Les références

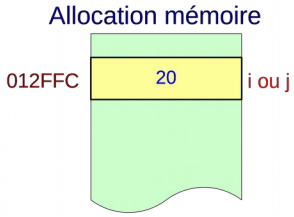
Nous avons dit qu'une donnée est une case de la mémoire qui possède un certain type. Pour pouvoir parler de cette case, on peut soit utiliser son adresse mémoire (qui est la méthode des pointeurs), soit faire référence directement à la donnée, ce qui est le cas quand on utilise un nom de variable. Ainsi, après la déclaration suivante :

```
int i = 18 ;
```

Le compilateur a réservé une case mémoire de 4 octets pour la variable i, et chaque fois que dans un programme on parle de i, le compilateur sait qu'il doit utiliser cette case là. Nous dirons que i est une référence pour la donnée correspondante de valeur 18.

En général un seul nom suffit pour une case mémoire. Mais dans certain cas, on a besoin d'un autre nom, d'une seconde référence. Pour la créer et l'initialiser, on utilise l'opérateur de référence '**&**'.

```
int& j = i ;  ou  int &j = i ;
```



Ceci signifie : j est une référence sur un entier qui est équivalent à i (on dit souvent que j est un alias de i). Dès lors, tous les usages de j dans la suite du programme seront équivalent à ceux de i. Par exemple, si l'on écrit :

```
int i = 18 ;
int& j = i ;
i++ ;
j++ ;
```

Une référence doit toujours être initialisée.

Référence vers un entier

A la fin de ces instructions, i et j seront tous deux égaux à 20. Plus exactement, l'unique case mémoire dénommée à la fois i et j contiendra la valeur 20.

Déclaration

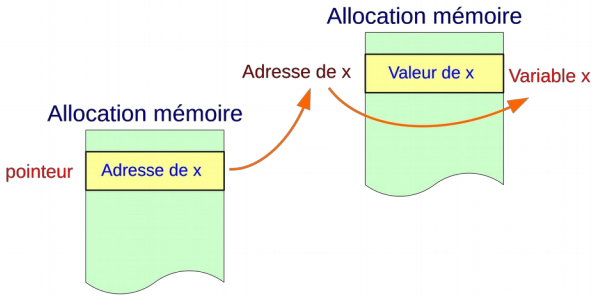
En interne, une référence maintient l'adresse de l'objet pour lequel elle est un alias. C'est exactement le même comportement que pour une variable classique. Une fois que l'adresse est déterminée, il n'est pas possible d'en changer. Une fois déclarée, une référence ne peut plus être modifiée, ce qui impose qu'elle soit impérativement initialisée lors de sa déclaration.

```
int main()
{
  int multiples[10], b=1;

  // Initialisation du tableau (plage de valeurs C++11)
  for (int& multiple : multiples) { multiple=b; b*=2; }

  // Visualisation du tableau (plage de valeurs C++11)
  cout << '[';
  for (int multiple : multiples) cout << multiple << ' ';
  cout << "\b]\n";

  return 0;
}
```

```
[1 2 4 8 16 32 64 128 256 512]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Les pointeurs

Lorsqu'un programme entre en exécution, des cases de cette mémoire sont réservées pour chaque variables, le nombre de cases étant fonction du type de la variable. Chaque variable se trouve quelque part en mémoire. On dira plus précisément qu'elle se trouve à telle adresse. Il est possible d'accéder à une variable :



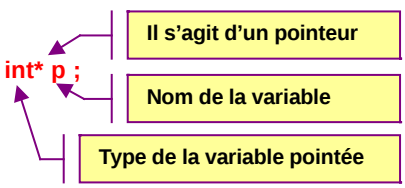
- Par son nom, comme nous l'avons fait jusqu'à présent,
- Par son adresse, en utilisant un pointeur.

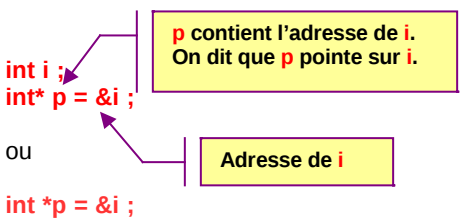
Dans nos précédents programmes, l'accès à une variable (plus précisément à son contenu) se faisait par l'intermédiaire de son nom. Mais au lieu d'accéder par ce nom directement à la variable concernée, on peut aussi choisir un chemin d'accès indirect par le biais de l'adresse de la variable. Pour cela, on utilise ce que l'on appelle un pointeur.

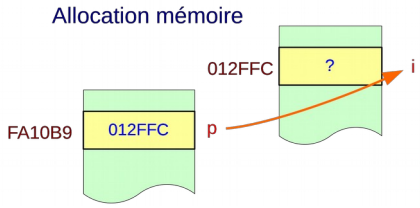
Un pointeur désigne une variable qui contient l'adresse d'une autre variable. On dit aussi que le pointeur renvoie ou 'pointe' (d'où le nom) vers la variable concernée, cela via son contenu consistant en une adresse de variable. Les pointeurs sont parfois appelés 'indirections'.

Déclaration :

Comme toute variable, un pointeur doit être déclaré préalablement à son utilisation. Un pointeur est défini en préfixant l'identificateur avec l'opérateur de déréférencement '*'. Le programmeur doit également indiquer le type de la variable pointée. Il est possible, mais non obligatoire (contrairement aux références), d'initialiser les pointeurs. Il suffit alors d'indiquer l'adresse de la variable à pointer en utilisant l'opérateur de référence '&' (& indique l'adresse de).



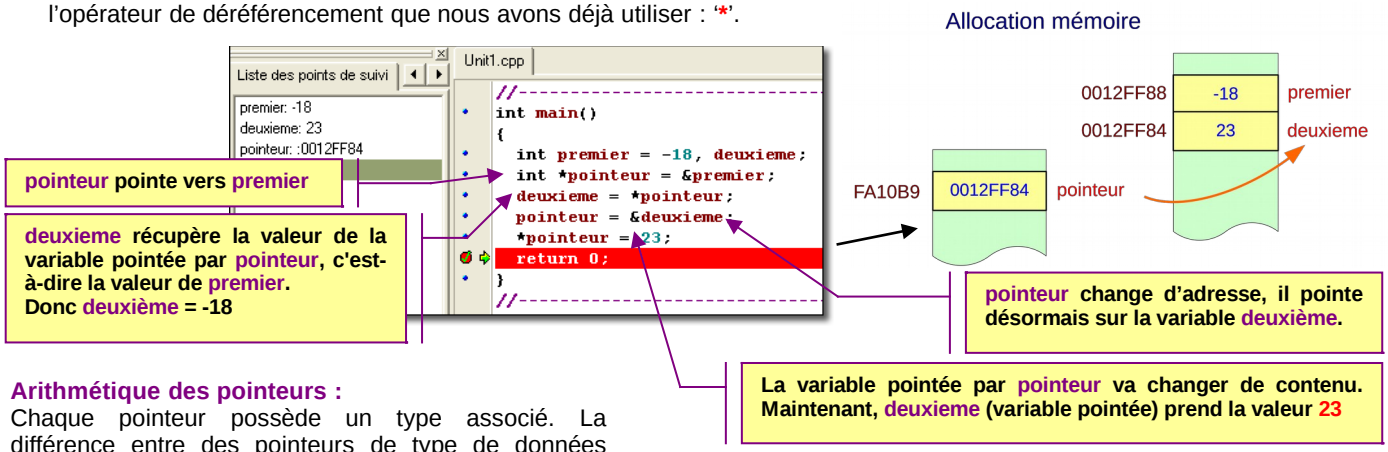




Accès indirect aux variables :

Il existe deux façons d'utiliser un pointeur.

- Soit nous nous intéressons à son contenu, auquel cas, c'est l'adresse d'une variable qui nous préoccupe. C'est notamment le cas lorsque nous décidons de pointer vers une autre variable. La syntaxe à utiliser est la même que pour tout autre variable, puisque c'est son contenu qui nous désirons changer.
- Soit nous décidons d'atteindre indirectement la variable pointée. Il est alors nécessaire d'utiliser une syntaxe adaptée pour bien préciser que c'est la variable pointée qui nous intéresse. Il existe un opérateur qui traite l'indirection et qui s'appelle l'opérateur de déréférencement que nous avons déjà utiliser : '*'.



Arithmétique des pointeurs :

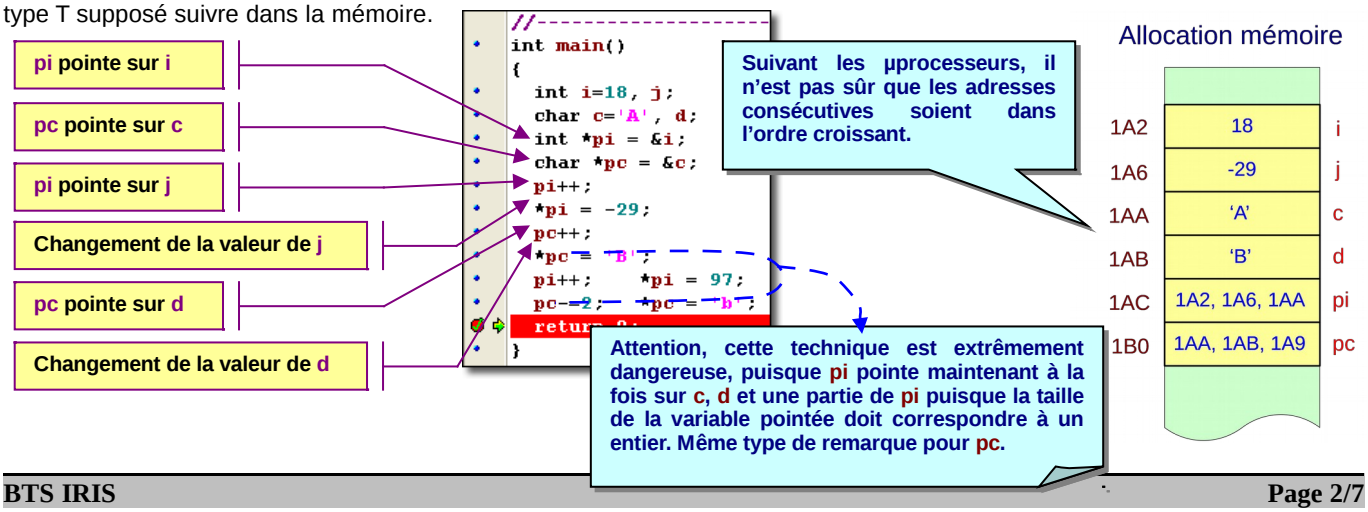
Chaque pointeur possède un type associé. La différence entre des pointeurs de type de données différents ne se trouve ni dans la représentation du pointeur, ni dans les valeurs (adresses) que le pointeur peut contenir – en effet toutes les adresses ont la même capacité mémoire. La différence est plutôt dans le type de la variable adressée. Le type du pointeur instruit le compilateur sur la façon dont il doit interpréter la mémoire trouvée à une adresse particulière ainsi que sur la quantité de mémoire que doit couvrir cette interprétation :

Lorsque nous utilisons la variable pointeur, suivant les syntaxes, nous avons :
pointeur → *'adresse de'* la variable pointée
***pointeur → *'contenu de'* la variable pointée**

- Un pointeur int adressant l'emplacement mémoire 1000 couvre l'espace d'adressage 1000-1003.
- Un pointeur double adressant mémoire 1000 couvre l'espace d'adressage 1000-1007.

Comme nous l'avons expérimenté dans le scénario précédent, il est tout à fait possible de modifier un pointeur, c'est-à-dire de changer l'adresse à laquelle il réfère. Nous avons d'ailleurs déjà utiliser l'opérateur d'affectation pour réaliser cela. On dispose de plusieurs autres opérateurs qui permettent de changer la valeur de l'adresse relativement à la précédente valeur.

L'opérateur d'incrémentation '++' par exemple agit de la même façon que sur des entiers. Cependant, le pointeur est augmenté de une position, non de un octet. En effet, le pointeur pointe sur un type T qui a une certaine taille en octets t (t=4 dans le cas d'un int) ; lorsqu'on écrit p++, dans ce cas, l'adresse est augmentée de t octets, afin de pointer sur l'élément de type T supposé suivre dans la mémoire.



On peut se demander légitimement si la technique de l'arithmétique des pointeurs est judicieuse puisqu'elle comporte de nombreux risques d'atteindre un emplacement mémoire non prévu au départ. Il existe deux types de risque :

- La nouvelle adresse ne correspond pas au type prévu, et les nouvelles valeurs proposées sont totalement intempestives.
- On risque d'atteindre une adresse qui se trouve en dehors des données déclarées et qui peut, suivant le cas, soit détruire vos lignes de codes (autodestruction), soit carrément atteindre un autre programme et provoquer le même genre d'inconvénient. (Normalement, les systèmes d'exploitations récents se prémunissent contre ce genre d'agression).

Les pointeurs et les tableaux :

En fait, l'arithmétique des pointeurs a été mis en place pour permettre de se déplacer librement au sein d'un tableau ou au travers d'une chaîne de caractères (qui est également un tableau). La particularité d'un tableau, c'est justement de disposer de cases contiguës d'un même type d'éléments. Par ailleurs, n'oubliez pas que le nom du tableau correspond à un pointeur constant placé sur sa première case.

Lignes de code successives	*Explications*
int a[ ] = {11, 22, 33, 44} ;	Déclaration d'un tableau de quatre entiers initialisé respectivement par les valeurs a[0]=11, a[1]=22, a[2]=33, a[3]=44.
int *pa, x;	Déclaration d'un pointeur sur un entier ainsi que d'un entier appelé x.

pa = &a[0];
ou
pa = a;
	Copie dans pa l'adresse de la première case du tableau. Comme le nom du tableau correspond justement à un pointeur constant sur la première case du tableau, la deuxième écriture est préférable puisque plus concise.
	
	Le nom d'un tableau, sans référence à un indice, peut être utilisé dans un programme, à condition de le considérer comme une constante.

x = *pa;	Copie le contenu de l'entier pointé par pa (c'est-à-dire a[0]) dans x. x ← 11
pa++;	pa est incrémenté d'une unité et contient maintenant l'adresse de la deuxième case du tableau, c'est-à-dire l'adresse de a[1]. pa = &a[1]
x = *pa;	pa pointe sur a[1], *pa est le contenu de a[1]. x ← 22
x = *pa + 1;	Copie dans x la valeur a[1]+1. x ← 23
x = *(pa+1);	Comme les parenthèses sont prioritaires par rapport à l'opérateur d'indirection, le calcul intermédiaire pa+1 est effectué en premier, ce qui donne comme résultat l'adresse de a[2] puisque l'unité d'incrémentation est la taille de la variable pointée. L'entier contenu à cette adresse (c'est-à-dire a[2]) est copié dans x. x ← 33 Notez que si pa intervient dans le calcul, il n'est pas modifié, pa pointe toujours sur a[1].
x = *++pa;	En vertu des règles de priorité des opérateurs (* et ++ sont de niveau 2), l'opération la plus à droite est d'abord effectuée, c'est-à-dire ++pa. pa, qui est modifié par cette opération, contient maintenant l'adresse de a[2]. pa = &a[2] L'opération d'indirection * est ensuite effectuée. x contient dès lors a[2]. x ← 33
x = ++*pa;	*pa est d'abord évalué. pa pointant sur a[2], *pa est a[2]. La cellule a[2] est soumise à incrémentation, toujours de 1 car a[2] n'est pas un pointeur mais bien un entier. x ← a[2] ← 34 Bien que l'on ait à faire à une préincrémentation, * est prioritaire par rapport à ++ car ces deux opérateurs se situent au niveau 2 de priorité. Or, à ce niveau, l'opérateur le plus à droite dans l'expression est prioritaire.
x = *pa++;	La postincrémentation, même si elle paraît prioritaire ici, n'est effectuée qu'en dernier lieu, par définition de la postincrémentation. Les règles de prorité indiquent cependant que l'opération ++ porte sur pa et non sur *pa. Le contenu de *pa (c'est-à-dire 34) est copié dans x. x ← 34 pa est ensuite incrémenté et contient maintenant l'adresse de a[3]. pa = &a[3]

Similitude entre les pointeurs et les tableaux :

Nous venons de voir que l'accès à un élément quelconque de tableau peut se faire non seulement par le nom du tableau accompagné d'un indice, mais aussi par un pointeur manipulé par des opérations spécifiques d'arithmétique de pointeurs. En voici un autre exemple :

En utilisant les indices	Avec l'arithmétique des pointeurs
int x[10] ; for (int i=0 ; i<10 ; i++) x[i] = i;	int x[10], *px = x ; for (int i=0 ; i<10 ; *px++ = i++) ;

En fait px et x, l'un comme l'autre sont des pointeurs. Il existe une seule différence. Un tableau est un pointeur constant qui pointe toujours sur la première case du tableau. Le fait qu'il y ait similitude entre les tableaux et les pointeurs est très important. Cela veut dire que lorsque nous aurons affaire à un pointeur, il sera possible d'avoir une écriture spécifiquement prévu pour les tableaux en utilisant notamment les indices. De la même manière, lorsque nous aurons affaire à un tableau, il sera également possible d'avoir une écriture spécifiquement prévu pour les pointeurs en utilisant l'arithmétique des pointeurs en faisant attention toutefois à ce que l'adresse du tableau ne change pas puisqu'elle doit être constante. Utilisons ce principe pour x et px :

En utilisant l'arithmétique des pointeurs pour le tableau	Avec l'indice de tableau sur un pointeur
int x[10] ; for (int i=0 ; i<10 ; i++) *(x+i) = i ;	int x[10], *px = x ; for (int i=0 ; i<10 ; i++) px[i] = i ;

L'exemple ci-dessus mérite quelques explications. Il est impératif de bien comprendre ce qui se passe au niveau du compilateur lorsqu'il analyse l'écriture d'un tableau. Revenons justement sur la variable x qui représente un tableau d'entier de 10 cases :

- x → adresse du premier élément du tableau → équivalent à &x[0]
- x+1 → adresse du deuxième élément du tableau → équivalent à &x[1]
- x+2 → adresse du troisième élément du tableau → équivalent à &x[2]
- x+n → adresse du n^{ème} élément du tableau → équivalent à &x[n]

Donc lorsqu'on écrit &x[n], cela veut bien dire : adresse de la case du tableau x indicé par n qui se traduit plus simplement par x+n, qui dans ce cas correspond plus à un décalage par rapport à l'adresse d'origine (première case du tableau). De toute façon, quel que soit l'écriture, nous sommes là en contact avec une adresse.

Lorsque nous voulons maintenant atteindre la donnée relative à cette adresse. Pour la syntaxe liée aux indices, il suffit d'enlever le symbole '&' (qui indique bien une adresse). Pour l'arithmétique des pointeurs, il est nécessaire de déréférencer en utilisant le symbole ' * '.

Donnée relative à : &x[n] ou x+n ⇔ x[n] ou *(x+n)

De toute façon, en interne, lorsque le compilateur rencontre une écriture utilisant la notation en indice, il sait que c'est un pointeur, et il transforme donc cette écriture pour avoir l'équivalent en arithmétique de pointeur. Lorsque nous avons :

t[i] le compilateur transforme systématiquement cette écriture par *(t+i)
t[4] le compilateur transforme systématiquement cette écriture par *(t+4)

Incidemment :

*(t+4) ⇔ *(4+t) puisque l'addition est commutative donc → t[4] ⇔ 4[t]

Différence entre pointeur et tableau :

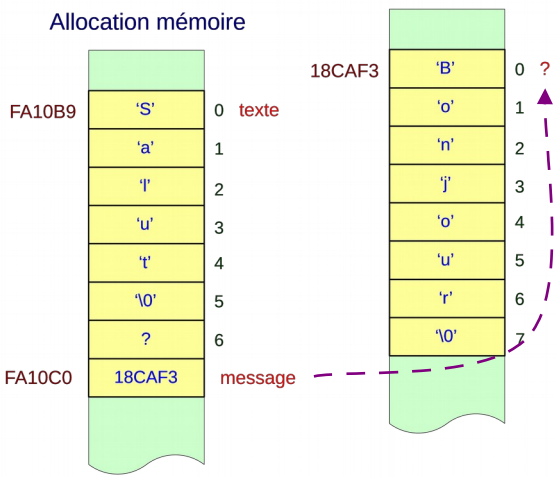
Tableau	Pointeur
Pointeur constant, mais avec une réservation d'une zone mémoire correspondant à la capacité du tableau pour stocker les données	Pointeur variable. On peut donc modifier son contenu. Par contre, il n'y a pas de réservation mémoire de la variable pointée. Cette réservation doit être réalisée au préalable.

Les chaînes de caractères :

Tout ce que nous avons pu dire ou faire sur les tableaux s'applique également pour les chaînes de caractères. D'ailleurs, on utilise très souvent un pointeur de caractère pour référencer une chaîne. Cela dépend du cas de figure. Lorsque nous décidons de prendre un tableau, cela veut dire que notre variable chaîne de caractères aura plusieurs valeurs possibles au cours du programme, et nous sommes donc obligé de faire une réservation mémoire suffisante pour pouvoir stocker la plus grande des chaînes admissible (le tableau est donc impératif). Lorsque nous avons besoin que d'une seule valeur de chaîne qui servira généralement de message, à ce moment là, il est préférable de prendre un pointeur qui sera initialiser par la constante littérale.

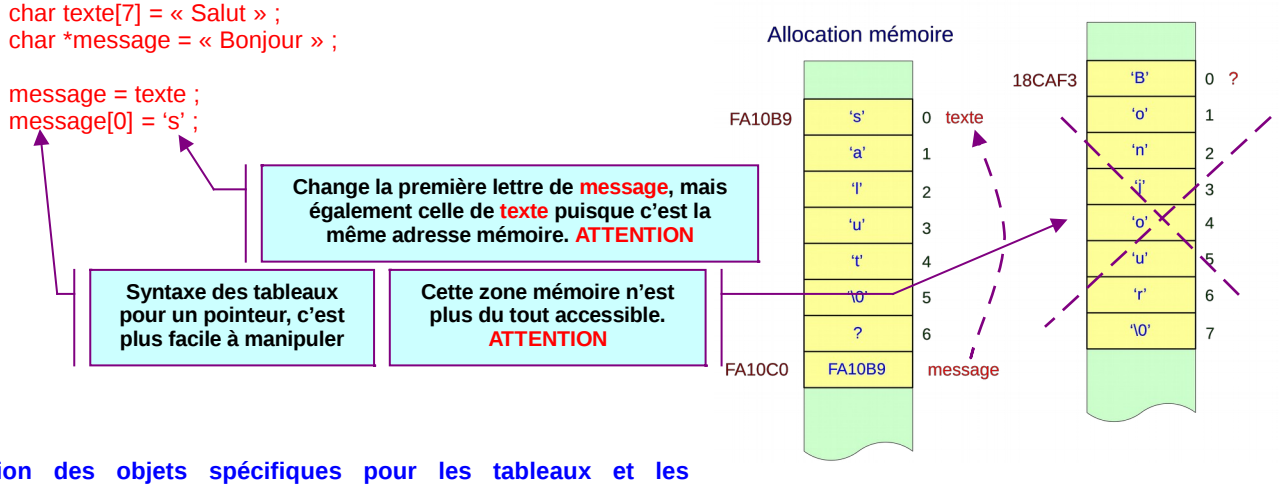
```
char texte[7] = « Salut » ;          // tableau de caractères
char *message = « Bonjour » ;        // pointeur vers un caractère
```

Rappel sur les constantes littérales chaînes
La constante littérale chaîne est un tableau de caractères qui ne porte pas de nom (anonyme) et qui se trouve quelque part en mémoire



La variable message est un pointeur de caractère, il est donc tout à fait possible de changer sa valeur au cours du temps. Attention toutefois, cela reste un pointeur, ce n'est pas une chaîne de caractères en tant que tel. Si vous proposer une nouvelle affectation, vous ne copiez pas la nouvelle chaîne, vous pointez vers cette nouvelle chaîne. Par ailleurs, si vous faites cette affectation, l'adresse de la chaîne précédente est perdue, ce qui fait que l'ancienne chaîne n'est plus du tout accessible.

```
char texte[7] = « Salut » ;
char *message = « Bonjour » ;

message = texte ;
message[0] = 's' ;
```



Utilisation des objets spécifiques pour les tableaux et les chaînes :

Vu la difficulté d'utiliser correctement les tableaux et les chaînes de caractères, notamment dans le cadre de l'affectation (les tableaux et les chaînes sont des pointeurs constants), nous utiliserons dorénavant des objets spécifiques représentant ces deux types d'entités, respectivement la classe ***vector<>*** et la classe ***string***.

```
vector<int> entiers; // Tableau dynamique d'entiers
string chaine;       // Chaîne de caractères de taille quelconque
```

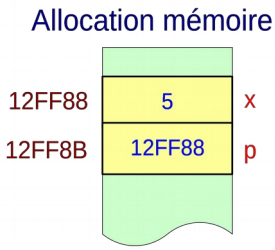
Pointeur, adresse de pointeur et variable pointée :

Le symbole '**&**' de calcul d'adresse peut porter sur un pointeur : il s'agit de l'adresse d'implantation en mémoire du pointeur.

```
int x=5, *p = &x ;
```

déterminez :

```
p = ?        // désigne le contenu du pointeur
&p = ?       // désigne l'adresse du pointeur
*p = ?       // désigne le contenu de la variable pointée.
```



Pointeur non initialisé :

```
char *p ;
cout << p ;
*p = 'A' ;
```

Le pointeur n'a pas été initialisé. Il contient dès lors une valeur aléatoire. Par conséquent, il pointe de façon aléatoire quelque part en mémoire.

Affichage d'une chaîne pointée par p, à ce moment là, sa longueur est totalement aléatoire ! (Il faut que cout trouve le caractère terminal '\0')

Extrêmement dangereux, on modifie le contenu d'une case mémoire sans l'avoir localisée. Il se peut que se soit une case mémoire importante.

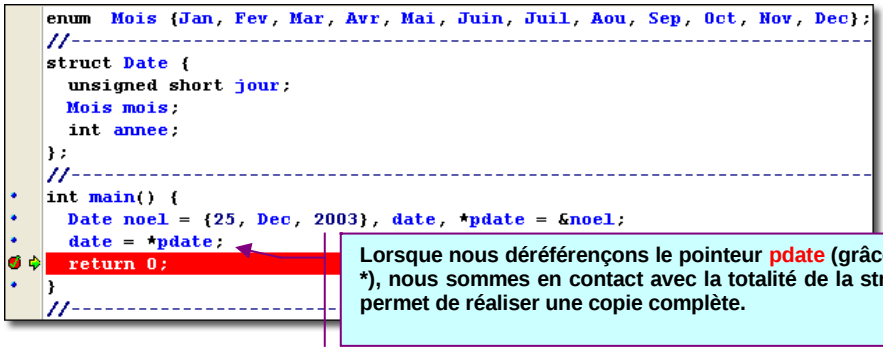
ATTENTION
Si nous avons de la chance, p pointe sur une zone inoccupée de la mémoire et tout se déroulera sans conséquences dramatiques. Mais si p pointe sur une zone de la mémoire qui contient des instructions ou des données, vous détruisez cette zone. Avec les conséquences que l'on peut imaginer…
Si nous désirons qu'un pointeur ne pointe sur rien momentanément, il est préférable de l'initialiser à 0.

Règle sur les pointeurs
Un pointeur doit toujours être initialisé avant d'être utilisé pour consultation ou modification des valeurs pointées. Il n'est pas impératif que cette initialisation soit réalisée au moment de la déclaration, elle peut-être faite plus tard, le tout, c'est que le pointeur soit prêt avant d'être utiliser.

C++11
Il existe dorénavant un nouveau mot clé du langage qui permet de spécifier un pointeur nul, ou plus précisément qu'il ne point sur rien. Il s'agit de *nullptr*.

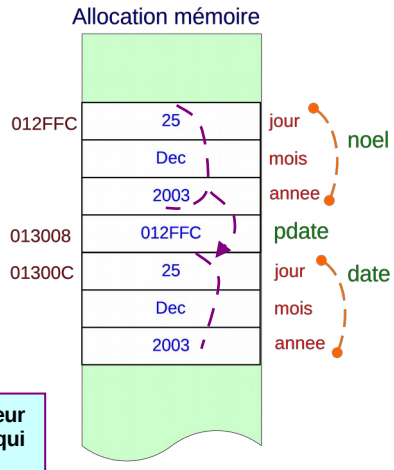
Les pointeurs sur des structures ou des unions :

Les pointeurs peuvent intégrer des adresses de variable de n'importe quel type, et bien évidemment, c'est aussi vrai pour les variables de type structure ou union. Les exemples que nous prendrons, seront uniquement proposés pour des structures, mais la syntaxe demeure identique pour les unions.



```
enum  Mois {Jan, Fev, Mar, Avr, Mai, Juin, Juil, Aou, Sep, Oct, Nov, Dec};
//-------------------------------------------------------------------------
struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
};
//-------------------------------------------------------------------------
int main() {
  Date noel = {25, Dec, 2003}, date, *pdate = &noel;
  date = *pdate;
  return 0;
}
//-------------------------
```

Lorsque nous déréférençons le pointeur pdate (grâce à l'opérateur *), nous sommes en contact avec la totalité de la structure, ce qui permet de réaliser une copie complète.



Dans l'exemple ci-dessus, nous désirions travailler avec la totalité de la structure pointée. Nous remarquons que la syntaxe est identique au pointeur de type entier ou caractère. Il suffit d'utiliser l'opérateur de déréférencement. Par contre, nous avons souvent besoin de d'accéder à un seul des champs de la structure. Il faut que le système procède en deux étapes. En premier, il faut déréférencer la structure (grâce à l'opérateur de déréférencement **' * '**) et ensuite accéder au champ désiré (grâce à l'opérateur de champ **'.'**). Exemple :

```
(*pdate).jour = 12 ;          // change le jour de noel
```

Attention, les parenthèses sont nécessaires à cause de la priorité des opérateurs. '.' Est prioritaire sur '*'. Sans les parenthèses, le compilateur aurait interprété :

```
*(pdate.jour) = 12 ;          // par cette écriture, c'est le champ de la structure qui est déréférencé → Erreur compilation
```

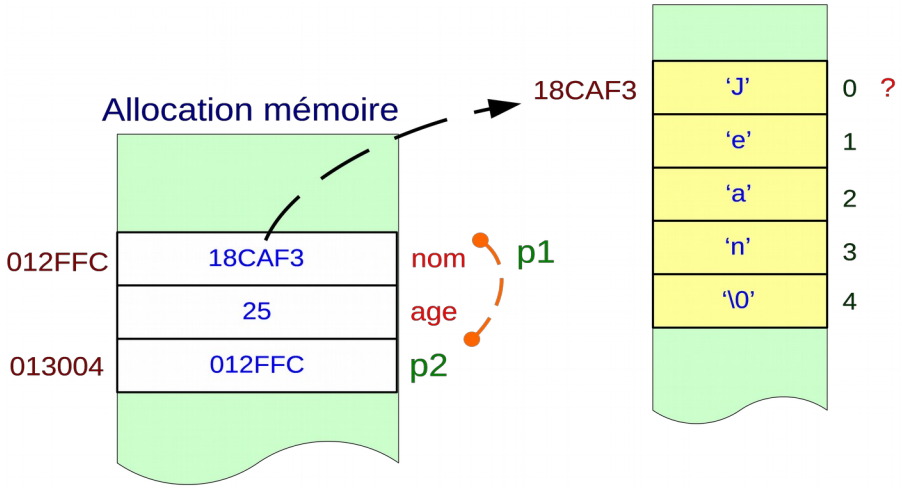
Cette syntaxe est un peu lourde. Les concepteurs du langage ont donc prévus un opérateur spécialisé sur l'accès à un champ d'une structure (ou d'une union) pointée. Il s'agit de l'opérateur **'→'**. Du coup, l'exemple précédent devient :

```
pdate→jour = 12 ;          // jour est un champ d'une structure pointée par pdate.
```

Exemple :

Déclaration	Question	Réponse

struct {
  char *nom ;
  int age ;
} p1 = { "Jean", 25 }, *p2 = &p1;
	Afficher nom de p1
	Afficher age de p1
	Afficher 1ère lettre du nom de p1
	Afficher nom de p2
	Afficher age de p2
	Afficher 1ère lettre du nom de p2
	Que se passe t-il après
	p2 --
	p1.nom++
	*(p2+1)→nom
		cout << p1.nom ;
		cout << p1.age ;
		cout << *p1.nom ; ⇔ cout << p1.nom[0] ;
		cout << p2→nom ;
		cout << p2→age ;
		cout << *p2→nom ; ⇔ cout << p2→nom[0] ;
		
		p2 = 012FF4 // décrémentation d'une structure
		p1.nom = 18CAF4 // 2ème lettre du nom
		'e' // représente la 2ème lettre du nom de p1

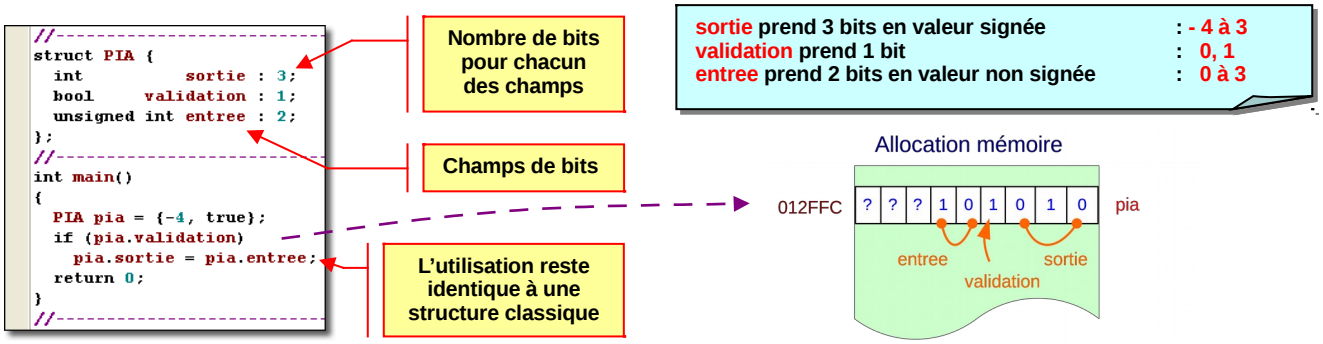


Les structures de bits

Le langage C++ permet de définir et manipuler des champs dont la taille est inférieure à un octet et qui se mesure en nombre de bits. Un champ de bits aura un type de donnée entier (également caractère ou même booléen), signé ou non signé. L'identificateur de champ de bits est suivi d'un deux points **' :'**, puis d'une expression constante indiquant le nombre de bits.

Les champs de bits définis consécutivement dans le corps de la structure sont regroupés dans des bits adjacents du même entier, permettant ainsi de compresser la mémoire.

Un champ de bit est accédé de la même manière que les autres données membres d'une structure.



Les champs de bits peuvent procurer des facilités dans certains cas ; ils sont surtout utiles dans des applications très techniques faisant intervenir le matériel ou les périphériques.

Union

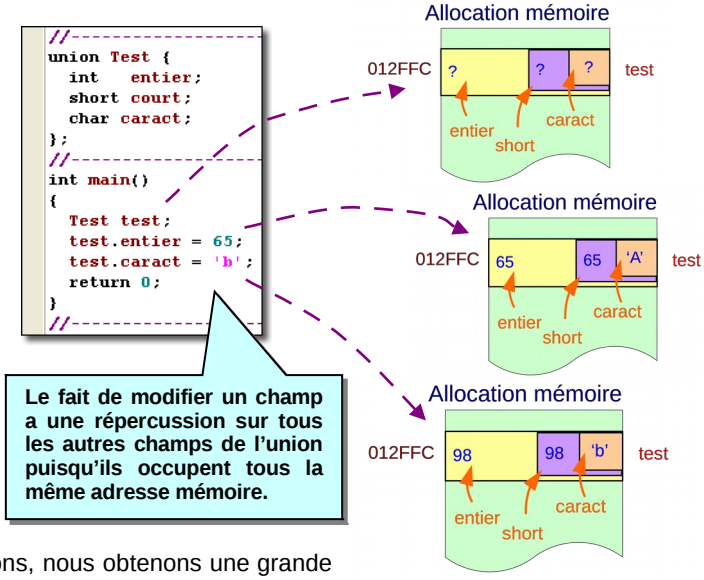
Une union est une sorte de structure spéciale. Les données membres dans une union sont stockées en mémoire de façon à ce qu'elles se recouvrent. Chaque membre commence à la même adresse mémoire. La quantité mémoire allouée à une union est celle nécessaire pour contenir la plus grande de ses données membres. Seul un membre à la fois peut être effecté d'une valeur.

Déclaration
Une union se déclare exactement de la même façon qu'une structure. C'est uniquement l'organisation mémoire interne qui est différente. Le mot réservé est **union**. Il est donc possible de déclarer un nouveau type et d'y associer toutes les variables nécessaires au moment de la déclaration. On utilise également les mêmes règles pour la déclaration des champs.

```
union  Type
{
      type1  champ1 ;
      type2  champ2, champ3 ;
      …
      typeN   champN ;
}  variable1, variable2, … ;
```

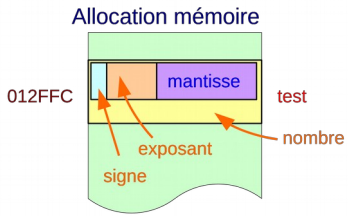
Nom donné à l'union
Nouveau type défini par l'utilisateur

Variables associées.



Le fait de modifier un champ a une répercussion sur tous les autres champs de l'union puisqu'ils occupent tous la même adresse mémoire.

Lorsque nous combinons les structures de bits avec des unions, nous obtenons une grande richesse d'expression. Il sera alors possible de manipuler des données dotées d'une infrastructure très compliquée, avec au contraire, une utilisation d'une simplicité décourcertante. Par exemple, il est possible de représenter un nombre réel avec sa représentation à virgule flottante, et en même temps dans sa représentation en notation scientifique avec la séparation de la mantisse, de l'exposant et de la valeur du signe.



Structure anonyme

A cet instant là, x.nombre = -1.5F

```
union Reel {
  float nombre;
  struct{
    unsigned mantisse : 23;
    unsigned exposant : 8;
    int signe : 1;
  };
};
int main()
{
  Reel x;
  x.nombre = 1.5F;
  x.signe = 1;
  return 0;
}
```

La plupart du temps, une union propose une alternative de représentation.