Cette question devrait être faite sans utiliser un compilateur.

Pour chaque case de cette question, indiquez le type qu'il faut mettre devant le nom de variable dans la déclaration pour que l'affectation soit correcte et qu'elle ne change pas le type de la valeur (ex. : bool x = 2; est une affectation correcte mais change le 2 d'un entier à un booléen, donc pas accepté comme réponse). Aucune des réponses n'est "const" ni une référence à quelque chose qui pouvait être affecté par copie.
Indiquez 'X' comme type si l'expression ne peut pas être correcte.

Réponse : (régime de pénalités : 0 %)

```
// Soient les déclarations:
struct B { int a; char b; short** c; };
struct A { B** a; B b; string c; A* d; };
struct C { double a; float* b; A** c; };
A* b(int x);
A a;
C c;

// Indiquez les bons types devant les variables suivantes:
```

char*	d = new char;
A	e = *(b(3));
B*	f = a.d->a[0];
char*	g = &(a.a[0]->b);
B	h = b(1)->d->b;
A	i = **(c.c);
A*	j = b(5);
float	k = c.b[0];
X	m = a.a->c;
B	n = a.d->a[0][0];

Soient les classes

```
1 class PolyBook {
2 public:
3     // 2
4 private:
5     // 1
6 }
7
8 class Etudiant {
9     // ...
10 }
```

Identifier le constructeur et l'attribut de la classe PolyBook.

Si **PolyBook** est une agrégation par pointeur de **Etudiant**,	1. Etudiant *etud_; 2. PolyBook(Etudiant* etud) : etud_(etud) {}
Si **PolyBook** est composé directement de **Etudiant**,	1. Etudiant etud_; 2. PolyBook(const Etudiant& etud) : etud_(etud) {}
Si **PolyBook** est une agrégation par référence de **Etudiant**,	1. Etudiant &etud_; 2. PolyBook(Etudiant& etud) : etud_(etud) {}
Si **PolyBook** est composé par pointeur de **Etudiant**,	1. Etudiant *etud_; 2. PolyBook(Etudiant* etud) : etud_(nullptr) { etud_ = new Etudiant (etud); }
Si **PolyBook** est composé par référence de **Etudiant**,	Aucune solution

La réponse correcte est :

Si **PolyBook** est une agrégation par pointeur de **Etudiant**,

→ 1. Etudiant *etud_; 2. PolyBook(Etudiant* etud) : etud_(etud) {},

Si **PolyBook** est composé directement de **Etudiant**,

→ 1. Etudiant etud_; 2. PolyBook(const Etudiant& etud) : etud_(etud) {},

Si **PolyBook** est une agrégation par référence de **Etudiant**,

→ 1. Etudiant &etud_; 2. PolyBook(Etudiant& etud) : etud_(etud) {},

Si **PolyBook** est composé par pointeur de **Etudiant**,

→ 1. Etudiant *etud_; 2. PolyBook(const Etudiant& etud) : etud_(nullptr) { etud_ = new Etudiant(etud); },

Si **PolyBook** est composé par référence de **Etudiant**,

→ Aucune solution

Soit la classe Piece ci-dessous. On aimerait pouvoir créer de nouveaux objets à partir d'une pièce déjà existante.

La copie d'un pièce doit respecter ces règles:

- Un objet copié doit avoir son identifiant (id_) incrémenté de 1 par rapport à l'original.
- Un objet copié doit avoir les même fabriquants que l'objet de base. Cependant ajouter un fabriquant après la copie ne doit impacter que l'objet concerné.
- Tous les autres attributs doivent être copiés tels qu'ils sont.

Vous faites vos définitions à l'extérieur de la classe et vous ne pouvez pas changer les signatures des méthodes ou en ajouter de nouvelles par rapport aux déclarations dans la classe ci-dessous.

```
1  class Piece
2  {
3  public:
4          Piece(const string& nom, const unsigned int id) :
5                  nom_(nom), id_(id)
6          {
7                  maxFab_    = 10;
8                  nombreFab_ = 0;
9
10                 fabriquants_ = new string[maxFab_];
11         }
12
13         Piece(const Piece& p);
14
15         ~Piece()
16         {
17                 delete[] fabriquants_;
18         }
19
20         void ajouterFab(const string& fab);
21
22         string getNom() { return nom_; }
23         unsigned getId() { return id_; }
24         unsigned getPrix() { return prix_; }
25         unsigned getNombreFab() { return nombreFab_; }
26         unsigned getNombreFabMax() { return maxFab_; }
27         string* getFabriquants() { return fabriquants_; }
28
29 private:
30         string   nom_;
31         unsigned id_;
32         double   prix_;
33
34         unsigned maxFab_;
35         unsigned nombreFab_;
36
37         string* fabriquants_;
38 };
```

Test	Résultat

`Piece p6("roue", 0);`
`Piece p7(p6);`
 	
`cout << (p7.getId() == p6.getId() + 1);`
`cout << (p7.getPrix() == p6.getPrix());`
`cout << (p7.getNom() == p6.getNom());`
`cout << (p7.getNombreFab() == p6.getNombreFab());`
`cout << (p7.getNombreFabMax() == p6.getNombreFabMax());`
	11111

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
1  Piece::Piece(const Piece& p): nom_(p.nom_), id_(p.id_ + 1), prix_(p.prix_),
2                                maxFab_(p.maxFab_), nombreFab_(p.nombreFab_)
3  {
4      fabriquants_ = new string[maxFab_];
5      for (auto i : range(nombreFab_))
6      {
7          fabriquants_[i]=p.fabriquants_[i];
8      }
9  }
```

- Modifiez le code déjà écrit afin de rendre la classe Point générique (ses attributs seront d'un type T).
- Vous devez aussi mettre la définition du constructeur à l'extérieur de la classe plutôt que dans la classe comme il l'est actuellement (cette partie de la question n'est pas testée par les tests CodeRunner).
- Ne changez pas les noms des attributs et laissez-les public.

Nous voulons qu'un code comme le suivant ait deux points dont les types des attributs sont différents (on compile en C++17):

```
Point pointInt(3, 4);
Point pointDouble(3.5, 4.5);
```

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
1  template <typename T>
2  class Point
3  {
4  public:
5      Point(T i, T j);
6
7      T i_;
8      T j_;
9  };
10
11 template <typename T>
12 Point<T>::Point(T i, T j): i_(i), j_(j) {}
```