (*ptr)  (int, int, char *);

void  (*ptr2) (int);
```

Les pointeurs sur les fonctions

- **Fonction sans retour et sans arguments**
- **Fonction sans retour et avec arguments**
- **Fonction avec retour et sans arguments**
- **Fonction avec retour et avec arguments**

Les fonctions sans retour/sans arguments

Fonction sans retour et **sans arguments**

- On appelle **procédure ou fonction sans retour**, toutes fonctions ayant le type void comme type de retour : **void ma_fonction(....);**
- On appelle **fonction sans arguments** (ou sans paramètres), toutes fonctions ayant le type void en argument : **void ma_fonction(void);**
- Ainsi la déclaration d'un pointeur sur ce type de fonctions, à savoir sans arguments et sans retour, est comme ceci :

```
void (*pointeurSurFonction)(void);
```

Les fonctions sans retour/sans arguments

Fonction sans retour et **sans arguments**

Exemple :

```
void ma_fonction(void);          /*Prototype*/

void ma_fonction(void)           /*La fonction*/
{
   printf("Bonjour!\n");
}

int main(void)
{
   void (*pointeurSurFonction)(void);  /*Déclaration du pointeur*/

   pointeurSurFonction = ma_fonction;  /*Sauvegarde de l'adresse de la fonction
                                         dans le pointeur adéquat*/

   return 0;
}
```

Les fonctions sans retour/avec arguments

Fonction sans retour et **avec arguments**

- Une fonction est dite **avec argument** si dans la liste de ses arguments, il y en a, au moins un, différent du type void :

```
void ma_fonction(int argint);
```

- La déclaration d'un pointeur sur ce type de fonctions est comme ceci :

```
void (*pointeurSurFonction)(int)
```

Les fonctions sans retour/avec arguments

Fonction sans retour et **avec arguments**

- Même principe pour les fonctions à plusieurs arguments :

Exemple :

```
void ma_fonction(int argint,char * argchar,float leng);

void (*pointeurSurFonction)(int,char *,float);
```

- La sauvegarde de l'adresse de la fonction reste la même :

```
pointeurSurFonction = ma_fonction;
```

Les fonctions avec retour/sans arguments

Fonction **avec retour**

- Une fonction est dite **"avec retour"** quand le type de retour est différent du void.

Sans arguments

La déclaration d'un pointeur sur une fonction du genre **int ma_fonction(void)** est :

```
int (*pointeurSurFonction)(void)
```

Avec arguments

La déclaration d'un pointeur sur une fonction du genre **int * ma_fonction(double * f,int n)** est :

```
int * (*pointeurSurFonction)(double*,int);
```

Appels de fonctions en utilisant leurs adresses

Opérateur d'appel de fonctions

- L'appel aux fonctions est caractérisé par la présence de parenthèses **'()'**. Dans le cas général, **on appellera une fonction à partir de son pointeur** ainsi :

(*nom_du_pointeur_sur_fonction)(liste_d'arguments)

- La valeur de retour peut être récupérée ainsi :

variable_receptrice = (*nom_du_pointeur_sur_fonction)(liste_d'arguments)

Appels de fonctions en utilisant leurs adresses

Exemple 1 : Fonction sans retour et sans arguments

```
void afficherBonjour(void)
{
   printf("Bonjour\n");
}
int main (void)
{
    void (*pointeurSurFonction)(void);      /*déclaration du pointeur*/
    pointeurSurFonction = afficherBonjour;   /*Initialisation*/

    (*pointeurSurFonction)();              /*Appel de la fonction*/

    return 0;
}
```

Bonjour

Appels de fonctions en utilisant leurs adresses

Exemple 2 : Fonction sans retour et avec arguments

```
void somme(int a, int b)
{
	printf("La somme est %d\n",a+b);
}

int main (void)
{
	void (*pointeurSurFonction)(int, int); /*déclaration du pointeur*/
	pointeurSurFonction = somme; /*Initialisation*/
	(*pointeurSurFonction)(2,3); /*Appel de la fonction*/
	return 0;
}
```

La somme est 5

Appels de fonctions en utilisant leurs adresses

Exemple 3 : Fonction avec retour et sans arguments

```
int saisirNombre(void)
{
	int n;
	printf("Saisissez un nombre entier : ");
	scanf("%d",&n);
	return n;
}

int main (void)
{
	int (*pointeurSurFonction)(void); /*déclaration du pointeur*/
	int nombre;
	pointeurSurFonction = saisirNombre; /*Initialisation*/
	nombre = (*pointeurSurFonction)(); /*Appel de la fonction*/
	printf("Le nombre est : %d",nombre);
	return 0;
}
```

```
Saisissez un nombre entier : 4
Le nombre est : 4
```

Appels de fonctions en utilisant leurs adresses

Exemple 4 : Fonction avec retour et avec arguments

```
int sommeNombres(int a, int b)
{
	return a+b;
}

int main (void)
{
	int (*pointeurSurFonction)(int,int); /*déclaration du pointeur*/
	int somme;
	pointeurSurFonction = sommeNombres; /*Initialisation*/
	somme = (*pointeurSurFonction)(5,10); /*Appel de la fonction*/
	printf("La somme est : %d", somme);
	return 0;
}
```

```
La somme est : 15
```

Tableau de pointeurs sur fonctions

- La **déclaration d'un tableau de pointeurs sur fonction** se fait sous la forme suivante :

type_de_retour (* nom_du_tableau_de_pointeurs_sur_fonctions [taille_du_tableau])

(liste_des_arguments);

- Son utilisation est soumise à toutes les règles d'utilisation des tableaux ordinaires, à savoir le premier indice est 0 et le dernier est *taille* – 1

- Tous les pointeurs qui seront stockés dedans doivent avoir le même type (les fonctions qu'on mettra dedans doivent avoir le même prototype, càd le même type de retour et les mêmes arguments).

Retourner un pointeur sur fonction

Méthode normale

- Si l'on souhaite **retourner un pointeur sur fonctions** dans une de nos fonctions, on utilise la syntaxe suivante :

type_de_retour_de_la_fonction_pointee (* nom_de_la_fonction_de_renvoi (liste_arguments))(liste_arguments_fonction_pointee)

Retourner un pointeur sur fonction

Méthode normale

- **Exemple pour une fonction sans arguments :**

```
int fonction1(void)
{
        return 1;
}

int (* fonction2(void))(void)
{
        return fonction1;   /*Ici le retour d'un pointeur sur fonction*/
}
```

Retourner un pointeur sur fonction

Méthode normale

- **Exemple pour une fonction avec arguments :**

```
int fonction1(double,double)
{
        return 1;
}

int (* fonction2(char str[]))(double,double)
{
        printf("%s",str);    /*Affichage de la chaine passée en paramètre*/
        return fonction1; /*Ici le retour d'un pointeur sur fonction*/
}
```

Retourner un pointeur sur fonction

Utilisation d'un typedef

- Avec cette méthode, on va pouvoir **déclarer un pointeur sur fonction**, en mettant **typedef** à sa gauche :

typedef type_retour (* nom_du_pointeur) (liste_arguments)

- Et ceci **dans le champ des déclarations globales** (c'est-à-dire en dehors des fonctions).

Retourner un pointeur sur fonction

Utilisation d'un typedef

- **Exemple pour une fonction sans arguments :**

```
typedef int (*ptrFonction)(void);

int fonction1(void)
{
    return 1;
}

ptrFonction fonction2(void)
{
    return fonction1;       /*Ici le retour d'un pointeur sur fonction*/
}
```

Retourner un pointeur sur fonction

Utilisation d'un typedef

- **Exemple pour une fonction avec arguments :**

```
typedef int (*ptrFonction)(double,double);

int fonction1(double,double)
{
        return 1;
}

ptrFonction fonction2(char str[])
{
        printf("%s",str); /*Affichage de la chaine passée en paramètre*/
        return fonction1; /*Ici le retour d'un pointeur sur fonction*/
}
```

Passage de pointeur de fonctions en paramètre

- Il s'agit de passer un pointeur sur fonction en paramètre à une autre fonction.

Méthode normale

```
void fonction1(int n)
{
   printf("fonction 1 appel N° %d\n",n);
}

void fonction2(int n, void (*ptrfonction)(int))
{
    (*ptrfonction)(n);      /*Appel de la fonction pointée par ptrfonction*/
}
int main(void)
{
       fonction2(13,fonction1);
}
```

Passage de pointeur de fonctions en paramètre

Utilisation de typedef

- Permet d'utiliser le pointeur de fonction comme n'importe quel autre type

```
typedef void (*ptrfonction)(int);

void fonction1(int n)
{
	printf("fonction 1 appel N° %d\n",n);
}
void fonction2(int n, ptrfonction ptr)
{
	(*ptr)(n);	/*Appel de la fonction pointée par ptrfonction*/
}
int main(void)
{
	fonction2(13,fonction1);
}
```

Les pointeurs sur les fonctions

Exercice

Ecrire un programme qui affiche un menu avec les calculs d'addition, soustraction, multiplication et division à l'utilisateur, et lui demande d'en choisir un. Ensuite il lui demande de saisir les opérandes.
Le programme devra faire le calcul choisi avec les opérandes entrées, et afficher le résultat.

Etapes :

1)Les fonctions de calcul : écrire 4 fonctions traitants les différents calculs (addition, soustraction, multiplication et division)
2)Déclaration et initialisation d'un tableau de pointeurs sur fonctions : créer un tableau de pointeurs sur fonctions, qu'on initialisera avec les fonctions de calcul
3)Affichage du menu : créer une fonction qui va afficher un menu, proposant à l'utilisateur de choisir une opération
4)La fonction main : appel de la fonction, lecture des opérandes et affichage du résultat

Les pointeurs sur les fonctions

Exercice - Correction

Les fonctions de calcul

```
double addition(double n1,double n2)
{
    return n1 + n2;
}
double soustraction(double n1,double n2)
{
return n1 - n2;
}
double multiplication(double n1,double n2)
{
    return n1 * n2;
}
double division(double n1,double n2)
{
    return n1 / n2;
}
```

Les pointeurs sur les fonctions

Exercice - Correction

Déclaration et initialisation d'un tableau de pointeurs sur fonctions

double (*listeFonctions[4])(double,double) = {addition,soustraction,multiplication,division};

- Pour déclarer un pointeur sur fonction : **double (*listeFonctions)(double,double)**
- Donc la déclaration d'un tableau est de la forme : **double (*listeFonctions[4])(double,double)**
- Ce tableau est ensuite initialisé avec : **{addition,soustraction,multiplication,division};**

Les pointeurs sur les fonctions

Exercice - Correction

Affichage du menu

```c
double (* affichMenu(void))(double,double)        /*Le retour correspondant à un pointeur sur fonction*/
{
    int choix;                                    /*La variable pour le choix*/
    do{
        printf("------------------------------MENU-------------------------------------\n");
        printf("Veuillez choisir une operation (en choisissant un nombre entre 1 et 4) :\n");
        printf("1 pour addition\n");
        printf("2 pour soustraction\n");
        printf("3 pour multiplication\n");
        printf("4 pour division\n");
        printf("Votre choix : ");
        scanf("%d",&choix);

    }while(choix < 1 || choix > 4 );              /*En cas d'erreur de saisie*/

    return listeFonctions[choix - 1];             /*On renvoi le pointeur sur la fonction de calcul choisie*/
}
```

Les pointeurs sur les fonctions

Exercice - Correction

La fonction main

```c
int main (void)
{
    double (*fonctionDeCalcul)(double,double);        /*déclaration du pointeur*/
    double n1,n2;

    fonctionDeCalcul = affichMenu();                  /*On charge la fonction de calcul choisie*/

    printf("Saisissez les operandes : ");             /*On lit les opérandes de notre calcul*/
    scanf("%lf",&n1);
    scanf("%lf",&n2);

    /*On appelle la fonction choisie et on affiche le résultat.*/
    printf("le resultat du calcul est : %f\n",(*fonctionDeCalcul)(n1,n2));
    return 0;
}
```