الأكاديمية العربية الدولية
Arab International Academy

لغة البرمجة C++

C++ Programming Language

عبدالقادر العبدالله

كلية العلوم – تخصص البرمجة

الأكاديمية العربية الدولية
Arab International Academy

المحاور

- دوال داخل الكائنات: الأساليب في C++
- تعريف الأساليب داخل وخارج الفئة
- معاملات الدوال وأنواع الإرجاع
- المُنشئات: الأنواع والاستدعاء
- التحكم في الوصول: عام، خاص، محمي
- التغليف Encapsulation وتطبيقه عبر get/set
- الوراثة: المفاهيم والبناء العملي
- الوراثة متعددة المستويات والمتعددة
- تعدد الأشكال Polymorphism وتجاوز الدوال
- أمثلة عملية شاملة على الوراثة وتعدد الأشكال

المخرجات المتوقعة من الدرس

- فهم كيفية تعريف الأساليب داخل وخارج الفئة في .C++
- استخدام المُنشئات لتهيئة الكائنات وتحديد أنواعها (افتراضي، ذو معاملات، نسخة).
- تطبيق مبدأ التغليف باستخدام محددات الوصول والدوال العامة مثل get و .set
- إنشاء برامج تستخدم الوراثة وإعادة استخدام الكود بين الفئات.
- التمييز بين أنواع الوراثة: متعددة المستويات والمتعددة.
- استخدام مبدأ تعدد الأشكال لكتابة دوال قابلة للتخصيص في الفئات المشتقة.
- تحليل وتنفيذ كود يعتمد على مبادئ OOP في تصميم الكائنات والواجهات.

فهم أساليب الفئات: دوال داخل الكائنات

في برمجة C++، تُعرف الأساليب Methods بأنها دوال تنتمي إلى الفئة Class تُحدد هذه الدوال السلوكيات أو الإجراءات التي يمكن أن تؤديها الكائنات Objects التي تُنشأ من تلك الفئة. بعبارة أخرى، إذا كانت سمات الفئة attributes تمثل البيانات أو الخصائص (ما هو الكائن)، فإن الأساليب تمثل الإجراءات أو الأفعال التي يمكن للكائن القيام بها (ماذا يفعل الكائن). تعمل هذه الأساليب على البيانات الداخلية للكائن، مما يسمح بتفاعل منظم مع حالته.

يُعد هذا المفهوم أساسيًا في البرمجة الشيئية (OOP)، حيث يضع حجر الزاوية لفهم كيفية دمج البيانات والسلوكيات معًا داخل كيان واحد، وهو الفئة. هذا الدمج هو ما يمكّن من إنشاء كائنات ذاتية الاحتواء يمكنها إدارة بياناتها الخاصة وتوفير واجهة محددة للتفاعل معها، مما يعزز تنظيم التعليمات البرمجية ويجعلها أكثر قابلية للفهم والصيانة.

تعريف الأساليب: داخل تعريف الفئة

تُعد أبسط طريقة لتعريف الأساليب في C++ هي تضمين تعريف الدالة بالكامل داخل تعريف الفئة نفسها. هذا الأسلوب مناسب بشكل خاص للدوال الصغيرة والمبسطة التي لا تتطلب الكثير من التعليمات البرمجية، حيث يوفر وضوحًا فوريًا للوظيفة داخل سياق الفئة. ومع ذلك، بالنسبة للمشاريع الأكبر أو الأساليب الأكثر تعقيدًا، قد يؤدي هذا النهج إلى تضخم تعريفات الفئات، مما يؤثر على قابلية القراءة ويجعل من الصعب فهم الهيكل العام للفئة بسرعة.

في المثال التالي، يتم تعريف الدالة myMethod() مباشرة داخل الفئة MyClass. عند إنشاء كائن من MyClass واستدعاء myMethod()، يتم تنفيذ التعليمات البرمجية الموجودة داخل الدالة.

تعريف الأساليب: داخل تعريف الفئة

يُظهر هذا المثال كيف يتم دمج تعريف الدالة مباشرة مع تعريف الفئة، مما يجعله خيارًا مباشرًا للتطبيقات البسيطة. ومع ذلك، فإن هذا الأسلوب يمهد الطريق لفهم الحاجة إلى طرق بديلة لتعريف الأساليب، خاصة عندما تصبح الدوال أكثر تعقيدًا أو عندما يكون الهدف هو فصل الواجهة عن التنفيذ لتحسين تنظيم التعليمات البرمجية.

```
class MyClass {
public:
  void myMethod() { // الدالة معرفة داخل الفئة
    cout << "Hello from inside the method!";
  }
};
int main() {
  MyClass myObj;
  myObj.myMethod(); // استدعاء الدالة
  return 0;
}
```

تعريف الأساليب: خارج تعريف الفئة

تحقيق تنظيم أفضل للتعليمات البرمجية، خاصة في المشاريع الكبيرة، يمكن تعريف الأساليب خارج تعريف الفئة. يتضمن هذا النهج الإعلان عن الدالة داخل الفئة أولاً، ثم تعريفها بشكل منفصل خارج الفئة باستخدام عامل حل النطاق Scope Resolution Operator (::) تُعد هذه الممارسة أساسية في C++ل أنها تسمح بفصل واجهة الفئة (ما تفعله الفئة) عن تنفيذها (كيف تفعله)، مما يحسن من قابلية القراءة والصيانة.

يُشير عامل (::) إلى أن الدالة myMethod تنتمي إلى MyClass، حتى عندما يتم تعريفها خارج نطاق الفئة مباشرة.

هذا الفصل بين الإعلان والتنفيذ هو ممارسة شائعة في تطوير C++الاحترافي، حيث غالبًا ما توضع الإعلانات في ملفات الرؤوس .(h) وتوضع التعريفات في ملفات المصدر (cpp) هذا يقلل من أوقات التجميع ويعزز نمطية التعليمات البرمجية، مما يجعل إدارة المشاريع الكبيرة أكثر سهولة.

تعريف الأساليب: خارج تعريف الفئة

يُظهر هذا المثال مرونة C++في تنظيم التعليمات البرمجية، حيث يمكن للمطورين اختيار الأسلوب الأنسب بناءً على حجم المشروع وتعقيد الأساليب.

يُعد هذا الفصل ضروريًا لبناء أنظمة برمجية قابلة للتوسع والصيانة، حيث يمكن للمطورين العمل على أجزاء مختلفة من التعليمات البرمجية دون التأثير على الأجزاء الأخرى بشكل مباشر.

```
class MyClass {
public:
  void myMethod(); // إعلان الدالة داخل الفئة
};
// تعريف الدالة خارج الفئة
void MyClass::myMethod() {
  cout << "Hello from outside the method!";
}
int main() {
  MyClass myObj;
  myObj.myMethod();
  return 0;
}
```

معاملات الأساليب وأنواع الإرجاع

مامًا مثل الدوال العادية، يمكن للأساليب في C++أن تقبل معاملات Parameters وتُرجع قيمًا Return Values تُتيح هذه المرونة للأساليب أداء عمليات ديناميكية بناءً على المدخلات الخارجية، مما يجعل الكائنات تفاعلية ووظيفية بشكل أكبر. تسمح المعاملات للأساليب بتلقي البيانات من خارج الكائن، بينما تُتيح أنواع الإرجاع للأساليب إرسال نتائج عملياتها إلى التعليمات البرمجية التي استدعتها.

إن قدرة الأساليب على معالجة المدخلات وإنتاج المخرجات أمر بالغ الأهمية لتمكين الكائنات من أداء مهام مفيدة. على سبيل المثال، يمكن لدالة add في فئة Calculator أن تقبل رقمين كمعاملات وتُرجع مجموعهما، مما يجعل الكائن قادرًا على إجراء عمليات حسابية بناءً على بيانات محددة. هذا التفاعل بين الأساليب والبيانات، سواء كانت داخلية أو خارجية، هو ما يجعل الكائنات مكونات برمجية قوية.

```
class Calculator {
public:
  int add(int a, int b) { // دالة بمعاملات ونوع إرجاع
    return a + b;
  }
};
int main() {
  Calculator calc;
  int result = calc.add(5, 3);
  cout << "Sum: " << result; // الإخراج: Sum: 8
  return 0;
}
```

يُوضح هذا المثال كيف تُستخدم المعاملات a و b لتمرير قيم إلى الدالة add ، وكيف تُرجع الدالة قيمة int تمثل المجموع. تُعد هذه القدرة على التعامل مع المدخلات والمخرجات أساسية لتصميم كائنات يمكنها التفاعل بفعالية مع بقية النظام البرمجي .

المُنشئات في C++ (C++ Constructors)

المُنشئ Constructor في C++ هو دالة خاصة تُستدعى تلقائيًا عند إنشاء كائن من فئة ما.

وظيفته الأساسية هي تهيئة سمات الكائن (attributes)، مما يضمن أن الكائن في حالة صالحة وجاهز للاستخدام فور إنشائه.

هذا يمنع الأخطاء المحتملة التي قد تنجم عن استخدام بيانات غير مهيأة، ويُعد جزءًا حيويًا من دورة حياة الكائن في C++.

يتميز المُنشئ بعدة خصائص فريدة تميزه عن الدوال العادية:

يجب أن يحمل نفس اسم الفئة التي ينتمي إليها ، ويجب أن يكون دائمًا عامًا (public) ، ولا يُرجع أي قيمة على الإطلاق، ولا حتى (void) .

هذه الخصائص تجعل من السهل التعرف على المُنشئات وتفهم دورها الفوري عند إنشاء الكائنات.

يُعد المُنشئ بمثابة "شهادة ميلاد" للكائن، حيث يضمن تكوينه الصحيح منذ اللحظة الأولى لوجوده.

Q1

1. صح أم خطأ: يتم تعريف الدوال دائمًا داخل الفئة في C++ فقط.

2. صح أم خطأ: يمكن للأساليب أن تُعيد قيمة باستخدام return.

3. ما الكلمة التي تُستخدم لفصل اسم الكلاس عن اسم الدالة عند تعريف الأساليب خارج الفئة؟
   أ) ::
   ب) ->
   ج) .
   د) *::

4. صح أم خطأ: من الأفضل دائمًا تعريف كل الأساليب داخل الفئة لسهولة الصيانة.

5. صح أم خطأ: الأساليب يمكن أن تستقبل معاملات مثل الدوال العادية.

الأجوبة

1. خطأ
2. صح
3. أ
4. خطأ
5. صح

بناء المُنشئ الأساسي والاستدعاء التلقائي

يتسم بناء المُنشئ الأساسي في C++ بالبساطة، حيث يتم استخدام اسم الفئة متبوعًا بقوسين فارغين () ثم كتلة التعليمات البرمجية الخاصة بالمُنشئ. إن أهم ما يميز المُنشئات هو استدعاؤها التلقائي؛ فبمجرد الإعلان عن كائن من الفئة، يتم استدعاء المُنشئ الخاص بها بشكل فوري دون الحاجة إلى استدعاء صريح كما هو الحال مع الدوال العادية. هذا السلوك التلقائي يُعد ميزة أساسية في C++، حيث يضمن تهيئة الكائنات بشكل صحيح بمجرد إنشائها، مما يقلل من فرص الأخطاء الناتجة عن بيانات غير مهيأة. قد يتوقع المبتدئون استدعاء المُنشئ يدويًا، ولكن فهم هذا الاستدعاء التلقائي هو نقطة تحول في إدراك كيفية عمل الكائنات في (C++)

```
class MyClass {
public:
  MyClass() { // المُنشئ
    cout << "Object created and constructor called!";
  }
};
int main() {
  MyClass myObj; // هذا يستدعي المُنشئ تلقائيًا
  return 0;
}
```

يوضح هذا المثال البسيط كيف أن مجرد إنشاء الكائن myObj يؤدي إلى تنفيذ التعليمات البرمجية داخل المُنشئ MyClass(). هذا التفاعل السلس بين إنشاء الكائن وتهيئة حالته هو جوهر فائدة المُنشئات في C++.

المُنشئات ذات المعاملات: تهيئة السمات

تُتيح المُنشئات في C++ قبول معاملات Parameters تمامًا مثل الدوال العادية، مما يوفر مرونة كبيرة في تهيئة سمات الكائن بقيم أولية مختلفة. تُعد هذه القدرة ضرورية لإنشاء كائنات متنوعة من نفس مخطط الفئة، حيث يمكن لكل كائن أن يبدأ بحالة فريدة خاصة به. على سبيل المثال، عند إنشاء كائن Car، يمكن تمرير قيم مثل BMW للعلامة التجارية، وX5 لموديل، و1999 للسنة، مباشرة إلى المُنشئ. داخل المُنشئ ذي المعاملات، تُستخدم القيم المُمررة لتعيين السمات المقابلة للكائن، مما يضمن أن الكائن يُنشأ بالبيانات المطلوبة. تُعد هذه المرونة حاسمة لبناء أنظمة معقدة تتطلب كائنات ذات حالات أولية مخصصة، وتُظهر كيف يمكن للمُنشئات أن تتجاوز مجرد التهيئة الأساسية لتُصبح أدوات قوية لإدارة بيانات الكائن عند الإنشاء.

```
class Car {
public:
  string brand;
  string model;
  int year;
  Car(string x, string y, int z) { // مُنشئ ذو معاملات
    brand = x;
    model = y;
    year = z;
  }
};
int main() {
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n"
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n"
  return 0;
}
```

يُظهر هذا المثال كيف يتم استخدام المُنشئ Car(string x, string y, int z) لتهيئة كائنين مختلفين من الفئة Car ، كل منهما بسماته الفريدة. هذا يُبرز كيف تُمكن المُنشئات ذات المعاملات من إنشاء كائنات ذات حالات أولية متنوعة، مما يزيد من قابلية الفئة للاستخدام في سيناريوهات متعددة

تعريف المُنشئات خارج الفئة

تمامًا مثل الدوال العادية، يمكن الإعلان عن المُنشئات داخل تعريف الفئة ثم تعريفها بشكل منفصل خارج الفئة. تُعد هذه الممارسة مفيدة للحفاظ على تعريف الفئة نظيفًا وموجزًا، خاصة عندما تكون المُنشئات معقدة أو تحتوي على تعليمات برمجية كبيرة. يُستخدم عامل حل النطاق :: لربط تعريف المُنشئ الخارجي بالفئة التي ينتمي إليها. يُعد فصل الإعلان عن التعريف ممارسة قياسية في تطوير C++ على نطاق واسع، حيث تُوضع الإعلانات عادةً في ملفات الرؤوس (h) وتُوضع التعريفات في ملفات المصدر (cpp) هذا يُساهم في تحسين نمطية التعليمات البرمجية، ويقلل من تبعيات التجميع، ويُسهل إدارة المشاريع المعقدة. يُعكس هذا النمط الاتساق في تصميم اللغة، حيث تُعامل المُنشئات، على الرغم من طبيعتها الخاصة، بطريقة مشابهة للدوال العادية فيما يتعلق بالتنظيم.

```
class Car {
public:
  string brand;
  string model;
  int year;
  Car(string x, string y, int z); // إعلان المُنشئ
};
// تعريف المُنشئ خارج الفئة
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}
int main() {
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

يُظهر هذا المثال كيف يتم الإعلان عن المُنشئ Car داخل الفئة ثم يتم تعريفه خارجها باستخدام Car::Car(...) هذا الفصل يُعزز تنظيم التعليمات البرمجية ويُسهم في بناء تطبيقات C++ أكثر قابلية للصيانة والتوسع.

أنواع المُنشئات (أبعد من الأساسي)

تُقدم C++أنواعًا مختلفة من المُنشئات لتلبية سيناريوهات التهيئة المتنوعة، مما يُبرز تركيز اللغة على إدارة الموارد بدقة ودلالات الكائن. تُعد هذه الأنواع ضرورية لفهم كيفية تهيئة الكائنات في سياقات مختلفة، وتُشير إلى أن تهيئة الكائنات أكثر تعقيدًا من مجرد تعيين قيم أولية.

تشمل الأنواع الرئيسية للمُنشئات ما يلي:

- المُنشئ الافتراضي Default Constructor: هو مُنشئ لا يأخذ أي معاملات. إذا لم يتم تعريف أي مُنشئ في الفئة، فإن المترجم يقوم بتوفير مُنشئ افتراضي تلقائيًا. يمكن للمبرمج أيضًا تعريف مُنشئ افتراضي خاص به لتهيئة السمات بقيم محددة.
- المُنشئ ذو المعاملات Parameterized Constructor: يأخذ معاملات لتهيئة سمات الكائن بقيم محددة عند الإنشاء. تم تناول هذا النوع بالتفصيل في الشرائح السابقة.
- مُنشئ النسخ Copy Constructor: هو مُنشئ خاص يأخذ كائنًا من نفس الفئة كمعامل عادةً عن طريق مرجع ثابت (const ClassName) ويُستخدم لنسخ قيم سمات كائن موجود إلى كائن جديد. تُعد هذه الآلية حاسمة لضمان النسخ الصحيح للكائنات، خاصة تلك التي تدير موارد ديناميكية.
- المُنشئ الذي لا يفعل شيئًا Do Nothing Constructor: هو مُنشئ لا يحتوي على أي تعليمات برمجية بداخله. على الرغم من اسمه، فإنه لا يزال يُستدعى عند إنشاء الكائن، ولكنه لا يقوم بأي تهيئة صريحة.

أمثلة شاملة للمُنشئات

تُوضح الأمثلة التالية التطبيق العملي لأنواع مختلفة من المُنشئات في C++، مما يُسهم في ترسيخ فهم كيفية تهيئة الكائنات في سيناريوهات متنوعة. يُعد الانتقال من المُنشئات الافتراضية الضمنية إلى المُنشئات ذات المعاملات ومُنشئات النسخ الصريحة مؤشرًا على كيفية منح C++ للمبرمجين تحكمًا متزايدًا في إنشاء الكائنات وتهيئتها، مما يؤدي إلى تعليمات برمجية أكثر قوة وقابلية للتنبؤ.

مثال (المُنشئ الافتراضي - المُعرف من قبل المستخدم):

يُظهر هذا المثال مُنشئًا افتراضيًا مُعرفًا من قبل المستخدم

يقوم بتهيئة سمة val بقيمة 20.

```
class Calc {
    int val;
public:
    Calc() { // مُنشئ افتراضي مُعرف من قبل المستخدم
        val = 20;
    }
};
int main() {
    Calc c1; // يستدعي المُنشئ الافتراضي المُعرف من قبل المستخدم
    cout << c1.val; // 20 :الإخراج
    return 0;
}
```

مثال (مُنشئ النسخ - البناء)

يُوضح هذا المثال كيفية تعريف مُنشئ نسخ يقوم بنسخ سمات كائن موجود إلى كائن جديد. تُعد عملية التهيئة عبر مُنشئ النسخ، والتي تُعرف أيضًا بتهيئة النسخ، أمرًا بالغ الأهمية لضمان أن الكائنات المنسوخة تحتوي على بيانات صحيحة ومستقلة.

تُسلط هذه الأمثلة الضوء على كيفية استخدام المُنشئات المختلفة لضمان تهيئة الكائنات بشكل صحيح، سواء كانت تهيئة افتراضية، أو تهيئة بقيم محددة، أو تهيئة عن طريق نسخ كائن آخر.

```
class Calc {
    int a, b;
public:
    Calc(int x, int y) { a = x; b = y; } // مُنشئ ذو معاملات
    Calc(const Calc &obj) { // مُنشئ النسخ
        a = obj.a;
        b = obj.b;
    }
};
int main() {
    Calc C1(10, 20);
    Calc C2 = C1; // يستدعي مُنشئ النسخ
    Calc C3(C1);  // يستدعي مُنشئ النسخ أيضًا
    // كنسخ مستقلة من C1 يمكننا الآن استخدام C2 و C3
    return 0;
}
```

التحكم في الوصول: عام، خاص، ومحمي

تُعد مُحددات الوصول (Access Specifiers) في C++ كلمات مفتاحية تُستخدم للتحكم في كيفية الوصول إلى أعضاء الفئة (السمات والدوال) من خارج الفئة. تُعد هذه المُحددات أساسية لمفهوم إخفاء المعلومات والتغليف في البرمجة الشيئية، مما يُعزز نمطية التعليمات البرمجية ويمنع التعديلات الخارجية غير المقصودة للحالة الداخلية للكائن.

هناك ثلاثة مُحددات وصول رئيسية في (C++) :

- عام (public) : الأعضاء المُعلن عنها كـ publicيمكن الوصول إليها وتعديلها من أي مكان خارج الفئة. تُشبه هذه الأعضاء الباب الأمامي للمنزل، فهي متاحة للجميع.
- خاص (private) : الأعضاء المُعلن عنها كـ private لا يمكن الوصول إليها أو رؤيتها من خارج الفئة. تُشبه هذه الأعضاء الدرج المقفل، حيث لا يمكن لأي شخص من الخارج الوصول إليه.
- محمي (protected) : الأعضاء المُعلن عنها كـ protected لا يمكن الوصول إليها من خارج الفئة، ولكن يمكن الوصول إليها في الفئات الموروثة (الفئات الفرعية). تُشبه هذه الأعضاء غرفة مخصصة للعائلة فقط، حيث يمكن للأطفال (الفئات الفرعية) الدخول إليها، بينما لا يستطيع الآخرون ذلك.

بشكل افتراضي، إذا لم يتم تحديد مُحدد وصول، فإن جميع أعضاء الفئة تُعتبر (private) .

يُعد هذا التحكم في الرؤية أمرًا حيويًا لبناء تعليمات برمجية قوية وقابلة للصيانة من خلال تحديد واجهات واضحة وحماية البيانات الداخلية

أهمية السمات الخاصة

تُعد الممارسة الجيدة في برمجة C++هي الإعلان عن سمات الفئة (Class Attributes) كـ (private) قدر الإمكان.

هذا النهج لا يُقلل فقط من فرصة تلف التعليمات البرمجية بشكل عرضي، بل يُعد أيضًا مكونًا رئيسيًا لمفهوم التغليف (Encapsulation) من خلال جعل السمات خاصة، يتم إجبار التعليمات البرمجية الخارجية على التفاعل مع الكائن فقط من خلال واجهته العامة المُحددة (أي، من خلال الدوال العامة)، مما يؤدي إلى تحسين سلامة البيانات وتسهيل عملية تصحيح الأخطاء.

عندما تكون السمات خاصة، لا يمكن تعديلها مباشرة من خارج الفئة. بدلاً من ذلك، يجب أن يتم الوصول إليها أو تعديلها باستخدام دوال عامة (تُعرف غالبًا باسم دوال get و set داخل نفس الفئة. هذا يضمن أن أي تغييرات على البيانات تتم بطريقة مُتحكم بها، مما يسمح للمطورين بتضمين منطق التحقق من الصحة أو الآثار الجانبية (مثل التسجيل) عند الوصول إلى البيانات أو تعديلها.

هذا الفصل بين البيانات والوصول إليها يُعزز نمطية التعليمات البرمجية ويجعلها أكثر مرونة للتغيير في المستقبل

أمثلة توضيحية للتحكم في الوصول

تُقدم الأمثلة التالية توضيحًا عمليًا لكيفية تأثير مُحددات الوصول على إمكانية الوصول إلى أعضاء الفئة، مما يُسهم في ترسيخ فهم مفهوم حماية البيانات.

يُعد ظهور رسالة الخطأ (error: y is private) في المثال بمثابة دليل بصري قوي للمتعلمين، حيث يُظهر بشكل مباشر تأثير التحكم في الوصول ويُعزز مفهوم حماية البيانات.

مثال (عام مقابل خاص):يُوضح هذا المثال أن الأعضاء العامة (x) يمكن الوصول إليها وتعديلها من الدالة ()main، بينما لا يمكن الوصول إلى الأعضاء الخاصة (y)، مما يؤدي إلى خطأ.

```
class MyClass {
public:     // مُحدد وصول عام
  int x;    // سمة عامة
private:    // مُحدد وصول خاص
  int y;    // سمة خاصة
};
int main() {
  MyClass myObj;
  myObj.x = 25;  // (عام) مسموح
  // myObj.y = 50; // سيؤدي إلى خطأ - (خاص) ممنوع: error: y is private
  return 0;
}
```

مثال (خاص افتراضيًا)

يُبين هذا المثال أنه إذا لم يتم توفير مُحدد وصول، فإن أعضاء الفئة تُصبح خاصة افتراضيًا.

تُسلط هذه الأمثلة الضوء على الآثار المباشرة لاستخدام مُحددات الوصول، وتُعزز فهم كيفية تطبيق مبادئ حماية البيانات في C++.

```
class MyClass {
  int x;    // سمة خاصة (افتراضيًا)
  int y;    // سمة خاصة (افتراضيًا)
};
```

مقارنة مُحددات الوصول في C++

يُقدم هذا الجدول مقارنة موجزة بين مُحددات الوصول الثلاثة في C++، مُسلطًا الضوء على إمكانية الوصول إلى أعضاء الفئة من سياقات مختلفة. يُعد هذا الجدول أداة مرجعية قيمة للمتعلمين، حيث يُلخص الخصائص المميزة لكل مُحدد وصول ويُقدم تشبيهات من الحياة الواقعية لتعزيز الفهم

مُحدد الوصول	يمكن الوصول إليه من داخل الفئة	يمكن الوصول إليه من خارج الفئة	يمكن الوصول إليه من الفئة المشتقة	افتراضي لأعضاء الفئة	تشبيه
`public`	نعم	نعم	نعم	لا	الباب الأمامي
`private`	نعم	لا	لا	نعم	الدرج المقفل
`protected`	نعم	لا	نعم	لا	غرفة العائلة فقط

يُساعد هذا الجدول في فهم كيفية عمل مُحددات الوصول معًا لتوفير طبقات مختلفة من الحماية للبيانات داخل الفئات، مما يُسهم في تصميم تعليمات برمجية أكثر أمانًا وتنظيمًا.

التغليف: إخفاء البيانات الحساسة

التغليف (Encapsulation) هو مبدأ أساسي في البرمجة الشيئية يعني ضمان إخفاء البيانات "الحساسة" عن المستخدمين.

يتحقق هذا المبدأ بشكل أساسي عن طريق الإعلان عن متغيرات الفئة أو سماتها كـ private، مما يعني أنه لا يمكن الوصول إليها مباشرة من خارج الفئة.

يُعد هذا التحكم في الوصول أمرًا حيويًا لمنع التعديل المباشر وغير المُتحكم فيه لبيانات الكائن.

تُشبه هذه العملية إدارة راتب الموظف؛ فالراتب هو بيانات خاصة (لا يمكن للموظف تغييرها مباشرة)، وفقط المدير (من خلال دوال عامة مُصرح بها) يمكنه تحديثها أو مشاركتها عند الاقتضاء.

يُعزز هذا النهج نمطية التعليمات البرمجية ويُقلل من التبعيات غير الضرورية بين أجزاء مختلفة من النظام، مما يؤدي إلى سهولة الصيانة وتصحيح الأخطاء عن طريق تحديد التغييرات والتحكم في التفاعلات.

تطبيق التغليف باستخدام دوال Get و Set

للسماح بالوصول المُتحكم فيه إلى الأعضاء الخاصة، يتم توفير دوال عامة تُعرف باسم دوال (get) الوصول و (set) التعديل.

تُستخدم دالة set لكتابة أو تعديل قيمة سمة خاصة، بينما تُستخدم دالة get لقراءة أو استرداد قيمة سمة خاصة. هذا يضمن أن جميع التفاعلات الخارجية مع البيانات الخاصة تتم من خلال واجهة مُحددة، مما يسمح بتطبيق منطق التحقق من الصحة أو الآثار الجانبية (مثل التسجيل) عند الوصول إلى البيانات أو تعديلها.

إن استخدام دوال get و set يُعد نمطًا قياسيًا في البرمجة الشيئية لتطبيق التغليف. تُشكل هذه الدوال "بوابات" للبيانات الخاصة، مما يضمن أن أي عملية على البيانات تتم وفقًا للقواعد التي تحددها الفئة. تُمكن هذه الآلية المطورين من تغيير التنفيذ الداخلي للفئة دون التأثير على التعليمات البرمجية الخارجية التي تتفاعل معها، مما يُعزز مرونة التعليمات البرمجية وقابليتها للتوسع.

Q2

1. صح أم خطأ: المُنشئ يتم استدعاؤه يدويًا عند إنشاء الكائن.

2. ما نوع المُنشئ الذي لا يحتوي على معاملات؟
   أ) parameterized constructor
   ب) default constructor
   ج) copy constructor
   د) do-nothing constructor

3. صح أم خطأ: يجب أن يكون اسم المُنشئ مطابقًا لاسم الفئة.

4. صح أم خطأ: السمات المُعلنة بدون محدد وصول تكون عامة افتراضيًا.

5. صح أم خطأ: يمكن تعديل السمات الخاصة من خارج الفئة مباشرة.

الأجوبة

1. خطأ
2. ب
3. صح
4. خطأ
5. خطأ

مثال عملي على التغليف: إدارة راتب الموظف

يُقدم هذا المثال توضيحًا عمليًا لكيفية تطبيق التغليف في C++ باستخدام دوال "get" و "set" لإدارة سمة salary (الراتب) الخاصة. يُظهر المثال بوضوح أن سمةsalary ، المُعلن عنها كـprivate ، لا يمكن الوصول إليها مباشرة من خارج الفئة .Employee بدلاً من ذلك، يتم تعديلها وقراءتها فقط من خلال الدوال العامة ()setSalary و .()getSalary

في هذا المثال، يتم إنشاء كائن myObj من الفئة .Employee ثم تُستخدم myObj.setSalary(50000) لتعيين قيمة 50000 لراتب الكائن .myObj بعد ذلك، تُستخدم cout << myObj.getSalary(); لاسترداد وطباعة قيمة الراتب. يُوضح هذا كيف أن السمات الخاصة تُحمى وتُتاح فقط من خلال واجهة عامة مُتحكم بها، مما يجعل مفهوم التغليف ملموسًا وواضحًا .

```cpp
#include <iostream>
using namespace std;
class Employee {
private:
  int salary; // سمة خاصة
public:
  void setSalary(int s) { // دالة set
    salary = s;
  }
  int getSalary() { // دالة get
    return salary;
  }
};
int main() {
  Employee myObj;
  myObj.setSalary(50000); // تعيين القيمة باستخدام دالة set
  cout << myObj.getSalary(); // طباعة القيمة باستخدام دالة get
  return 0;
}
```

فوائد التغليف: أمان البيانات والتحكم

يُقدم التغليف العديد من الفوائد الهامة في تطوير البرمجيات، مما يجعله مبدأ تصميم أساسيًا للتعليمات البرمجية القوية والقابلة للصيانة. تُترجم هذه الفوائد مباشرة إلى تقليل الأخطاء، وتسهيل التعاون بين فرق العمل، ودورة حياة تطوير برمجيات أكثر استقرارًا.

تشمل الفوائد الرئيسية للتغليف ما يلي:

- تحكم أفضل في البيانات: يسمح التغليف للمطورين بتعديل جزء واحد من التعليمات البرمجية دون التأثير على الأجزاء الأخرى. هذا يعني أن التغييرات الداخلية في كيفية تخزين البيانات أو معالجتها لا تتطلب تعديلات واسعة النطاق في التعليمات البرمجية الخارجية التي تستخدم الفئة.
- زيادة أمان البيانات: يمنع التغليف الوصول المباشر وغير المصرح به إلى البيانات الحساسة وتعديلها. من خلال إجبار جميع التفاعلات على المرور عبر دوال get و set العامة، يمكن للمطورين فرض قواعد العمل والتحقق من صحة المدخلات قبل تحديث البيانات.
- المرونة: يُتيح التغليف إضافة منطق التحقق من الصحة داخل دوال. هذا يضمن أن البيانات تظل متسقة وصالحة.قابلية
- الصيانة: يُصبح تغيير التنفيذ الداخلي للفئة أسهل بكثير دون كسر التعليمات البرمجية الخارجية التي تعتمد عليها. طالما أن الواجهة العامة دوال get و set تظل كما هي، يمكن تغيير التفاصيل الداخلية بأمان.

أساسيات الوراثة: إعادة استخدام التعليمات البرمجية

الوراثة (Inheritance) هي آلية قوية في البرمجة الشيئية تسمح لفئة جديدة (الفئة المشتقة أو الابن) بوراثة السمات (Attributes) والدوال (Methods) من فئة موجودة بالفعل (الفئة الأساسية أو الأب).

تُعد هذه الآلية مفيدة بشكل أساسي لـ إعادة استخدام التعليمات البرمجية، حيث تُجنب تكرار التعليمات البرمجية وتعزز بناء هيكل هرمي للعلاقات بين الفئات.

يمكن تشبيه الوراثة بالعلاقة بين الوالد والطفل، حيث يرث الطفل صفات معينة من والديه. في البرمجة، يُمكن لفئة "سيارة" أن ترث خصائص عامة من فئة "مركبة" (مثل العلامة التجارية وطريقة إصدار الصوت)، ثم تُضيف خصائصها الفريدة (مثل الموديل).

تُعزز الوراثة مفهوم "هي-نوع-من" (is-a) ؛ فـ "السيارة هي نوع من المركبات". يُعد هذا الوضوح الدلالي أمرًا حيويًا لتصميم تسلسلات هرمية للفئات قابلة للتوسع والفهم، مما يُسهم في بناء تعليمات برمجية أكثر كفاءة وتنظيمًا.

بناء جملة وراثة الفئة: علاقات الأب والابن

لتطبيق الوراثة في C++، تُستخدم علامة النقطتين الرأسيتين (:) في إعلان الفئة المشتقة، متبوعة بمُحدد الوصول (مثل public) واسم الفئة الأساسية.

تُحدد هذه البنية العلاقة بين الفئة المشتقة والفئة الأساسية، وتُشير إلى أن الفئة المشتقة ستكتسب خصائص وسلوكيات الفئة الأساسية.

البنية العامة للوراثة هي كالتالي:

```
// الفئة الأساسية
class BaseClass {
  // سمات ودوال
};

// الفئة المشتقة
class DerivedClass : public BaseClass {
  // سمات ودوال جديدة
};
```

بناء جملة وراثة الفئة: علاقات الأب والابن

في هذه البنية:

- DerivedClass : هي الفئة الجديدة التي ترث.
- (:) : هو رمز الوراثة.
- (public) : هو مُحدد وصول للوراثة. يُحدد هذا المُحدد كيفية ظهور الأعضاء العامة والمحمية للفئة الأساسية في الفئة المشتقة. على سبيل المثال، إذا كانت الوراثة public، فإن الأعضاء العامة للفئة الأساسية تظل عامة في الفئة المشتقة، والأعضاء المحمية تظل محمية. ويُعد هذا جانبًا حاسمًا في تصميم الواجهة للفئة المشتقة.
- BaseClass : هي الفئة الموجودة التي تُورَث منها الأعضاء.

يُعد فهم هذا البناء أمرًا ضروريًا لإنشاء تسلسلات هرمية للفئات بشكل صحيح، حيث تُمكن هذه الآلية من بناء تعليمات برمجية مُنظمة وقابلة لإعادة الاستخدام.

مثال عملي على الوراثة: تسلسل المركبات والسيارات

يُقدم هذا المثال توضيحًا عمليًا لوراثة++C ، حيث ترث فئة Car (الفئة المشتقة) من فئة Vehicle (الفئة الأساسية). يُظهر هذا المثال بوضوح كيف يمكن للفئة المشتقة الوصول إلى السمات والدوال الموروثة من الفئة الأساسية، بالإضافة إلى سماتها ودوالها الخاصة .

```
// الفئة الأساسية
class Vehicle {
public:
  string brand = "Ford"; // سمة من الفئة الأساسية
  void honk() { // دالة من الفئة الأساسية
    cout << "Tuut, tuut! \n";
  }
};

// الفئة المشتقة
class Car : public Vehicle { // Car ترث من Vehicle
public:
  string model = "Mustang"; // سمة من الفئة المشتقة
};

int main() {
  Car myCar; // إنشاء كائن من فئة Car
  myCar.honk(); // استدعاء دالة honk() من فئة Vehicle (الموروثة)
  cout << myCar.brand + " " + myCar.model; // الوصول إلى سمات من Car و Vehicle
  return 0;
}
```

عند تنفيذ هذا المثال، سيكون الإخراج :

Tuut, tuut! \nFord Mustang.

يُبرز هذا الإخراج كيف أن الكائن myCar ، على الرغم من كونه من نوع Car ، يمكنه استدعاء الدالة honk() الموروثة من Vehicleوالوصول إلى السمة brand من Vehicle ، بالإضافة إلى الوصول إلى سمته الخاصة model.

هذا يُوضح بفعالية مبدأ إعادة استخدام التعليمات البرمجية من خلال الوراثة، مما يُسهم في بناء تعليمات برمجية أكثر كفاءة وتنظيمًا .

مفاهيم الوراثة متعددة المستويات والمتعددة

يمكن أن تمتد الوراثة في C++ إلى ما هو أبعد من علاقة الأب والابن الواحدة، مما يسمح ببناء تسلسلات هرمية أكثر تعقيدًا للفئات. تُعد هذه المفاهيم متقدمة وتُضيف مرونة كبيرة إلى تصميم الفئات، ولكنها تتطلب فهمًا دقيقًا لكيفية عملها.

- الوراثة متعددة المستويات (Multilevel Inheritance) : تحدث عندما ترث فئة (B) من فئة أساسية (A)، ثم ترث فئة أخرى (C) من الفئة المشتقة (B) .

يُشكل هذا سلسلة من الوراثة A -> B -> C ، حيث ترث الفئة C جميع سمات ودوال الفئتين A و B.

تُعد هذه الآلية مفيدة لنمذجة العلاقات الهرمية الطبيعية، مثل "حيوان" يرث منه "ثديي"، ثم يرث منه "كلب".

- الوراثة المتعددة (Multiple Inheritance) :تحدث عندما ترث فئة (C) من عدة فئات أساسية (A) و (B) في نفس الوقت A, B -> C . تُمكن هذه الآلية الفئة المشتقة من دمج خصائص وسلوكيات من مصادر متعددة.

على الرغم من قوتها، يمكن أن تُقدم الوراثة المتعددة تعقيدات مثل "مشكلة المعين" (Diamond Problem) حيث قد ترث الفئة المشتقة نفس العضو من فئات أساسية مختلفة عبر مسارات متعددة، مما يتطلب حلولًا خاصة.

تُبرز هذه الأنواع من الوراثة مرونة نموذج كائن C++، مما يسمح بإنشاء تسلسلات هرمية معقدة للفئات.

مُحددات الوصول في الوراثة

يُعد فهم كيفية تأثير مُحددات الوصول (public, private, protected) على الأعضاء الموروثة من الفئة الأساسية إلى الفئة المشتقة أمرًا بالغ الأهمية في C++.
يُحدد وضع مُحدد الوصول في سطر الوراثة على سبيل المثال، (class Derived : public Base) كيفية ظهور أعضاء الفئة الأساسية في الفئة المشتقة، وبالتالي،
كيفية الوصول إليها من خارج الفئة المشتقة. يُعد هذا قرارًا تصميميًا حاسمًا يؤثر على واجهة الفئة المشتقة .
القاعدة العامة لـ الوراثة العامة(public Inheritance) ، وهي الأكثر شيوعًا، هي كالتالي:

- الأعضاء public في الفئة الأساسية تظل public في الفئة المشتقة .
- الأعضاء protected في الفئة الأساسية تظل protected في الفئة المشتقة .
- الأعضاء private في الفئة الأساسية لا يمكن الوصول إليها مباشرة في الفئة المشتقة (على الرغم من أنها لا تزال موجودة كجزء من الكائن) .

تُحدد هذه القواعد الواجهة التي تُقدمها الفئة المشتقة للعالم الخارجي، وتُسهم في الحفاظ على مبادئ التغليف وإخفاء المعلومات حتى في التسلسلات الهرمية المعقدة للفئات.
تُشير هذه الآلية إلى أن اختيار مُحدد وصول الوراثة يُسبب تغييرًا مباشرًا في مستوى إمكانية الوصول للأعضاء الموروثة، مما يُؤثر على كيفية تفاعل التعليمات البرمجية مع الفئة المشتقة .

تعدد الأشكال: "أشكال متعددة" من العمل

تعدد الأشكال (Polymorphism) هو مبدأ أساسي في البرمجة الشيئية يعني "أشكال متعددة ."يحدث هذا المفهوم عندما تكون هناك فئات متعددة مرتبطة ببعضها البعض من خلال الوراثة، مما يسمح بتنفيذ إجراء واحد بطرق مختلفة .في جوهره، يُمكن تعدد الأشكال الكائنات المختلفة من الاستجابة لنفس استدعاء الدالة بطرق فريدة، بناءً على نوع الكائن الفعلي .

تُعد العلاقة مع الوراثة حاسمة؛ فغالبًا ما يستخدم تعدد الأشكال دوالًا موروثة من فئة أساسية يتم تجاوزها (overridden) لاحقًا في الفئات المشتقة لأداء مهام مختلفة .يُعزز هذا المفهوم إعادة استخدام التعليمات البرمجية والمرونة، حيث يسمح للمطورين بكتابة تعليمات برمجية عامة يمكنها التفاعل مع كائنات من أنواع مختلفة بطريقة موحدة، مع السماح لكل نوع بتوفير سلوكه الخاص.

يُعد تعدد الأشكال تجسيدًا لمبدأ "افتح للتوسع، أغلق للتعديل"، مما يعني أن النظام يمكنه التعامل مع أنواع جديدة من الكائنات دون الحاجة إلى تعديل التعليمات البرمجية الموجودة، مما يُسهم في تصميم أنظمة قابلة للتوسع والفهم.

تجاوز الدوال لسلوكيات متنوعة

يُعد تجاوز الدوال (Method Overriding) الآلية الأساسية التي يُحقق بها تعدد الأشكال في C++.

يسمح هذا المفهوم للفئات المشتقة بتوفير تطبيقها الخاص لدالة مُعرفة بالفعل في فئتها الأساسية.
على الرغم من أن الدالة تحمل نفس الاسم في الفئة الأساسية والفئة المشتقة، إلا أن سلوكها يختلف بناءً على نوع الكائن الذي يستدعيها .

الهدف من تجاوز الدوال هو السماح بسلوك متخصص لأنواع مختلفة من الكائنات مع الحفاظ على واجهة مشتركة.

على سبيل المثال، إذا كان لدينا فئة أساسية Animal تحتوي على دالة animalSound() ، يمكن لكل فئة مشتقة مثل Pig أو Dog تجاوز هذه الدالة لتوفير صوتها الخاص مثل ("wee wee") أو ("bow wow") هذا يُمكن المطورين من كتابة تعليمات برمجية عامة تتفاعل مع الكائنات من خلال واجهة الفئة الأساسية، بينما يُترك تحديد السلوك الفعلي لنوع الكائن المحدد في وقت التشغيل.

يُؤدي تجاوز الدوال إلى استجابة كائنات الفئات المشتقة بشكل فريد لنفس استدعاء الدالة، مما يُسفر عن سلوك متعدد الأشكال.

مثال تعدد الأشكال: أصوات الحيوانات (تعريفات الفئات)

يُقدم هذا المثال الكلاسيكي توضيحًا ممتازًا لتعدد الأشكال من خلال تسلسل هرمي بسيط للفئات يُمثل الحيوانات وأصواتها.

تُظهر تعريفات الفئات كيف تُحدد الفئة الأساسية Animal واجهة مشتركة، بينما تُخصص الفئات المشتقة Pig و Dog هذه الواجهة بسلوكيات فريدة.

مثال (الفئة الأساسية):تُعرف الفئة Animal بدالة عامة animalSound() تُصدر صوتًا عامًا للحيوان.

```
class Animal {
public:
  void animalSound() {
    cout << "The animal makes a sound \n";
  }
};
```

مثال (الفئة المشتقة Pig)

ترث فئة Pig من Animal وتُعيد تعريف دالة animalSound() لتُصدر صوت الخنزير .

```
class Pig : public Animal {
public:
  void animalSound() {
    cout << "The pig says: wee wee \n";
  }
};
```

مثال (الفئة المشتقة Dog)

ترث فئة Dog أيضًا من Animal وتُعيد تعريف دالة animalSound() لتُصدر صوت الكلب .

```
class Dog : public Animal {
public:
  void animalSound() {
    cout << "The dog says: bow wow \n";
  }
};
```

يُوضح هذا النمط بوضوح كيف تُحدد الفئة الأساسية سلوكًا افتراضيًا أو عامًا، بينما تُوفر الفئات المشتقة تطبيقاتها المتخصصة لنفس الدالة. هذا يُعد نمطًا أساسيًا للتصميم متعدد الأشكال، حيث تُمكن الكائنات من الاستجابة بشكل مختلف لنفس الرسالة.

مثال تعدد الأشكال: أصوات الحيوانات (الدالة الرئيسية)

تُعد الدالة الرئيسية (main) هي الجزء الحاسم الذي يُظهر تعدد الأشكال في العمل. من خلال إنشاء كائنات من الفئة الأساسية والفئات المشتقة، ثم استدعاء نفس الدالة (animalSound()) على كل كائن، يمكن ملاحظة السلوكيات المتنوعة التي تُنتجها كل فئة. يُبرز هذا المثال كيف تُنتج نفس استدعاء الدالة مخرجات مختلفة بناءً على النوع الفعلي للكائن في وقت التشغيل .

```
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound(); // الإخراج: The animal makes a sound
  myPig.animalSound();    // الإخراج: The pig says: wee wee
  myDog.animalSound();    // الإخراج: The dog says: bow wow
  return 0;
}
```

عندما يتم استدعاء myAnimal.animalSound() ، يتم تنفيذ الدالة من فئة Animal. ولكن عندما يتم استدعاء myPig.animalSound() ، يتم تنفيذ الدالة التي تم تجاوزها في فئة Pig. وبالمثل، يتم تنفيذ الدالة التي تم تجاوزها في فئة Dog عند استدعاء myDog.animalSound(). تُوضح هذه المخرجات المختلفة بوضوح مفهوم "الأشكال المتعددة" للدالة animalSound() ، مما يُسلط الضوء على جوهر تعدد الأشكال .

تحقيق المرونة باستخدام تعدد الأشكال

يُتيح تعدد الأشكال للمطورين كتابة تعليمات برمجية عامة يمكنها العمل مع كائنات من أنواع مختلفة، طالما أنها تشترك في واجهة فئة أساسية مشتركة. تُعد هذه المرونة ذات قيمة هائلة في تصميم الأنظمة المعقدة، حيث تُمكن من إضافة أنواع جديدة من الكائنات دون الحاجة إلى تعديل التعليمات البرمجية الموجودة التي تتفاعل معها.

على الرغم من أن الأمثلة السابقة تُظهر تجاوز الدوال، إلا أن تحقيق تعدد الأشكال الحقيقي في وقت التشغيل (runtime polymorphism) في C++، خاصة عند التعامل مع مؤشرات أو مراجع لفئات أساسية، يتطلب استخدام الدوال الافتراضية (Virtual Functions) . تُمكن الدوال الافتراضية المترجم من تحديد الدالة الصحيحة التي يجب استدعاؤها في وقت التشغيل بناءً على النوع الفعلي للكائن الذي يُشير إليه المؤشر أو المرجع، وليس النوع المُعلن عنه.

يُعد هذا جانبًا متقدمًا من تعدد الأشكال يُضاف إلى قوة C++في بناء أنظمة ديناميكية وقابلة للتوسع، ويُشير إلى المستوى التالي من الفهم الذي يمكن للمتعلم السعي إليه.

Q3

1. صح أم خطأ: الفئة المشتقة ترث من الفئة الأساسية باستخدام:
2. ما نوع الوراثة الذي يرث من أكثر من فئة؟
   أ inheritance
   ب multilevel inheritance
   ج multiple inheritance
   د dual inheritance
3. صح أم خطأ: في تعدد الأشكال، يمكن لنفس الدالة أن تُستدعى بسلوك مختلف حسب نوع الكائن.
4. صح أم خطأ: عند تجاوز دالة، يجب أن يكون اسمها مختلفًا عن الدالة الأصلية.
5. صح أم خطأ: الوراثة تساعد في تقليل تكرار الكود وتحسين تنظيمه.

الأجوبة

1. صح
2. ج
3. صح
4. خطأ
1. صح

المصادر

تم الإعتماد على W3SCOOLS في تجهيز هذه المحاضرة

https://www.w3schools.com/cpp/cpp_functions.asp

شكرا لكم