Programmation impérative

Cours 2 : Pointeurs en C

Catalin Dima

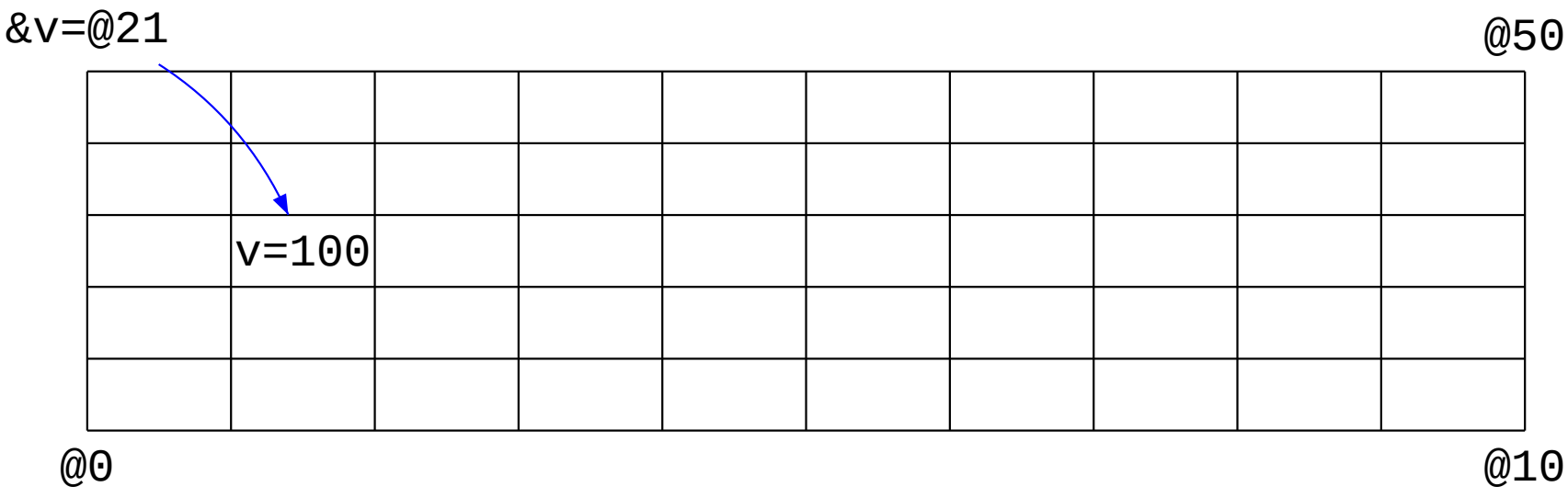
Valeurs et adresses de variables

```
char v=100;
```

- Valeur `x=100`.
- Adresse : où est cette valeur stockée ?

Toute variable est associée à une zone de mémoire où, à chaque instant, la valeur de la variable est stockée.

- Opérateur d'adressage `&` : `&v` est l'adresse de la variable `v`.
- Afficher une adresse (dans un `printf`) : format `%p`.

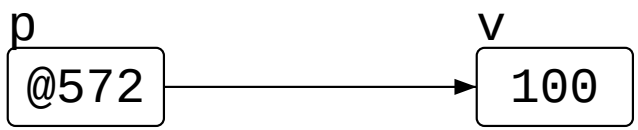


Pointeurs

- Pointeur = type de données permettant de stocker une adresse.
- Déclaration `type * ptr`.
- `ptr` peut se voir affecter une valeur qui est l'adresse d'une autre variable :
  `ptr = &a`, avec a déclaré comme `type a`.
- Opérateur `*` unaire préfix : retrouver la valeur stockée à l'adresse contenue dans la case mémoire du pointeur.
  - La variable `ptr` se voit affectée une case mémoire de la taille d'une adresse de l'ordinateur (souvent 4 octets), et stocke une adresse `@adr1`.
  - Cette valeur (le contenu du pointeur) représente elle-même une autre adresse `@adr2`!
  - `*ptr` permet de retrouver le contenu stocké à l'adresse `@adr2`.

Pointeurs (2)

```
int v=100;
int *p;  // p peut stocker des adresses d'entiers sur 4 octets
p=&v;    // donc, par exemple, l'adresse de v !
if (*p=100) printf("c'est normal !");
```



- Une même adresse en mémoire peut être associée à plusieurs types de données :

```
int v=100;
int *p;
char *q;
p = &v;
q = &v;  // q ''pointe'' sur le premier octet de v !
printf("valeur de p = %p, adresse de v = %p, contenu de p = %",p,&v,*p);
```

- Avertissement du compilateur lorsqu'une telle affectation est effectuée ! (à l'adresse v on a tout de même stocké un entier sur 4 octets !)

Pointeurs constants

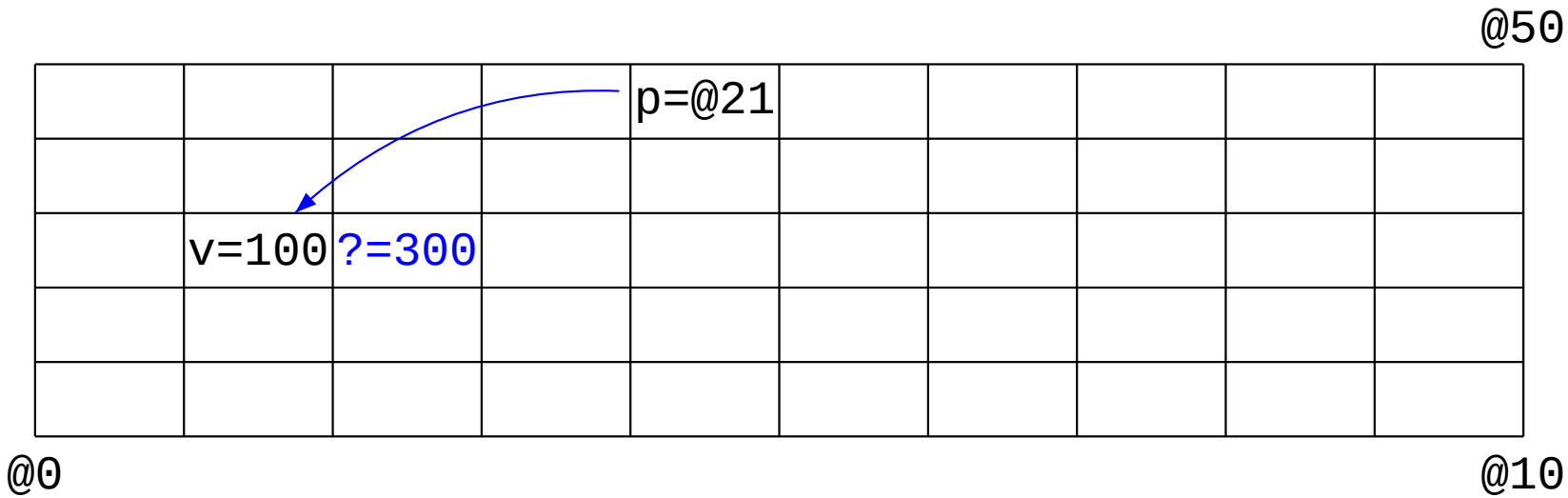
- Comme toute variable, un pointeur peut être déclaré constant :

```
int a,b;
const int *p=&b; // on ne peut plus affecter de nouvelle valeur à p !
p = &a;          // erreur de compilation ! pas de programme généré !
```

Arithmétique des pointeurs

- Certaines opérations d'entiers fonctionnent aussi sur les pointeurs :
  - `+, -` avec le 2 argument de type entier.

```
unsigned char v=100, *p=&v;
printf("val. p=%p", val. p+1=%p, contenu (p+1)=%d",p,(p+1),*(p+1));
```



- Mais si le type pointé est stocké sur plusieurs octets, alors l'addition se fait en multipliant avec la taille du type pointé :

```
int v=100, *p=&v;
// p+1 va être égal à @25 !
```

Pointeurs constants et tableaux

- Une variable tableau n'est autre qu'pointeur constant !

```
const char *a;
char b[100];   // presque la même chose que le déclarer *b
```

  - Presque la même chose, car les 100 cases de b sont réservées en mémoire,
  - ... alors que, pour une correctitude sémantique, a doit se voir affecté une case mémoire en utilisation pour une autre variable...
  - ... ou une case libre – à voir la semaine prochaine (`malloc`).

- L'indexation des tableaux correspond exactément à l'arithmétique des pointeurs :

```
*a = b; // a pointe désormais sur la première case du tableau
a[0] = 2   // identique à b[0]=2 et à *a=0 !
*(b+2) = 3 // identique à b[2]=3 !
a[5] = 100 // identique à ....
```

Opérations sur les tableaux ?

- Il n'existe pas d'affectation simple de tableaux, comme :

```
int a[3], b[3] = {1,2,3};
a=b;     // grosse erreur de compil !
```

- Pour la simple raison que l'affectation a est une affectation à une variable qui est un pointeur constant de `int` !
- Pour copier un tableau dans un 2e, il est obligatoire d'écrire une petite boucle ! (hélas !)
- Pareil pour copier une chaîne de caractères :

```
unsigned char a[100]; // chaîne de max. 99 caractères !
a = "nouvelle chaine de caracteres";  // la même grosse erreur de compil !
```

- Il existe des fonctions permettant de copier les chaînes de caractères (voir plus loin).

Paramètres des fonctions

- Lors de chaque appel d'une fonction, less variables et les paramètres des fonctions se voient réservé un espace de mémoire local à la fonction, différent de l'espace des autres fonctions.
- Une fois cet espace créé, les paramètres actuels dans l'appel sont placés dans les cases mémoire réservées pour chaque paramètre.
- À la fin de l'exécution d'une fonction, tout cet espace devient libre, et toutes les valeurs qui y ont été stockées disparaissent
  - ... à l'exception des variables déclarées comme `static` – à voir plus tard.
- La seule valeur qui n'est pas perdue est celle calculée pour le `return`.
- Ce qui implique que, si jamais on modifie les valeurs des paramètres dans une fonction, les valeurs modifiées seront perdues à la fin de l'exécution.
- Mais alors comment faire si on veut que des modifications de valeurs de paramètres soient utilisables après la fin de l'exécution de la fonction ?
  - Cas classique : échanger les valeurs de deux variables :

```
void ech(int x, int y){
  int z=x;
x=y;
y=z;
}
int main(){
 int a=0,b=1;
 ech(a,b);
 printf"a=%d,b=%",a,b); // est-ce que ça affiche a=1 et b=0 ?
}
```

Paramètres des fonctions

- Non, ça n'affiche pas `a=1,b=0` !

```
void ech(int x, int y){
  int z=x;
x=y;
y=z;
}
int main(){
int a=0,b=1;
ech(a,b);
printf"a=%d,b=%",a,b); // est-ce que ça affiche a=1 et b=1 ?
}
```

- Pour la simple raison que `x` et `y` sont des paramètres de `ech`,
- ... et les valeurs `0` et `1` seront affectés à des cases mémoire locales à la fonction `ech`,
- ... qui seront effacées à la fin de l'exécution de celle-ci !

Pointeurs et fonctions

- Les pointeurs arrivent à la rescousse !

```
void ech(int *x, int *y){
  int z=*x;
*x=*y;
*y=z;
}
int main(){
int a=0,b=1;
ech(&a, &b);
printf"a=%d,b=%",a,b); // est-ce que ça affiche a=1 et b=1 ?
}
```

- Voyons voir ce qui se passe dans les cases de la mémoire de l'ordinateur...

Tableaux et fonctions

Mais alors comment ça marche pour les tableaux, si on veut les modifier dans une fonction ?

- Réponse : on les donne en tant que tels, car un tableau est un pointeur,
- ... donc une variable dont la valeur est une adresse qui ne pointe pas dans la mémoire de la fonction,
- ... car constante et affectée avant l'appel de la fonction !

Exemple : renverser les cases d'un tableau, sans utilisation de tableau supplémentaire :

```
void renverser(int tab[], int taille){
      // taille nécessaire, car variable et impossible à retrouver en regardant
   int aux,i;
for (i=0; i<aux; i++){
    aux = tab[i];
tab[i]=tab[taille-i];
tab[taille-i]=aux;
}
}
```

Pointeurs et `scanf`

- On comprend pourquoi `scanf` prend en paramètres des `&qqchose` :
- Ce sont des adresses (donc des valeurs de type `*type`) que `scanf` attend !

```
int a, *b=&a;
scanf("%d",&a); // on lit à l'adresse de la variable a
scanf("%d",b);  // on lit à la même adresse !
```

Fonctions de chaînes de caractères

Bibliothèque `string.h` :

- `size_t strlen(char chaine[])` : retourne la longueur de la chaîne de caractères passée en paramètre.
- `char* strcpy(char dest[], char source[])` : copie la chaîne de caractères `source` dans la `dest`.
- `char* strcat(char s1[], char s2[])` : rajoute `s2` à la fin de `s2`, en écrasant le code ASCII `0` à la fin de `s1`.
  - Donc `s1` sort modifié de cette fonction !
- `int strcmp(char s1[], char s2[])` : compare lexicographiquement les deux chaînes et renvoie -1, 0 ou 1 comme résultat.
  - `strcmp("abc","abcd") == -1`.
  - `strcmp("abc","abc") == 0`.
  - `strcmp("abcd","abc") == 1`.

Fonctions de chaînes de caractères

Bibliothèque `string.h` :

- `size_t strlen(char chaine[])` : retourne la longueur de la chaîne de caractères passée en paramètre.
- `char* strcpy(char dest[], char source[])` : copie la chaîne de caractères `source` dans la `dest`.
- `char* strcat(char s1[], char s2[])` : rajoute `s2` à la fin de `s2`, en écrasant le code ASCII `0` à la fin de `s1`.
  - Donc `s1` sort modifié de cette fonction !
- `int strcmp(char s1[], char s2[])` : compare lexicographiquement les deux chaînes et renvoie -1, 0 ou 1 comme résultat.
  - `strcmp("abc","abcd") == -1`.
  - `strcmp("abc","abc") == 0`.
  - `strcmp("abcd","abc") == 1`.
- Problème de débordement de tableau : utiliser les versions n :
  - `char* strncpy(char dest[], char source[], int dmax)` : copie au plus `dmax` caractères de `source` dans `dest`.
  - `char* strncat(char s1[], char s2[], int dmax)` : rajoute au plus `dmax` caractères de `s2` à la fin de `s1`.
  - `int strncmp` : pareil (lire dans les pages du `man`).