Programmation Impérative

FLIN202

Vincent BOUDET

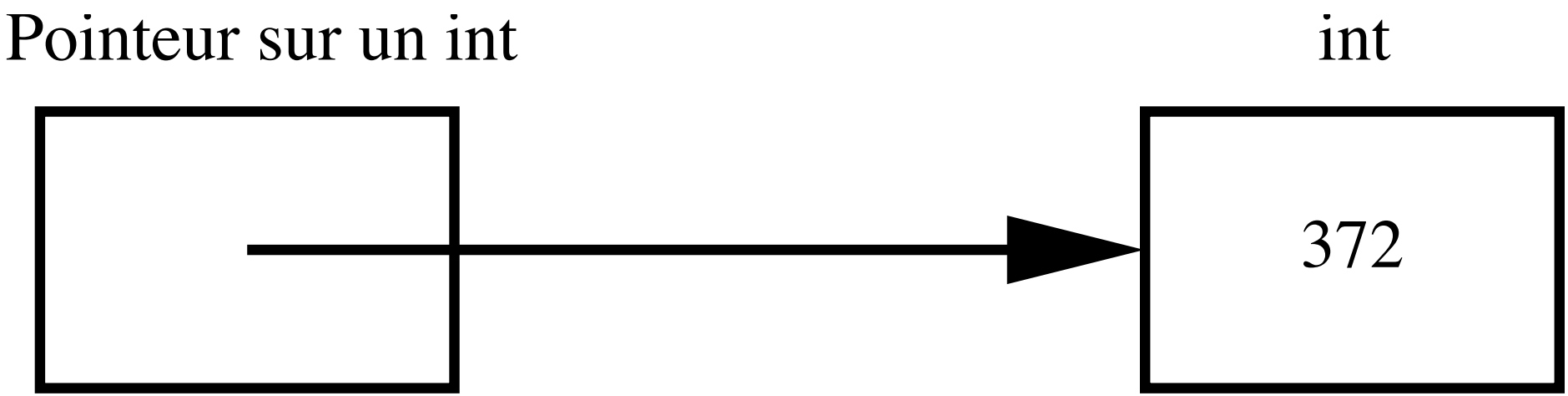
Janvier 2008

Sommaire

1. Pointeurs
   - Définition
   - Les pointeurs en C
   - Pointeurs en paramêtre
   - Tableau dynamique
   - Pointeurs et struct
   - Pointeur en retour de fonction

Notion de pointeurs

Une variable de type pointeur est une case mémoire qui contient un entier positif donnant l'adresse d'une autre case.



Historique

La manipulation des pointeurs étaient interdites à l'utilisateur dans les premiers langages. Seul les compilateurs utilisaient les pointeurs en interne pour dresser la liste des variables. Pour chaque variable, le compilateur mémorise l'emplacement mémoire où est stockée la valeur de la variable. On a donc bien affaire à des pointeurs.
Depuis le langage PASCAL, les pointeurs font partie de tous les langages.
Un pointeur n'est pas un concept unique et difficile. Ce n'est en fin de compte qu'un indice d'un très grand tableau qu'est la mémoire de l'ordinateur.

Déclaration

Un programmeur qui a besoin d'un pointeur va déclarer une variable de type "pointeur sur une variable de type T" où T est un type connu (ou défini) du langage.

- Cette déclaration indique que la variable pourra contenir un entier positif qui représente l'adresse d'une autre case. Habituellement cette case correspond à une autre variable.

Déclaration

Un programmeur qui a besoin d'un pointeur va déclarer une variable de type "pointeur sur une variable de type T" où T est un type connu (ou défini) du langage.

- Elle indique aussi le type T que devra avoir cette variable désignée.

  Avantage : On connaît la taille exacte de la place occupée par cette variable désignée.

  Restriction : Quand on veut changer de variable désignée, c'est obligatoirement pour une autre variable de même type.

Déclaration

Un programmeur qui a besoin d'un pointeur va déclarer une variable de type "pointeur sur une variable de type T" où T est un type connu (ou défini) du langage.

- Comme toutes les déclarations de variables, elle entraîne la réservation d'une place pour la variable déclarée (donc la place d'un entier).

Déclaration

```
nomType * ident
```

déclare une variable de type pointeur vers une variable de type nomType.

Exemple

```
int * p ;
double * z ;
float * k ;
```

Remplissage I

Un pointeur va contenir l'adresse d'une case mémoire, mais ce n'est pas le programmeur qui choisit les numéros des cases qu'il utilise.
Les possibilités pour mettre une valeur dans un pointeur sont restreintes.

Remplissage II

Remplissage par l'adresse d'une variable du programme

Opérateur "Adresse de"

L'opérateur & appliqué à une variable délivre l'adresse de celle-ci. Cette adresse pourra être affectée à une variable de type pointeur. On peut ainsi écrire :

```
int i ;
int * pi ;
pi=&i ;
```

voir schéma au tableau.

Remplissage II

- Affectation entre pointeurs : si p et q sont des pointeurs sur des variables de même type, on peut écrire $p = q$. p et q désigneront alors la même case mémoire.

Remplissage III

Initialisation de précaution

Lorsqu'on déclare un pointeur, la case réservée contient n'importe quoi. Si on l'utilise en oubliant de la remplir, on va accéder à une case mémoire qui ne nous appartient pas (autre programme, autre utilisateur...) et provoque une erreur de segmentation. Pour éviter ce problème, on dispose d'une valeur spéciale NULL. Cette valeur indique que pour le moment le pointeur ne contient pas une vraie adresse.

Remplissage IV

Par l'adresse d'une variable créée spécifiquement

C'est l'ulisation la plus fréquente et la plus puissante. Si on a déclarée une variable *p* "pointeur sur une variable de type T", une instruction existe dans chaque langage pour faire 2 choses à la fois :

- créer une variable de tpe T
- placer l'adresse de cette variable dans le pointeur

Cette nouvelle variable n'a pas de nom propre. Elle porte un nom dérivé de celui du pointeur, qui signifie "la variable dont l'adresse est actuellement dans le pointeur"

Remplissage IV

```
ident = (nomType *) malloc (sizeof(nomType));
```

crée une variable de type `nomType` et place son adresse dans `ident`. Cette variable pointée ou désignée par le pointeur s'appelle `*ident`.

Exemples

```
int * p ;
p=(int *) malloc (sizeof (int) ) ;
double * z ;
z = (double *) malloc (sizeof (int) ) ;
```

Opérateur "Indirection"

L'opérateur * appliqué à un pointeur donne la valeur pointée :

```
int i ;
int * pi ;
pi=&i ;
*pi=2 ;
```

Nettoyage

Quand le programmeur a créé une variable spécifiquement, il doit l'effacer explicitement quand il n'en a plus besoin.

```
free(ident)
```

rend au système la place occupée par la variable pointée (pas celle du pointeur). C'est en quelque sorte l'opération inverse de malloc.

Directives obligatoires

Il faut ajouter au début du programme :

```
#include <stdlib.h>
```

pour que les instructions de création et de nettoyage soient connues.

Voir explications au tableau.

Exemple

```
#include<stdio.>
#include<stdlib.h>
int  main()
{
        int  *p ; int  *  q ;
        p=NULL ;
        q=(int  *)  malloc  (sizeof(int)) ;
        *q=3 ;
        p=q ;
        *p=*q+1 ;
        p=(int  *)malloc(sizeof(int)) ;
        *p=*q+1 ;
}
```

Pointeurs en paramêtre

Comme n'importe quelle variable, un pointeur peut être passée en paramêtre.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
void incr(int * p)
{
        *p=*p+1 ;
}
int main()
{
        int i=1 ;
        int * pi ;
        pi=&i ;
        incr(pi) ;
        printf("i vaut %d\n",i) ;
}
```

Pointeurs en paramêtre

J'aurais pu écrire aussi :

Exemple

```
#include <stdio.h>
#include <stdlib.h>
void incr(int * p)
{
        *p=*p+1 ;
}
int main()
{
        int i=1 ;
        incr(&i) ;
        printf("i vaut %d\n",i) ;
}
```

Pointeurs en paramêtre

Vous avez bien compris...

Un paramêtre modifiable n'est rien d'autre qu'un pointeur sur ce paramêtre.

Quand on fait un appel à une fonction, les valeurs des paramêtres effectifs sont copiées dans les paramêtres formels. Il se passe la même chose avec les pointeurs, la valeur copiée étant l'adresse d'une autre variable, ce qui permet de manière détournée de modifier une variable.

Création I

Quand on a déclaré un pointeur `nomType * ident ;` on peut exécuter une instruction `malloc` différente de celle qui a été vue précédemment : on peut demander de réserver de la place pour plusieurs variables de type `nomType` contigües.

```
ident = (nomType *)malloc(nbPlaces*sizeof(nomType)) ;
```

où `nbPlaces` est une expression entière indiquant combien de variables on veut. Dans ce cas, l'adresse placée dans `ident` est celle de la première variable.

Exemple

```
int * p ; p=(int *) malloc(5 * sizeof(int)) ;
double * z ; p=(double *) malloc(3 * sizeof(double)) ;
```

Création II

Noms des variables créées

- La première variable s'appelle `*ident`, la deuxième `*(ident+1)`, la i-ème `*(ident+i)`, etc.

Création II

Noms des variables créées

- La première variable s'appelle `*ident`, la deuxième `*(ident+1)`, la i-ème `*(ident+i)`, etc.
- Notation équivalente : on peut écrire `ident[i]` à la place de `*(ident+i)` : tout s'écrit alors comme si on avait un tableau habituel.

Pointeurs et struct

On peut bien sur créer des pointeurs sur des objets de type struct. Pour illustrer tout ceci, nous allons utiliser la structure suivante :

Définition de notre structure

```
struct noteAn
{
int numId ;
float note ;
} ;
```

On déclarera un pointeur sur une structure de la manière suivante :

```
struct noteAn * pn;
pn=(struct noteAn *) malloc (sizeof(struct noteAn));
```

Accès aux champs de la variable pointée

```
(*pn).numId
(*pn).note
```

Il existe une notation équivalente :

```
pn->numId
```

Exemple élémentaire

Syntaxe

```
#include <stdio.h>
#include <stdlib.h>
int * f()
{
        int *p; p=(int *)malloc(sizeof(int));
        printf("valeur :");
        scanf("%d",&(*p));
        return p;
}
int main()
{
        int *z; z=f();
        printf("Valeur contenue : %d\n",*z);
}
```

Exemple Utile

Pour alléger un peu la syntaxe, on peut écrire une fonction qui crée un tableau dynamique.

Tableau dynamique

```
#include <stdio.h>
#include <stdlib.h>
int * CreerTableau(int n)
{
        int *p;
        p=(int *)malloc(n*sizeof(int));
        return p;
}
int main()
{
        int * z;
        z=CreerTableau(10);
        ...
```

Un dernier exemple...

Pointeurs et struct (bis)

```
#include<stdio.h>
#include<stdlib.h>
struct liNote
{int num;int nbNotes;double* notes;};
int main()
{ struct liNote l1;
  printf("numero : ");scanf("%d",&l1.num);
  printf("nombre de notes : ");scanf("%d",&l1.nbNotes);
  l1.notes=(double*)malloc (l1.nbNotes*sizeof(double));
  printf(" notes \n ");
  for (int i=0;i<l1.nbNotes;i++)
    {scanf("%lf",&l1.notes[i]);}
  for (int i=0;i<l1.nbNotes;i++)
    {printf("%g ",l1.notes[i]);}
  printf("\n ");
```