Support de cours

Cours:

Initiation à la programmation (en C++)

Vidéo:

Pointeurs intelligents (partie 1)

Concepts (extraits des sous-titres générés automatiquement) :

Deuxième unique. New unique. Types de pointeurs. Variable bb. Vocab de smart pointers. Adresse d'un objet. Pointeurs de type unique. Zone mémoire. Pointer de façon unique. Pointeurs intelligents. Restitution de la mémoire. Notion de personne. Grande rigueur. Type unique. Pointeurs.

vers la recherche de séquences vidéo
(dans Initiation à la programmation (en C++).)

vers la vidéo

EPFL

Pointeurs «intelligents» (C++11)
(Partie 1)

Initiation à la programmation (C++)

Vincent Lepetit, Jean-Cédric Chappelier et Jamila Sam

...

notes

résumé

0m 0s

Les pointeurs dont nous venons de détailler l'usage jusqu'ici sont les bons

notes

résumé

0m 1s

C++11 Pointeurs intelligents (1/2)

EPFL

Pour faciliter la gestion de l'allocation dynamique de mémoire
et éviter l'oubli des `delete`,
C++11 introduit la notion de
« pointeur intelligent » (*smart pointer*)
(dans la bibliothèque `memory`)

☞ Ces pointeurs font leur propre `delete` au moment opportun (*garbage collecting*)

vieux pointeurs « à la C », qui sont sans doute les plus polyvalents, les plus puissants, mais aussi les plus dangereux. En effet, nous avons vu que leur utilisation nécessite une grande rigueur. Tout ce qui est alloué par le programmeur comme mémoire pointée par ces pointeurs doit être systématiquement désalloué par le programmeur. Et cette grande rigueur est parfois difficile à respecter de façon systématique. Si l'on veut allouer de la mémoire dynamiquement, on peut depuis C++11 avoir recours à des types de pointeurs un petit peu plus confortables d'utilisation, qui nécessitent moins de précautions de la part du programmeur, il s'agit des pointeurs intelligents que je vais brièvement vous présenter maintenant. Les pointeurs intelligents que vous pourrez trouver dans la littérature anglophone sous le vocab de smart pointers, sont utilisables en incluant la bibliothèque memory. Ils ont la spécificité de faire leur propre delete au moment opportun. Ça veut dire qu'ils gèrent la restitution de la mémoire associée par leur propres moyens. Le programmeur n'a plus à se préoccuper de la restitution de la mémoire, et donc on va éviter par ce biais-là l'oubli de delete qui cause de nombreux soucis avec les pointeurs « à la C ». La gestion de la mémoire par les pointeurs intelligents fait appel à des

notes

résumé

0m 5s

C++11 unique_ptr

EPFL

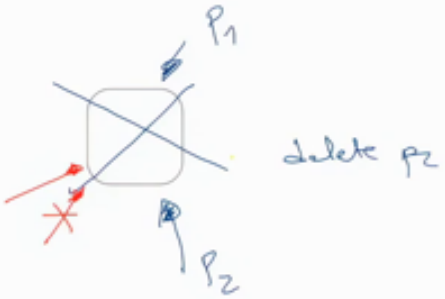
Les `unique_ptr` pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire »)

☞ évite les confusions

⟹ Ne peut être copié

mais peut être « déplacé », « transmis » plus loin

unique_ptr<int> p2 (new i



notes

résumé

1m 13s

C++11 unique_ptr

EPFL

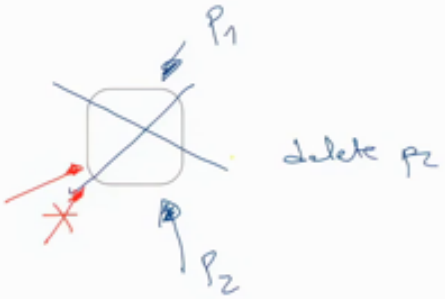
Les `unique_ptr` pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire »)

☞ évite les confusions

⟹ Ne peut être copié

mais peut être « déplacé », « transmis » plus loin

unique_ptr<int> p2 (new i



à l'heure n'est désormais plus possible. Cela va éviter les confusions justement que nous avons décrites. Et il en découle une conséquence assez logique, vu qu'il ne peut y avoir qu'un seul et unique unique_ptr pointant sur la zone mémoire, et bien les unique_ptr ne peuvent pas être copiés. Concrètement, en C++11, si je veux déclarer, initialiser un unique_ptr, et bien, je vais avoir recours à la syntaxe suivante. Donc, unique_ptr. Je dois indiquer sur quel type de zone mémoire est capable de pointer le unique_ptr. Donc ici je déclare un new unique_ptr capable de pointer sur un entier, et évidemment je lui donne un nom. Pour l'initialiser, et bien, c'est un pointeur donc c'est une adresse et donc

notes

résumé

C++11 unique_ptr, exemples

EPFL

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

je peux l'initialiser grâce à l'adresse d'un objet dynamiquement alloué. Ce que je sais faire pour un entier par exemple, par le biais de ce genre de tournure. Maintenant si j'essaie de créer une copie de ce unique_ptr, donc en déclarant un pointeur p1 et en lui copiant la valeur de p2, et bien ceci en C++11 n'est pas licite. Ceci pour garantir qu'il n'y ait pas justement deux pointeurs de type unique_ptr pointant sur la même zone mémoire dynamiquement allouée. Les unique_ptr ne peuvent donc pas être copiés, mais en revanche ils peuvent être déplacés, c'est-à-dire transmis plus loin. Le C++11 met en effet en oeuvre ce que l'on appelle la move semantic, qui est une notion qui dépasse un peu le cadre de ce cours, mais que je vais brièvement esquisser par la suite.

notes

résumé

3m 49s

C++11 unique_ptr, exemples

EPFL

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

Donc par rapport à la syntaxe, pour bien établir la syntaxe, un petit exemple de déclaration initialisation d'un unique_ptr. Donc ici, je déclare un unique_ptr px, qui est capable de pointer sur un entier. Un unique_ptr est un pointeur comme un autre dans le sens où il va contenir une adresse, et ici il contient l'adresse d'une zone dynamiquement allouée d'un entier dynamiquement alloué. Pour le reste, et bien, les unique_ptr vont s'employer comme les autres pointeurs

notes

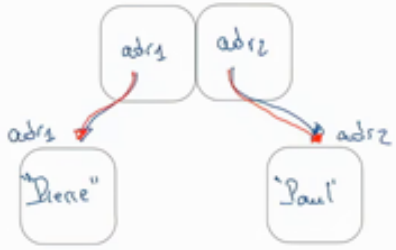
résumé

4m 38s

C++11 unique_ptr, exemples

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

```
vector<unique_ptr<string>> noms;
noms.push_back(unique_ptr<string>(new string("Pierre")));
noms.push_back(unique_ptr<string>(new string("Paul")));
```



qu'on a vu jusqu'ici. Par exemple, on peut accéder au contenu pointé par le unique_ptr au moyen de l'étoile comme nous avions l'habitude de le faire avec les pointeurs « à la C ». On peut bien sûr parfaitement déclarer un vecteur, un tableau dynamique de unique_ptr. Ici, vous avez l'exemple d'un tableau dynamique qui s'appelle noms et dont les cases sont des unique_ptr capables de pointer sur des zones mémoires de type chaîne de caractères. Au moment de l'initialisation de ce vecteur, nous avons recours à de l'allocation dynamique pour aboutir à la situation suivante. Donc, notre tableau noms contient deux cases, deux cellules, dont chacune contient l'adresse d'un objet dynamiquement alloué. Donc, la première case contient l'adresse d'une zone dynamiquement allouée contenant la chaîne de caractères Pierre, et la deuxième case du tableau contient l'adresse d'une zone dynamiquement allouée contenant la chaîne de caractères Paul. Donc ici, chaque case du tableau va pointer de façon unique vers une chaîne de caractères grâce à la notion de unique_ptr.

notes

résumé

5m 1s

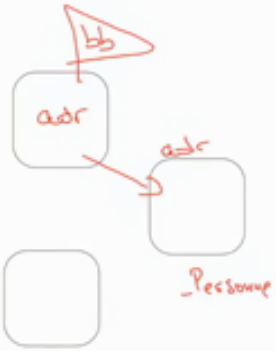
C++11 unique_ptr, exemples

EPFL

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

```
vector<unique_ptr<string>> noms;
noms.push_back(unique_ptr<string>(new string("Pierre")));
noms.push_back(unique_ptr<string>(new string("Paul")));
```

```
unique_ptr<Personne> naissance(string nom) {
  unique_ptr<Personne> bb(new Personne);
  // .. initialise le contenu pointé par bb
  return bb;
}
// ...
unique_ptr<Personne> adresse_quidam( naissance("Pierre") );
```



Dernier exemple pour illustrer notamment la notion de déplacement par rapport à celui de copie. Donc ici, on imagine que dans un programme on dispose d'un type Personne qui est par exemple une structure représentant la notion de personne et dont l'un des champs serait par exemple le nom de la personne de type chaîne de caractères. Donc, on veut au travers de la fonction naissance créer un nouvel objet de type Personne et retourner l'adresse de cet objet stocké dans un unique_ptr. Donc la ligne de code qui est en charge de créer ce nouvel objet est la suivante. Et la création de l'objet se fait par une allocation dynamique. On crée dynamiquement un objet de type Personne. L'adresse de cet objet est stockée dans la variable bb qui est de type unique_ptr sur une personne. Donc, on imagine ici que par la suite on fasse les initialisations nécessaires sur la variable bb, notamment en mettant le nom de la structure de données avec le nom transmis à la fonction et puis on transmet au monde extérieur l'adresse de l'objet en question que nous venons de créer. Donc, nous nous trouvons dans cette situation finale. Nous avons une variable bb qui contient l'adresse d'un objet dynamiquement alloué qui serait de type Personne, donc ici le type stocké est le type Personne. Plaçons-nous maintenant dans le contexte de l'utilisation de la fonction naissance.

notes

résumé

6m 13s

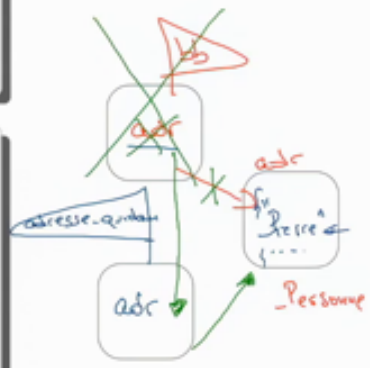
C++11 unique_ptr, exemples

EPFL

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

```
vector<unique_ptr<string>> noms;
noms.push_back(unique_ptr<string>(new string("Pierre")));
noms.push_back(unique_ptr<string>(new string("Paul")));
```

```
unique_ptr<Personne> naissance(string nom) {
  unique_ptr<Personne> bb(new Personne);
  // .. initialise le contenu pointé par bb
  return bb;
}
// ...
unique_ptr<Personne> adresse_quidam( naissance("Pierre") );
```



Donc ici, je déclare une variable adresse_quidam qui est de type unique_ptr sur Personne et je l'initialise au moyen du unique_ptr retourné par l'appel à la fonction naissance. Donc, on pourrait imaginer ici que l'on soit dans cette situation, donc la variable bb contient l'adresse d'une personne, de la personne dont le nom est Pierre, et ici, on pourrait imaginer que en réalité ce qu'on a fait ici c'est une copie de la valeur de bb dans une autre variable de type unique_ptr qui s'appellerai adresse_quidam. Donc ceci peut être parfaitement pris pour une copie, mais en réalité ce qui se passe c'est un déplacement, c'est-à-dire que ici la valeur qui était dans bb est déplacée vers la nouvelle variable, adresse_quidam, et donc la variable bb n'est plus utilisable en soi telle qu'elle est, ce qui veut dire que l'on a toujours qu'un seul objet pointant vers l'objet personne, et donc on n'a pas cassé le fait qu'un unique_ptr

notes

résumé

7m 37s

C++11 unique_ptr

EPFL

Les `unique_ptr` pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire »)

☞ évite les confusions

$\Longrightarrow$ Ne peut être copié
mais peut être « déplacé », « transmis » plus loin

Note : si l'on veut libérer un `unique_ptr` avant le garbarge collector (c'est-à-dire faire le `delete` nous-même), on peut utiliser la fonction spécifique `reset()` :

```
ptr.reset()
```

Remet en plus `ptr` à `nullptr`.

pointe vers un objet unique. Donc ici, on a implicitement utilisé ce qu'on appelle en C++11 la move semantic. Ce qui se passe dans ce contexte précis n'est pas une copie, mais c'est un déplacement. C++11 sait en effet que à ce stade-là nous utilisons un déplacement et non pas une copie de par le fait que ce que nous affectons à l'objet de type unique_ptr est en réalité une donnée transitoire issue d'un appel de fonction. C'est cette donnée transitoire issue de cet appel de fonction qui est affectée au unique_ptr, et dans ce cas-ci, et bien, on sait que il y a un déplacement qui a été mis en oeuvre et non pas une copie.

notes

résumé

8m 49s

Donc, les unique_ptr sont des pointeurs intelligents, ce qui veut dire que la mémoire qui y est associée est restituée automatiquement, on n'a pas besoin de s'en préoccuper. Si par contre, on voulait libérer cette mémoire de façon explicite, c'est aussi possible de le faire, et donc de faire le delete nous-même, et donc à ce moment-là, on utilise une fonctionnalité spécifique à ces pointeurs, la fonctionnalité reset. Et donc à ce moment-là, nous faisons l'appel ptr.reset() qui aura pour conséquence de libérer la mémoire associée à ce pointeur et d'affecter la valeur nullptr au pointeur intelligent ptr. pointeur intelligent ptr.

notes

résumé

9m 32s