1

Mémoire et Pointeurs

L'objectif de ce chapitre est de démystifier la différence entre un **pointeur** et une **référence** et de démontrer quand il est préférable d'en utiliser un plus que l'autre. Ce chapitre est fondamental et il est fortement conseillé de faire (et refaire) les exemples et de bien assimiler cette théorie. Nous pouvons affirmer, sans l'ombre d'un doute, que la notion de pointeurs en C++ est ce qui effraie le plus les nouveaux programmeurs. Alors, commençons par une petite révision.

Les variables

La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder directement à cette valeur.

Adressage direct : Accès au contenu d'une variable par l'identifiant (nom) de la variable.

Exemple :

```
int varA;
varA = 10;
```

En programmation, nous utilisons des variables pour stocker des informations. Lorsque vous déclarez une variable en C++ (ou dans tout autre langage de programmation), l'ordinateur alloue de la mémoire, selon la taille demandée, pour cette variable quelque part dans la mémoire. L'exemple de code suivant affiche la taille que les types de bases du C++ occupent dans la mémoire :

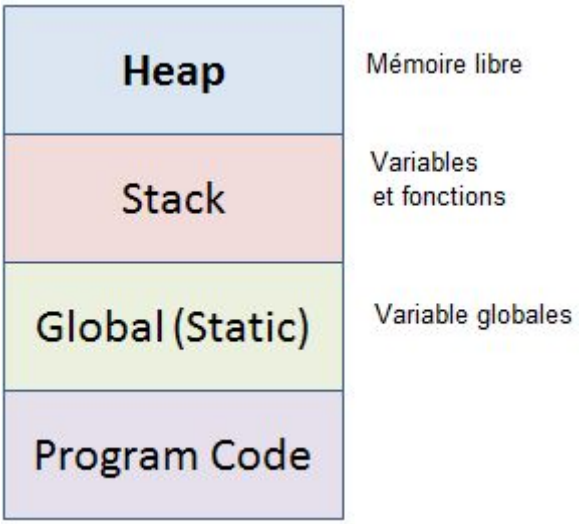
```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "char: " << sizeof(char) << endl;
    cout << "short: " << sizeof(short) << endl;
    cout << "int: " << sizeof(int) << endl;
    cout << "long: " << sizeof(long) << endl;
    cout << "float: " << sizeof(float) << endl;
    cout << "double: " << sizeof(double) << endl;
}
```

Valeurs possible par type :

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	

Mémoire

La mémoire dans un ordinateur est une **succession d'octets** (8 bits), organisés les uns à la suite des autres, accessibles par **une adresse**. La mémoire pour stocker des variables est organisée en deux catégories, c'est-à-dire la **pile** (stack) et le **tas** (heap).



En résumé, **votre code compilé réside en mémoire**. Ensuite, un espace est prévu pour les **variables globales**. Nous allons voir la notion de statique dans un autre chapitre. Mais pour l'instant, sachez que ce sont des variables accessibles de partout. Ce qui nous intéresse ici est la **Stack** et la **Heap**.

Nous allons premièrement nous intéresser à la Stack. Cette partie de la mémoire de notre programme est automatiquement gérée par le programme. Vous n'avez rien à faire pour détruire une variable sur la stack. Par exemple, lorsque vous appelez une fonction, ses paramètres sont copiés sur la Stack et lorsqu'elle se termine, la mémoire est automatiquement détruite.

```cpp
#include <iostream>
using namespace std;

void Add(int a, int b, int c)
{
    a = b + c;
    cout << b << " + " << c << " = " << a << endl;
}

int main()
{
    int a = 0;
    int b = 10;
    int c = 20;

    Add(a, b, c);
    cout << b << " + " << c << " = " << a << endl;

    return 0;
}
```

Lorsque la fonction “Add” est appelée, de la mémoire est allouée à celle-ci et les paramètres sont **copiés** sur la Stack. À la ligne 6, c’est la **copie** de “a” sur la Stack qui prend la valeur de b + c et non la variable originale de la ligne 12. Pour régler ce problème, nous allons avoir besoin de **références** ou de **pointeurs**.

Références

Une variable référence est un **alias**, un autre nom, pour une variable qui existe déjà. Lorsqu’une référence est associée à une variable, la référence peut être utilisée comme une variable normale.

Pour déclarer une référence, vous devez ajouter l'opérateur **(&)** après le type de la variable.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int varA = 20;
7      int &ref = varA;
8
9      cout << varA << endl;
10     cout << ref << endl;
11     return 0;
12 }
```

À la ligne 7, vous devez lire :

ref, qui est une référence d'un type int, prend la valeur de varA.

Vous allez le voir dans la section sur les pointeurs, les références sont beaucoup plus simples à utiliser. En fait, dans la plupart des langages de haut niveau, comme Java ou C#, les pointeurs vous sont cachés et il ne vous reste que des références.

Une référence peut être passée en paramètre à une fonction et la copie sur la Stack référence toujours la variable originale. L'exemple suivant est une fonction qui passe un paramètre **par référence** :

```cpp
#include <iostream>
using namespace std;

void Add(int &a, int b, int c)
{
    a = b + c;
    cout << b << " + " << c << " = " << a << endl;
}

int main()
{
    int a = 0;
    int b = 10;
    int c = 20;

    Add(a, b, c);
    cout << b << " + " << c << " = " << a << endl;

    return 0;
}
```

La référence est un alias à la variable originale. La valeur sera donc réellement modifiée, et lorsque le programme détruira la mémoire de la fonction Add, c'est la référence qui est détruite.

Les pointeurs

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable, nous pouvons copier l'adresse de cette variable dans une variable spéciale appelée **pointeur**. Ensuite, nous pouvons retrouver l'information de la variable en passant par le pointeur.

Adressage indirect : Accès au contenu d'une variable en passant par un pointeur qui contient l'adresse de la variable.

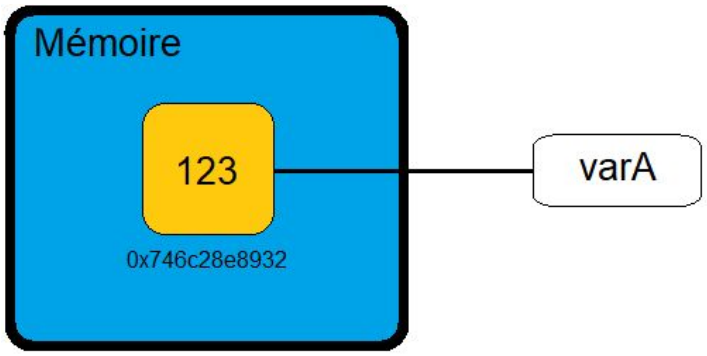
Un pointeur est une **variable** qui contient une **adresse** mémoire où vit une **valeur**. Ce concept est utilisé partout dans un programme en C++ et il serait impensable de maîtriser ce langage sans le connaitre. C'est un sujet difficile à saisir, alors il en vaut le coup de le diviser en morceaux et de bien comprendre chaque section avant de passer à la prochaine.

Adresse mémoire

Comme nous l'avons vu dans la section sur les variables, lorsque l'on déclare une variable, l'ordinateur alloue de l'espace dans la mémoire et y accroche une étiquette avec le nom de la variable.

```
1  int main()
2  {
3      int varA = 123;
4      return 0;
5  }
```

Dans cet exemple, l'ordinateur alloue **4 octets** quelque part dans la mémoire et lui donne l'étiquette "varA". Finalement, à cet endroit dans la mémoire, la valeur de "123" est stockée.



Chaque variable ne possède qu'une et une seule adresse et chaque adresse correspond à une seule variable. Il existe donc au moins deux façons d'accéder à une variable. On peut atteindre la case jaune de l'image précédente par :

- Le nom de la variable;
- L'adresse de la variable.

Mais pourquoi ne pas toujours utiliser la variable par son nom ou une référence?

Nous allons voir plus loin que de passer par l'adresse est parfois nécessaire en C++. Commençons premièrement par voir comment connaître l'adresse d'une variable.

Afficher l'adresse

En C++, le symbole pour obtenir l'adresse d'une variable est l'esperluette **(&)**. Pour **V**, étant une variable dans la mémoire de l'ordinateur :

```
&V
```

Se lis, l'adresse de V.

Si nous voulons afficher l'adresse de la variable varA, nous devons écrire simplement &varA.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int varA = 123;
    cout << "L'adresse est : " << &varA << endl;
    return 0;
}
```

Résultat :

```
L'adresse est : 0x78adf0a235ec
```

Vous aurez un résultat différent, car l'adresse change d'une exécution à l'autre. Même si elle contient des lettres, cette adresse est un nombre. Celui-ci est simplement écrit en hexadécimal, une autre façon de représenter de large nombre. En décimal, cette adresse correspond à l'adresse mémoire **132,688,461,837,804**. Cette information n'est utile que pour l'humain, car au final, dans l'ordinateur, c'est en binaire. Mais qui est capable de lire ce nombre en binaire? 111100010101101111100001010001000110101111101100.

Les pointeurs

Les adresses sont des nombres. Vous connaissez plusieurs types permettant de stocker des nombres : int, float, double. Mais, il nous faut utiliser un type un peu particulier : le pointeur. **Un pointeur est une variable qui contient l'adresse d'une autre variable**. Retenez bien cette phrase. Elle peut vous sauver la vie dans un examen.

Déclarer un pointeur

Pour déclarer un pointeur il faut, comme pour les variables, deux choses :

- Un type;
- Un identifiant (un nom).

Pour le nom, les mêmes règles que pour les variables s'appliquent. Le type d'un pointeur a une petite subtilité. Il faut indiquer quel est le type de variable dont on veut stocker l'adresse et ajouter une étoile **(*)**.

```
1  int main()
2  {
3      int* ptr;
4      return 0;
5  }
```

Cette notation a un léger inconvénient, c'est qu'elle ne permet pas de déclarer plusieurs pointeurs sur la même ligne, comme ceci :

```
int* ptr1, ptr2, ptr3;
```

Si l'on procède ainsi, seul `ptr1` sera un pointeur, les deux autres variables seront des entiers tout à fait standards.

Pour le moment, ce pointeur ne contient aucune adresse connue. **Cette situation est dangereuse**. Si vous essayez d'utiliser le pointeur, vous ne savez pas quelle adresse de la mémoire vous manipulez. Cela peut être n'importe quelle adresse. Par exemple, celle qui contient votre mot de passe Windows ou celle stockant l'heure actuelle. Une mauvaise manipulation des pointeurs peut créer de graves conséquences. Il ne faut donc jamais déclarer un pointeur sans lui donner une valeur initiale. Par conséquent, pour être tranquille, il faut toujours déclarer un pointeur en lui **donnant une adresse** ou la valeur **nullptr** :

```
1  int main()
2  {
3      int var = 123;
4
5      int* ptrA;
6      int* ptrB = nullptr;
7      int* ptrC = &var;
8
9      return 0;
10 }
```

5: ne faites pas ça...;
6: Assignez l'adresse 0 (ou nullptr) à votre pointeur;
7: Si vous connaissez déjà son adresse, assignez-la à votre pointeur.

Stocker un adresse

Maintenant que nous avons déclaré une variable pointeur, il n'y a plus qu'à y mettre une valeur. Vous savez déjà comment obtenir l'adresse d'une variable à l'aide de l'opérateur (**&**).

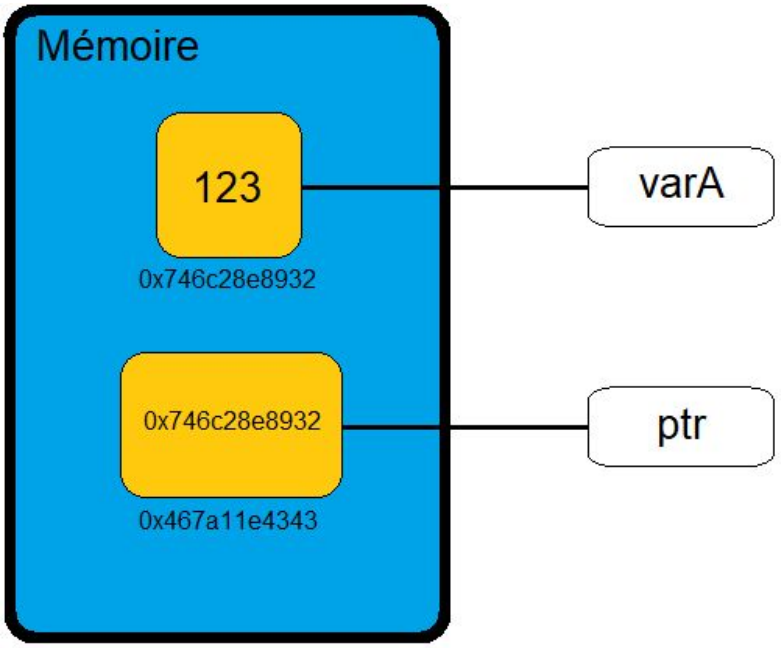
Pour **V**, étant une variable qui prend de l'espace dans la mémoire, et **P**, un pointeur qui contient une adresse :

```
P = &V
```

P est un **pointeur** qui prend comme **valeur**, **l'adresse** de la **variable V**.

```
1  int main()
2  {
3      int varA = 123;
4      int* ptr = &varA;
5      return 0;
6  }
```

La ligne 4 est celle qui nous intéresse. Elle écrit l'adresse de la variable **varA** dans le pointeur **ptr**. On dit alors que le pointeur ptr pointe sur varA. Voyons comment tout cela se déroule dans la mémoire grâce à une image.



On retrouve quelques éléments familiers : le nom varA désigne un espace mémoire qui se retrouve à l'adresse 0x746c28e8932.

La nouveauté est le pointeur. À l'adresse mémoire 0x467a11e4343, il y a une variable nommée ptr qui a pour valeur l'adresse 0x746c28e8932, c'est-à-dire l'adresse de la variable varA.

Comme pour toutes les variables, on peut afficher le contenu d'un pointeur.

```
1  int main()
2  {
3      int varA = 123;
4      int* ptr = &varA;
5
6      cout << "L'adresse de varA est " << &varA << endl;
7      cout << "La valeur de ptr est " << ptr << endl;
8      return 0;
9  }
```

Résultat :

```
L'adresse de varA est 0x731fbc04dd1c
La valeur de ptr est 0x731fbc04dd1c
```

C'est confirmé, la valeur du pointeur est bien l'adresse de la variable pointée. On a bien réussi à stocker une adresse !

Voilà, vous savez tout ou presque. Cela peut sembler absurde pour le moment, mais faites-moi confiance : les choses vont progressivement s'éclairer pour vous. Si vous avez compris la théorie jusqu'à présent, alors vous pouvez vous attaquer à des programmes plus complexes. Vous aurez la chance de tester vos connaissances à la fin de ce chapitre et tout au long du cours.

Pratique d'indirection:

```
1  int main()
2  {
3      int a = 5, b = 2, c = 17;
4      int* p = NULL;
5
6      p = &a;
7      cout << "p pointe sur a : " << *p << endl;
8
9      p = &b;
10     cout << "puis sur b : " << *p << endl;
11
12     p = &c;
13     cout << "et enfin sur c : " << *p << endl;
14
15     return 0;
16 }
```

Accéder à la valeur pointée

Le rôle d'un pointeur est d'accéder à une variable **sans passer par son nom**. Voici comment faire : il faut utiliser l'étoile **(*)** sur le pointeur pour afficher la valeur de la variable pointée. Pour **P**, étant un pointeur sur une variable :

```
*P = 123
```

La valeur de la mémoire pointée par **P** prend valeur de 123.

```
1  int main()
2  {
3      int varA = 123;
4      int* ptr = &varA;
5
6      cout << "La valeur de varA est " << varA << endl;
7      cout << "La valeur de ptr est " << *ptr << endl;
8      return 0;
9  }
```

À la ligne 7, le programme effectue les étapes suivantes :

1. Aller à l'adresse mémoire nommée ptr;
2. Lire la valeur enregistrée;
3. Aller à l'adresse pointée;
4. Lire la valeur stockée dans la mémoire;
5. Afficher cette valeur : ici, ce sera 123.

En utilisant l'étoile, **on accède à la valeur de la variable pointée**. C'est ce qui s'appelle **déréférencer un pointeur**. Voici donc un deuxième moyen d'accéder à la valeur de varA.

Pouvez-vous modifier le code que nous avons vu plutôt sur les paramètres passés par copie pour éviter la copie avec des pointeurs et réellement modifier la valeur de la variable originale?

```
1  #include <iostream>
2  using namespace std;
3
4  void Add(int* a, int b, int c)
5  {
6      *a = b + c;
7  }
8
9  int main()
10 {
11     int a = 0;
12     int b = 10;
13     int c = 20;
14
15     Add(&a, b, c);
16
17     cout << a << " = " << b << " + " << c << endl;
18 }
19
```

Effectivement, pour un pointeur, c'est l'adresse de la variable qui y sera copiée sur la stack, et donc, l'indirection modifie la variable d'origine.

Si vous avez déjà fait des tableaux en C++, et bien, vous utilisez des pointeurs sans le savoir. Dans l'exemple suivant :

```
1  int main()
2  {
3      int a[5] = { 10, 12, 21, 13, 54 };
4
5      cout << a << endl;
6      cout << a[1] << endl;
7      cout << *(a + 1) << endl;
8
9      return 0;
10 }
```

NOTE: les tableaux résident dans la mémoire de façon continue.

La ligne 3 déclare un tableau de 5 éléments. En réalité, "a" est un pointeur qui pointe sur le premier élément. Aux lignes 6 et 7 affichent la valeur de la position 1 du tableau. On peut donc parcourir un tableau avec un pointeur :

```
1  void Print(int *ptr, int n)
2  {
3      for(int i = 0; i < n; i++)
4      {
5          cout << *(ptr + i) << endl;
6      }
7  }
8
9  int main()
10 {
11     int a[5] = { 10, 12, 21, 13, 54 };
12     Print(a, 5);
13
14     return 0;
15 }
```

Exercice: ajouter le code pour copier un tableau en utilisant les pointeurs **t1** et **t2**.

```
void Print(int* t)
{
    while (*t++)
    {
        cout << *t << " ";
    }

    cout << endl;
}

int main()
{
    int tableau1[10] = { 5, 3, 9, 11, 4, 132, 45, 2, 89, 0 };
    int tableau2[10] = { -1, -1, -1, -1, -1, -1, -1, -1, -1, 0 };
    int* t1 = tableau1;
    int* t2 = tableau2;

    Print(tableau2);

    // Copie de tableau ici...

    Print(tableau2);
    return 0;
}
```

Il existe un moyen de pointer sur un peu tout et n'importe quoi: un pointeur **void**. C'est comme un pointeur avec un type, excepté que vous ne pouvez pas faire d'indirection comme d'habitude. Il faut effectuer une conversion pour pouvoir récupérer la valeur contenue à l'adresse en question.

```
int main()
{
    int a = 45;
    float b = 32.5f;
    void* p;

    p = &a;
    cout << "a = " << *((int*)p) << endl;

    p = &b;
    cout << "b = " << *((float*)p) << endl;

    return 0;
}
```

Récapitulatif de la notation

Cette notation peut parfois sembler compliquée. L'étoile a deux significations différentes et on utilise l'esperluette alors qu'elle sert déjà pour les références. Essayons donc de récapituler le tout.

Pour une variable N :

- N est un identifiant qui permet d'accéder à une valeur (c'est le nom de la variable);
- &N permet d'accéder à l'adresse de la variable.

Pour un pointeur P :

- P permet d'accéder à la valeur du pointeur, c'est-à-dire à **l'adresse** de la variable pointée;
- *P permet d'accéder à la **valeur** de la variable pointée.

C'est ce qu'il faut retenir de cette section. Je vous invite à tester tout cela avec les nombreux exercices du chapitre pour vérifier que vous avez bien compris comment afficher une adresse, comment utiliser un pointeur, etc. Sans ces connaissances, vous n'irez pas bien loin en C++. Il faut impérativement s'entraîner pour bien comprendre. Les meilleurs sont tous passés par là et je peux vous assurer qu'ils ont aussi soufferts en découvrant les pointeurs.

Allocation dynamique

Dans certains cas, le programmeur doit gérer la mémoire lui-même. C'est-à-dire, quand elle doit être créé, quand elle est détruite, etc. Jusqu'à maintenant, lors de la déclaration d'une variable, le programme effectue deux étapes :

1. Il demande à l'ordinateur de lui fournir une zone dans la mémoire. En termes techniques, on parle **d'allocation de la mémoire**.
2. Il remplit cette case avec la valeur fournie. On parle alors **d'initialisation de la variable**.

Tout cela est entièrement automatique. De même, lorsqu'on arrive à la fin d'une fonction, le programme rend la mémoire utilisée à l'ordinateur. C'est ce qu'on appelle la **libération de la mémoire**. C'est à nouveau automatique. Tout ceci se faisait automatiquement sur la **stack**. Nous allons maintenant apprendre à le faire manuellement sur la **heap**, et pour cela, nous avons besoin des pointeurs.

Allouer un espace mémoire

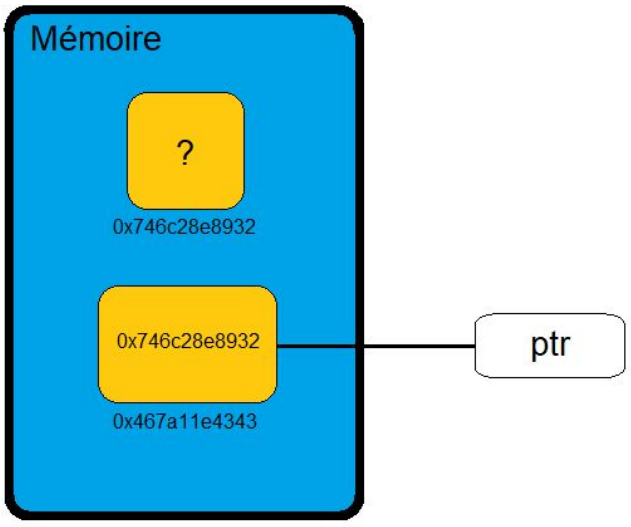
La heap est l'espace mémoire d'un programme que vous devez gérer vous-même. (avec **new**, **delete**, **malloc**, ou **free**.) Contrairement à la stack, la gestion de cette mémoire n'est pas automatique et c'est au programmeur de la demander et de s'assurer qu'elle est éventuellement libérée. Pour demander manuellement de la mémoire, il faut utiliser l'opérateur **new** ou **malloc**. Ces opérateurs nous renvoient un pointeur vers l'espace de la mémoire que l'ordinateur nous a alloué.

```
1  int main()
2  {
3      int* p;
4      int* T;
5
6      p = (int*)malloc(sizeof(int));
7      *p = 1;
8
9      T = (int*)malloc(sizeof(int) * 10);
10
11     for (int i = 0; i < 10; i++)
12     {
13         *(T + i) = 0;
14         T[i] = 0;
15     }
16 }
```

Avec l'opérateur malloc, nous devons à la ligne 6, indiquer le type de variable pointée ainsi que la taille que la variable prendra en mémoire. Bien que cette façon de faire soit beaucoup plus flexible, l'opérateur new se charge de tout pour nous comme on peut le voir dans l'exemple qui suit :

```
1  int main()
2  {
3      int* ptr = new int;
4      int* T;
5      *ptr = 123;
6
7      T = new int[10];
8
9      for (int i = 0; i < 10; i++)
10     {
11         *(T + i) = 0;
12         T[i] = 0;
13     }
14 }
```

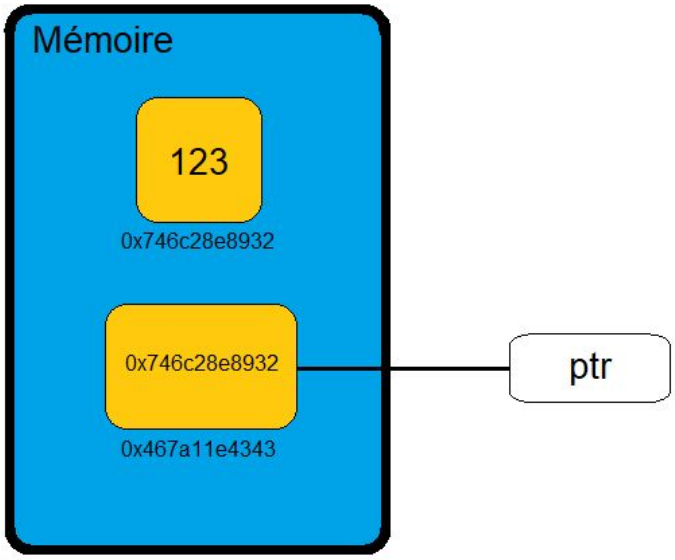
La ligne 6 demande de la quantité de mémoire nécessaire pour stocker un entier et retourne l'adresse dans le pointeur.



Sur cette image, **ptr** contient l'adresse d'une variable qui n'a pas d'identifiant. Si vous changez la valeur de **ptr**, vous perdez le seul moyen d'accéder à cette emplacement dans la mémoire. Vous ne pourrez donc plus l'utiliser ni la supprimer ! Elle sera définitivement perdue, mais elle continuera à prendre de la place dans la mémoire. C'est ce qu'on appelle une **fuite de mémoire**. Il faut donc faire attention.

Une fois allouée manuellement, la variable s'utilise comme n'importe quelle autre. On doit juste se rappeler qu'il faut y accéder par le pointeur en le déréférençant. C'est-à-dire, en ajoutant un étoile **(*)** devant le pointeur (ligne 7).

Après l'exécution de cette exemple, la valeur 123 sera attribuée à notre variable.



À part son accès un peu spécial (via *ptr), nous avons donc une variable en tout point semblable à une variable avec identifiant. Il ne faut pas oublier que nous devons maintenant libérer la mémoire que l'ordinateur nous a allouée.

Libérer la mémoire

Une fois que vous n'avez plus besoin de la mémoire, vous devez la rendre à l'ordinateur avec l'aide de l'opérateur **delete** ou **free**. Delete peut être utilisé pour un pointeur ou delete[] pour un tableau de pointeur créé avec un new.

Une fois que la mémoire est rendue à l'ordinateur, elle pourra être utilisée pour autre chose. Le pointeur, lui, existe toujours et il pointe toujours sur la même adresse. Vous ne devez plus l'utiliser, et pour être sûr, il est conseillé de mettre la valeur du pointeur à **nullptr**.

```
1  int main()
2  {
3      int* tabA = new int[10];
4      int* ptr = new int();
5
6      delete[] tabA;
7      delete ptr;
8
9      tabA = nullptr;
10     ptr = nullptr;
11
12     return 0;
13 }
```

N'oubliez pas de libérer la mémoire que vos applications utilisent, surtout pour des machines plus restreintes en mémoire comme un téléphone intelligent ou une console portative. Si vous ne le faites pas, votre programme risque d'utiliser de plus en plus de mémoire, jusqu'au moment où il n'y aura plus aucune mémoire disponible; et que votre application plante violemment.

Modifier les exemples du début de la précédente section pour libérer votre mémoire.

Quand utiliser les pointeurs

Les différences entre un pointeur et une référence peuvent devenir déroutantes. Tout au long de ce cours, vous allez les revoir et tout deviendra plus clair à force de les utiliser. Le tableau qui suit compare les principaux points qui différencient les références des pointeurs :

	Pointeur	Référence
1	Contient une adresse	Est l'alias d'une variable
2	Peut être initialisé à tout moment.	Doit être initialisée à la définition
3	Peut changer l'adresse	Ne peut changer d'objet
4	Peut être utilisé dans un tableau	Ne peut être utilisée dans un tableau
5	Doit utiliser un opérateur (*) pour obtenir la valeur à l'adresse	Aucun opérateur n'est nécessaire
6	Peut être nul	Ne peut être nulle

Quiz

1) Complétez le tableau suivant pour chaque instruction du programme ci-dessous :

```
1     int A = 1;
2     int B = 2;
3     int C = 3;
4     int *P1 = nullptr;
5     int *P2 = nullptr;
6
7     P1 = &A;
8     P2 = &C;
9     *P1 = (*P2)++;
10    P1 = P2;
11    P2 = &B;
12    *P1 -= *P2;
13    ++*P2;
14    *P1 *= *P2;
15    A = ++*P2 * *P1;
16    P1 = &A;
17    *P2 = *P1 /= *P2;
```

Lignes	A	B	C	P1	P2
7	1	2	3	&A	nullptr
8	1	2	3	&A	&C
9	3	2	3	&A	&C
10	3	2	4	&C	&C
11	3	2	4	&C	&B
12	3	2	2	&C	&B
13	3	3	2	&C	&B
14	3	3	6	&C	&B
15	24	4	6	&C	&B
16	24	4	6	&A	&B
17	6	6	6	&A	&B

2) Soit **P**, un pointeur qui 'pointe' sur un tableau **A** :

```
int A[] = { 12, 23, 34, 45, 56, 67, 78, 89, 90 };
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions :

a) *P + 2 =
b) *(P + 2) =
c) &A[4] - 3 =
d) A + 3 =
e) P + (*P - 10) =
f) *(P + *(P + 8) - A[7]) =

3) Implémentez la fonction Swap qui a comme paramètre deux pointeurs vers les entiers a et b de sorte à ce que le programme échange le contenu des deux variables.

```
#include<iostream>
using namespace std;

void Swap(int *pA, int *pB)
{
    // ??
}

int main()
{
    int a = 10;
    int b = 20;

    // Avant Swap: affiche 10, 20
    cout << a << ", " << b << endl;

    // Swap ici

    // Après Swap: affiche 20, 10
    cout << a << ", " << b << endl;
    return 0;
}
```

4) Écrire une fonction qui a comme paramètre un tableau d'entiers, la taille du tableau, et 2 pointeurs vers des entiers min et max. La fonction doit renvoyer dans les entiers pointés par min et max, respectivement les plus petits et les plus grands entiers du tableau.

```
#include<iostream>
using namespace std;

void Minmax(int *p, int n, int *pmin, int *pmax)
{
    // ??
}

int main()
{
    int tn[] = { 12, 23, 36, 5, 46, 9, 25 };
    int min, max;

    // Minmax ici

    cout << "Min, Max : " << min << " " << max << endl;
    return 0;
}
```

5) Complétez le code suivant pour faire un bubble sort.

```
void Swap(int* val1, int* val2)
{
    int temp = *val1;
    *val1 = *val2;
    *val2 = temp;
}

void PrintTable(int* table, int size)
{
    // Afficher le contenu du tableau
}

void Sort(int* table, int size)
{
    // Trier le tableau
}

int main()
{
    int table[] = { 4, 6, 3, 7, 9, 2, 1, 10, 5, 8 };
    PrintTable(table, 10);
    Sort(table, 10);
    PrintTable(table, 10);

    return 0;
}
```