INF 111, L1, 2004/2005

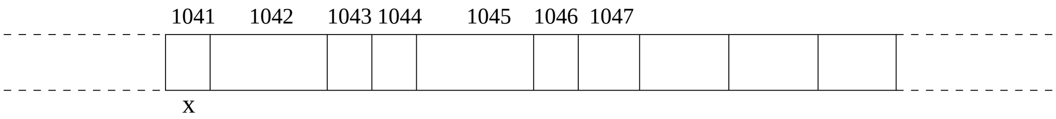
Décembre 2004

Aide Mémoire Pointeurs

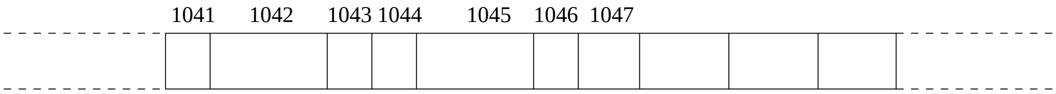
Laure Gonnord
`http://www-verimag.imag.fr/~gonnord/`

1 Un petit cours

Lorsque l'on déclare une variable, on réserve (on dit que l'on "alloue") une place en mémoire (par exemple sur le disque dur) pour elle (plus ou moins grande suivant le type). Par exemple, si on écrit `int x;`, on peut représenter la mémoire comme ci-dessous :

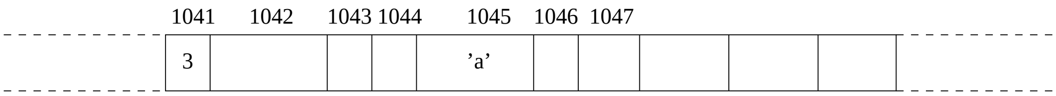


Le numéro `1041` en face de la case `x` indique où est située la variable `x` dans la mémoire, en un mot son "adresse". A chaque instant de l'exécution d'un programme, l'ordinateur sait où sont situées les variables qui sont utilisées, il y a quelque part une structure (mettons, un tableau), qui récapitule tout ça. Par exemple supposons que l'on a déclaré un caractère `char y`, on se retrouve avec le schéma suivant :



x	int	1041
y	char	1045

Maintenant l'initialisation ne pose pas de problème, si on veut donner à `y` la valeur `'a'`, il suffit d'écrire `y = 'a';` et l'ordinateur mettra le charactère `'a'` dans l'adresse (la case) `1045`. Idem pour l'instruction `x = 3;` :



On peut bien entendu récupérer l'adresse de `x`, avec `&x`. Cette adresse est notamment utilisée pour stocker une valeur en la demandant à l'utilisateur : l'instruction `scanf("%d",&x);`, (`x` a préalablement été déclarée, donc son adresse existe) va demander une valeur entière à l'utilisateur, et la stocker à l'adresse de `x`, c'est-à-dire à l'adresse `1041`. L'ancienne valeur stockée à cette adresse, si il y en avait une, est écrasée.

2 Utilisation des pointeurs

Petit rappel sur les appels de fonctions Supposons que l'on ait écrit une fonction `f` (pour une procédure, c'est pareil) qui prend en paramètre deux arguments, ainsi que le programme principal suivant :

```
int f (int a, char c)
{
   int tmp
   if (c == 'z')
      {
        a = 2+a;
        tmp = a;
      }
   else
      {
        a = 4+a;
        tmp = a;
      }
   return tmp;
}
```

```
int main(void)
{
   int x, char y;

   x = 4;
   y = 'z';
   printf("%c",f(x,y));

   return 0;
}
```

Lorsque l'on exécute la fonction `main`, on appelle la fonction `f` sur l'entier `x`, qui vaut `2` à ce moment là, et le caractère `y`, qui vaut `'z'` à ce moment là. La fonction `f` est donc appelée sur les paramètres `(2,'z')`, et tout se passe comme si on avait directement fait appel à la fonction de la façon suivante : `f(2,'z')` [1]. En particulier, lorsque la fonction `f` est exécutée, on "invente" un paramètre `a`, qui est un entier et qui vaut `2`, et un paramètre `c` qui est un caractère et qui vaut `'z'`, on leur fait subir les calculs à l'intérieur de la fonction `f`, ils sont utiles pour calculer la valeur de la fonction, mais sont oubliés ensuite. En aucun cas la valeur `6` ne sera stockée quelque part lorsque la fonction `f` aura fini de s'exécuter. NI DANS `a`, NI DANS `x`, CETTE VALEUR EST OUBLIÉE.

Passage par adresse Quelquefois on aimerait bien pouvoir modifier les paramètres d'une fonction. Par exemple, comme on ne peut pas retourner deux entiers à la fois comme résultat d'une fonction, il peut être commode de pouvoir modifier les paramètres entiers d'une fonction, tout en enregistrant ces modifications. Pour celà, on va utiliser les pointeurs :

```
void g(int *pa, int *pb)
{
   *pa = *pa + 1;
   *pb = *pb + 1;
}
```

```
int main(void)
{
   int a,b;
   a = 0;
   b = 3;
   g(&a,&b);
   printf("%d %d",a,b);
   return 0;
}
```

La fonction `g` ne prend plus en argument deux entiers, mais deux **pointeurs** sur les entiers, autrement dit elle prend en argument deux "adresses de variables entières". `int *pa` désigne la valeur entière pointée. Remarquez l'utilisation des étoiles.
Comment se déroule l'appel ? La fonction `g` ayant besoin de deux adresses, on lui donne les adresses de `a` et `b`. La fonction `g` agit sur les contenus des cases d'adresses `&a` et `&b`, autrement dit la fonction `g` a enregistré les modifications faites sur les variables `a` et `b`. Le programme imprime donc `1 4`.

[1] On appelle celà le passage de paramètres (à une fonction) par VALEUR

Remarque 1 Quand on utilise des tableaux comme paramètres de fonctions/procédures, on a vu que les modifications étaient enregistrées. Heureusement car une fonction ne peut pas retourner un tableau. Mais pourquoi donc ? Parce que, en fait, lorsque l'on écrit `int t[N]`, on écrit en fait `int *t[0]` (autrement dit c'est un pointeur sur la première case du tableau). Ouf, c'est cohérent !

3 Échange de deux variables

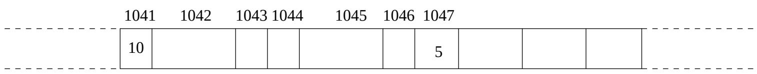
Soit le programme suivant :

```c
void echange(int *x, int *y)
{
 int tmp;
 tmp = *x;
 *x = *y;
 *y = tmp;
}
```

```c
int main (void)
{
  int a,b;

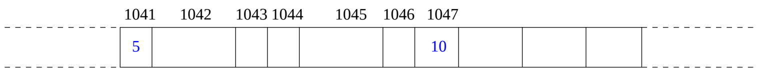
  a = 5;
  b = 10;
  exchange(&a, &b);
  printf("a = %d, b = %d\n",a,b);
  return 0;
}
```

La procédure prend en paramètre deux adresses (pointeurs sur des entiers), et échange les valeurs pointées par ces adresses. Le programme principal commence par déclarer et initialiser deux variables entières `a` et `b`, autrement dit on a en mémoire quelque chose comme ça :



a	int	1047
b	int	1041

Ensuite, le programme principal appelle la procédure `echange` sur les adresses des variables `a` et `b`, et donc celà échange les contenus des cases :



a	int	1047
b	int	1041

Le programme principal fait ensuite une impression à l'écran du contenu des variables `a` et `b`, qu'il va chercher aux adresses qu'il connaît, donc il imprime : `a = 10, b = 5`.

4 Et pour la route ...

4.1 Programme mystère 1

```
int main(void)
{
  char c[8]="grumpf";
  printf("%s\n",c);

  scanf("%c",&c[4]);
  printf("%s\n",c);

  return 0;
}
```

4.2 Programme mystère 2

```
int hopla(char c, char *u)
{
  if (c == 'u')
    {
      *u = *u + 4;
      return 2;
    }
  else
    {
      *u = *u + 3;
      return 4;
    }
}
```

```
int main(void)
{
  char c[8]="grumpf";
  char c2[8]="ploup";

  int x = hopla(c[2], c2);
  printf("%d\n", x);

  printf("%s %s\n", c, c2);

  return 0;
}
```

4.3 Correct ou non ?

Source : `http://www-gtr.iutv.univ-paris13.fr/Cours/Mat/LangageC/TDTP/td08.html`

On écrit la fonction suivante :

```
void Zero( int *p )
{
   *p = 0;
}
```

Les appels suivants sont-ils corrects ? Si oui, que font-ils ?

- ```
  int *a;
  Zero(a);
  ```
- ```
  int *c;
  int x = &c;
  Zero(x);
  ```
- ```
  int x;
  int *p = &x;
  Zero(p);
  ```
  ```