Q: *Pourquoi garder une copie des fichiers Postscript du cours dans mon répertoire si je sais où l'original se trouve sur le web ?*

R: Pour accéder à l'original au cas où je n'ai pas de connexion à Internet.

Ou alors pour l'imprimer plus tard ...

Sinon, je ne garde que le lien qui montre le fichier sur la page web du cours.

Voilà une incarnation du concept du pointeur en dehors du langage C++ !

Mais, les pointeurs, c'est trivial !?

```
#include <iostream.h>

int *p,**q,***r;
void y(int **&x){ **x=6 };

main ( ) {
  r = &q;
  p = new int;
  *r = &p;
  ***r = 5;
  y(q);
  cout << *p << endl;
}
```

Les Pointeurs

Préparation et explications supplémentaires concernant **§17** du logiciel QC++

- la confusion entre **&** et *****
  (problème de C++, pas du concept des pointeurs)
- les variables et les adresses
- les pointeurs et les références
- allocation dynamique: **new** et **delete**
- structures dynamiques de données

Mar 21, 2001 17:23

Q: *La confusion entre **&** et ***** ?*

R: Ils apparaissent

- comme *opérateurs sur les* variables

- dans les *déclarations des* variables
- dans les *listes de paramètres* des fonctions

et chaque fois avec une autre signification.

Q: *<u>Qu'est–ce qui se passe quand on déclare une variable?</u>*

R: 1) **Réservation** d'espace dans la mémoire
à une certaine adresse (normalement sur la "pile")

2) **Association** de la variable
avec l'adresse de cette espace

```
int v;
```

&v

0

```
int z=5;
```

&z

5

"<u>&v</u>" <u>signigfie:</u>
"*l'adresse de l'espace qui contient la valeur de la variable v*"

Le type **`int`** définit la taille de l'espace réservé.

<u>variable</u>: nom *logique* au niveau de la *programmation*
généré par *l'utilisateur*

<u>adresse</u>: nom *interne* (on utilise α) au niveau de la *machine*
généré par *le compilateur*

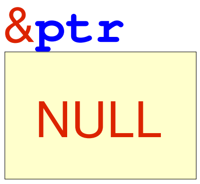
En fait, si **`v`** est une variable de n'importe quel type,
on peut toujours accéder à son adresse
en utilisant **l'opération d'adressage &**
au niveau du langage C++.

Mais, **où** est–ce qu'on met le résultat de &**`v`** **?**
Ou quelles sont les variables de type "*adresse de qqch*" ?

Q: *Qu'est–ce qu'un pointeur?*

R: C'est (le type d') une variable qui contient comme valeur l'adresse d'un espace réservé dans la mémoire.

`int *ptr;` Déclaration d'un pointeur `ptr`, c–à–d d'une variable qui contient l'adresse d'un espace réservé pour un `int`.



L'initialisation donne d'abord l'adresse **NULL**, qui n'est associée à aucun espace dans le mémoire.

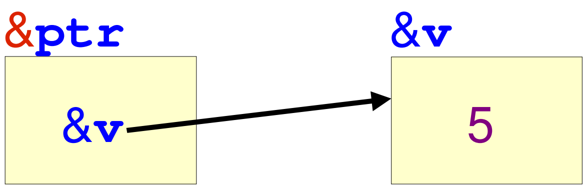
Comme `ptr` est une variable, une adresse `&ptr` montre son contenu.

Q: *Comment peut–on utiliser un pointeur?*

R1: Par affectation.

```
int v=5;
int *ptr;
ptr = &v;
```

Affectation d'une adresse à l'espace qui est accessible par l'adresse `&ptr`.

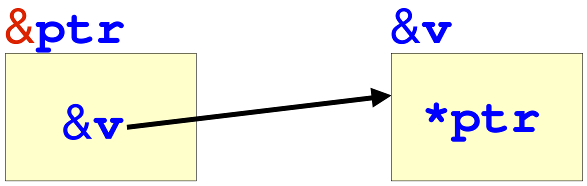


Q: *Comment peut–on utiliser un pointeur?*

R2: Par l’opération d’**indirection** `*` .

Si α est une adresse (nom interne), alors $*\alpha$ retourne le contenu de l’espace qui se trouve à cette adresse.

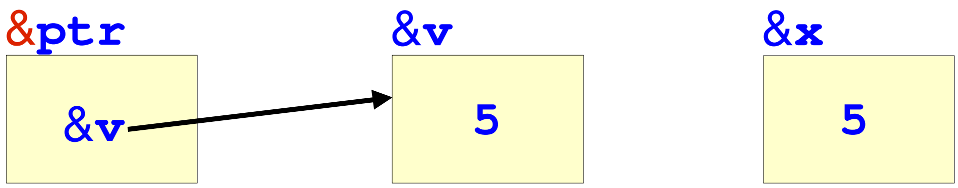
Si **`ptr`** est un(e variable de type) pointeur, **`*ptr`** permet d’accéder au contenu de l’espace qui se trouve à l’adresse contenue dans **`&ptr`**.



`*ptr` peut se trouver des deux côtés d’une affectation!

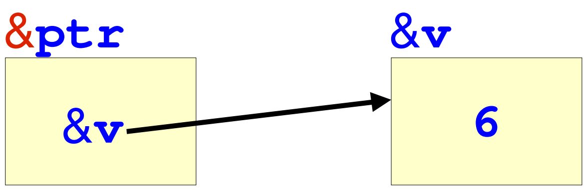

```
int x,v=5,*ptr;
ptr = &v;
x = *ptr;
```

Affectation du contenu de l’espace qui est accessible par le pointeur **`ptr`** à la variable **`x`**.




```
int v=5,*ptr;
ptr = &v;
*ptr = 6;
```

Affectation d’une valeur à l’espace qui est accessible par le pointeur **`ptr`**.

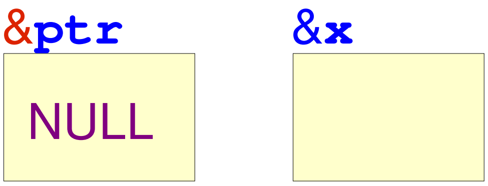


Q: *Qu'est–ce qui se passe si `ptr` ne contient pas d'adresse différente de NULL?*

R: Essayez ...

```
int x,*ptr;
x = *ptr;
```

Affectation du contenu de l'espace qui est accesible par le pointeur **`ptr`**, alors qu'il se trouve à NULL, à la variable **`x`**.



Q: *Comment peut–on accéder à une variable v ?*

R: – simplement en utilisant le nom **v** de la variable

– en utilisant l'operation d'**adressage &** pour récupérer l'adresse **&v** de la variable **v**,

suivi par une demande explicite, en utilisant l'opération d'**indirection ***, du contenu de l'espace qui se trouve à cette adresse, donc, par ***&v**

Donc **v** et ***&v** donnent le même résultat.

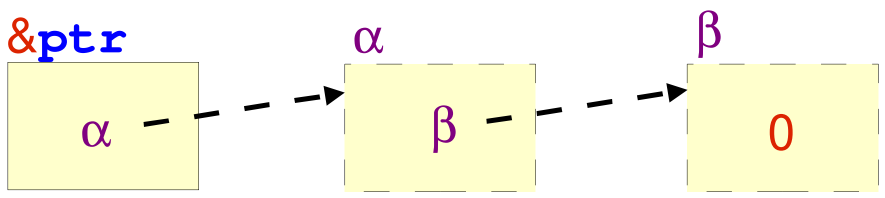
Problème !?
On utilise le symbole **`*`** pas seulement
comme opérateur sur les variables et adresses,
mais aussi dans les déclarations.

`int *ptr;`	*Déclaration* d'un pointeur, c–à–d d'une variable qui contient l'adresse d'un espace réservé pour un **`int`**.
`*ptr = 5;`	*Affectation* d'une valeur à l'espace qui est accessible par le pointeur **`ptr`**, c–à–d à l'adresse qui se trouve dans &**`ptr`**

Problème!?
La définition suivante définit aussi un pointeur.

```
int **ptr;
```

Déclaration d'un pointeur **`ptr`**,
qui contient l'adresse d'un autre pointeur
qui contient l'adresse d'un **`int`**.



Q: *Qu'est–ce qu'une référence?*

R: C'est un nom alternatif ou "alias" pour un objet. Malheureusement, on utilise aussi le "&" pour la déclaration des références ...

```
int v = 42;
int &x = v;
x = 43;
```

&v,&x

43

Répétition: passage par référence

- pour retourner des résultat sans utilisation de return
- pour retourner plusieurs résultats à la fois
- pour améliorer l'efficacité

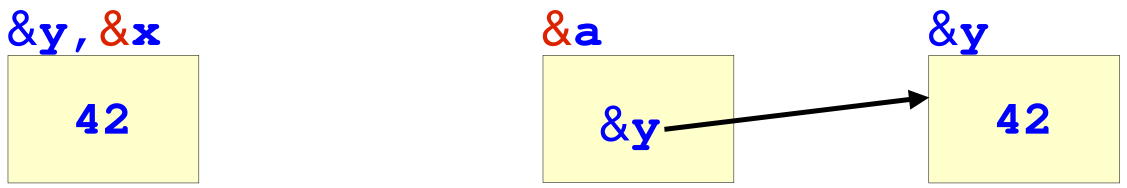
Répétition: passage par référence / par valeur (§13.15)

```
void calcul(int &x){
   x = 42;
}

main(){
   int y = 10;
   calcul(y);
}
```

```
void calcul(int *a){
   *a = 42;
}

main(){
   int y = 10;
   calcul(&y);
}
```

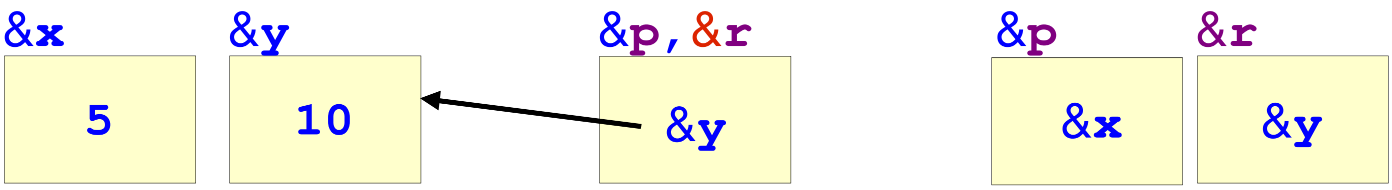


Combinaison: passage des pointeurs par référence

```
int x=5, y=10, *p;

void calcul (int *&r){ r = &y; };

main ( ) {
   p=&x;
   calcul(p);
   cout << "*p = " << *p << endl;
}
```



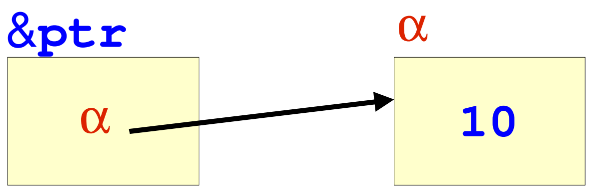
(par référence) (par valeur)

Q: *Faut–il toujours utiliser "&" pour obtenir une adresse ?*

R: Non, on peut aussi créer des espaces avec
des adresses anonymes, c–à–d
des adresses qui ne sont pas associées à une variable.

```
int *ptr;
ptr = new int;
*ptr = 10;
```

Création d'un espace *dynamique* dans le mémoire.




```
...
```

```
delete ptr;
```

Destruction d'un espace *dynamique* dans le mémoire.

Structures dynamiques (I):

- génération dynamique des "cellules" liées

comme "cellules"
on utilise des structures qui contiennent
– une ou plusieurs *données*
– un ou plusieurs *pointeurs* à la "cellule" suivante

```
struct Cell {
  int donnee;
  Cell *suivant;           /* définition récursive */
};

Cell c,*ptr;
```

Structures dynamiques (II):

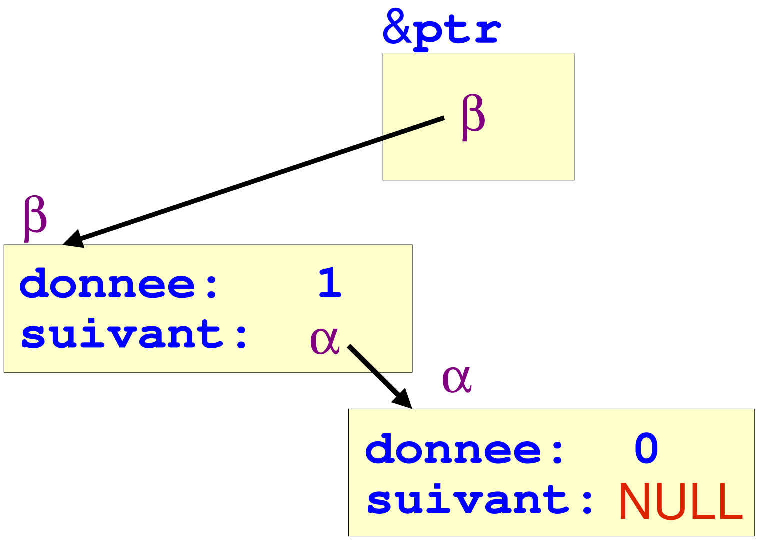
```
struct Cell2;        /* déclaration avant l'utilisation! */

struct Cell1 {       /* définition récursive */
  int donnee1;
  Cell2 *suivant1;
};

struct Cell2 {       /* définition récursive */
  int donnee2;
  Cell1 *suivant2;
};
```


Structures dynamiques (III):

- terminaison des structures
  avec des pointeurs contenant NULL



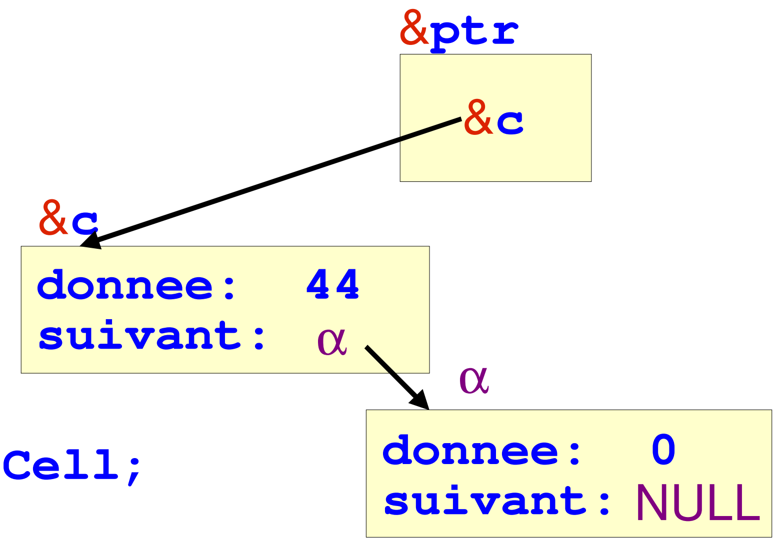
Abréviation pour pointeurs de structure:

```
struct Cell {
  int donnee;
  Cell *suivant;
};

Cell c,*ptr;
ptr = &c;

c.donnee = 42;
(*ptr).donnee = 43;
ptr->donnee = 44;

ptr->suivant = new Cell;
```



Opérations standard sur les structures dynamiques:

- *insérer*
  créer une nouvelle cellule pour la nouvelle donnée
  trouver la "place" de l’insertion selon un critère d’ordre

- *trouver* (lire, extraire, ...)
  selon un critère d’égalité ou bien successivement

- *réorganiser* (enlever, trier, ...)
  ne pas oublier `delete` pour toutes les cellules enlevées
  qui étaient créées par `new`.

Nous approfondirons ce sujet après Pâques ...

Stratégies d'accès à des structures dynamiques:

- **FIFO** (first–in–first–out)
  ex: queue / file

- **LIFO** (last–in–first–out)
  ex: stack / pile

- autres ... (least–recently–used)

Sommaire: Quels sont les avantages et les désavantages?

- efficacité
  (copier une structure coûte plus que copier un pointeur)

- alias
  (plusieurs noms pour la même chose)

- affectation "sur place"
  (pas nécessaire de synchroniser des changements)

- complexité des "protocoles" (jeu d'échange d'information)

- lisibilité/compréhensibilité des programmes . . .

A propos de lisibilité?

```
#include <iostream.h>

int *p,**q,***r;
void y(int **&x){ **x=6 };

main ( ) {
  r = &q;
  p = new int;
  *r = &p;
  ***r = 5;
  y(q);
  cout << *p << endl;
}
```