Chapitre 11

LES POINTEURS

Dans le chapitre dédié aux fonctions, nous avons évoqué les passages de paramètres par valeur et par adresse. Nous connaissons l'opérateur unaire `&`, nous l'avons notamment utilisé avec la fonction `scanf()`. Cette dernière place la valeur saisie par l'utilisateur à l'emplacement mémoire spécifié par l'adresse (dans la variable). Mais comment créer une fonction qui, tout comme `scanf()`, acceptera non plus une variable mais son adresse, nous permettant ainsi de modifier directement son contenu ? C'est ici que la notion de pointeur entre en jeu ...

11.0.1 L'opérateur unaire &

L'opérateur adresse `&` retourne l'adresse d'une variable en mémoire.

Exemple 11.1. : *Adresse d'une variable.*

```
1  int maVariable = 8;
2  printf("Valeur de maVariable = %d\n", maVariable);
3  printf("Adresse de maVariable (format complet sur 64 bits) : %p\n", &maVariable);
4  printf("Adresse tronquée (sur 32 bits uniquement) de maVariable : %x\n", &maVariable);
```

En compilant l'exemple 11.1 sur une machine 64 bits, vous obtiendrez un avertissement/*warning* et constaterez à l'exécution que la ligne 4 n'affiche que les dernières digits de l'adresse. Le spécificateur de format **%p** (pointeur) permet l'affichage d'une adresse mémoire au complet et en héxadécimal.

11.0.2 Définition

Un **pointeur** est une variable contenant une **adresse mémoire**.

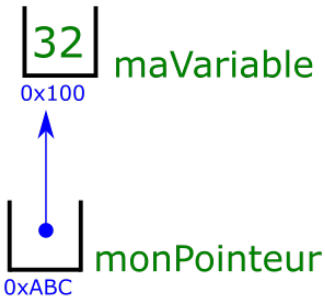

Fig. 11.1. Représentation d'un pointeur.

11.0.3 Déclaration

Une variable de type pointeur se déclare à l'aide du type de la case à pointer suivi du symbole *** (opérateur d'indirection)** puis de l'identificateur de votre pointeur.

Syntaxe :

```
type_a_pointer *identificateur
```

Exemple 11.2. : *Déclaration d'un pointeur.*

```
int *monPointeurSurTypeEntier;
```

Attention :

- **Déclarer un pointeur ne réserve pas un emplacement pour une information pointée.**
- Utiliser un pointeur non initialisé est une erreur difficile à déceler. En effet, le programme peut très bien fonctionner à certains moments (si la zone pointée n'est pas utilisée) et donner des résultats imprévisibles voir planter à d'autres moments.

11.0.4 Affectation, initialisation

Pour initialiser un pointeur, nous devons lui affecter une adresse mémoire.

Syntaxe :

```
identificateur = &mémoire // nous utilisons les opérateurs d'affectation et d'adresse
```

Exemple 11.3. : *Initialisation d'un pointeur.*

```
int maVariable = 8;
int *monPointeur;
monPointeur = &maVariable;
```

Remarque : il n'est pas possible d'affecter une constante à un pointeur, excepté pour le nombre 0 qui est déclaré équivalent à NULL dans stdio.h et représente un pointeur ne pointant sur rien.

11.0.5 Contenu de la variable pointée

L'opérateur d'indirection * s'utilise dans deux cas distincts. Nous avons vu le premier qui sert à déclarer un pointeur. Le second cas d'utilisation permet d'accéder au contenu de la variable pointée.

Exemple 11.4. : *Contenu de l'adresse pointée.*

```
int maVariable = 8;
int *monPointeur;                    // Premier cas d'usage de l'opérateur d'indirection
monPointeur = &maVariable;
printf("valeur = %d", *monPointeur);  // Second cas d'usage de l'opérateur d'indirection
```

Exercice 1. : *Représentation graphique*

```
int m = 20;
int n, *ptr;
ptr = &m;
n = *ptr;
*ptr = 30;
ptr = &n;
```

*Pour **chaque ligne**, représentez les trois variables : `m`, `n` et `ptr`, conformément à la convention graphique de la figure 11.1.*

Exemple 11.5. : *Résumé par l'exemple*

```
char c = 'A';
int i = 52;
float f = 23.2;

char *charPtr = &c;
int *intPtr = &i;
float *floatPtr = &f;

printf("c = %c \n", *charPtr);
printf("i = %d \n", *intPtr);
printf("f = %.2f \n", *floatPtr);

printf("l'adresse de la variable c = %p ", &c);
printf(" = contenu du pointeur charPtr = %p \n", charPtr );

printf("c = %c = contenu de la variable pointée par charPtr = %c \n", c, *charPtr);
```

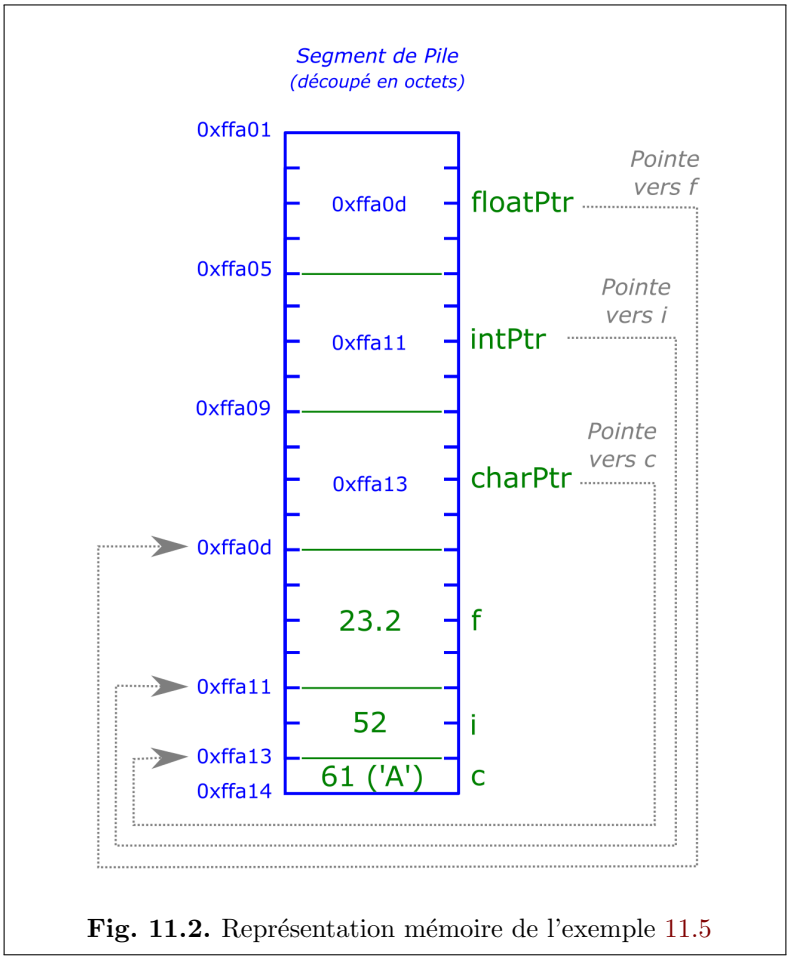

Fig. 11.2. Représentation mémoire de l'exemple 11.5

Remarques :

— La figure 11.2 est une représentation mémoire du **segment de Pile** suite à l'exécution des instructions de l'exemple 11.5.

— L'architecture utilisée est en **32 bits** : les entiers et les réels sont codés sur 4 octets. Les pointeurs occupent en général la plus grande taille directement gérable par le processeur donc ici 4 octets également.

— Les variables sont **arbitrairement placées** de manière consécutive dans le segment de pile, par ordre de déclaration (la pile grandit vers le haut). En pratique, la répartition des adresses peut être différente. Cela est notamment du au travail d'optimisation de la mémoire réalisé par le compilateur.

Exercice 2. : *Déroulé d'exécution*
2.1) Quelles seront les lignes affichées à l'écran par l'exemple 11.5 ?
2.2) Compilez le programme et vérifiez votre réponse.

11.0.6 Allocation dynamique

Il n'est pas toujours possible de fixer la taille des données au moment où nous rédigeons le code source. Il paraît alors utile de pouvoir réserver de la mémoire en cours d'exécution. Le langage C offre cette possibilité qui se nomme : l'allocation dynamique de mémoire. Plusieurs fonctions de la librairie standard sont dédiées à la gestion dynamique de la mémoire.

La fonction `malloc`

Elle alloue en mémoire un espace de **n octets** (contigus non initialisés). La valeur de retour est un pointeur non typé (void) sur le début de la zone allouée. En cas d'erreur, la fonction retourne le pointeur **NULL** de type **void**.

Exemple 11.6. : *Utilisation de `malloc` (convertion implicite)*

```
int *p;
p = malloc(12);
assert( p != NULL );
```

Dans l'exemple 11.6, le pointeur `p` est déclaré sur le type entier (`int`), alors que `malloc` retourne un pointeur de type `void`. A la compilation, un avertissement/*warning* peut apparaître (sans conséquence pour l'exécution). Pour éliminer ce warning, il faut exécuter une convertion de type explicite :

Exemple 11.7. : *Utilisation de `malloc` (convertion explicite)*

```
int *p;
p = (int *) malloc(12);  // Le cast au retour du malloc est à éviter.
if( p == NULL){
    printf("Error : could not allocate heap memory.\n");
    exit(-1);
}
```

Conseils :

- *Dans l'ancienne version du langage C, le cast du retour du pointeur était requis. Mais aujourd'hui, `malloc()` renvoie un `void *` qui sera promu automatiquement vers le type ad hoc. Le transtypage est donc inutile et déconseillé, car il peut masquer l'oublie d'inclure la bibliothèque `stdlib.h`.*
- *N'oubliez pas que la taille en octet d'un type de donnée peut varier selon le processeur utilisé (8/16/32/64 bits).*
- *Vérifiez systématiquement que l'allocation dynamique s'est bien déroulée.*

Exemple 11.8. : *Utilisation de `malloc` et `sizeof`*

```
int elts = 3;
int *p;
p = malloc( sizeof(int) * elts );
assert( p != NULL );
```

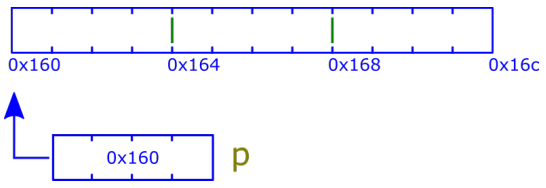

Fig. 11.3. Allocation dynamique avec `malloc`, Exemple 11.8

Remarque : d'autres fonctions d'allocation existent. Nous vous laisserons les découvrir par vous même (ex : `realloc`).

Attention une erreur courante est d'appliquer l'opérateur `sizeof()` sur un pointeur plutôt que sur les données stockées en mémoire.

Exemple 11.9. : *Taille des pointeurs*

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

#define N 6

int main(void){
    printf("Attention la taille d'un type est different de la taille d'un pointeur !\n");
    printf("Voici quelques exemples :\n");
    printf("char   :\t %ld octet.\t\t char *   :\t %ld octets.\n", sizeof(char), sizeof(char *));
    printf("short  :\t %ld octet.\t\t short *  :\t %ld octets.\n", sizeof(short), sizeof(short *));
    printf("int    :\t %ld octets.\t\t int *    :\t %ld octets.\n", sizeof(int), sizeof(int *));
    printf("float  :\t %ld octets.\t\t float *  :\t %ld octets.\n", sizeof(float), sizeof(float *));
    printf("double :\t %ld octets.\t\t double * :\t %ld octets.\n", sizeof(double), sizeof(double *));
    printf("En C, la taille d'un pointeur correpondra généralement à la taille du bus d'adresse :\n");
    printf("4 octets sur une architecture x86 (32 bits), 8 octets en x64.\n");
    printf("Certains compilateurs réduisent la taille des pointeurs\n");
    printf("par souci de rétrocompatibilité.\n");
    return EXIT_SUCCESS;
}
```

La fonction `free`

Libère la zone mémoire précédement allouée au pointeur.

```c
  free(p);  // Libération de la zone mémoire pointée par p.
  p = NULL; // Affectation à NULL pour ne plus pointer sur la zone désallouée.
```

Fuite de mémoire *(memory leak)*

Une fuite mémoire est une grave erreur de programmation. Au delàs du fait qu'elle peut faire planter la machine qui exécute le programme *(pensez aux microcontrôleurs)*, ce type d'erreur, comme le dépassement de bornes d'un tableau introduit une faille de sécurité dans le programme.
Il est donc impératif que pour chaque instruction d'allocation `malloc()`, corresponde une instruction de libération `free()`.

Dangling pointer *(pointeur pendouillant)*

L'instruction `free` libère une zone mémoire que le système pourra réutiliser ultérieurement. Toutefois le/les pointeur(s) pointant(s) sur cette zone ne sont pas modifié(s) lors de la libération. Il(s) pointe(nt) donc toujours en direction de cette zone mémoire et leur(s) utilisation risque de corrompre la mémoire ou de créér un comportement inattendu (*use-after-free*). Un *dangling pointer* est un pointeur qui contient l'adresse mémoire d'un élément qui a été libéré. Pour nous prémunir de tout problème nous affectons à `NULL` le(s) pointeur(s) après libération de la zone mémoire.

11.0.7 Arithmétique sur les pointeurs

Nous pouvons essentiellement **déplacer** un pointeur par addition, soustraction, incrémentation ou décrémentation d'un entier `n`. Cet entier `n` représente un nombre d'**élément**. Par exemple, si ptr pointe sur une zone mémoire d'entiers codés sur 4 octets, p++ pointe sur l'élément suivant, p– pointe sur l'élément précédent.

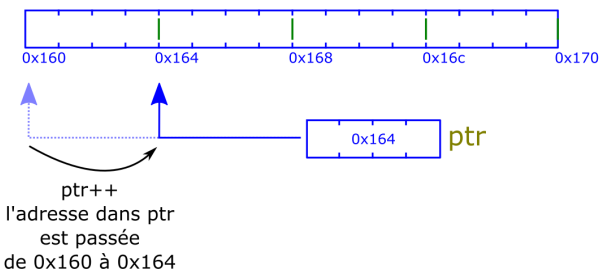

Fig. 11.4. Incrémentation d'un pointeur

Nous pouvons soustraire deux pointeurs, le résultat sera le nombre d'éléments qui séparent **les adresses pointées**. En revanche l'addition de deux pointeurs est interdite.

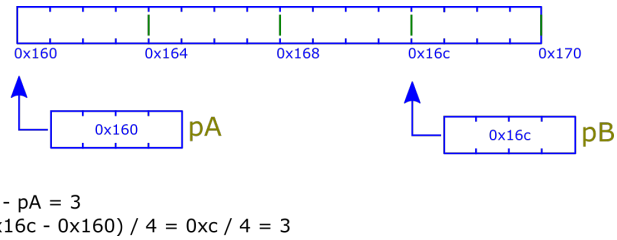

Fig. 11.5. Soustraction entre pointeurs

11.0.8 Fonction qui prend en paramètre un pointeur

Le passage des paramètres en C se fait par valeur. Le paramètre formel est toujours une valeur. Maintenant que nous savons utiliser les pointeurs, voyons comment modifier la valeur d'une variable définie en dehors d'une fonction.

Pour implémenter le passage par adresse vu dans le cours de méthode, nous allons passer un pointeur.

Exemple 11.10. : *Passage de paramètre par adresse*

```
void carre(int *n){   // Ici n est un paramètre formel,
    *n = *n * *n;     // au sein de la fonction c'est un pointeur sur entier
}                     // qui reçoit pour valeur une adresse.


int main(void){
    int x = 4 ;
    carre(&x) ;
    printf("carre = %d \n\n", x);
    return EXIT_SUCCESS ;
}

// Ou

int main(void){
    int *x = malloc(sizeof(int)) ;
    assert( x != NULL);
    *x = 4 ;
    carre(x) ;
    printf("carre = %d \n\n", *x);
    free(x);
    x = NULL;
    return EXIT_SUCCESS ;
}
```

11.1 Retour sur la fonction scanf

Nous avons vu au chapitre 5 que pour saisir un caractère ou un nombre, nous passons l'adresse de ce caractère :

```
  char c;
  printf("Tapez une lettre : ");
  scanf("%c",&c);
```

Nous pouvons donc procéder ainsi :

```
  char *adr = malloc(1);      // réservation de 1 octet
  if( adr == NULL){
      printf("Error : could not allocate heap memory.\n");
      exit(-1);
  }
  printf("Tapez une lettre : ");
  scanf("%c",adr);
  printf("saisie = %c \n", *adr);
  free(adr);
  adr = NULL;
```

Nous saisissons ici le contenu de l'adresse adr. Cette méthode s'applique aux caractères (`char`), aux entiers (`int`), aux réels (`float`).