Algorithmes et programmation II : Les pointeurs

Souheib Baarir[1]

[1]Université Paris Ouest Nanterre La Défense.
Laboratoire d'informatique de Paris 6.
Souheib.baarir@u-paris10.fr

Licence Mia - 2010/2011

Grandes lignes du cours

Introduction

- Toute variable manipulée dans un programme est stockée quelque part en **mémoire centrale**.

Introduction

- Toute variable manipulée dans un programme est stockée quelque part en **mémoire centrale**.
- La mémoire peut être assimilée à un “tableau” dont chaque élément est identifié par une ‘adresse’.

Introduction

- Toute variable manipulée dans un programme est stockée quelque part en **mémoire centrale**.
- La mémoire peut être assimilée à un “tableau” dont chaque élément est identifié par une ‘adresse’.
- Pour retrouver une variable, il suffit, donc, de connaître l’adresse de l’élément-mémoire ou elle est stockée.

Introduction

- Toute variable manipulée dans un programme est stockée quelque part en **mémoire centrale**.
- La mémoire peut être assimilée à un “tableau” dont chaque élément est identifié par une ‘adresse’.
- Pour retrouver une variable, il suffit, donc, de connaître l’adresse de l’élément-mémoire ou elle est stockée.
- C’est le compilateur qui fait le lien entre l’identificateur d’une variable et son adresse en mémoire.

Introduction

- Toute variable manipulée dans un programme est stockée quelque part en **mémoire centrale**.
- La mémoire peut être assimilée à un “tableau” dont chaque élément est identifié par une ‘adresse’.
- Pour retrouver une variable, il suffit, donc, de connaître l’adresse de l’élément-mémoire ou elle est stockée.
- C’est le compilateur qui fait le lien entre l’identificateur d’une variable et son adresse en mémoire.
- Il peut être cependant plus intéressant de décrire une variable non plus par son identificateur mais directement par son adresse !

Lvalue : adresse et valeur

Définition

On appelle **Lvalue (left value)** toute expression du langage pouvant être placé à gauche d'un opérateur d'affectation.

Lvalue : adresse et valeur

Définition

On appelle **Lvalue (left value)** toute expression du langage pouvant être placé à gauche d'un opérateur d'affectation.

Caractérisation

Une Lvalue est caractérisée par :

- son **adresse** : *i.e.*, l'adresse mémoire à partir de laquelle l'objet est stocké ;
- sa **valeur** : *i.e.*, ce qui est stocké à cette adresse.

Lvalue : adresse et valeur

Définition

On appelle **Lvalue (left value)** toute expression du langage pouvant être placé à gauche d'un opérateur d'affectation.

Caractérisation

Une Lvalue est caractérisée par :

- son **adresse** : *i.e.*, l'adresse mémoire à partir de laquelle l'objet est stocké ;
- sa **valeur** : *i.e.*, ce qui est stocké à cette adresse.

Une variable est un exemple concret de Lvalue.

Lvalue : exemple

```
int i , j ;
    i =1;
    j=i ;
```

Lvalue : exemple

```
int i , j ;
    i =1;
    j=i ;
```

Si le compilateur a placé la variable *i* à l'adresse 4831830000 et *j* à l'adresse 4831830004, alors :

Lvalue : exemple

```
int i , j ;
    i =1;
    j=i ;
```

Si le compilateur a placé la variable *i* à l'adresse 4831830000 et *j* à l'adresse 4831830004, alors :

Lvalue	Adresse	Valeur
i	4831830000	1
j	4831830004	1

Lvalue : exemple

```
int i , j ;
    i =1;
    j=i ;
```

Si le compilateur a placé la variable *i* à l'adresse 4831830000 et *j* à l'adresse 4831830004, alors :

Lvalue	Adresse	Valeur
i	4831830000	1
j	4831830004	1

Remarque

L'adresse d'une lvalue est un entier (16 bits, 32 bits ou 64 bits) et ce quelque soit le type de la valeur de lvalue.

Lvalue : récupérer l'adresse

L'opérateur adresse

Pour acceder à l'adresse d'une variable (lvalue) nous disposons de l'opérateur unaire &.

Par l'exemple :

```
int i;
printf("l'adresse de i = %ld",&i);
```

Si le compilateur a placé la variable *i* à l'adresse 4831830000 alors l'affichage sera : l'adresse de i = 4831830000

Outline

Notion de pointeur

Définition

Un pointeur est une lvalue dont la valeur est égale à l'adresse d'une autre lvalue.

Notion de pointeur

Définition

Un pointeur est une lvalue dont la valeur est égale à l'adresse d'une autre lvalue.

Déclaration

`type *nomPointeur` (où `type` est le type de l'élément pointé).

Notion de pointeur

Définition

Un pointeur est une lvalue dont la valeur est égale à l'adresse d'une autre lvalue.

Déclaration

`type *nomPointeur` (où `type` est le type de l'élément pointé).

Exemple :

```
int i=3;
int *p;
p = &i;
```

Notion de pointeur

Définition

Un pointeur est une lvalue dont la valeur est égale à l'adresse d'une autre lvalue.

Déclaration

`type *nomPointeur` (où `type` est le type de l'élément pointé).

Exemple :

```
int i=3;
int *p;
p = &i;
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	4830000

Pointeur : opérateur unaire d'indirection (1/3)

Problème

Comment peut-on accéder directement à l'element pointé par la valeur d'un pointeur ?

Exemple :

```
int i=3;
int *p;
p = &i;
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	4830000

Solution

Utilisation d'un nouvel opérateur unaire : " * "

Pointeur : opérateur unaire d'indirection (2/3)

Exemple :

```
int i=3;
int *p;
p = &i;
printf("La valeur de *p = %d",*p);
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	4830000
*p	4830000	3

Pointeur : opérateur unaire d'indirection (3/3)

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    *p1=*p2
}
```

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    p1=p2;
}
```

Pointeur : opérateur unaire d'indirection (3/3)

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    *p1=*p2
}
```

Lvalue	Adresse	Valeur
i	4830000	3
j	4830004	6
p1	4835984	4830000
p2	4835982	4830004

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    p1=p2;
}
```

Pointeur : opérateur unaire d'indirection (3/3)

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    *p1=*p2
}
```

Lvalue	Adresse	Valeur
i	4830000	3
j	4830004	6
p1	4835984	4830000
p2	4835982	4830004

```c
int
main(int arv,char * arg[])
{
    int i=3,j=6;
    int *p1,*p2;
    p1 = &i;
    p2 = &j;
    p1=p2;
}
```

Lvalue	Adresse	Valeur
i	4830000	6
j	4830004	6
p1	4835984	4830000
p2	4835982	4830004

Outline

Arithmétique des pointeurs

Arithmétique des pointeurs

- La valeur d'un pointeur est un entier.

Arithmétique des pointeurs

- La valeur d'un pointeur est un entier.
- On peut appliquer à un pointeur quelque opérations arithmétiques :
  - **Addition** d'un entier à un pointeur.
  - **Soustraction** d'un entier à pointeur.
  - **Différence** entre deux pointeurs (de même type).

Arithmétique des pointeurs : l'addition et la soustraction

- Soit i un entier et p un pointeur sur un element de type *type*,
- l'expression $p' = p + i$ (resp. $p' = p - i$) désigne un pointeur sur un element de type *type*,
- la valeur de p' est égale à la valeur de p incrémenté (resp. décrémenté) de $i * sizeof(type)$.

```
int
main(int arv,char * arg[]) {
    int i=5;
    int *p1,*p2;
    p1 = &i + 2;
    p2 = p1 - 2;
}
```

Si &i = 4830000 alors :

Arithmétique des pointeurs : l'addition et la soustraction

- Soit i un entier et p un pointeur sur un element de type *type*,
- l'expression $p' = p + i$ (resp. $p' = p - i$) désigne un pointeur sur un element de type *type*,
- la valeur de p' est égale à la valeur de p incrémenté (resp. décrémenté) de $i * sizeof(type)$.

```
int
main(int arv,char * arg[]) {
    int i=5;
    int *p1,*p2;
    p1 = &i + 2;
    p2 = p1 - 2;
}
```

Lvalue	Adresse	Valeur
i	4830000	5
p1	4830004	4830008
p2	4830008	4830000

Si &i = 4830000 alors :

Arithmétique des pointeurs : la différence

- Soit p et q deux pointeurs sur des elements de type *type*,
- l'expression $p - q$ désigne un entier dont la valeur est : $(p - q)/sizeof(type)$,

```
int
main(int arv,char * arg[]) {
    int i=5;
    int *p1,*p2;
    p1 = &i + 2;
    p2 = p1 - 2;
    int j = p2 - p1;
}
```

Si &i = 4830000 alors :

Arithmétique des pointeurs : la différence

- Soit p et q deux pointeurs sur des elements de type *type*,
- l'expression $p - q$ désigne un entier dont la valeur est : $(p - q)/sizeof(type)$,

```
int
main(int arv,char * arg[]) {
    int i=5;
    int *p1,*p2;
    p1 = &i + 2;
    p2 = p1 - 2;
    int j = p2 - p1;
}
```

Lvalue	Adresse	Valeur
i	4830000	5
p1	4830004	4830008
p2	4830008	4830000
j	4830016	2

Si &i = 4830000 alors :

Outline

Allocation dynamique

Initialisation d'un pointeur

Avant toute utilisation, un pointeur doit être initialisé (sinon, il peut pointer sur n'importe quelle région de la mémoire !) :

- soit par l'affectation d'une valeur "nulle" à un pointeur : $p = NULL$;

Initialisation d'un pointeur

Avant toute utilisation, un pointeur doit être initialisé (sinon, il peut pointer sur n'importe quelle région de la mémoire !) :

- soit par l'affectation d'une valeur "nulle" à un pointeur : $p = NULL$;
- soit par l'affectation de l'adresse d'une autre variable (lvalue) : $p = \&i$;

Initialisation d'un pointeur

Avant toute utilisation, un pointeur doit être initialisé (sinon, il peut pointer sur n'importe quelle région de la mémoire !) :

- soit par l'affectation d'une valeur "nulle" à un pointeur : $p = NULL$;
- soit par l'affectation de l'adresse d'une autre variable (lvalue) : $p = \&i$;
- **soit par l'allocation dynamique d'un nouvel espace-mémoire...**

Allocation dynamique : définition

Définition

L'allocation dynamique est l'opération qui consiste à réserver un espace-mémoire d'une taille définie.

- L'allocation dynamique en C se fait par l'intermédiaire (entre autre) de la fonction de la libraire standard `stdlib.h` :

```
char* malloc(nombreOctets)
```

- Par défaut, cette fonction retourne un `char *` pointant vers une espace mémoire de taille `nombreOctets`.
- Il faut convertir le de sortie de `malloc` à l'aide d'un cast, pour des pointeur qui ne sont pas des `char *`

Allocation dynamique : exemple

```c
#include <stdlib.h>

int main(int arv,char * arg[]) {
    int i = 3, *p;
    p = (int*) malloc(sizeof(int));
    *p = i;
}
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	?

Allocation dynamique : exemple

```c
#include <stdlib.h>

int main(int arv, char * arg[]) {
    int i = 3, *p;
    p = (int*) malloc(sizeof(int));
    *p = i;
}
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	4830008
*p	4830008	?

Allocation dynamique : exemple

```c
#include <stdlib.h>

int main(int arv,char * arg[]) {
    int i = 3, *p;
    p = (int*) malloc(sizeof(int));
    *p = i;
}
```

Lvalue	Adresse	Valeur
i	4830000	3
p	4830004	4830008
*p	4830008	3

Allocation dynamique : la libération mémoire

Définition

C'est l'opération qui consiste à libérer l'espace-mémoire alloué.

- En C, la libération mémoire se fait par l'intermédiaire de la fonction de la libraire standard `stdlib.h` :

```
void free(nomPointeur)
```

- Tout espace-mémoire alloué dynamiquement via `malloc` (ou équivalent) doit obligatoirement être désalloué par `free`, sinon nous rencontrons le fameux problème de fuite mémoire (memory leak).

Pointeurs et tableaux

Pointeurs et tableaux à une dimension

À retenir

Tout tableau en C est en fait un pointeur constant !

Soit `int tab[N]` un tableau alors `tab` est un pointeur constant qui a pour valeur `&tab[0]`.

Pointeurs et tableaux à une dimension

À retenir

Tout tableau en C est en fait un pointeur constant !

Soit `int tab[N]` un tableau alors `tab` est un pointeur constant qui a pour valeur `&tab[0]`.

```
int main(int arv,char * arg[]) {
    int tab[5]={1,2,3,4,5};
    int i, *p;
    p = tab;
    for (i=0 ; i<5 ; ++i)
      printf("*(p+i) = %d = p[i] = %d = tab[i] = %d \n",
             *(p+i), p[i], tab[i]);
}
```