https://simplifycpp.org

مسودة

إتقان البرمجة كائنية التوجه OOP في لغة C++ الحديثة

إعداد: أيمن الحراكي

إتقان البرمجة كائنية التوجّه OOP
في لغة C++ الحديثة

تمت الاستعانة بأدوات ذكاء اصطناعي حديثة في بعض مراحل الصياغة واستكشاف الأفكار، وذلك تحت إشراف كامل، مع المراجعة والتحقق والتصحيح، مع احتفاظ المؤلف الكامل بحقوق التأليف والمسؤولية العلمية.

إعداد : أيمن الحراكي

simplifycpp.org

يناير 2026

المحتويات

مقدمة المؤلف

يسعدني وبحماسٍ كبير أن أقدّم ترجمة للإصدار الثاني من هذا الكتاب للغة العربية، والذي تم تحديثه وتوسيعه بشكل ملحوظ ليعكس أحدث التطورات في Modern C++، بما في ذلك الميزات التي تم تقديمها في C++20 و C++23. على مرّ السنوات، واصلت C++ تطورها المستمر، متبنيةً تحسينات قوية في اللغة مثل concepts و coroutines و ranges و modules، إلى جانب تحسينات في نماذج أمان الذاكرة، مع الحفاظ في الوقت نفسه على أدائها الفائق وتحكمها منخفض المستوى الذي لا يُضاهى.

كُتب الإصدار الأول من هذا الكتاب بهدف واحد واضح: مساعدة المطورين، سواء المبتدئين أو ذوي الخبرة، على فهم مبادئ البرمجة كائنية التوجه (OOP -- Object-Oriented Programming) و Modern C++ بأسلوب منظم، عملي، وسهل الوصول. ومنذ صدوره، أبرزت ملاحظات القرّاء أهمية تضمين تطبيقات واقعية، ودراسات حالة، ورؤى أعمق حول أحدث ميزات اللغة. وقد جاء هذا الإصدار الثاني ليستجيب مباشرةً لتلك الملاحظات، حيث يقدّم:

- تغطية شاملة لميزات C++20 و C++23، بما في ذلك concepts، و coroutines، وranges، و modules، وغيرها.
- فصولاً موسّعة حول تقنيات OOP المتقدمة، وأنماط التصميم الحديثة، وبرمجة القوالب المتقدمة (template metaprogramming).
- دراسات حالة ومشاريع من العالم الحقيقي توضّح أفضل الممارسات في الأداء، وإدارة الذاكرة، وتصميم التطبيقات القابلة للتوسع.

- ملاحق محدّثة تتضمن ملخصات عملية (cheat sheets)، وتمارين برمجية، ومراجع لأهم الأدوات والمكتبات وموارد التعلم.

هذا الكتاب موجّه لشريحة واسعة من القرّاء. فهو مناسب لمطوري C++ المحترفين الراغبين في تحديث قواعدهم البرمجية، وللطلاب الذين يسعون إلى بناء أساس قوي في كلٍ من OOP و Modern C++، وكذلك للمبرمجين القادمين من لغات أخرى والراغبين في فهم القوة والمرونة والأداء الفريد الذي تقدمه C++. يركّز هذا النهج على الفهم العميق للمفاهيم، وتطبيقها عملياً، وكتابة شيفرة نظيفة، قابلة للصيانة، وعالية الأداء.

تبقى الفلسفة الكامنة وراء هذا الإصدار كما كانت في الإصدار الأول: C++ ليست مجرد لغة برمجة، بل أداة لحل مشكلات العالم الحقيقي بكفاءة. ومن خلال تعلّم كيفية توظيف ميزات Modern C++ بشكل صحيح، يمكن للمطورين كتابة شيفرة آمنة، ومعبرة، وسهلة الصيانة، دون التفريط في مستوى التحكم الدقيق الذي تنفرد به C++.

آمل أن يكون هذا الإصدار الثاني مرجعًا شاملاً ودليلاً عملياً في آنٍ واحد، يلهم القرّاء على الاستكشاف، والتجربة، وإتقان Modern C++. وأن يصبح رفيقًا موثوقًا في رحلتكم نحو أن تكونوا مطوري C++ محترفين وواثقين.

أيمن الحراكي

التمهيد

في هندسة البرمجيات، تتطور التقنيات بسرعة، وتتغير المعايير، وتُصقَل الأدوات باستمرار. ومع ذلك، ورغم هذه التحولات، تبقى بعض الأفكار التأسيسية راسخة لأنها تعالج مشكلة جوهرية: كيفية إدارة التعقيد. وتُعد البرمجة كائنية التوجه (Object-Oriented Programming -- OOP) إحدى هذه الأفكار. فهي ليست مجرد أسلوب برمجي، بل منهج منظم للتفكير في الأنظمة البرمجية التي تنمو في الحجم والمسؤولية وطول العمر.

يتناول هذا الكتاب، *Object-Oriented Programming in Modern C++*، مفهوم OOP من منظور Modern C++—وهي لغة تجمع بين التحكم منخفض المستوى والتجريد عالي المستوى، وتواصل تطورها ضمن إطار صارم من التقييس الرسمي. وبدلاً من التعامل مع OOP على أنه مجموعة قواعد أو أنماط تُحفَظ، يقدّم هذا الكتاب OOP بوصفه تخصصًا هندسيًا يجب تطبيقه بوعيٍ مقصود، مع إدراكٍ كاملٍ للتكلفة، والعمر التشغيلي، والصحة (correctness).

في جوهره، يشجّع OOP المطورين على نمذجة الأنظمة على هيئة كائنات متعاونة تغلف الحالة (state) والسلوك (behavior) معًا. وعند تطبيقه بشكل صحيح، يقود ذلك إلى شيفرة أسهل للفهم، وأسهل للتوسعة، وأكثر مقاومة للتغيير. لكن عندما يُطبّق بصورة آلية أو دون انضباط، يمكن أن يُدخل ترابطًا غير ضروري، وتسلسلات وراثية هشة، وتعقيدًا خفيًا. وتحدّ Modern C++ من كثير من هذه المخاطر عبر توفير دعم لغوي أقوى للتعبير عن النية (intent)، والملكية (ownership)، والثوابت/القيود الداخلية (invariants).

منذ C++11، شهدت اللغة تحولاً عميقًا. فقد أعادت ميزات مثل إدارة الموارد وفق RAII، والمؤشرات الذكية (smart pointers)، ودلالات النقل (move semantics)، وتعبيرات lambda، وتقييم constexpr، وconcepts تعريف ما يعنيه التصميم الكائني الفعّال في C++. إن Modern C++ لا ترفض OOP؛ بل تُهذّبه وتُحسّنه، وتدمجه مع التجريد وقت الترجمة (compile-time abstraction)، ودلالات القيمة

(value semantics)، ونماذج الملكية الصريحة (explicit ownership models).

لماذا Modern C++

لطالما عُرفت C++ بأدائها وقدرتها التعبيرية، لكنها تعرضت أيضًا للنقد بسبب التعقيد وسوء الاستخدام. وترجع كثير من هذه الانتقادات إلى أساليب ما قبل العصر الحديث (pre-modern idioms) التي كانت تعتمد بشكل كبير على انضباط المبرمج بدلاً من ضمانات اللغة. تعالج Modern C++ هذه الإشكالات عبر جعل التعبير عن الصحة أسهل، وجعل إساءة الاستخدام أصعب تبريرًا.

واليوم، تُستخدم C++ على نطاق واسع في برمجة الأنظمة (systems programming)، والبرمجيات المضمنة (embedded software)، ومحركات الألعاب (game engines)، والأنظمة المالية، والبنى التحتية ذات الزمن الحقيقي (real-time infrastructure)، والمكتبات الحساسة للأداء (performance-critical libraries). ولا تُعد استمرارية حضورها أمرًا عارضًا؛ بل هي نتيجة تطور مقصود موجّه بخبرة عملية ومبادئ تصميم رسمية. يتبنى هذا الكتاب ذلك الاتجاه الحديث، ويتعامل مع C++ لا كلغة قديمة، بل كمنظومة أدوات حية تتمحور حول الهندسة.

ماذا يعلّم هذا الكتاب

هذا الكتاب موجّه للقراء الذين يرغبون في فهم OOP في C++ على مستوى احترافي. وهو يفترض إلماماً أساسياً بالبرمجة، لكنه لا يفترض إتقاناً مسبقاً لـC++ أو لتصميم البرمجيات الكائني. على امتداد الفصول، سيستكشف القارئ:

- مبادئ **OOP** الأساسية
  تُناقَش مفاهيم التغليف (encapsulation)، والتجريد (abstraction)، والوراثة (inheritance)، وتعدد الأشكال (polymorphism) لا كشعارات، بل كآليات ذات تكاليف وفوائد محددة ضمن Modern C++.

- ممارسات التصميم الحديثة
  التركيز على التركيب بدل الوراثة (composition over inheritance)، وRAII، والملكية الصريحة (explicit ownership)، والتصميم المعتمد على الواجهات (interface-based design) المتوافق مع ISO C++ Core Guidelines.

- إدارة الذاكرة والموارد
  يشرح الكتاب كيف تستبدل Modern C++ إدارة الذاكرة اليدوية الهشة بتحكم حتمي بعمر الكائنات يعتمد على النطاق (deterministic, scope-based lifetime control).

- ميزات اللغة المتقدمة
  تُقدَّم القوالب (templates)، وconcepts، وتقييم constexpr، والبرمجة العامة (generic programming) بوصفها أدوات مكملة تُعيد تشكيل التصميم الكائني بدلاً من أن تستبدله.

- اعتبارات هندسية من العالم الحقيقي
  توضح أمثلة عملية كيف تؤثر قرارات OOP في الأداء، وقابلية الصيانة، والصحة، والتطور طويل الأمد للأنظمة البرمجية.

رحلة منضبطة

البرمجة ليست فقط كتابة شيفرة تعمل اليوم، بل تصميم أنظمة تبقى مفهومة وصحيحة غدًا. يشجّع هذا الكتاب عقلية منضبطة: مساءلة التجريدات، وجعل الملكية صريحة، وتفضيل الوضوح على الاستعراض.

يُشجَّع القرّاء على التجربة بالأمثلة، ومناقشة قرارات التصميم المعروضة، ومقارنة البدائل الممكنة. تكافئ Modern C++ المهندسين الذين يفكرون بعناية في الواجهات، والأعمار (lifetimes)، والمسؤوليات.

لقد لعب مجتمع C++—عبر لجان المعايير (standards committees)، ومشاريع المصدر المفتوح، والنقاشات المهنية—دورًا حاسمًا في تشكيل الصورة الحديثة للغة. ويُعد التفاعل مع هذا المجتمع جزءًا أساسيًا من إتقان C++ كأداة احترافية.

الخاتمة

لا يهدف *Object-Oriented Programming in Modern C++* إلى أن يكون مقدمة سريعة أو قائمة أنماط. إنه استكشاف منظم لكيفية تموضع التصميم الكائني داخل Modern C++ بوصفه تخصصًا هندسيًا.

بحلول نهاية هذا الكتاب، ينبغي أن يكون القرّاء قادرين على تطبيق مبادئ OOP بثقة، وفهم متى تكون مناسبة، والتعرّف على الحالات التي تقدم فيها التقنيات البديلة حلولاً أفضل. الهدف ليس كتابة شيفرة أكثر كائنية، بل كتابة C++ أفضل.

وأنت تنتقل إلى الفصل الأول، اقترب من المادة بانتباه للتفاصيل واستعداد للإعادة النظر في الافتراضات. أدوات Modern C++ قوية، لكن قيمتها الحقيقية تظهر في مدى حسن توظيفها وبقدر ما تُطبَّق بوعيٍ ومسؤولية.

الفصل ١: مقدمة في البرمجة كائنية التوجه باستخدام C++

- تعريف البرمجة كائنية التوجه.
- تطور البرمجة كائنية التوجه في C++.
- الفروقات بين C++ الكلاسيكية وModern C++ في تطبيق OOP.

تُعد البرمجة كائنية التوجه (OOP -- Object-Oriented Programming) نموذجًا تصميميًا يركّز على تنظيم البرمجيات حول أنواع محددة بوضوح، تجمع بين السلوك (behavior) والحالة (state)، مع فرض ثوابت داخلية (invariants) بشكل صارم. في C++، لا تُعامل OOP على أنها أيديولوجيا مستقلة، بل تقنية هندسية منضبطة تُستخدم لإدارة التعقيد، والتعبير الدقيق عن النية التصميمية، والحفاظ على صحة النظام عبر الزمن.
وعلى خلاف اللغات الكائنية البحتة، تدمج C++ بين البرمجة الكائنية، والإجرائية، والعامة، والوظيفية. ووفقًا لإرشادات ISO C++ Core Guidelines، يجب تطبيق تقنيات OOP بشكل واعٍ ومقصود، فقط عندما تؤدي إلى تحسين الوضوح، والأمان، وقابلية الصيانة، دون المساس بالأداء.

١.١ تعريف البرمجة كائنية التوجه

في C++، تُعرَّف البرمجة كائنية التوجه على أنها منهجية لبناء البرمجيات حول أنواع (types) تمثل مسؤوليات (responsibilities) محددة، لا مجرد استعارات مباشرة من العالم الواقعي. تُغلِّف الكائنات الحالة والسلوك معًا، وتفرض ثوابت الصنف عبر المُنشئات (constructors) وواجهات مضبوطة بعناية.
الأهداف الأساسية للبرمجة كائنية التوجه في C++ هي:

- تغليف الحالة والثوابت الداخلية
- توفير واجهات عامة واضحة ومحدودة
- التعبير الصريح عن الملكية (ownership) وعمر الكائن
- الاستخدام المنضبط لتعدد الأشكال

١.١.١ المفاهيم الجوهرية للبرمجة كائنية التوجه

تعيد Modern C++ تفسير مفاهيم OOP الكلاسيكية ضمن قيود هندسية صارمة كما تحددها إرشادات Core Guidelines.

١. التغليف

يعني التغليف الفصل الواضح بين الواجهة والتنفيذ، ومنع الوصول غير المنضبط إلى حالة الكائن. تكون بيانات الصنف خاصة، وتُنشأ الثوابت عند البناء، وتتم جميع التعديلات عبر دوال عضو محددة بدقة.

٢. التجريد

التجريد في C++ يعني إظهار ما يحتاجه المستخدم فقط، مع إخفاء تفاصيل التنفيذ. ويتحقق ذلك عادةً باستخدام أصناف أساس مجردة، وواجهات غير مالكة، وواجهات برمجية صغيرة ومركّزة.

٣. الوراثة

تُحجز الوراثة حصريًا لنمذجة قابلية الاستبدال الحقيقية (true substitutability). وتحذّر إرشادات Core Guidelines بشدة من استخدام الوراثة لغرض إعادة استخدام التنفيذ فقط، حيث يُفضَّل التركيب ما لم يكن تعدد الأشكال وقت التشغيل مطلوبًا فعليًا.

٤. تعدد الأشكال

يتيح تعدد الأشكال الوصول إلى تطبيقات مختلفة عبر واجهة مشتركة. وتُفرّق C++ بوضوح بين تعدد الأشكال وقت الترجمة (القوالب) وتعدد الأشكال وقت التشغيل (الدوال الافتراضية).

٢.١.١ C++ والبرمجة كائنية التوجه

تُعد C++ لغة متعددة المناهج، تُشكّل OOP أحد أساليبها المكملة. فهي تمنح تحكمًا مباشرًا في عمر الكائن، وتخطيط الذاكرة، والأداء، مع توفير آليات تجريد قوية عند الحاجة.
في التصميم الحديث، لا يقتصر التركيز على الأصناف والكائنات فقط، بل يمتد ليشمل دلالات القيمة (value semantics)، وRAII، والتعبير الصريح عن الملكية، بدل الاعتماد المفرط على هياكل وراثية معقدة.

الأصناف والكائنات في C++

يُعرِّف الصنف نوعًا محددًا، يشمل ثوابته وعملياته وتمثيله الداخلي، بينما يمثّل الكائن تجسيدًا فعليًا لذلك النوع بعمر محدد وواضح.

مثال: صنف مُغلف بشكل صحيح

```cpp
#include <iostream>
#include <string>

class Car {
public:
    Car(std::string brand, std::string model, int year)
        : brand_{std::move(brand)},
          model_{std::move(model)},
          year_{year} {}

    void display_info() const {
        std::cout << "Car Info: "
                  << brand_ << " "
                  << model_ << ", "
                  << year_ << '\n';
    }

private:
    std::string brand_;
    std::string model_;
    int year_;
};
```

```
int main() {
    Car my_car{"Toyota", "Corolla", 2021};
    my_car.display_info();
}
```

الشرح

- جميع بيانات الصنف خاصة ولا يمكن الوصول إليها مباشرة.
- تُنشأ الثوابت الداخلية عند البناء.
- الدوال العضوية ملتزمة بمبدأ const-correctness.

الناتج:

```
Car Info: Toyota Corolla, 2021
```

ا. التغليف

يضمن التغليف عدم تعديل الحالة الداخلية للكائن بشكل عشوائي. وتُفرض قواعد الرؤية باستخدام محددات الوصول:

- private: تفاصيل التنفيذ والثوابت
- protected: استخدام محدود لدعم الوراثة
- public: واجهات مستقرة ومحدودة

مثال: تغليف مع فرض الثوابت

```
class BankAccount {
public:
```

```
    explicit BankAccount(double initial_balance)
        : balance_{initial_balance} {}

    void deposit(double amount) {
        if (amount > 0) {
            balance_ += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount <= balance_) {
            balance_ -= amount;
            return true;
        }
        return false;
    }

    double balance() const noexcept {
        return balance_;
    }

private:
    double balance_{0.0};
};
```

٢. التجريد

يُخفي التجريد تفاصيل التنفيذ خلف واجهة مستقرة، بحيث يتعامل المستخدم مع السلوك لا مع التمثيل الداخلي.

مثال: تجريد قائم على الواجهات

```
class BeverageMaker {
public:
    virtual ~BeverageMaker() = default;
    virtual void prepare() = 0;
};

class CoffeeMachine final : public BeverageMaker {
public:
    void prepare() override {
        boil_water();
        brew();
        pour();
    }

private:
    void boil_water();
    void brew();
    void pour();
};
```

٣. الوراثة

تعبّر الوراثة عن علاقة is-a ويجب أن تلتزم بمبدأ الاستبدال لLiskov. ولا يجوز استخدامها لمجرد إعادة استخدام التنفيذ.

مثال: وراثة متعددة الأشكال صحيحة

```
class Animal {
```

```
public:
    virtual ~Animal() = default;
    virtual void sound() const = 0;
};

class Dog final : public Animal {
public:
    void sound() const override {
        std::cout << "Barking\n";
    }
};
```

٤. تعدد الأشكال

يتيح تعدد الأشكال وقت التشغيل الوصول إلى سلوكيات مختلفة عبر واجهة مشتركة عند الحاجة إلى الإرسال الديناميكي.

مثال: تعدد أشكال آمن وقت التشغيل

```
#include <memory>
#include <vector>

void make_sound(const std::vector<std::unique_ptr<Animal>>& animals) {
    for (const auto& a : animals) {
        a->sound();
    }
}
```

٣.١.١ الخلاصة

البرمجة كائنية التوجه في C++ لا تهدف إلى محاكاة العالم الحقيقي حرفيًا، ولا إلى تضخيم عدد الأصناف، بل إلى تصميم تجريدات دقيقة، وفرض الثوابت عبر نظام الأنواع، والتعبير الصريح عن الملكية وعمر الكائن.

وعند تطبيقها وفق إرشادات ISO C++ Core Guidelines وممارسات C++23 الحديثة، تصبح OOP أداة هندسية قوية لبناء أنظمة برمجية آمنة، وعالية الكفاءة، وقابلة للصيانة على المدى الطويل.

١.٢ تطور البرمجة كائنية التوجه في C++

لعبت البرمجة كائنية التوجه (OOP -- Object-Oriented Programming) دورًا حاسمًا في تشكيل هندسة البرمجيات الحديثة. وقد مثّل دمجها في C++ محاولة واعية للجمع بين الكفاءة منخفضة المستوى والتحكم الدقيق بالأداء، وبين آليات تنظيم عالية المستوى قادرة على إدارة أنظمة كبيرة وطويلة العمر.

يمتد تطور OOP في C++ عبر عدة عقود، ويعكس عملية تنقيح مستمرة لمبادئ التصميم، لا تحوّلاً مفاجئًا في المنهج. فلم تتبنَّ C++ الأفكار الكائنية كما هي، بل أعادت تكييفها لتتعايش مع التحكم المباشر بالذاكرة، والأداء الحتمي، والتجريد وقت الترجمة.

يتتبع هذا القسم المسار التاريخي لتطور OOP في C++، ويبرز محطاته الأساسية، ويشرح كيف أعادت Modern C++ صياغة التصميم الكائني وفق قيود هندسية صارمة.

١.٢.١ نماذج البرمجة قبل ظهور OOP

قبل إدخال التقنيات الكائنية، كانت معظم البرمجيات تُبنى باستخدام البرمجة الإجرائية. حيث تُنظَّم البرامج حول دوال تعمل على بيانات مشتركة أو ذات بنية ضعيفة.

تكون البرمجة الإجرائية فعّالة للبرامج الصغيرة أو ذات النطاق المحدود، إلا أنها تتدهور بسرعة مع ازدياد حجم الأنظمة. ومع نمو قواعد الشيفرة، تعاني التصاميم الإجرائية عادةً من:

- ترابط غير مضبوط بين البيانات
- تبعيات ضمنية بين الدوال
- صعوبة فرض الثوابت الداخلية
- ضعف إعادة الاستخدام لمكوّنات متماسكة

تُعد لغة C، السلف المباشر لـC++، مثالاً واضحًا على هذا النموذج.

مثال: البرمجة الإجرائية في C

```c
#include <stdio.h>

void print_student(const char* name, int age, float gpa) {
    printf("Student: %s\n", name);
    printf("Age: %d\n", age);
    printf("GPA: %.2f\n", gpa);
}

int main(void) {
    print_student("John Doe", 20, 3.5f);
    return 0;
}
```

رغم صحة هذا الأسلوب، إلا أنه لا يوفّر آليات لفرض الثوابت، أو تجميع السلوكيات المرتبطة، أو التحكم في الوصول إلى الحالة المشتركة. ومع ازدياد حجم الأنظمة، أصبحت هذه القيود مكلفة بشكل متزايد.

٢.٢.١ نشأة C++ والتصميم كائني التوجه

نشأت C++ في أوائل ثمانينيات القرن الماضي كامتداد للغة C على يد **Bjarne Stroustrup**. وكان اسمها الأولي *C with Classes*، وكان هدفها الأساسي إضافة آليات تجريد أقوى دون التضحية بالأداء أو التحكم منخفض المستوى.
شملت الأهداف التصميمية المبكرة لـC++:

- الحفاظ على التوافق مع C
- إدخال أنواع معرّفة من قبل المستخدم مع ثوابت واضحة
- تمكين التفكير المعياري في الأنظمة الكبيرة

وأدى ذلك إلى إدخال:

- الأصناف والكائنات
- التغليف والتحكم في الوصول
- الوراثة والإرسال الافتراضي

وعلى عكس العديد من اللغات الكائنية اللاحقة، لم تتخلَّ C++ عن البرمجة الإجرائية. بل أصبحت OOP أداة من بين عدة أدوات، تُستخدم عند ملاءمتها للمشكلة المطروحة.

٣.٢.١ محطات رئيسية في تطور OOP داخل C++

- الأصناف والتغليف (1983)
  أتاحت الأصناف تجميع البيانات والعمليات في وحدات متماسكة مع تحكم صريح في الوصول، مما شكّل الأساس لمفهوم التغليف.

- الوراثة (1985)
  أدخلت الوراثة علاقات هرمية بين الأنواع، ومكّنت تعدد الأشكال المنضبط وإعادة استخدام الواجهات.

- الدوال الافتراضية وتعدد الأشكال (أواخر الثمانينيات)
  أتاح الإرسال الافتراضي تعدد الأشكال وقت التشغيل، مما سمح بالتعامل مع كائنات من أنواع مختلفة عبر واجهات مشتركة.

- القوالب والبرمجة العامة (**C++98**)
  حوّلت القوالب ومكتبة القوالب القياسية (STL) التركيز نحو تعدد الأشكال وقت الترجمة والتصميم القائم على القيم.

- إصلاحات **Modern C++** (**C++11** وما بعدها)
  غيّرت المؤشرات الذكية، ودلالات النقل، وتعبيرات lambda، وconstexpr جذريًا طريقة التعبير عن الملكية، والعمر، والتجريد في التصميم الكائني.

قلّلت هذه المحطات تدريجيًا من الاعتماد على الهياكل الوراثية الهشة، وعززت التصاميم الأكثر أمانًا ووضوحًا.

٤.٢.١ تطور المفاهيم الجوهرية لـOOP في C++

١. التغليف

تطور التغليف من مجرد التحكم في الوصول إلى مفهوم أقوى يتمثل في فرض الثوابت. تؤكد Modern C++ على التهيئة عبر المُنشئات، والبيانات الخاصة، والواجهات العامة الدنيا.

مثال حديث على التغليف

```
class Student {
public:
    Student(std::string name, int age, double gpa)
        : name_{std::move(name)}, age_{age}, gpa_{gpa} {}

    void print() const {
        std::cout << name_ << ", " << age_ << ", " << gpa_ << '\n';
    }

private:
    std::string name_;
    int age_;
    double gpa_;
};
```

٢. الوراثة

لا تزال الوراثة مدعومة، لكن Modern C++ تثني عن استخدامها للإعادة استخدام التنفيذ. وتوصي إرشادات ISO C++ Core Guidelines باستخدامها فقط لتحقيق الاستبدال متعدد الأشكال الحقيقي.

٣. تعدد الأشكال

ينقسم تعدد الأشكال في Modern C++ إلى:

- تعدد أشكال وقت الترجمة باستخدام القوالب وconcepts
- تعدد أشكال وقت التشغيل باستخدام الدوال الافتراضية

ويُفضَّل تعدد الأشكال وقت الترجمة متى أمكن، نظرًا لانعدام كلفة وقت التشغيل وضمانات الأنواع الأقوى.

٤. التجريد

تحوّل التجريد نحو تصميم قائم على الواجهات مع أصناف أساس غير مالكة ودلالات ملكية صريحة.

تجريد قائم على الواجهات

```
class Vehicle {
public:
    virtual ~Vehicle() = default;
    virtual void drive() = 0;
};

class Car final : public Vehicle {
public:
    void drive() override {
        std::cout << "Driving a car\n";
```

```
    }
};
```

١.٢.٥ إسهامات Modern C++ في تحسين التصميم الكائني

قدّمت معايير Modern C++ (C++11–C++23) ميزات حسّنت البرمجة كائنية التوجه بشكل جوهري:

- المؤشرات الذكية للتعبير الصريح عن الملكية وRAII
- دلالات النقل لنقل الموارد بكفاءة
- تعبيرات **lambda** لتعريف السلوك محليًا
- **Concepts** لتقييد الواجهات العامة
- **constexpr** الموسّع للتقييم وقت الترجمة

أسهمت هذه الميزات في تقليل الاعتماد على التصاميم كثيفة الوراثة، وعززت تجريدات أوضح وأكثر أمانًا.

١.٢.٦ الخلاصة

يعكس تطور البرمجة كائنية التوجه في C++ انتقالاً تدريجيًا من تصميم متمحور حول الأصناف إلى هندسة متمحورة حول الثوابت والملكية الواعية.
لا ترفض OOP Modern C++، بل تُنقّيها. وعند تطبيقها وفق إرشادات ISO C++ Core Guidelines، تتعايش التقنيات الكائنية مع البرمجة العامة، وRAII، والتجريد وقت الترجمة لإنتاج برمجيات فعّالة، قابلة للتوسع، وقابلة للصيانة عبر عقود.
إن فهم هذا التطور شرط أساسي لكتابة C++ حديثة تحترم قيود الأداء وتضمن الصحة طويلة الأمد.

١.٣ الفروقات بين C++ الكلاسيكية وModern C++ في البرمجة كائنية التوجه

أدّى تطور لغة C++ منذ معاييرها المبكرة وحتى Modern C++ (C++11 وما بعدها) إلى إدخال تغييرات جوهرية حسّنت بشكل كبير الأمان، والقدرة التعبيرية، وصحة التصميم الكائني.
وفّرت C++ الكلاسيكية الآليات الأساسية للبرمجة كائنية التوجه، بما في ذلك الأصناف، والوراثة، والدوال الافتراضية. إلا أنها اعتمدت بدرجة كبيرة على انضباط المبرمج بدلاً من دعم اللغة نفسها لضمان الصحة. تعالج Modern C++ هذه القيود من خلال إدخال دلالات ملكية صريحة، وتجريدات أكثر أمانًا، وضمانات أقوى وقت الترجمة.
يقارن هذا القسم بين C++ الكلاسيكية (ما قبل C++11) وModern C++ (C++11–C++23) من منظور التصميم الكائني، مُبرزًا كيف تعيد الميزات الحديثة صياغة أفضل الممارسات.

١.٣.١ نظرة عامة على الفروقات الأساسية

١. المُنشئات وإدارة الذاكرة

٢. الوراثة وتعدد الأشكال

٣. المؤشرات الذكية ودلالات الملكية

٤. دلالات النقل

٥. استنتاج الأنواع باستخدام auto

٦. تعبيرات lambda

٧. إمكانات القوالب الحديثة

٨. البرمجة وقت الترجمة باستخدام constexpr

٩. التوازي وأمان الكائنات

٢.٣.١ المُنشئات وإدارة الذاكرة

C++ الكلاسيكية

في C++ الكلاسيكية، كانت المُنشئات والمُدمّرات مسؤولة عن إدارة الموارد يدويًا. واعتمد تخصيص الذاكرة الديناميكي على new وdelete، مما جعل إدارة العمر الصحيحة مسؤولية كاملة على عاتق المبرمج.

نمط كلاسيكي

```
class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource released\n"; }
};

int main() {
    Resource* r = new Resource();
    delete r;   //
}
```

يُعد هذا النمط هشًا، ومعرّضًا لتسريبات الذاكرة والسلوك غير المعرّف.

Modern C++

تفرض Modern C++ مبدأ RAII باستخدام أنواع من المكتبة القياسية تُعبّر عن الملكية بشكل صريح. وتصبح إدارة الذاكرة حتمية وآمنة في وجود الاستثناءات بحكم التصميم.

نمط حديث

```
#include <memory>

int main() {
```

```
    auto r = std::make_unique<Resource>();
}
```

يُحرَّر المورد تلقائيًا عند خروج الكائن المالك من النطاق.

٣.٣.١ الوراثة وتعدد الأشكال

C++ الكلاسيكية

كانت الوراثة وتعدد الأشكال متاحة، لكن دون ضوابط كافية. ومن المشاكل الشائعة: غياب المُدمّرات الافتراضية، وإعادة التعريف غير المقصودة، واستخدام المؤشرات الخام بشكل غير آمن.

تعدد أشكال كلاسيكي

```
class Animal {
public:
    virtual void sound() {
        std::cout << "Animal sound\n";
    }
};

int main() {
    Animal* a = new Animal{};
    delete a;
}
```

يُعرّض هذا التصميم البرنامج لسلوك غير معرّف عند استخدامه تعدديًا.

Modern C++

تعزّز Modern C++ التصميم التعددي باستخدام override، وfinal، وفرض المُدمّرات الافتراضية.

تعدد أشكال حديث

```
class Animal {
public:
    virtual ~Animal() = default;
    virtual void sound() const = 0;
};

class Dog final : public Animal {
public:
    void sound() const override {
        std::cout << "Barking\n";
    }
};
```

هذا التصميم صريح، وآمن، وذاتي التوثيق.

٤.٣.١ المؤشرات الذكية ودلالات الملكية

C++ الكلاسيكية

كانت الملكية ضمنية وغير موثقة. وغالبًا ما استُخدمت المؤشرات الخام لتمثيل علاقات امتلاك، مما أدى إلى أخطاء تصميمية خطيرة.

Modern C++

تُعبَّر الملكية بشكل صريح باستخدام المؤشرات الذكية:

- std::unique_ptr للملكية الحصرية
- std::shared_ptr للملكية المشتركة
- std::weak_ptr لكسر الدورات المرجعية

مثال

```
#include <memory>

auto widget = std::make_unique<Widget>();
```

يتماشى هذا الأسلوب مع إرشادات Core Guidelines التي توصي بتجنب المؤشرات المالكة الخام.

٥.٣.١ دلالات النقل

C++ الكلاسيكية

كان النسخ هو الآلية الأساسية لنقل الموارد، مما أدى في كثير من الحالات إلى كلفة أداء غير ضرورية.

Modern C++

تسمح دلالات النقل بنقل الموارد بكفاءة دون تكرارها.

مثال

```
class Data {
public:
    Data() = default;
    Data(const Data&) { std::cout << "Copy\n"; }
    Data(Data&&) noexcept { std::cout << "Move\n"; }
};

Data a;
Data b = std::move(a);
```

تُعد دلالات النقل ركيزة أساسية في التصميم الكائني الحديث.

٦.٣.١ استنتاج الأنواع باستخدام auto

C++ الكلاسيكية

أدّت التصريحات الصريحة عن الأنواع إلى شيفرة مطوّلة وأقل قابلية للصيانة.

Modern C++

يتيح auto استنتاج النوع بدقة مع الحفاظ على التحقق الساكن.

```
auto count = std::vector<int>{1, 2, 3}.size();
```

يحسّن هذا الأسلوب قابلية القراءة ويجعل الشيفرة أكثر مقاومة للإعادة الهيكلة.

٧.٣.١ تعبيرات Lambda

C++ الكلاسيكية

كانت الاستدعاءات الراجعة تتطلب مؤشرات دوال أو أصناف دوال (functors).

Modern C++

توفّر تعبيرات lambda تعريفًا محليًا للسلوك مع نية تصميمية واضحة.

```
std::for_each(values.begin(), values.end(),
              [](int v) { std::cout << v << ' '; });
```

تتكامل lambdas بسلاسة مع الخوارزميات العامة.

٨.٣.١ إمكانات القوالب الحديثة

حسّنت Modern C++ القوالب بشكل كبير من خلال:

- القوالب المتغيرة (Variadic Templates)
- تعبيرات الطي (Fold Expressions)
- Concepts (C++20)

مثال

```
template<typename... Args>
void print(const Args&... args) {
    (std::cout << ... << args) << '\n';
}
```

أصبحت القوالب آلية رئيسية لتعدد الأشكال وقت الترجمة.

٩.٣.١ البرمجة وقت الترجمة باستخدام constexpr

C++ الكلاسيكية

كانت الحسابات وقت الترجمة محدودة بالثوابت البسيطة والماكروات.

Modern C++

يتيح constexpr تنفيذ منطق معقّد وقت الترجمة.

مثال

```
constexpr int factorial(int n) {
    int r = 1;
    for (int i = 2; i <= n; ++i) r *= i;
    return r;
}

constexpr int value = factorial(5);
```

يحسّن constexpr الحديث الأداء، والأمان، والقدرة التعبيرية.

١.٣.١٠ التوازي وأمان الكائنات

لم توفّر C++ الكلاسيكية نموذج توازي معياري. أما Modern C++ فقد قدّمت نموذج ذاكرة، وخيوط تنفيذ، وذريّات، وآليات مزامنة، مما أتاح تصميمًا كائنيًا آمنًا في البيئات المتوازية.

١.٣.١١ الخلاصة

يمثل الانتقال من C++ الكلاسيكية إلى Modern C++ تحوّلاً من صحة قائمة على الانضباط إلى صحة تفرضها اللغة نفسها.

لا تتخلى Modern C++ عن البرمجة كائنية التوجه، بل تُنقّيها. ومن خلال دمج OOP مع RAII، ودلالات الملكية الصريحة، ودلالات النقل، والتجريد وقت الترجمة، تمكّن C++ المهندسين من بناء أنظمة فعّالة، ومتينة، وقابلة للتوسع.

إن فهم هذه الفروقات أمر جوهري لكتابة C++ حديثة تحترم إرشادات ISO C++ Core Guidelines ومتطلبات هندسة البرمجيات على نطاق واسع.

الفصل ٢: المفاهيم الأساسية للبرمجة كائنية التوجه

- الأصناف والكائنات.
- الوراثة وأنواعها (public، private، protected).
- التغليف والتجريد.
- تعدد الأشكال وأنواعه (وقت الترجمة مقابل وقت التشغيل).

١.٢ الأصناف والكائنات في C++

تتمحور البرمجة كائنية التوجه (OOP) حول تنظيم البرمجيات باستخدام وحدات مسؤولية محددة بوضوح. في C++، تُعبَّر هذه الوحدات من خلال الأصناف والكائنات. توفر الأصناف آلية لتعريف أنواع جديدة تجمع البيانات مع العمليات التي تحافظ على صحتها، بينما تمثل الكائنات تجسيداً فعلياً لتلك الأنواع وقت التشغيل.
في Modern C++، لا تُعد الأصناف والكائنات أدوات بنيوية فحسب، بل تشكل الأساس للتعبير عن الثوابت الداخلية، والملكية، وعمر الكائن، وحدود التجريد. يقدّم هذا القسم هذه المفاهيم باستخدام ممارسات حديثة ومتوافقة مع الإرشادات.

١.١.٢ ما هو الصنف؟

يُعرِّف الصنف في C++ نوعاً مُعرّفاً من قبل المستخدم، ويحدّد:

- البيانات التي يمتلكها
- العمليات التي يمكن تنفيذها على تلك البيانات
- قواعد الوصول التي تحافظ على الصحة

الصنف ليس كائناً بحد ذاته؛ بل وصف يُنشأ منه كائنات. تفرض الأصناف المصمَّمة جيداً الثوابت الداخلية وتُخفي تفاصيل التنفيذ.

الصيغة العامة

```
class TypeName {
public:
    // Public interface

private:
    // Internal representation
```

```
};
```

يشجّع تصميم Modern C++ على الفصل الواضح بين الواجهة والتنفيذ، مع إبقاء أعضاء البيانات خاصة في الغالب.

٢.١.٢ مثال: تعريف صنف

لنفترض صنفًا بسيطًا يمثّل سيارة:

```
#include <string>
#include <iostream>

class Car {
public:
    Car(std::string brand, std::string model, int year)
        : brand_{std::move(brand)},
          model_{std::move(model)},
          year_{year} {}

    void print() const {
        std::cout << "Brand: " << brand_
                  << ", Model: " << model_
                  << ", Year: " << year_ << '\n';
    }

private:
    std::string brand_;
    std::string model_;
    int year_;
};
```

مناقشة يوضح هذا الصنف عدة مبادئ في Modern C++:

- أعضاء البيانات خاصة للحفاظ على الثوابت
- التهيئة تتم عبر المُنشئ
- الالتزام بمبدأ const-correctness للتعبير عن النية
- الأنواع المالكة للموارد تستخدم RAII (مثل std::string)

٣.١.٢ ما هو الكائن؟

الكائن هو تجسيد فعلي لصنف ما. تُخصَّص الذاكرة للكائن عند إنشائه، ويُحكَم عمره بواسطة النطاق وقواعد الملكية.

إنشاء كائن

```
Car myCar{"Toyota", "Camry", 2020};
```

في هذه اللحظة، يُنفَّذ المُنشئ ويصبح الكائن مُهيّأً بالكامل.

٤.١.٢ استخدام الكائنات

```
int main() {
    Car myCar{"Toyota", "Camry", 2020};
    myCar.print();
}
```

يمتلك الكائن myCar حالته، ويكشف عن سلوكه عبر واجهة محددة جيداً.

٥.١.٢ محددات الوصول والتغليف

توفر C++ محددات وصول للتحكم في الرؤية:

- public: جزء من الواجهة
- private: التمثيل الداخلي
- protected: متاح للأصناف المشتقة

لا يعني التغليف إخفاء البيانات اعتباطيًا، بل الحفاظ على الثوابت وتقليل الترابط.

مثال

```
class Car {
public:
    explicit Car(std::string vin)
        : vin_{std::move(vin)} {}

    const std::string& vin() const {
        return vin_;
    }

private:
    std::string vin_;
};
```

هنا يصبح رقم الهيكل (VIN) غير قابل للتغيير بعد الإنشاء، مما يضمن الاتساق.

٦.١.٢ المُنشئات والمُدمّرات

تُنشئ المُنشئات الثوابت، وتُحرِّر المُدمّرات الموارد. في Modern C++ نادرًا ما تُستخدم المُدمّرات مباشرةً لإدارة الذاكرة، إذ تتكفل أنواع RAII بالتنظيف تلقائيًا.

مثال

```
class Logger {
public:
    Logger() {
        std::cout << "Logger initialized\n";
    }

    ~Logger() {
        std::cout << "Logger destroyed\n";
    }
};
```

تُستدعى المُدمّرات تلقائيًا عند انتهاء عمر الكائن.

٧.١.٣ مبدأ RAII وعمر الكائن

تعتمد Modern C++ على مبدأ **Resource Acquisition Is Initialization** (RAII). تمتلك الكائنات الموارد، ويُربط تحرير الموارد بتدمير الكائن.
يوفّر هذا النموذج:

- تنظيفًا حتميًا
- أمانًا في وجود الاستثناءات
- دلالات ملكية واضحة

٨.١.٣ الخلاصة

تشكل الأصناف والكائنات العمود الفقري للتصميم الكائني في C++. في Modern C++، لا تُستخدم هذه المفاهيم لنمذجة كيانات العالم الحقيقي فحسب، بل للتعبير عن الملكية، وفرض الثوابت، وإدارة العمر بأمان.

ومن خلال الالتزام بالتغليف، والتهيئة عبر المُنشئات، ومبادئ RAII، يمكن بناء أنظمة متينة، وقابلة للصيانة، وعالية الكفاءة. وتُمثّل هذه المفاهيم الأساس لموضوعات أكثر تقدمًا كالوراثة، وتعدد الأشكال، والتجريد، التي ستُناقش في الفصول اللاحقة.

٢.٢ الوراثة وأشكالها في C++ (public، protected، private)

تُعد الوراثة من أكثر آليات البرمجة كائنية التوجه (OOP) وضوحًا، لكنها في الوقت نفسه من أكثرها سوء استخدام. في C++، تمكّن الوراثة صنفًا من اشتقاق سلوك من صنف آخر، إلا أن Modern C++ تتعامل مع الوراثة باعتبارها أداة دلالية، لا مجرد آلية لإعادة استخدام الشيفرة.
وفقًا لإرشادات ISO C++ Core Guidelines، يجب أن تُستخدم الوراثة أساسًا لنمذجة قابلية الاستبدال (علاقات "is-a``") وواجهات تعدد الأشكال. وعند إساءة استخدامها، قد تؤدي الوراثة إلى ترابط شديد، وهياكل هرمية هشة، ودلالات ملكية غير واضحة.
يشرح هذا القسم الوراثة في C++ مع التركيز على:

- المعنى المفاهيمي الصحيح
- أنواع علاقات الوراثة
- تأثير محددات الوصول
- أفضل الممارسات الحديثة

١.٢.٢ ما هي الوراثة؟

الوراثة هي آلية يُوسّع من خلالها الصنف المشتق واجهة أو سلوك الصنف الأساس. ويمكن للصنف المشتق أن:

- يعيد استخدام السلوك
- يعيد تعريف الدوال الافتراضية
- يوسّع الوظائف المتاحة

المصطلحات

- الصنف الأساس: يعرّف الواجهة والسلوك المشترك
- الصنف المشتق: يخصّص أو يطبّق تلك الواجهة

الصيغة العامة

```
class Base {
    // Base interface
};

class Derived : access-specifier Base {
    // Specialized behavior
};
```

يحدد محدد الوصول كيفية كشف واجهة الصنف الأساس من خلال الصنف المشتق.

٢.٣.٢ الأنواع المفاهيمية للوراثة

تدعم C++ عدة أشكال بنيوية للوراثة. تصف هذه الأشكال كيفية ارتباط الأصناف، لا كيفية التحكم في الوصول.

١. الوراثة الأحادية
يرث صنف مشتق من صنف أساس واحد.

٢. الوراثة المتعددة
يرث الصنف من أكثر من صنف أساس. وهي قوية، لكنها تتطلب تصميمًا حذرًا لتجنب الغموض والتعقيد.

٣. الوراثة متعددة المستويات
يصبح الصنف المشتق أساسًا لصنف آخر، مُشكِّلاً سلسلة.

٤. الوراثة الهرمية
ترث عدة أصناف مشتقة من نفس الصنف الأساس.

٥. الوراثة الهجينة
مزيج من الأشكال السابقة، وغالبًا ما تتضمن واجهات وmixins.

تشجّع Modern C++ بشدة على تقليل عمق الوراثة وتجنب الهياكل المعقدة ما لم يكن تعدد الأشكال ضروريًا فعلاً.

٣.٢.٢ الوراثة ومحددات الوصول

عند الاشتقاق من صنف أساس، تتيح C++ ثلاثة أنماط وصول: **public**، **protected**، و**private**. لا تغيّر هذه الأنماط الصنف الأساس نفسه، بل تغيّر كيفية كشف واجهته عبر الصنف المشتق.

الوراثة العامة (**public**)

تحافظ الوراثة العامة على واجهة الصنف الأساس وتُنمذج علاقة ``is-a'' الحقيقية. وهي الشكل الوحيد المناسب لتعدد الأشكال.

- تبقى الأعضاء العامة عامة
- تبقى الأعضاء المحمية محمية
- تظل الأعضاء الخاصة غير متاحة

مثال

```
class Animal {
public:
    virtual ~Animal() = default;
    virtual void eat() const {
```

```
        std::cout << "Animal is eating\n";
    }
};

class Dog : public Animal {
public:
    void bark() const {
        std::cout << "Dog is barking\n";
    }
};
```

الاستخدام

```
Animal* a = new Dog{};
a->eat();   // Polymorphic access
delete a;
```

تُعبّر الوراثة العامة عن قابلية الاستبدال: كل Dog هو Animal.

الوراثة الخاصة (private)

تُعبّر الوراثة الخاصة عن إعادة استخدام التنفيذ، لا عن قابلية الاستبدال. تصبح واجهة الصنف الأساس خاصة داخل الصنف المشتق.

- تتحول الأعضاء العامة والمحمية إلى خاصة
- لا يوجد تعدد أشكال

مثال

```
class Engine {
public:
    void start() {
        std::cout << "Engine started\n";
    }
};

class Car : private Engine {
public:
    void drive() {
        start(); // allowed internally
        std::cout << "Car driving\n";
    }
};
```

مناقشة نادرًا ما تُنصح الوراثة الخاصة في Modern C++. فالتركيب (composition) غالبًا بديل أوضح وأكثر أمانًا.

الوراثة المحمية (protected)

تقيّد الوراثة المحمية واجهة الصنف الأساس لتكون متاحة للأصناف المشتقة فقط. وتُستخدم غالبًا داخل الأطر والمكتبات.

- تتحول الأعضاء العامة والمحمية إلى محمية
- لا تكون متاحة لمستخدمي الصنف المشتق

مثال

```
class Employee {
public:
    void work() {
        std::cout << "Employee working\n";
    }
};

class Manager : protected Employee {
public:
    void manage() {
        work(); // accessible
        std::cout << "Manager managing\n";
    }
};
```

هذا الشكل غير شائع في شيفرة التطبيقات.

٤.٢.٢ الوراثة مقابل التركيب

يفضّل تصميم Modern C++ التركيب على الوراثة. ولا ينبغي استخدام الوراثة إلا عندما:

- توجد علاقة "is-a`` حقيقية
- يكون تعدد الأشكال مطلوبًا
- يكون الصنف الأساس مصممًا للوراثة

وفي غير ذلك، يوفّر التركيب:

- ترابطًا أقل

- ملكية أوضح
- قابلية اختبار أفضل

٥.٢.٢ مثال توضيحي: هيكلية تعدد أشكال

```
class Animal {
public:
    virtual ~Animal() = default;
    virtual void speak() const = 0;
};

class Lion final : public Animal {
public:
    void speak() const override {
        std::cout << "Roar\n";
    }
};

class Bird final : public Animal {
public:
    void speak() const override {
        std::cout << "Chirp\n";
    }
};
```

يتميّز هذا التصميم بأنه:

- يستخدم الوراثة العامة
- يعتمد واجهات افتراضية خالصة

- يمنع الاشتقاق غير المقصود لاحقًا

٦.٢.٢ الخلاصة

الوراثة في C++ أداة قوية، لكنها متخصصة وتتطلب انضباطًا. ويستلزم استخدامها الصحيح فهم معناها الدلالي وآثارها الميكانيكية معًا.

- الوراثة العامة: لنمذجة علاقات "is-a`` متعددة الأشكال
- الوراثة الخاصة: لإعادة استخدام التنفيذ
- الوراثة المحمية: لاستخدامات داخلية في الأطر

تشجّع Modern C++ على هياكل سطحية، وواجهات صريحة، وتصميم يبدأ بالتركيب. وعند استخدام الوراثة بانضباط، فإنها تمكّن من بناء أنظمة كائنية تعبيرية، وقابلة للتوسّع، وصحيحة من الناحية الهندسية.

٣.٢ التغليف والتجريد في Modern C++

يُعد كلٌّ من التغليف (Encapsulation) والتجريد (Abstraction) ركيزتين مترابطتين ولكن مختلفتين جوهريًا في البرمجة كائنية التوجه (OOP). ومعًا، يوفّران الانضباط البنيوي اللازم لإدارة التعقيد في أنظمة البرمجيات كبيرة الحجم وطويلة العمر.
في Modern C++، لا تُعد هذه المفاهيم أفكارًا نظرية فحسب، بل هي أدوات هندسية تُستخدم لفرض الثوابت، وضبط الاعتماديات، وتعريف واجهات مستقرة عبر قواعد شيفرة تتطور بمرور الوقت.
يستعرض هذا القسم:

- المعنى الدقيق للتغليف
- دور التجريد في تصميم الأنظمة
- كيفية تعبير Modern C++ عن كلا المفهومين
- أوجه الاختلاف في الهدف والاستخدام

١.٣.٢ التغليف

التغليف هو ممارسة ربط البيانات مع العمليات التي تحافظ على صحتها، مع منع التلاعب الخارجي المباشر بالحالة الداخلية للكائن.
لا يقتصر التغليف على جعل الأعضاء خاصة (private) فقط. بل يتمثل هدفه الحقيقي في فرض ثوابت الصنف (Class Invariants)، أي الشروط التي يجب أن تبقى صحيحة دائمًا ليظل الكائن في حالة صالحة.
في C++، يتحقق التغليف من خلال:

- التحكم في الوصول (private، protected، public)
- تصميم دقيق للدوال العضوية
- قواعد صريحة للملكية وعمر الكائن

الخصائص الأساسية للتغليف

- لا يمكن تعديل الحالة الداخلية بشكل عشوائي
- تمر جميع التغييرات عبر عمليات مضبوطة
- تُمنع الحالات غير الصالحة منذ لحظة الإنشاء

مثال على التغليف: فرض الثوابت

```
#include <string>
#include <stdexcept>

class Employee {
public:
    explicit Employee(std::string name, int age)
        : name_(std::move(name)), age_(age)
    {
        if (age_ <= 0) {
            throw std::invalid_argument("Invalid age");
        }
    }

    const std::string& name() const noexcept {
        return name_;
    }

    int age() const noexcept {
        return age_;
    }
```

```
    void promote() noexcept {
        ++level_;
    }

private:
    std::string name_;
    int age_;
    int level_{1};
};
```

في هذا المثال:

- الأعضاء البيانية خاصة ولا يمكن الوصول إليها خارجيًا
- تُفرض صلاحية الكائن أثناء الإنشاء
- لا توجد دوال ضبط (setters) تسمح بحالة غير صالحة

هنا، يحمي التغليف صحة الكائن، لا مجرد إخفاء البيانات.

٢.٣.٢ التجريد

التجريد هو عملية تعريف ماذا يفعل الكائن مع إخفاء كيف يحقق ذلك. ويسمح للمستخدمين بالتعامل مع الأنظمة عبر واجهات مستقرة دون الاعتماد على تفاصيل التنفيذ.
يقلل التجريد العبء الذهني، ويفصل مكونات النظام عن بعضها، مما يجعل الأنظمة أسهل في التوسعة والاستبدال والتحليل.
في Modern C++، يُعبَّر عن التجريد من خلال:

- الأصناف الأساس المجردة
- الدوال الافتراضية الخالصة
- الواجهات غير المالكة

مثال على التجريد: تصميم قائم على الواجهات

```cpp
#include <iostream>

class Shape {
public:
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};
```

```cpp
class Circle final : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle\n";
    }
};

class Rectangle final : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle\n";
    }
};
```

```cpp
void render(const Shape& shape) {
    shape.draw();
}
```

يضمن هذا التجريد أن:

- يعتمد العملاء على السلوك فقط لا على التنفيذ
- يمكن إضافة أشكال جديدة دون تعديل الشيفرة الحالية
- يتيح تعدد الأشكال الاستبدال أثناء التشغيل

٣.٣.٢ التغليف مقابل التجريد: الفرق المفاهيمي

رغم اقترانهما غالبًا، يحل التغليف والتجريد مشكلتين مختلفتين:

- التغليف يحمي الصحة الداخلية للكائن
- التجريد يدير التعقيد الخارجي

يجيب التغليف عن السؤال:

كيف نمنع سوء الاستخدام ونحافظ على الثوابت؟

ويجيب التجريد عن السؤال:

كيف نسمح بالاستخدام دون كشف تفاصيل التنفيذ؟

قد يكون الصنف مغلفًا جيدًا دون أن يكون مجردًا، وقد تكون الواجهة مجردة تمامًا دون أن تكشف أي بيانات.

٤.٣.٢ منظور التصميم في **Modern C++**

تشجّع Modern C++ على:

- تغليف قوي مع أقل قدر ممكن من القابلية للتغيير
- تجريد عبر واجهات ضيقة ومستقرة
- دلالات صريحة للملكية وعمر الكائن

يُثبَّط الإفراط في استخدام دوال الجلب والضبط عندما تكون مجرد كشف للحالة الداخلية، ويُفضَّل بدلاً من ذلك تصميم واجهات موجهة للسلوك.

٥.٣.٢ الخلاصة

يُعد التغليف والتجريد مبدأين متكاملين لكنهما مختلفان:

- التغليف يحمي صحة الكائن وسلامته
- التجريد يبسّط التفاعل ويقلل الترابط

يركّز التغليف إلى الداخل، بحماية الثوابت والحالة الداخلية، بينما يركّز التجريد إلى الخارج، محدداً ما يُسمح للعملاء برؤيته واستخدامه.
إن إتقان هذين المفهومين يمكّن من تصميم أنظمة C++ متينة، وقابلة للصيانة، وقابلة للتطور—وهي مهارة أساسية للممارسة الاحترافية في Modern C++.

٤.٢ تعدد الأشكال وأنواعه (وقت الترجمة مقابل وقت التشغيل) في C++

يُعد تعدد الأشكال (Polymorphism) أحد الأعمدة الأساسية في البرمجة كائنية التوجه (OOP) في C++. إذ يتيح لواجهة واحدة أن تمثل عدة أشكال تنفيذية مختلفة، مما يسمح بكتابة شيفرة عامة، قابلة للتوسعة، وسهلة إعادة الاستخدام.
في جوهره، يمكّن تعدد الأشكال من التعامل مع كائنات من أنواع مختلفة من خلال نوع أساس مشترك، مع الحفاظ على السلوك الخاص بكل نوع. وتُعد هذه القدرة ضرورية لبناء الأنظمة القابلة للتوسعة، والأطر البرمجية (Frameworks)، والبنى المعمارية واسعة النطاق.
يتناول هذا القسم:

١. مفهوم تعدد الأشكال وهدفه
٢. الشكلين الرئيسيين لتعدد الأشكال في C++
٣. أمثلة عملية لكل نوع
٤. اعتبارات الأداء، والأمان، والتصميم
٥. خلاصة تقنية موجزة

١.٤.٢ ما هو تعدد الأشكال؟

يرجع مصطلح تعدد الأشكال إلى اللغة اليونانية، ويعني «أشكالاً متعددة». وفي تصميم البرمجيات، يشير إلى قدرة كائنات مختلفة على الاستجابة لاستدعاء الدالة نفسها بطرق مختلفة.
في C++، يتيح تعدد الأشكال:

- كتابة الشيفرة اعتماداً على التجريدات بدل الأنواع الملموسة
- استبدال المكونات دون تعديل الشيفرة المستدعية

- توسيع السلوك عبر الوراثة أو القوالب
- تقليل الترابط وتحسين قابلية الصيانة

تدعم C++ نوعين مختلفين جذريًا من تعدد الأشكال:

- تعدد الأشكال وقت الترجمة (Compile-time / Static Polymorphism)
- تعدد الأشكال وقت التشغيل (Run-time / Dynamic Polymorphism)

لكل نوع هدف مختلف، وله مفاضلات خاصة من حيث الأداء والتصميم.

٢.٤.٢ أنواع تعدد الأشكال في C++

تعدد الأشكال وقت الترجمة (التعدد الساكن)

تعدد الأشكال وقت الترجمة يُحسم بالكامل أثناء عملية الترجمة. إذ يحدد المترجم أي دالة أو عملية سيتم استخدامها اعتماداً فقط على بنية الشيفرة والأنواع والمعاملات.
يمتاز هذا النوع بأنه يضيف صفر كلفة زمنية أثناء التشغيل، ولذلك يُفضَّل بشدة في المسارات الحساسة للأداء.
ويُنفَّذ باستخدام:

- التحميل الزائد للدوال
- التحميل الزائد للمعاملات
- القوالب (Templates) – التعدد البارامتري

التحميل الزائد للدوال يسمح التحميل الزائد بتعريف عدة دوال تحمل الاسم نفسه مع اختلاف نوع أو عدد أو ترتيب المعاملات.

مثال: التحميل الزائد للدوال

```cpp
#include <iostream>
#include <string>

class Printer {
public:
    void print(int value) {
        std::cout << "Integer: " << value << std::endl;
    }

    void print(double value) {
        std::cout << "Double: " << value << std::endl;
    }

    void print(const std::string& value) {
        std::cout << "String: " << value << std::endl;
    }
};

int main() {
    Printer p;
    p.print(10);
    p.print(3.14);
    p.print("Hello C++");
}
```

ملاحظات أساسية:

- يتم اختيار الدالة أثناء الترجمة
- يعتمد الاختيار على أنواع المعاملات

- تؤدي الحالات الغامضة إلى أخطاء ترجمة

التحميل الزائد للمعاملات يسمح التحميل الزائد للمعاملات باستخدام المعاملات المدمجة مع الأنواع المعرفة من قبل المستخدم مع الحفاظ على الصياغة الطبيعية.

مثال: التحميل الزائد للمعاملات

```cpp
#include <iostream>

class Complex {
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

int main() {
    Complex c1(2.5, 1.5);
    Complex c2(1.5, 2.5);
    Complex c3 = c1 + c2;
```

```
    c3.display();
}
```

إرشادات تصميمية:

- يجب أن يحافظ المعامل على معناه البديهي
- تجنب السلوك المفاجئ أو غير المتوقع
- فضّل الدوال المسماة عندما يكون القصد غير واضح

تعدد الأشكال وقت التشغيل (التعدد الديناميكي)

تعدد الأشكال وقت التشغيل يُحسم أثناء تنفيذ البرنامج، ويُنفَّذ باستخدام الوراثة والدوال الافتراضية. وعلى عكس التعدد وقت الترجمة، تعتمد الدالة المستدعاة على النوع الديناميكي للكائن، لا على النوع الساكن للمؤشر أو المرجع.
يتيح ذلك:

- الاستبدال الديناميكي للمكونات
- المعماريات القائمة على الإضافات (Plugins)
- تصميم الأطر والواجهات

الدوال الافتراضية تُمكِّن الدالة الافتراضية من الربط الديناميكي عبر مؤشر أو مرجع إلى الصنف الأساس.

مثال: تعدد الأشكال وقت التشغيل

```
#include <iostream>

class Shape {
```

```
public:
    virtual ~Shape() = default;

    virtual void draw() const {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    Shape* shapes[] = { new Circle(), new Rectangle() };

    for (const Shape* s : shapes) {
        s->draw();
    }
```

```
    for (Shape* s : shapes) {
        delete s;
    }
}
```

ملاحظات مهمة:

- يتم اختيار الدالة أثناء التشغيل عبر جدول افتراضي (vtable)
- يسمح بتغيير السلوك دون تعديل الشيفرة المستدعية
- يتطلب وجود دالة افتراضية واحدة على الأقل في الصنف الأساس

٣.٤.٢ الفروق الأساسية بين التعدد وقت الترجمة ووقت التشغيل

تعدد الأشكال وقت الترجمة	تعدد الأشكال وقت التشغيل
يُحسم أثناء الترجمة	يُحسم أثناء التنفيذ
يعتمد على التحميل الزائد والقوالب	يعتمد على الوراثة والدوال الافتراضية
دون كلفة زمنية أثناء التشغيل	يتطلب ربطًا ديناميكيًا
أسرع وأكثر قابلية للتنبؤ	أكثر مرونة وقابلية للتوسعة
تُكتشف الأخطاء مبكرًا	يُحدد السلوك أثناء التشغيل

٤.٤.٢ اعتبارات التصميم والأداء

- فضّل التعدد وقت الترجمة في المسارات الحساسة للأداء
- استخدم التعدد وقت التشغيل عندما يجب أن يتغير السلوك ديناميكيًا
- تجنب سلاسل الوراثة العميقة
- احرص دائمًا على جعل مُدمِّر الصنف الأساس افتراضيًا
- استخدم override لاكتشاف الأخطاء أثناء الترجمة

٥.٤.٢ الخلاصة

- يمكّن تعدد الأشكال من كتابة شيفرة C++ مرنة وقابلة لإعادة الاستخدام
- يوفر التعدد وقت الترجمة كفاءة وأمانًا عاليين
- يتيح التعدد وقت التشغيل سلوكًا ديناميكيًا واستبدالاً مرنًا
- يوازن التصميم الاحترافي في C++ بين النوعين بوعي وانضباط

الفصل ٣: ميزات البرمجة الكائنية الحديثة في Modern C++

- قوائم التهيئة (Initializer Lists).
- البُنّاءات والمُدمِّرات (الافتراضي، النسخ، النقل).
- دلالات النقل ومراجع Rvalue.
- المؤشرات الذكية (unique_ptr, shared_ptr, weak_ptr) وإدارة ذاكرة الكائنات.
- البُنّاءات المفوَّضة والموروثة.
- دوال Lambda واستخدامها في البرمجة الكائنية.

١.٣ قوائم التهيئة

تُعد قوائم التهيئة (Initializer Lists) ميزةً أساسية في لغة C++، إذ تُحدِّد الكيفية التي تُنشأ بها الكائنات وتُوضَع في حالة صالحة منذ لحظة وجودها الأولى. وهي تلعب دورًا محوريًا في الأداء، والصحة البرمجية، ووضوح التصميم، خصوصًا في الشيفرات الكائنية وإدارة الموارد.
في Modern C++، لم تعد قوائم التهيئة مجرد اختصارٍ نحوي، بل أصبحت الآلية الأساسية والمُفضَّلة لتهيئة أعضاء الأصناف والأصناف الأساسية.
يتناول هذا القسم:

١. ماهية قوائم التهيئة ومتى تُنفَّذ
٢. لماذا تُعد أساسية في Modern C++
٣. كيفية عملها عمليًا
٤. تهيئة الأعضاء، وتهيئة الأصناف الأساسية، والحاويات
٥. الأخطاء الشائعة، وقواعد الترتيب، وأفضل الممارسات

١.١.٣ ما هي قوائم التهيئة؟

قائمة التهيئة هي قائمة تعبيرات تُستخدم لتهيئة أعضاء الصنف والأصناف الأساسية قبل تنفيذ جسم البُنّاء.
تأتي قائمة التهيئة مباشرة بعد قائمة معاملات البُنّاء، ويبدأ تعريفها بعلامة النقطتين (:).

الصيغة العامة:

```
ClassName::ClassName(parameters)
    : member1(value1),
      member2(value2),
      BaseClass(baseArgs)
```

```
{
    // Constructor body
}
```

خصائص أساسية:

- تتم التهيئة قبل تنفيذ جسم البُنّاء
- تُهيَّأ الأعضاء مباشرة، لا عن طريق الإسناد
- تُهيَّأ الأصناف الأساسية قبل أعضاء الصنف المشتق
- يتبع ترتيب التهيئة ترتيب التصريح عن الأعضاء، لا ترتيب القائمة

قوائم التهيئة تُحدِّد كيفية نشوء الكائن، لا ما يحدث له بعد أن يكون قد وُجد بالفعل.

٣.١.٢ أهمية قوائم التهيئة

ليست قوائم التهيئة اختيارية في كثير من الحالات، وهي موصى بها بشدة في جميع البُنّاءات تقريبًا.

الكفاءة

من دون قائمة تهيئة:

- يُنشأ العضو إنشاءً افتراضيًا
- ثم يُعاد إسناد قيمة جديدة له داخل جسم البُنّاء

أما مع قائمة التهيئة:

- يُنشأ العضو مرة واحدة مباشرة بالقيمة الصحيحة

وهذا يتجنب العمل غير الضروري، ويحسن الأداء، خصوصًا مع الأنواع غير البسيطة مثل الحاويات، والسلاسل النصية، والمؤشرات الذكية.

الأعضاء الثابتة والمراجع

الأعضاء الثابتة (const) والمراجع:

- يجب تهيئتها عند الإنشاء
- لا يمكن إسناد قيم لها لاحقًا

ولذلك تكون قوائم التهيئة إلزامية في هذه الحالات.

تهيئة الأصناف الأساسية

يجب إنشاء الصنف الأساسي بالكامل قبل تنفيذ جسم بُنّاء الصنف المشتق. ولا توجد آلية تمرير معاملات إلى بُنّاء الصنف الأساسي سوى عبر قائمة التهيئة.

الصحة والأمان

تساعد قوائم التهيئة على:

- منع إنشاء كائنات مهيأة جزئيًا
- تثبيت الثوابت (Invariants) مبكرًا
- تقليل الأخطاء الخفية المرتبطة بحالات غير مهيأة

٣.١.٣ استخدام قوائم التهيئة

تهيئة الأعضاء الأساسية

مثال: تهيئة المتغيرات العضوية

```
#include <iostream>
```

```cpp
class Rectangle {
    int width;
    int height;

public:
    Rectangle(int w, int h)
        : width(w), height(h)
    {}

    void display() const {
        std::cout << "Width: " << width
                  << ", Height: " << height << std::endl;
    }
};

int main() {
    Rectangle rect(10, 20);
    rect.display();
}
```

المخرجات:

```
Width: 10, Height: 20
```

هنا تم تهيئة كلا العضوين مباشرة، دون إنشاء افتراضي ثم إعادة إسناد.

قاعدة ترتيب التهيئة

قاعدة مهمة:

تُهيَّأ الأعضاء حسب ترتيب التصريح عنها في الصنف، وليس حسب ترتيبها في قائمة التهيئة.

مخالفة هذه القاعدة قد تؤدي إلى أخطاء دقيقة يصعب اكتشافها.
أفضل ممارسة: اكتب قائمة التهيئة دائمًا بالترتيب نفسه الذي صُرِّح به عن الأعضاء.

تهيئة الأصناف الأساسية

عند وجود وراثة، يجب أن يُنشأ الصنف الأساسي أولاً.

مثال: تهيئة الصنف الأساسي

```
#include <iostream>

class Shape {
protected:
    int area;

public:
    explicit Shape(int a) : area(a) {}

    void displayArea() const {
        std::cout << "Area: " << area << std::endl;
    }
};

class Square : public Shape {
    int sideLength;

public:
```

```
    Square(int side)
        : Shape(side * side), sideLength(side)
    {}

    void display() const {
        std::cout << "Side Length: " << sideLength << std::endl;
        displayArea();
    }
};

int main() {
    Square square(5);
    square.display();
}
```

المخرجات:

```
Side Length: 5
Area: 25
```

نقاط أساسية:

- يُهيَّأ الصنف الأساسي Shape أولاً
- ثم تُهيَّأ أعضاء الصنف المشتق
- ويُنفَّذ جسم البُنّاء أخيرًا

تهيئة الحاويات والأنواع المعقدة

تُعد قوائم التهيئة مثالية لتهيئة أنواع مكتبة Standard Library.

مثال: تهيئة std::vector

```
#include <iostream>
#include <vector>

class DataContainer {
    std::vector<int> data;

public:
    explicit DataContainer(std::vector<int> values)
        : data(std::move(values))
    {}

    void display() const {
        for (int v : data) {
            std::cout << v << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    DataContainer container({1, 2, 3, 4, 5});
    container.display();
}
```

المخرجات:

```
1 2 3 4 5
```

هذا الأسلوب يتجنب النسخ الزائد ويُنشئ الحاوية مباشرة في صورتها النهائية.

٤.١.٣ أخطاء شائعة

- تهيئة الأعضاء داخل جسم البُنّاء بدل قائمة التهيئة
- تجاهل تحذيرات ترتيب التهيئة
- نسيان تهيئة الصنف الأساسي صراحة
- تنفيذ منطق معقد داخل تعبيرات التهيئة

٥.١.٣ أفضل الممارسات

- فضّل دائمًا قوائم التهيئة على الإسناد
- التزم بترتيب التصريح عن الأعضاء
- استخدم قوائم التهيئة في جميع البُنّاءات
- اجعل تعبيرات التهيئة بسيطة وحتمية
- اربط قوائم التهيئة بمبادئ RAII

٦.١.٣ الخلاصة

تُعد قوائم التهيئة حجر الأساس في دلالات الإنشاء في Modern C++.
فهي توفر:

- تهيئة مباشرة وفعّالة للأعضاء
- دعمًا إلزاميًا للأعضاء الثابتة والمراجع
- إنشاءً صحيحًا للأصناف الأساسية
- شيفرة أنظف، وأكثر أمانًا، وأسهل صيانة

إتقان قوائم التهيئة ضرورة لكل من يسعى إلى كتابة شيفرة C++ احترافية، عالية الأداء، وصحيحة—خصوصًا في الأنظمة الكائنية والمعتمدة على إدارة الموارد.

٢.٣ البُنّاءات والمُدمِّرات (الافتراضي، النسخ، النقل)

في Modern C++، تُحدِّد البُنّاءات والمُدمِّرات دورة حياة الكائن بالكامل. فهي الآلية الأساسية التي تُمكِّن الكائنات من اكتساب الموارد وتحريرها، وتثبيت الثوابت (Invariants)، والتفاعل بأمان مع النظام المحيط.

إن الفهم العميق للبُنّاءات والمُدمِّرات ضروري من أجل:

- إدارة الموارد بشكل صحيح (RAII)
- كتابة شيفرة حساسة للأداء
- النسخ الآمن ونقل الكائنات
- بناء أنظمة قابلة للصيانة وآمنة من حيث الاستثناءات

يتناول هذا القسم:

١. نظرة عامة على البُنّاءات والمُدمِّرات
٢. البُنّاءات الافتراضية
٣. بُنّاءات النسخ
٤. بُنّاءات النقل
٥. المُدمِّرات
٦. أفضل الممارسات وقواعد التصميم الحديثة

١.٢.٣ نظرة عامة على البُنّاءات والمُدمِّرات

البُنّاءات

البُنّاء (Constructor) هو دالة عضوية خاصة تُستدعى تلقائيًا عند إنشاء الكائن. ووظيفته الأساسية هي تهيئة الكائن ووضعه في حالة صالحة وقابلة للاستخدام.

خصائص أساسية:

- يحمل الاسم نفسه للصنف
- لا يمتلك نوع إرجاع
- يمكن تحميله تحميلاً زائداً
- يُنفَّذ مرة واحدة فقط لكل كائن

أنواع البُنّاءات الشائعة:

- البُنّاء الافتراضي
- بُنّاء النسخ
- بُنّاء النقل

المُدمِّرات

المُدمِّر (Destructor) هو دالة عضوية خاصة تُستدعى تلقائيًا عند تدمير الكائن. ومهمته تحرير أي موارد يملكها الكائن.
خصائص أساسية:

- يُسمّى على الشكل ~ClassName
- لا يستقبل معاملات
- لا يمتلك نوع إرجاع
- يُستدعى مرة واحدة فقط لكل كائن

تُعد المُدمِّرات العمود الفقري لمبدأ **RAII (Resource Acquisition Is Initialization)**.

٢.٢.٣ البُنّاءات الافتراضية

البُنّاء الافتراضي هو بُنّاء يمكن استدعاؤه من دون تمرير أي معاملات.
سلوك المُترجم:

- يُنشئه تلقائيًا إذا لم يُعرَّف أي بُنّاء
- يُلغى إن وُجد أي بُنّاء مُعرَّف من قبل المستخدم

مثال: بُنّاء افتراضي

```cpp
#include <iostream>

class MyClass {
    int value;

public:
    MyClass() : value(0) {
        std::cout << "Default constructor called. Value: "
                  << value << std::endl;
    }
};

int main() {
    MyClass obj;
}
```

المخرجات:

```
Default constructor called. Value: 0
```

ملاحظة أساسية: تضمن قوائم التهيئة أن يُنشأ العضو مباشرة في حالة صحيحة منذ البداية.

٣.٢.٣ بُنّاءات النسخ

بُنّاء النسخ يُستخدم لتهيئة كائن جديد كنسخة من كائن موجود.

التوقيع القياسي:

```
ClassName(const ClassName& other);
```

متى يُستدعى؟

- تمرير الكائنات بالقيمة
- إرجاع الكائنات بالقيمة
- تهيئة كائن من كائن آخر

مثال: بُنّاء النسخ

```
#include <iostream>

class MyClass {
    int value;

public:
    explicit MyClass(int v) : value(v) {}

    MyClass(const MyClass& other)
        : value(other.value) {
        std::cout << "Copy constructor called. Value: "
                  << value << std::endl;
    }
```

```
    int getValue() const { return value; }
};

int main() {
    MyClass obj1(10);
    MyClass obj2 = obj1;
}
```

المخرجات:

```
Copy constructor called. Value: 10
```

ملاحظة مهمة: النسخ السطحي للموارد المملوكة قد يؤدي إلى أخطاء تحرير مزدوج ما لم يُدار بعناية.

٤.٢.٣ بُنّاءات النقل

بُنّاء النقل ينقل ملكية الموارد من كائن مؤقت (rvalue) إلى كائن جديد.

التوقيع القياسي:

```
ClassName(ClassName&& other) noexcept;
```

أهمية بُنّاءات النقل:

- تجنب النسخ العميق المكلف
- تمكين إعادة تخصيص الحاويات بكفاءة
- تحسين الأداء في Modern C++

مثال: بُنّاء النقل

```cpp
#include <iostream>
#include <vector>

class MyClass {
    std::vector<int> data;

public:
    MyClass() = default;

    MyClass(std::vector<int>&& vec)
        : data(std::move(vec)) {
        std::cout << "Move constructor called." << std::endl;
    }

    void display() const {
        for (int v : data) {
            std::cout << v << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    std::vector<int> vec{1,2,3,4,5};
    MyClass obj(std::move(vec));
    obj.display();
}
```

المخرجات:

```
Move constructor called.
1 2 3 4 5
```

بعد النقل، يبقى الكائن المصدر صالحًا، ولكن في حالة غير محددة.

٥.٢.٣ المُدمِّرات

تتولى المُدمِّرات تحرير الموارد التي تم الحصول عليها أثناء الإنشاء.
المسؤوليات الشائعة:

- تحرير الذاكرة الديناميكية
- إغلاق الملفات
- تحرير الأقفال
- إعادة الموارد للنظام

مثال: مُدمِّر

```cpp
#include <iostream>

class MyClass {
    int* data;

public:
    explicit MyClass(int v)
        : data(new int(v)) {
        std::cout << "Constructor called." << std::endl;
```

```
    }

    ~MyClass() {
        delete data;
        std::cout << "Destructor called. Memory freed."
                  << std::endl;
    }
};

int main() {
    MyClass obj(10);
}
```

المخرجات:

```
Constructor called.
Destructor called. Memory freed.
```

نصيحة حديثة: فضّل استخدام المؤشرات الذكية بدل المؤشرات الخام للاستغناء عن المُدمِّرات اليدوية.

٦.٢.٣ قواعد التصميم الحديثة

قاعدة الصفر (Rule of Zero)

إذا كان الصنف لا يدير موارد مباشرة:

- لا تُعرِّف مُدمِّرًا
- ولا بُنّاءات نسخ أو نقل
- دع المُترجم يولِّد كل شيء

قاعدة الخمسة **(Rule of Five)**

إذا عرَّفت أحد العناصر التالية:

- المُدمِّر
- بُنّاء النسخ
- معامل الإسناد بالنسخ
- بُنّاء النقل
- معامل الإسناد بالنقل

فغالبًا ما تحتاج إلى تعريفها جميعًا.

٧.٢.٣ الخلاصة

تُحدِّد البُنّاءات والمُدمِّرات صحة دورة حياة الكائن وأمانها في C++.

- البُنّاءات الافتراضية تُنشئ الحالة الابتدائية
- بُنّاءات النسخ تُعرِّف دلالات التكرار
- بُنّاءات النقل تُمكِّن نقل الملكية بكفاءة
- المُدمِّرات تضمن تنظيفًا حتميًا للموارد

إن إتقان هذه الآليات شرط أساسي لكتابة شيفرة C++ فعّالة، وآمنة من حيث الاستثناءات، ومتوافقة مع RAII ومعايير الهندسة البرمجية الاحترافية.

٣.٣ دلالات النقل ومراجع Rvalue

تُعد دلالات النقل (Move Semantics) ومراجع **Rvalue** من الركائز الأساسية التي أُدخلت في C++11، وقد أحدثت نقلة نوعية في الأداء وإدارة الموارد في Modern C++. فبدلاً من نسخ الموارد مثل مخازن الذاكرة، وواصفات الملفات، والمقابس، تسمح دلالات النقل بنقل الملكية مباشرة، مما يلغي العمليات المكلفة وغير الضرورية.
تُعد هذه الآليات ضرورية لكتابة أنظمة عالية الأداء، وآمنة من حيث الاستثناءات، وقابلة للتوسع، خصوصًا عند التعامل مع كائنات كبيرة أو حاويات مكتبة STL.
يتناول هذا القسم:

١. مقدمة في دلالات النقل ومراجع Rvalue
٢. فهم فئات القيم (Value Categories) ومراجع Rvalue
٣. بُنّاءات النقل
٤. معاملات الإسناد بالنقل
٥. أفضل الممارسات وإرشادات Modern C++

١.٣.٣ مقدمة في دلالات النقل ومراجع Rvalue

دلالات النقل هي تقنية برمجية تُمكِّن من نقل الموارد بكفاءة من كائن إلى آخر بدلاً من تكرارها. ويُستفاد منها أساسًا عند التعامل مع الكائنات المؤقتة التي توشك على الانتهاء من عمرها.
الفكرة الأساسية:

إذا كان الكائن على وشك الزوال، فيمكن الاستيلاء على موارده بأمان بدلاً من نسخها.

أما مراجع **Rvalue** (&&) فهي آلية اللغة التي تسمح بالتعرّف على هذه الكائنات المؤقتة والتعامل معها.

٢.٣.٣ فهم مراجع Rvalue وفئات القيم

تنتمي تعبيرات C++ إلى فئات مختلفة تُسمى فئات القيم، وهي التي تُحدِّد كيفية ربطها بالمراجع.

فئات القيم

- Lvalue: له هوية ثابتة وعنوان في الذاكرة
- Prvalue: قيمة مؤقتة خالصة
- Xvalue: قيمة على وشك الانتهاء ويمكن نقلها

وتشمل Rvalues كلاً من prvalues وxvalues.

مراجع **Rvalue**

مرجع **Rvalue** يُصرَّح عنه باستخدام && ويستطيع الارتباط بالكائنات المؤقتة.

الصيغة:

```
Type&& variableName;
```

أهمية مراجع **Rvalue**:

- تمكين دلالات النقل
- السماح بتعديل الكائنات المؤقتة
- التمييز بين عمليات النسخ وعمليات النقل

التحميل الزائد بين **Lvalue** و**Rvalue**

```cpp
#include <iostream>

void print(int& x) {
    std::cout << "Lvalue reference: " << x << std::endl;
}

void print(int&& x) {
    std::cout << "Rvalue reference: " << x << std::endl;
}

int main() {
    int a = 10;
    print(a);     // lvalue
    print(20);    // rvalue
}
```

المخرجات:

```
Lvalue reference: 10
Rvalue reference: 20
```

يوضح هذا المثال كيف يختار المُترجم الدالة المناسبة اعتماداً على فئة القيمة.

٣.٣.٣ دلالات النقل وبُنّاءات النقل

بُنّاء النقل يُنشئ كائنًا جديداً عن طريق نقل الموارد من كائن موجود على وشك الانتهاء من عمره.

التوقيع القياسي:

```
ClassName(ClassName&& other) noexcept;
```

أهمية noexcept:

- تمكين تحسينات الحاويات
- شرط أساسي لإعادة التخصيص الآمن في حاويات STL

مثال: بُنّاء النقل

```
#include <iostream>
#include <vector>

class MyClass {
    std::vector<int> data;

public:
    explicit MyClass(std::vector<int>&& vec)
        : data(std::move(vec)) {
        std::cout << "Move constructor called." << std::endl;
    }

    void display() const {
        for (int v : data)
            std::cout << v << " ";
        std::cout << std::endl;
    }
};
```

```
int main() {
    std::vector<int> vec{1,2,3,4,5};
    MyClass obj(std::move(vec));
    obj.display();
    std::cout << "Vector size after move: "
              << vec.size() << std::endl;
}
```

المخرجات:

```
Move constructor called.
1 2 3 4 5
Vector size after move: 0
```

توضيح مهم: الدالة std::move لا تنقل شيئًا بحد ذاتها، بل تُحوِّل الكائن إلى rvalue لتفعيل دلالات النقل.

٤.٣.٣ معاملات الإسناد بالنقل

معامل الإسناد بالنقل ينقل الموارد بين كائنين موجودين مسبقًا.

التوقيع القياسي:

```
ClassName& operator=(ClassName&& other) noexcept;
```

المسؤوليات الأساسية:

- تحرير الموارد الحالية
- نقل الملكية من الكائن المصدر
- إبقاء الكائن المصدر في حالة صالحة

مثال: معامل الإسناد بالنقل

```cpp
#include <iostream>
#include <vector>

class MyClass {
    std::vector<int> data;

public:
    MyClass() = default;

    explicit MyClass(std::vector<int>&& vec)
        : data(std::move(vec)) {}

    MyClass& operator=(MyClass&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
        }
        return *this;
    }

    void display() const {
        for (int v : data)
            std::cout << v << " ";
        std::cout << std::endl;
    }
};

int main() {
    MyClass obj1(std::vector<int>{1,2,3});
```

```
    MyClass obj2(std::vector<int>{4,5,6});

    obj1 = std::move(obj2);

    obj1.display();
    obj2.display();
}
```

المخرجات:

```
4 5 6
```

بعد الإسناد بالنقل، يبقى obj2 صالحًا للاستخدام، لكنه فارغ المحتوى.

٥.٣.٣ أفضل الممارسات والإرشادات الحديثة

- فضّل دلالات النقل في الأنواع المالكة للموارد
- احرص دائمًا على وسم عمليات النقل بـnoexcept
- اتبع قاعدة الصفر أو قاعدة الخمسة
- استخدم std::move بوعي وقصد
- لا تفترض أن الكائنات المنقول منها غير صالحة

٦.٣.٣ الخلاصة

تُعد دلالات النقل ومراجع Rvalue أدوات أساسية في Modern C++.

- مراجع Rvalue (&&) تُمكِّن نقل الموارد

- بُنّاءات النقل تُحسِّن إنشاء الكائنات
- معاملات الإسناد بالنقل تُحسِّن إعادة الإسناد
- تعتمد حاويات STL بشكل كبير على دلالات النقل لتحقيق الأداء العالي

بإتقان هذه الميزات، يصبح المطوّر قادرًا على كتابة شيفرة C++ أسرع، وأكثر أمانًا، وقابلة للتوسع، ومستفيدة بالكامل من فلسفة التصميم الحديثة للغة.

٤.٣ المؤشرات الذكية (unique_ptr، shared_ptr، weak_ptr) وإدارة ذاكرة الكائنات

تُعد المؤشرات الذكية في Modern C++ من أهم الأدوات الهندسية لإدارة الذاكرة الديناميكية بشكل آمن وصحيح. وعلى عكس المؤشرات الخام (pointers raw) التي تتطلب إدارة يدوية باستخدام new وdelete، تقوم المؤشرات الذكية بإدارة دورة حياة الكائنات تلقائيًا، مما يُلغي فئات كاملة من الأخطاء الشائعة مثل: تسرب الذاكرة، والمؤشرات المتدلية، والحذف المزدوج.
لا تُعد المؤشرات الذكية مجرد تحسين اصطلاحي، بل هي تعبير صريح عن ملكية الذاكرة وسلوك عمر الكائن، وهو ما يتوافق تمامًا مع فلسفة RAII والتصميم الحديث في C++.
يتناول هذا القسم:

١. مقدمة في المؤشرات الذكية
٢. unique_ptr
٣. shared_ptr
٤. weak_ptr
٥. أفضل ممارسات إدارة ذاكرة الكائنات
٦. أمثلة تطبيقية
٧. الخلاصة

١.٤.٣ مقدمة في المؤشرات الذكية

المؤشرات الذكية هي أغلفة (wrappers) حول المؤشرات الخام، تقوم بإدارة حذف الكائنات الديناميكية تلقائيًا وفق قواعد واضحة للملكية والعمر.
توفر Modern C++ ثلاثة أنواع رئيسية:

- unique_ptr: ملكية حصرية لكائن واحد فقط.
- shared_ptr: ملكية مشتركة بين عدة مؤشرات باستخدام العدّ المرجعي.
- weak_ptr: مرجع غير مالك لكائن تُديره shared_ptr، ويُستخدم لكسر الدورات المرجعية.

اختيار النوع الصحيح يُعد قرارًا تصميميًا وليس مجرد تفصيل تنفيذي.

٣.٤.٣ unique_ptr

يحافظ unique_ptr على ملكية حصرية لكائن ديناميكي. لا يمكن نسخه، لكن يمكن نقله، مما يضمن وجود مالك واحد فقط في أي لحظة.
الخصائص الأساسية:

- حذف تلقائي للكائن عند الخروج من النطاق
- غير قابل للنسخ
- قابل للنقل لنقل الملكية

الصيغة:

```
std::unique_ptr<Type> ptr(new Type());
std::unique_ptr<Type> ptr = std::make_unique<Type>();
```

مثال تطبيقي:

```
#include <iostream>
#include <memory>

class MyClass {
```

```
public:
    MyClass() { std::cout << "Constructor called." << std::endl; }
    ~MyClass() { std::cout << "Destructor called." << std::endl; }
    void display() const {
        std::cout << "Display method called." << std::endl;
    }
};

int main() {
    std::unique_ptr<MyClass> ptr1 = std::make_unique<MyClass>();
    ptr1->display();

    //
    std::unique_ptr<MyClass> ptr2 = std::move(ptr1);
    ptr2->display();
}
```

المخرجات:

```
Constructor called.
Display method called.
Destructor called.
```

يضمن هذا التصميم عدم وجود أكثر من مالك واحد، مما يمنع الحذف المزدوج تلقائيًا.

٣.٤.٣ shared_ptr

يسمح shared_ptr بامتلاك كائن واحد من قبل عدة مؤشرات في آنٍ واحد. ويُحذف الكائن فقط عندما يصل العدّ المرجعي إلى الصفر.

الخصائص الأساسية:

- عدّ مرجعي تلقائي
- قابل للنسخ
- مناسب للملكية المشتركة

الصيغة:

```
std::shared_ptr<Type> ptr(new Type());
std::shared_ptr<Type> ptr = std::make_shared<Type>();
```

مثال تطبيقي:

```
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "Constructor called." << std::endl; }
    ~MyClass() { std::cout << "Destructor called." << std::endl; }
    void display() const {
        std::cout << "Display method called." << std::endl;
    }
};

int main() {
    std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();
    std::shared_ptr<MyClass> ptr2 = ptr1;
```

```
    ptr1->display();
    ptr2->display();

    std::cout << "Reference count: "
              << ptr1.use_count() << std::endl;
}
```

المخرجات:

```
Constructor called.
Display method called.
Display method called.
Reference count: 2
Destructor called.
```

يُستخدم shared_ptr بحذر، لأن الإفراط فيه قد يُخفي علاقات الملكية الحقيقية.

weak_ptr ٤.٤.٣

يوفر weak_ptr مرجعًا غير مالك لكائن تُديره shared_ptr. ولا يُؤثّر على العدّ المرجعي.
الخصائص الأساسية:

- لا يملك الكائن
- لا يزيد العدّ المرجعي
- يمنع التسرب الناتج عن الدورات المرجعية

الصيغة:

```
std::weak_ptr<Type> weakPtr = sharedPtr;
```

مثال تطبيقي:

```
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "Constructor called." << std::endl; }
    ~MyClass() { std::cout << "Destructor called." << std::endl; }
    void display() const {
        std::cout << "Display method called." << std::endl;
    }
};

int main() {
    std::shared_ptr<MyClass> sharedPtr = std::make_shared<MyClass>();
    std::weak_ptr<MyClass> weakPtr = sharedPtr;

    if (auto locked = weakPtr.lock()) {
        locked->display();
    }

    sharedPtr.reset();

    if (auto locked = weakPtr.lock()) {
```

```
        locked->display();
    } else {
        std::cout << "Object has been deleted." << std::endl;
    }
}
```

المخرجات:

```
Constructor called.
Display method called.
Destructor called.
Object has been deleted.
```

٥.٤.٣ أفضل ممارسات إدارة ذاكرة الكائنات

تُبسّط المؤشرات الذكية إدارة الذاكرة عبر:

- تنظيف تلقائي للموارد
- إلغاء الحذف اليدوي
- تعبير صريح عن الملكية

أفضل الممارسات:

- استخدم unique_ptr كخيار افتراضي
- استخدم shared_ptr فقط عند الحاجة الفعلية
- استخدم weak_ptr لكسر الدورات المرجعية

٦.٤.٣ الخلاصة

تُعد المؤشرات الذكية في Modern C++ عنصرًا أساسيًا لإدارة الذاكرة الآمنة:

- unique_ptr: ملكية حصرية وتنظيف تلقائي
- shared_ptr: ملكية مشتركة بعدّ مرجعي
- weak_ptr: مرجع غير مالك لمنع التسرب

باستخدام المؤشرات الذكية بشكل واعٍ، يصبح كود C++ أكثر أمانًا، وأكثر وضوحًا، وأقرب إلى التصميم الهندسي الاحترافي الذي تستهدفه المعايير الحديثة.

٥.٣ تفويض المُنشئات ووراثتها (Delegating and Inheriting Constructors)

قدّمت Modern C++ (ابتداءً من C++11) ميزتين مهمتين في مجال تهيئة الكائنات: تفويض المُنشئات ووراثة المُنشئات. تهدف هاتان الميزتان إلى تقليل تكرار الكود، وتحسين قابلية الصيانة، وتوحيد منطق التهيئة في الأنظمة الكائنية المعقّدة.
هاتان الميزتان لا تُغيّران سلوك البناء بحد ذاته، بل تُحسّنان طريقة التعبير عنه بما ينسجم مع مبادئ التصميم الحديث في C++.

١.٥.٣ مقدمة

- تفويض المُنشئات (**Constructor Delegation**): يسمح لمُنشئ باستدعاء مُنشئ آخر داخل نفس الصنف، بهدف تجنّب تكرار منطق التهيئة وضمان الاتساق.
- وراثة المُنشئات (**Inheriting Constructors**): تُمكّن الصنف المشتق من استخدام مُنشئات الصنف الأساسي مباشرة دون إعادة تعريفها، عندما تكون آلية التهيئة الأساسية كافية.

٢.٥.٣ تفويض المُنشئات (Constructor Delegation)

التعريف والغرض: تفويض المُنشئات يسمح لمُنشئ داخل الصنف أن يستدعي مُنشئًا آخر من نفس الصنف. بهذا الأسلوب، يتم تجميع منطق التهيئة في نقطة واحدة بدل تكراره في عدة مُنشئات.
هذا الأسلوب:

- يقلل التكرار
- يمنع اختلاف سلوك التهيئة بين المُنشئات
- يُسهّل الصيانة والتعديل

الصيغة والمثال:

```
#include <iostream>
#include <string>

class Person {
public:
    Person() : Person("Unknown", 0) {} //

    Person(const std::string& name, int age)
        : name(name), age(age) {}

    void display() const {
        std::cout << "Name: " << name
                  << ", Age: " << age << std::endl;
    }

private:
    std::string name;
    int age;
};

int main() {
    Person p1;                  //
    Person p2("Alice", 30);     //

    p1.display();
    p2.display();
}
```

الشرح:

- المُنشئ الافتراضي لا يحتوي منطق تهيئة خاص به
- يتم تفويض التهيئة بالكامل إلى المُنشئ الأساسي
- يضمن ذلك اتساق تهيئة الأعضاء في جميع الحالات

هذا الأسلوب يُعد مثاليًا عند وجود أكثر من مُنشئ يمثل حالات مختلفة لتهيئة نفس الكائن.

٣.٥.٣ وراثة المُنشئات (Inheriting Constructors)

التعريف والغرض: وراثة المُنشئات تسمح للصنف المشتق باستخدام مُنشئات الصنف الأساسي مباشرة، دون الحاجة إلى إعادة تعريفها، طالما أن سلوك التهيئة الأساسي مناسب.
تُستخدم هذه الميزة عندما:

- لا يضيف الصنف المشتق متطلبات تهيئة جديدة
- يكون الهدف هو إعادة استخدام واجهة البناء فقط

الصيغة والمثال:

```
#include <iostream>
#include <string>

class Base {
public:
    Base(int x, double y) : x(x), y(y) {}
    Base(const std::string& str)
        : x(static_cast<int>(str.length())), y(0.0) {}

    void display() const {
```

```
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }

private:
    int x;
    double y;
};

class Derived : public Base {
public:
    using Base::Base; //            Base
};

int main() {
    Derived d1(42, 3.14);  //     Base(int, double)
    Derived d2("Hello");   //     Base(const std::string&)

    d1.display();
    d2.display();
}
```

الشرح:

- التعليمة `using Base::Base` تُورّث جميع المُنشئات
- لا حاجة لإعادة تعريف المُنشئات في الصنف المشتق
- يمكن إضافة مُنشئات أو دوال إضافية عند الحاجة

وراثة المُنشئات تُستخدم غالبًا في التصاميم التي تعتمد على الأصناف الأساسية كواجهات أو طبقات بنيوية.

٤.٥.٣ أفضل الممارسات

تفويض المُنشئات:

- استخدمه لتوحيد منطق التهيئة
- تجنّب تكرار نفس التهيئة في أكثر من مُنشئ
- اجعل مُنشئًا واحدًا هو المصدر الحقيقي للتهيئة

وراثة المُنشئات:

- استخدمها عندما لا يضيف الصنف المشتق متطلبات تهيئة جديدة
- لا تستخدمها إذا كان الصنف المشتق يفرض قيودًا أو ثوابت مختلفة

٥.٥.٣ أمثلة إضافية

مثال (1): تفويض المُنشئات

```
#include <iostream>

class Rectangle {
public:
    Rectangle() : Rectangle(0, 0) {} //
    Rectangle(int w, int h)
        : width(w), height(h) {}

    void show() const {
        std::cout << "Width: " << width
                  << ", Height: " << height << std::endl;
    }
```

```
private:
    int width;
    int height;
};

int main() {
    Rectangle r1;
    Rectangle r2(10, 20);

    r1.show();
    r2.show();
}
```

مثال (2): وراثة المُنشئات

```
#include <iostream>
#include <string>

class Shape {
public:
    Shape(int w, int h) : width(w), height(h) {}
    Shape(const std::string& c)
        : width(0), height(0), color(c) {}

    void show() const {
        std::cout << "Width: " << width
                  << ", Height: " << height
                  << ", Color: " << color << std::endl;
```

```
    }

private:
    int width;
    int height;
    std::string color;
};

class Square : public Shape {
public:
    using Shape::Shape;
};

int main() {
    Square s1(15, 15);
    Square s2("Red");

    s1.show();
    s2.show();
}
```

٦.٥.٣ الخلاصة

توفر Modern C++ آليات قوية لتحسين تهيئة الكائنات:

- تفويض المُنشئات يقلل التكرار ويوحّد منطق التهيئة
- وراثة المُنشئات تُبسّط الأصناف المشتقة
- كلاهما يعزز الوضوح وقابلية الصيانة

الاستخدام الواعي لهاتين الميزتين يساعد في كتابة كود أنظف، وأكثر تعبيرًا، ومتوافقًا مع المعايير الهندسية الحديثة في C++.

٦.٣ دوال لامبدا واستخدامها في البرمجة الكائنية-(Lambda Functions in OOP)

تُعد دوال لامبدا إحدى أهم الإضافات التي جاءت مع C++11، وقد غيّرت بشكل جذري طريقة التعبير عن السلوك المؤقت والمحلي داخل البرامج. في سياق البرمجة كائنية التوجه (OOP)، تُستخدم دوال لامبدا لتمثيل السلوك بدقة ووضوح دون الحاجة إلى إنشاء أصناف أو دوال مستقلة.
في Modern C++، لا تُعد لامبدا مجرد اختصار لغوي، بل أداة تصميمية تُستخدم في:

- معالجة الأحداث (Event Handling)
- الاستدعاءات الراجعة (Callbacks)
- المقارنات المخصصة
- الدمج بين OOP والبرمجة الوظيفية

١.٦.٣ مقدمة حول دوال لامبدا

التعريف: دالة لامبدا هي كائن دالي مجهول (Anonymous Function Object) يُعرّف مباشرة داخل السياق الذي يُستخدم فيه، دون الحاجة إلى تعريف دالة مستقلة أو صنف جديد.

الصيغة العامة:

```
[capture](parameters) -> return_type {
    // body
};
```

مكونات الصيغة:

- capture: المتغيرات الملتقطة من النطاق الخارجي

- parameters: معاملات الدالة
- return_type: نوع القيمة المعادة (اختياري)
- body: جسم الدالة

في أغلب الحالات، يستنتج المُصرّف نوع القيمة المعادة تلقائيًا، مما يجعل الصيغة أكثر اختصارًا ووضوحًا.

٢.٦.٣ استخدام دوال لامبدا في OOP

في البرمجة الكائنية الحديثة، تُستخدم دوال لامبدا للتعبير عن السلوك لا الهوية. وهي مناسبة جدًا للحالات التي يكون فيها السلوك:

- محليًا
- قصير العمر
- مرتبطًا بسياق محدد

تشمل الاستخدامات الشائعة:

- معالجة الأحداث
- الاستدعاءات الراجعة
- المقارنات المخصصة
- البرمجة الوظيفية داخل التصميم الكائني

مثال: معالجة الأحداث

```
#include <iostream>
#include <functional>

class Button {
public:
    void setOnClickHandler(std::function<void()> handler) {
        onClick = handler;
    }

    void click() {
        if (onClick) {
            onClick();
        }
    }

private:
    std::function<void()> onClick;
};

int main() {
    Button btn;

    btn.setOnClickHandler([]() {
        std::cout << "Button clicked!" << std::endl;
    });

    btn.click();
}
```

التحليل:

- تم تمرير السلوك (الحدث) بدل تمرير كائن
- لا حاجة لتعريف صنف مخصص لمعالجة الحدث
- السلوك واضح وملاصق لمكان الاستخدام

مثال: الاستدعاءات الراجعة **(Callbacks)**

```
#include <iostream>
#include <vector>
#include <functional>

class Processor {
public:
    void process(const std::vector<int>& data,
                 std::function<void(int)> callback) {
        for (int value : data) {
            callback(value);
        }
    }
};

int main() {
    Processor proc;
    std::vector<int> data{1, 2, 3, 4, 5};

    proc.process(data, [](int value) {
        std::cout << "Processing value: " << value << std::endl;
    });
```

```
}
```

هنا يتم فصل آلية المعالجة عن السلوك المطلوب تنفيذُه، وهو مبدأ تصميمي مهم في الأنظمة المرنة.

مثال: مقارنات مخصصة

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers{5, 3, 8, 1, 2};

    std::sort(numbers.begin(), numbers.end(),
              [](int a, int b) {
                  return a > b;
              });

    for (int n : numbers) {
        std::cout << n << " ";
    }
}
```

تُستخدم لامبدا هنا لتعريف منطق المقارنة دون الحاجة إلى كائن Comparator منفصل.

مثال: البرمجة الوظيفية داخل OOP

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5};
    std::vector<int> results;

    std::transform(numbers.begin(), numbers.end(),
                   std::back_inserter(results),
                   [](int value) {
                       return value * 2;
                   });

    for (int n : results) {
        std::cout << n << " ";
    }
}
```

هذا الأسلوب يُظهر كيف تتكامل دوال لامبدا مع خوارزميات STL لبناء كود تعبيري وواضح.

٣.٦.٣ ميزات متقدمة في دوال لامبدا

الالتقاط بالقيمة مقابل المرجع

```cpp
int x = 10;

auto lambdaVal = [x]() mutable {
    x = 20;
    std::cout << "Inside lambda: " << x << std::endl;
};
```

```
lambdaVal();
std::cout << "Outside lambda: " << x << std::endl;
```

التوضيح:

- الالتقاط بالقيمة ينشئ نسخة محلية
- استخدام mutable يسمح بتعديل النسخة
- القيمة الأصلية تبقى دون تغيير

لامبدا قابلة للتعديل (mutable)

دون mutable، تكون المتغيرات الملتقطة بالقيمة للقراءة فقط. هذه القاعدة تُعزز السلامة الافتراضية في التصميم.

٤.٦.٣ الخلاصة

توفر دوال لامبدا في Modern C++ وسيلة قوية للتعبير عن السلوك داخل التصميم الكائني:

- تُقلل الحاجة إلى الأصناف الصغيرة المؤقتة
- تُقرّب السلوك من موضع الاستخدام
- تُعزز القابلية للقراءة والصيانة
- تُسهل الدمج بين OOP والبرمجة الوظيفية

الاستخدام الواعي لدوال لامبدا يُعد علامة واضحة على تصميم احترافي حديث في C++، خصوصًا في الأنظمة المعتمدة على الأحداث، والواجهات، والخوارزميات العامة.

الفصل ٤: التصميم الكائني المعتمد على C++20

- استخدام Concepts في التصميم الكائني.
- Constexpr وتأثيرها على قرارات التصميم.
- Coroutines ودورها في الكائنات غير المتزامنة.

١.٤ استخدام Concepts في التصميم الكائني

قدّمت C++20 ميزة Concepts كتحسين جوهري للبرمجة القالبية (Generic Programming)، حيث تسمح بفرض قيود صريحة على أنواع القوالب (Template Parameters). تُحوّل هذه الميزة المتطلبات الضمنية إلى عقود واضحة يُجبر المُصرّف على التحقق منها وقت الترجمة.
عند دمج Concepts مع البرمجة الكائنية (OOP)، نحصل على تصميم:

- أكثر وضوحًا
- أكثر أمانًا نوعيًا
- أسهل في الصيانة
- متوافق مع إرشادات ISO C++ Core Guidelines

هذا القسم يستعرض استخدام Concepts داخل التصميم الكائني بأسلوب هندسي احترافي متوافق مع C++20 وحتى C++23.

١.١.٤ مقدمة حول Concepts

التعريف: الـ Concept هو قيد يُعرّف في وقت الترجمة يحدد مجموعة الشروط التي يجب أن يحققها نوع ما ليكون صالحًا للاستخدام مع قالب معيّن.
بدل أن تفشل القوالب برسائل أخطاء غامضة، تُقدّم Concepts:

- توثيقًا ذاتيًا للمتطلبات
- أخطاء ترجمة دقيقة ومباشرة
- فصلاً واضحًا بين الواجهة والتنفيذ

الصيغة العامة:

```
//     Concept
template<typename T>
concept ConceptName = /* compile-time condition */;
```

٢.١.٤ تطبيق Concepts في OOP

١. فرض عقود الواجهات (Interface Contracts)

في التصميم الكائني، تُستخدم الواجهات لضمان التوافق السلوكي بين الكائنات. تسمح Concepts بفرض هذه العقود حتى في القوالب، دون الحاجة إلى وراثة أو دوال افتراضية.

مثال: فرض واجهة حسابية

```
#include <iostream>
#include <type_traits>

// Concept
template<typename T>
concept Arithmetic = std::is_arithmetic_v<T>;

//            Concept
template<Arithmetic T>
class Calculator {
public:
    explicit Calculator(T value) : value_{value} {}

    T add(T other) const { return value_ + other; }
    T subtract(T other) const { return value_ - other; }
```

```
private:
    T value_;
};

int main() {
    Calculator<int> intCalc(10);
    std::cout << intCalc.add(5) << "\n";
    std::cout << intCalc.subtract(3) << "\n";

    // Calculator<std::string> invalid; //
}
```

التحليل:

- تم منع الاستخدام الخاطئ عند الترجمة
- لم نحتج إلى وراثة أو دوال افتراضية
- الواجهة الحسابية أصبحت عقدًا صريحًا

٢. تصميم سمات الأنواع **(Type Traits)** بوضوح

تُسهّل Concepts تعريف سمات الأنواع بطريقة مباشرة وأكثر قابلية للفهم من SFINAE التقليدية.

مثال: Concept للأنواع القابلة للطباعة

```
#include <iostream>
#include <concepts>

// Concept                  std::cout
template<typename T>
concept Printable = requires(T t) {
```

```
    { std::cout << t } -> std::same_as<std::ostream&>;
};

template<Printable T>
void print(const T& value) {
    std::cout << value << "\n";
}

int main() {
    print(42);
    print(3.14);
    print("Hello");

    // print(std::vector<int>{1,2,3}); //
}
```

الفائدة التصميمية:

- توثيق مباشر للمتطلبات
- إزالة الغموض من واجهات القوالب
- أخطاء ترجمة مفهومة وقصيرة

٣. تحسين التخصيص القالبي **(Template Specialization)**

تجعل Concepts نية التخصيص (Specialization) واضحة وصريحة.

مثال: تخصيص باستخدام Concept

```
#include <iostream>
#include <type_traits>
```

```cpp
template<typename T>
concept Integer = std::is_integral_v<T>;

template<typename T>
struct TypeInfo {
    static void print() {
        std::cout << "General type\n";
    }
};

template<Integer T>
struct TypeInfo<T> {
    static void print() {
        std::cout << "Integer type\n";
    }
};

int main() {
    TypeInfo<int>::print();
    TypeInfo<double>::print();
}
```

المغزى:

- السلوك مقيّد بشروط واضحة
- لا التباس في سبب اختيار التخصيص

٤. دمج Concepts لقيود مركّبة

يمكن تركيب عدة Concepts لتمثيل متطلبات تصميم معقدة.

مثال: قيود مركّبة

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
concept Arithmetic = std::is_arithmetic_v<T>;

template<typename T>
concept ArithmeticWithDefault =
    Arithmetic<T> && std::is_default_constructible_v<T>;

template<ArithmeticWithDefault T>
class AdvancedCalculator {
public:
    AdvancedCalculator() : value_{} {}

    void setValue(T v) { value_ = v; }
    T getValue() const { return value_; }

private:
    T value_;
};

int main() {
    AdvancedCalculator<int> calc;
    calc.setValue(42);
    std::cout << calc.getValue() << "\n";
}
```

الفائدة:

- التعبير عن القيود أصبح جزءاً من التصميم
- سهولة التوسعة والصيانة

٣.١.٤ الخلاصة

توفر Concepts في C++20 أداة قوية لتعزيز التصميم الكائني الحديث:

- فرض عقود الواجهات دون وراثة
- تقليل الأخطاء وقت الترجمة
- تحسين وضوح القوالب وصيانتها
- دمج OOP مع البرمجة القالبية بأناقة

اعتماد Concepts في التصميم الكائني يُعد خطوة أساسية نحو كتابة C++ حديثة، آمنة، وقابلة للتطور، ومتوافقة تماماً مع فلسفة C++20 وما بعدها.

٢.٤ constexpr وتأثيرها على التصميم

مقدمة:
قدّمت C++20 تحسينات جوهرية على الميزة constexpr، مما سمح بتوسيع نطاق الحسابات التي يمكن تنفيذها في وقت الترجمة (Compile-Time) سواء للدوال أو للكائنات. هذه الميزة لا تهدف فقط إلى تحسين الأداء، بل تؤثر بشكل مباشر على وضوح التصميم، سلامة الأنواع، وقابلية الصيانة. عند تطبيق constexpr ضمن البرمجة الكائنية (OOP)، تُصبح قرارات التصميم أكثر دقة، وتُبنى الأصناف (Classes) على أسس قابلة للتحقق في وقت الترجمة، مما يقلل الأخطاء ويزيد من موثوقية الأنظمة الكبيرة.
يستعرض هذا القسم constexpr في C++ الحديثة (حتى C++23 مع التركيز على التأثيرات التصميمية العملية والأمثلة الهندسية المتوافقة مع ISO C++ Core Guidelines.

١.٢.٤ ما هي constexpr

constexpr هي مُحدِّد (Specifier) يُشير إلى أن قيمة أو دالة أو كائن يمكن تقييمه في وقت الترجمة إذا توفرت الشروط اللازمة.
في C++20، توسّعت قدرات constexpr لتشمل:

- دوال أكثر تعقيداً
- كائنات مُعرّفة من قبل المستخدم
- دعم جزئي للتعدد الشكلي (Virtual Functions)
- استخدام محدود للذاكرة الديناميكية

هذا التوسّع أزال كثيراً من القيود القديمة، وسمح بدمج constexpr بشكل واقعي داخل تصاميم OOP الحديثة.

٢.٢.٤ فوائد constexpr في تصميم OOP

١. تحسين الأداء
تنفيذ الحسابات في وقت الترجمة يقلل العبء وقت التشغيل، ويُدمج النتائج مباشرة داخل الملف التنفيذي.

٢. ضمانات أقوى وقت الترجمة
تفرض constexpr قيوداً صارمة: لا استثناءات غير معالجة، ولا تأثيرات جانبية غير متوقعة، مما يزيد من موثوقية النظام.

٣. وضوح النية التصميمية
استخدام constexpr يُعبّر صراحةً عن أن منطقًا معينًا يجب أن يكون ثابتًا وقابلاً للتقييم المبكر.

٤. تعزيز البرمجة القالبية
تتكامل constexpr بسلاسة مع القوالب (Templates)، مما يسمح ببناء تصميمات هجينة تجمع بين OOP والحسابات وقت الترجمة.

٣.٢.٤ أمثلة عملية

مثال 1: بُناة constexpr

```
#include <iostream>

class Point {
public:
    constexpr Point(int x, int y) : x_(x), y_(y) {}
    constexpr int getX() const { return x_; }
    constexpr int getY() const { return y_; }

private:
    int x_;
```

```
    int y_;
};

int main() {
    constexpr Point p1(10, 20);
    static_assert(p1.getX() == 10, "Compile-time assertion failed");

    Point p2(15, 25);
    std::cout << "(" << p2.getX() << ", " << p2.getY() << ")\n";
}
```

التحليل:

- الكائن p1 أُنشئ بالكامل وقت الترجمة
- تم التحقق من القيم باستخدام static_assert
- نفس الصنف يمكن استخدامه وقت التشغيل دون تغيير

مثال 2: constexpr مع دوال افتراضية

```
#include <iostream>

class Shape {
public:
    constexpr Shape(double w, double h) : w_(w), h_(h) {}
    virtual constexpr double area() const = 0;
    virtual ~Shape() = default;

protected:
    double w_;
```

```
    double h_;
};

class Rectangle : public Shape {
public:
    constexpr Rectangle(double w, double h) : Shape(w, h) {}
    constexpr double area() const override {
        return w_ * h_;
    }
};

int main() {
    constexpr Rectangle r(10.0, 20.0);
    static_assert(r.area() == 200.0, "Area error");
}
```

ملاحظة تصميمية: رغم دعم constexpr للدوال الافتراضية، إلا أن التقييم الكامل وقت الترجمة يتطلب معرفة النوع الفعلي للكائن.

مثال 3: constexpr مع القوالب

```
#include <array>

template<typename T>
class Matrix {
public:
    constexpr Matrix(std::array<std::array<T,3>,3> v) : values_(v) {}

    constexpr T determinant() const {
```

```
        return values_[0][0]*(values_[1][1]*values_[2][2] -
        ↪  values_[1][2]*values_[2][1])
             - values_[0][1]*(values_[1][0]*values_[2][2] -
             ↪  values_[1][2]*values_[2][0])
             + values_[0][2]*(values_[1][0]*values_[2][1] -
             ↪  values_[1][1]*values_[2][0]);
    }

private:
    std::array<std::array<T,3>,3> values_;
};

int main() {
    constexpr std::array<std::array<int,3>,3> v =
        {{{1,2,3},{0,1,4},{5,6,0}}};

    constexpr Matrix<int> m(v);
    static_assert(m.determinant() == 1);
}
```

الفائدة: دمج OOP مع الحسابات الرياضية وقت الترجمة بكفاءة عالية وبدون تكلفة وقت التشغيل.

مثال 4: Singleton باستخدام constexpr

```
class Logger {
public:
    static constexpr Logger& instance() {
        return instance_;
    }
```

```
    constexpr void log(const char*) const {}

private:
    constexpr Logger() = default;
    static constexpr Logger instance_{};
};

int main() {
    constexpr Logger& log = Logger::instance();
    log.log("Compile-time log");
}
```

تنبيه: هذا النمط مناسب فقط للحالات التي لا تتطلب حالة ديناميكية أو موارد وقت التشغيل.

٤.٢.٤ اعتبارات تصميمية

١. زمن الترجمة
الإفراط في constexpr قد يزيد زمن الترجمة بشكل ملحوظ.

٢. تعقيد الكائنات
ليست كل البُنى معقولة للتقييم الكامل وقت الترجمة.

٣. الاستثناءات
لا يمكن استخدام throw داخل constexpr.

٤. الدوال الافتراضية
الدوال غير الافتراضية أكثر قابلية للتقييم الحتمي وقت الترجمة.

٥.٢.٤ الخلاصة

توفر constexpr في C++20 وما بعدها أداة قوية لإعادة التفكير في تصميم OOP:

- أداء أعلى
- أخطاء أقل
- تصميم أوضح وأكثر تعبيرًا
- تكامل فعّال مع القوالب

عند استخدامها بانضباط، تُعد constexpr عنصرًا أساسيًا في بناء أنظمة C++ حديثة، آمنة، وقابلة للصيانة، ومتوافقة مع ISO C++ Core Guidelines.

٣.٤ الـ Coroutines ودورها في تصميم الكائنات غير المتزامنة

قدّمت C++20 ميزة Coroutines كإضافة جوهرية لتبسيط البرمجة غير المتزامنة. تسمح الـ Coroutines للدوال بإيقاف التنفيذ مؤقتًا (Suspend) ثم استئنافه لاحقًا من نفس النقطة، مما يوفّر نموذجًا واضحًا للتعدد التعاوني (Cooperative Multitasking).
في سياق البرمجة الكائنية (OOP)، تمكّن Coroutines المصمّم من بناء كائنات غير متزامنة بأسلوب نظيف وقابل للصيانة، دون اللجوء إلى Callbacks معقّدة أو آلات حالات يدوية. يستعرض هذا القسم استخدام Coroutines في OOP وفق C++ الحديثة (حتى C++23 وبما يتماشى مع ISO C++ Core Guidelines.

١.٣.٤ فهم Coroutines

التعريف:
Coroutine هي دالة يمكنها تعليق تنفيذها مؤقتًا ثم المتابعة لاحقًا، مع الاحتفاظ بالحالة الداخلية تلقائيًا. هذا الأسلوب يحسّن قابلية القراءة والصيانة مقارنة بالأساليب التقليدية غير المتزامنة.
المفاهيم الأساسية:

- دوال Coroutines: أي دالة تستخدم co_await أو co_yield أو co_return.

```
task<int> example() {
    co_return 42;
}
```

- **Awaitable Types**: كائنات تدعم co_await وتتحكم بآلية التعليق والاستئناف.
- **Tasks**: تمثّل ناتج العمليات غير المتزامنة، وغالبًا ما تُستخدم مع co_await للالتقاط النتائج.

٢.٣.٤ Coroutines في OOP

١. دوال عضو غير متزامنة

تسمح Coroutines بكتابة دوال عضو تقوم بعمليات طويلة دون حجب (Blocking) الخيط الرئيسي، مما يرفع من استجابة النظام.

مثال: قراءة ملف بشكل غير متزامن

```
#include <fstream>
#include <string>
#include <coroutine>
#include <future>

class FileReader {
public:
    struct Awaitable {
        std::ifstream& file;
        std::string buffer;

        bool await_ready() const { return false; }

        void await_suspend(std::coroutine_handle<> h) {
            std::async(std::launch::async, [this, h] {
                std::getline(file, buffer);
                h.resume();
            });
        }

        std::string await_resume() { return buffer; }
    };

    explicit FileReader(const std::string& name) : file(name) {}
```

```
    auto readLine() { return Awaitable{file}; }

private:
    std::ifstream file;
};
```

التحليل:

- الكائن Awaitable يدير التعليق والاستئناف.
- readLine تُرجع واجهة غير متزامنة دون كشف التفاصيل.

٣. تنفيذ عمليات غير متزامنة معقّدة

تُقلل Coroutines بشكل كبير من الشيفرة المتكررة عند التعامل مع الشبكات أو الإدخال/الإخراج.

مثال: محاكاة طلب HTTP غير متزامن

```
#include <string>
#include <coroutine>
#include <thread>
#include <chrono>

class HttpClient {
public:
    struct Awaitable {
        std::string url;
        std::string response;

        bool await_ready() const { return false; }

        void await_suspend(std::coroutine_handle<> h) {
```

```
            std::thread([this, h] {
                std::this_thread::sleep_for(std::chrono::seconds(2));
                response = "Response from " + url;
                h.resume();
            }).detach();
        }

        std::string await_resume() const { return response; }
    };

    auto fetch(const std::string& u) {
        return Awaitable{u};
    }
};
```

الفائدة التصميمية: المنطق غير المتزامن يبدو تسلسليًا وواضحًا، دون تشابك التحكم بالتنفيذ.

٣. دمج Coroutines مع الوراثة والتعدد الشكلي

تتكامل Coroutines بسلاسة مع مبادئ OOP المتقدمة.

مثال: صنف أساسي غير متزامن

```
#include <coroutine>
#include <thread>
#include <chrono>

class AsyncBase {
public:
    struct Awaitable {
        bool await_ready() const { return false; }
```

```
        void await_suspend(std::coroutine_handle<> h) {
            std::thread([h] {
                std::this_thread::sleep_for(std::chrono::seconds(1));
                h.resume();
            }).detach();
        }

        void await_resume() {}
    };

    auto doAsyncWork() { return Awaitable{}; }
};

class Derived : public AsyncBase {
public:
    auto performTask() {
        co_await doAsyncWork();
        //
    }
};
```

المغزى:

- السلوك غير المتزامن يُعرّف في الصنف الأساسي.
- الأصناف المشتقة تعيد استخدامه دون تعقيد إضافي.

٣.٣.٤ اعتبارات عملية

التعامل مع الاستثناءات:
تدعم Coroutines تمرير الاستثناءات، لكن يجب تصميم آلية الـ Promise بعناية لضمان سلامة

النظام.
ملاحظة تصميمية: إهمال إدارة الاستثناءات في سياق غير متزامن قد يؤدي إلى حالات فشل صامتة يصعب تتبعها.

٤.٣.٤ اعتبارات الأداء

رغم أن Coroutines تقلل التعقيد وتحسّن الوضوح، إلا أنها قد:

- تُنشئ تخصيصات على الهيب
- تُضيف كلفة تبديل سياق

لذلك يجب تحليل الاستخدام بعناية، خاصة في الأنظمة ذات المتطلبات الزمنية الصارمة.

٥.٣.٤ الخلاصة

توفر Coroutines في C++20 نموذجًا قويًا لتصميم كائنات غير متزامنة بأسلوب OOP:

- دوال عضو غير حاجبة
- تبسيط العمليات غير المتزامنة المعقّدة
- تكامل طبيعي مع الوراثة والتعدد الشكلي
- تحسين قابلية الصيانة والوضوح

عند استخدامها بانضباط، تُعد Coroutines أداة أساسية لبناء أنظمة C++ حديثة عالية الاستجابة، وآمنة، وقابلة للتوسع حتى C++23.

الفصل ٥: أنماط التصميم في C++

تُعد أنماط التصميم (Design Patterns) خلاصة خبرات تراكمية في هندسة البرمجيات، وتهدف إلى تقديم حلول مجرّبة لمشكلات متكررة في تصميم الأنظمة. في C++، لا تُستخدم هذه الأنماط كقوالب جاهزة فقط، بل تُعاد صياغتها بما يتوافق مع خصائص اللغة مثل التحكم الصارم في الذاكرة، الأداء العالي، ووضوح الملكية.

يتناول هذا الفصل مجموعة من أشهر أنماط التصميم المستخدمة في C++ الحديثة، مع تحليل هندسي يركّز على الاستخدام الصحيح، المزايا، والمخاطر المحتملة.

- نمط Singleton
- نمط Factory
- نمط Observer
- نمط Strategy
- نمط Command
- نمط Method Template

١.٥ نمط Singleton

مقدمة
نمط Singleton هو أحد أكثر أنماط التصميم شيوعاً، وأيضًا أكثرها إثارة للجدل. الهدف الأساسي منه هو ضمان وجود نسخة واحدة فقط من صنف معيّن، مع توفير نقطة وصول عامة ومنضبطة إليها.
يُستخدم هذا النمط غالبًا لإدارة موارد مشتركة مثل:

- إعدادات التطبيق
- أنظمة التسجيل (Logging)
- مديري الموارد أو الاتصالات

لكن سوء استخدام Singleton قد يؤدي إلى تصميم هشّ، صعب الاختبار، ومليء بالاعتماديات الخفية، لذلك يجب التعامل معه بحذر هندسي.

١.١.٥ ما هو نمط Singleton؟

يضمن نمط Singleton الخصائص التالية:

- نسخة وحيدة: لا يمكن إنشاء أكثر من كائن واحد من الصنف.
- وصول عام: يمكن الوصول إلى النسخة من أي مكان في النظام.
- تحكم بالإنشاء: يمنع الإنشاء الخارجي غير المنضبط.

الغاية ليست ``السهولة''، بل التحكم المركزي في مورد مشترك.

٢.١.٥ تنفيذ Singleton في C++

١. التنفيذ الأساسي

يعتمد التنفيذ التقليدي على:

- مُنشئ خاص (private)
- مؤشر ساكن للاحتفاظ بالنسخة
- دالة وصول عامة ساكنة

```cpp
#include <iostream>

class Singleton {
private:
    static Singleton* instance;
    Singleton() {}

public:
    static Singleton* getInstance() {
        if (!instance) {
            instance = new Singleton();
        }
        return instance;
    }

    void showMessage() const {
        std::cout << "Singleton instance accessed\n";
    }
};
```

```
Singleton* Singleton::instance = nullptr;
```

ملاحظة تصميمية: هذا التنفيذ غير آمن في البيئات متعددة الخيوط، ويُعد غير مناسب لـ C++ الحديثة.

٢. Singleton آمن مع تعدد الخيوط

لحماية الإنشاء من حالات السباق، يمكن استخدام std::mutex:

```
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;
    Singleton() {}

public:
    static Singleton* getInstance() {
        if (!instance) {
            std::lock_guard<std::mutex> lock(mtx);
            if (!instance) {
                instance = new Singleton();
            }
        }
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

تحليل: هذا الأسلوب يقلل الأخطاء لكنه يزيد التعقيد ويصعّب الصيانة.

٣. التهيئة الكسولة باستخدام متغير ساكن محلي

ابتداءً من C++11، يضمن المعيار أن التهيئة المحلية الساكنة آمنة خيوطيًا.

```
class Singleton {
private:
    Singleton() {}

public:
    static Singleton& instance() {
        static Singleton instance;
        return instance;
    }
};
```

هذا هو التنفيذ الموصى به في C++ الحديثة:

- آمن خيوطيًا
- بسيط
- دون مؤشرات أو حذف يدوي

٤. Singleton وإدارة الموارد الحديثة

رغم شيوع ذكر المؤشرات الذكية مع Singleton، إلا أن المتغير الساكن المحلي غالبًا يغني عنها تمامًا. استخدام unique_ptr لا يضيف فائدة حقيقية هنا.

٣.١.٥ تنويعات نمط Singleton

١. **Double-Checked Locking**

يهدف لتقليل كلفة القفل، لكنه تاريخيًا كان عرضة لأخطاء نموذج الذاكرة.

تحذير هندسي: حتى مع C++ الحديثة، هذا الأسلوب أقل وضوحًا وأصعب في المراجعة من المتغير الساكن المحلي.

٢. Singleton مع حقن الاعتماديات

يُستخدم عند الحاجة لاختبار Singleton أو تغيير سلوكه.

```
#include <memory>

class Database {
public:
    void connect() {}
};

class Service {
private:
    std::shared_ptr<Database> db;
    Service(std::shared_ptr<Database> d) : db(d) {}

public:
    static Service& instance(std::shared_ptr<Database> db) {
        static Service instance(db);
        return instance;
    }
};
```

الفائدة:

- تحسين قابلية الاختبار
- تقليل الاعتماديات الصلبة

٤.١.٥ أفضل الممارسات

- لا تستخدم Singleton افتراضيًا — اسأل أولاً: هل هو ضروري؟
- فضّل المتغير الساكن المحلي في C++11+
- تجنّب استخدامه كبديل عن التصميم الجيد
- احذر من الاعتماديات الخفية
- فكّر دائمًا بالاختبار والصيانة طويلة الأمد

٥.١.٥ الخلاصة

نمط Singleton أداة قوية لكنها حادّة:

- يُفيد في إدارة الموارد المشتركة
- قد يُضعف التصميم إذا أسيء استخدامه
- يتطلب انضباطًا معماريًا واضحًا

في C++ الحديثة، التنفيذ الصحيح والبسيط هو مفتاح الاستفادة من Singleton دون الوقوع في فخ التصميم العالمي غير المنضبط.

٢.٥ نمط Factory

مقدمة

يُعد نمط Factory أحد أهم أنماط الإنشاء (Creational Patterns) في هندسة البرمجيات. تكمن فكرته الأساسية في فصل عملية إنشاء الكائنات عن استخدامها، بحيث لا يعتمد الكود العميل على الأصناف الملموسة (Concrete Classes)، بل على واجهات أو تجريدات ثابتة.

هذا النمط بالغ الأهمية في الأنظمة الكبيرة والمعقدة، حيث:

- يختلف نوع الكائن المطلوب إنشاؤه حسب سياق التنفيذ
- يتغير منطق الإنشاء مع تطور النظام
- يُراد تقليل الارتباط المباشر بين الأجزاء

في C++، يتكامل Factory بشكل طبيعي مع:

- الوراثة وتعدد الأشكال
- المؤشرات الذكية
- مبادئ التصميم الحديثة مثل Open/Closed Principle

١.٢.٥ نظرة عامة على نمط Factory

التعريف

نمط Factory يعرّف واجهة لإنشاء الكائنات، بينما يترك قرار أي صنف ملموس سيتم إنشاؤه لطبقة متخصصة. وبهذا، يصبح الكود العميل معزولاً عن تفاصيل الإنشاء.

المكوّنات الأساسية

- Product: واجهة أو صنف مجرد يحدد سلوك الكائنات المنتَجة.
- **ConcreteProduct**: تطبيقات ملموسة لـ Product.

- **Creator**: يعرّف آلية أو دالة إنشاء تُعيد Product.
- **ConcreteCreator**: يحدد الصنف الملموس الذي سيتم إنشاؤه.

الفكرة الجوهرية:

استخدم الكائنات دون معرفة كيفية أو نوع إنشائها.

٢.٢.٥ تنفيذ نمط Factory في C++

ا. **Simple Factory**

أبسط أشكال Factory، حيث تُستخدم دالة أو صنف واحد لإنشاء كائنات مختلفة بناءً على مُدخلات.

```
#include <iostream>
#include <memory>
#include <string>

class Product {
public:
    virtual void use() const = 0;
    virtual ~Product() = default;
};

class ProductA : public Product {
public:
    void use() const override {
        std::cout << "Using Product A\n";
    }
};
```

```
class ProductB : public Product {
public:
    void use() const override {
        std::cout << "Using Product B\n";
    }
};

class ProductFactory {
public:
    static std::unique_ptr<Product> create(const std::string& type) {
        if (type == "A")
            return std::make_unique<ProductA>();
        if (type == "B")
            return std::make_unique<ProductB>();
        throw std::invalid_argument("Unknown product type");
    }
};
```

تحليل هندسي

- سهل الفهم والتنفيذ
- يخرق مبدأ Open/Closed عند إضافة أنواع جديدة
- مناسب للحالات البسيطة أو المؤقتة

٢. Factory Method

ينقل قرار الإنشاء إلى أصناف فرعية، مع الحفاظ على واجهة ثابتة.

```
#include <memory>
#include <iostream>
```

```
class Product {
public:
    virtual void use() const = 0;
    virtual ~Product() = default;
};

class ProductA : public Product {
public:
    void use() const override {
        std::cout << "Using Product A\n";
    }
};

class ProductB : public Product {
public:
    void use() const override {
        std::cout << "Using Product B\n";
    }
};

class Creator {
public:
    virtual std::unique_ptr<Product> create() const = 0;

    void execute() const {
        auto product = create();
        product->use();
    }
```

```
    virtual ~Creator() = default;
};

class CreatorA : public Creator {
public:
    std::unique_ptr<Product> create() const override {
        return std::make_unique<ProductA>();
    }
};

class CreatorB : public Creator {
public:
    std::unique_ptr<Product> create() const override {
        return std::make_unique<ProductB>();
    }
};
```

الفائدة التصميمية

- يحقق Open/Closed Principle
- يسمح بالتوسع دون تعديل الكود القائم
- مناسب للأطر (Frameworks) والمكتبات

٣. Abstract Factory

يُستخدم لإنشاء عائلات كاملة من الكائنات المرتبطة منطقيًا.

```
#include <memory>
#include <iostream>
```

```cpp
class ProductA {
public:
    virtual void use() const = 0;
    virtual ~ProductA() = default;
};

class ProductB {
public:
    virtual void use() const = 0;
    virtual ~ProductB() = default;
};

class ProductA1 : public ProductA {
public:
    void use() const override { std::cout << "ProductA1\n"; }
};

class ProductB1 : public ProductB {
public:
    void use() const override { std::cout << "ProductB1\n"; }
};

class AbstractFactory {
public:
    virtual std::unique_ptr<ProductA> createA() const = 0;
    virtual std::unique_ptr<ProductB> createB() const = 0;
    virtual ~AbstractFactory() = default;
};
```

```
class Factory1 : public AbstractFactory {
public:
    std::unique_ptr<ProductA> createA() const override {
        return std::make_unique<ProductA1>();
    }
    std::unique_ptr<ProductB> createB() const override {
        return std::make_unique<ProductB1>();
    }
};
```

متى يُستخدم؟

- عند الحاجة لضمان توافق الكائنات معًا
- في الأنظمة متعددة المنصات أو الواجهات
- عند تبديل عائلة كاملة من السلوكيات

٣.٢.٥ اعتبارات عملية

- المرونة: Factory يسهل إضافة أنواع جديدة دون كسر الكود.
- العزل: يفصل منطق الإنشاء عن منطق الاستخدام.
- التعقيد: الإفراط في استخدامه قد يؤدي إلى تضخم عدد الأصناف.
- التوازن: اختر أبسط شكل يحقق الهدف.

٤.٢.٥ الخلاصة

نمط Factory هو حجر أساس في تصميم الأنظمة المرنة في C++:

- يعزل الكود عن التفاصيل الملموسة

- يدعم التوسع والصيانة طويلة الأمد
- يتكامل بعمق مع OOP وتعدد الأشكال

الاختيار بين Abstract Factory و، Simple Factory، Factory Method يعتمد على حجم النظام، درجة التعقيد، ومتطلبات التوسع المستقبلية. الاستخدام الواعي لهذا النمط يُنتج تصميمًا نظيفًا، قابلاً للتطور، ومتوافقًا مع مبادئ C++ الحديثة.

٣.٥ نمط Observer

مقدمة
يُعد نمط Observer من أنماط السلوك (Behavioral Design Patterns) الأساسية، ويقوم على إنشاء علاقة واحد إلى متعدد بين الكائنات. حيث يحتفظ الكائن المُراقَب (Subject) بقائمة من المُراقِبين (Observers) ويقوم بإخطارهم تلقائيًا عند حدوث أي تغيير في حالته.
يُستخدم هذا النمط على نطاق واسع في:

- الأنظمة المعتمدة على الأحداث (Event-Driven Systems)
- أطر واجهات المستخدم الرسومية (GUI Frameworks)
- أنظمة الإشعارات والتنبيهات
- مراقبة البيانات في الزمن الحقيقي

القيمة الأساسية لهذا النمط تكمن في فصل منطق التغيير عن منطق الاستجابة.

١.٣.٥ نظرة عامة على نمط Observer

١. التعريف
يسمح نمط Observer لكائن واحد (Subject) بإخطار عدة كائنات أخرى (Observers) عند تغيّر حالته، دون أن يعرف هذا الكائن أي تفاصيل عن طبيعة أو تنفيذ المراقبين.

٢. المكوّنات الأساسية

- Subject: يحتفظ بالمراقبين ويوفر آليات الإضافة، الإزالة، والإخطار.
- Observer: واجهة تُعرّف دالة التحديث التي يستدعيها الـ Subject.
- ConcreteSubject: التطبيق الفعلي لـ Subject ويمثل الحالة المراد مراقبتها.
- ConcreteObserver: تطبيق فعلي لـ Observer يستجيب للتغييرات.

الفكرة الجوهرية:

غيّر الحالة في مكان واحد، ودع الآخرين يتفاعلون تلقائيًا.

٢.٣.٥ تنفيذ نمط Observer في C++

مثال تطبيقي: محطة طقس
لنفترض وجود محطة طقس تقوم بتحديث درجة الحرارة، ويوجد أكثر من جهاز عرض أو نظام تسجيل يحتاج للاستجابة لهذه التغييرات.

١. تعريف واجهة Observer

```
#include <iostream>

class Observer {
public:
    virtual void update(float temperature) = 0;
    virtual ~Observer() = default;
};
```

الدور:

- توحيد آلية التحديث
- تمكين تعدد الأشكال (Polymorphism)

٢. تعريف واجهة Subject

```
class Subject {
public:
    virtual void attach(Observer* observer) = 0;
    virtual void detach(Observer* observer) = 0;
```

```
    virtual void notify() = 0;
    virtual ~Subject() = default;
};
```

الدور:

- إدارة دورة حياة المراقبين
- فصل الإخطار عن التنفيذ الفعلي

٣. تنفيذ ConcreteSubject

```
#include <vector>
#include <algorithm>

class WeatherStation : public Subject {
private:
    std::vector<Observer*> observers;
    float temperature{0.0f};

public:
    void attach(Observer* observer) override {
        observers.push_back(observer);
    }

    void detach(Observer* observer) override {
        observers.erase(
            std::remove(observers.begin(), observers.end(), observer),
            observers.end()
        );
    }
```

```
    void notify() override {
        for (Observer* observer : observers) {
            observer->update(temperature);
        }
    }

    void setTemperature(float temp) {
        temperature = temp;
        notify();
    }
};
```

تحليل:

- لا يعرف نوع المراقبين
- مسؤول فقط عن الحالة والإخطار

٤. تنفيذ ConcreteObservers

```
class TemperatureDisplay : public Observer {
public:
    void update(float temperature) override {
        std::cout << "Display: Temperature = "
                  << temperature << "°C\n";
    }
};

class TemperatureLogger : public Observer {
public:
```

```
    void update(float temperature) override {
        std::cout << "Logger: Logged temperature "
                  << temperature << "°C\n";
    }
};
```

الفائدة:

- كل مراقب يفسر التحديث بطريقته
- لا تغيير في الـ Subject عند إضافة مراقب جديد

٥. استخدام النمط

```
int main() {
    WeatherStation station;

    TemperatureDisplay display;
    TemperatureLogger logger;

    station.attach(&display);
    station.attach(&logger);

    station.setTemperature(25.0f);
    station.setTemperature(30.0f);

    station.detach(&display);
    station.setTemperature(28.0f);

    return 0;
}
```

٣.٣.٥ اعتبارات عملية

- فصل الاعتماديات: الـ Subject لا يعتمد على المراقبين فعليًا.
- المرونة: إضافة أو إزالة مراقبين دون تعديل الكود القائم.
- الأداء: كثرة المراقبين قد تؤثر على الأداء.
- تعدد الخيوط: في الأنظمة متعددة الخيوط يجب تأمين التزامن.

في الأنظمة الكبيرة، قد يُستعاض عن Observer التقليدي بـ:

- Bus Event
- Queue Message
- Streams Reactive

٤.٣.٥ الخلاصة

يوفر نمط Observer آلية أنيقة لبناء أنظمة قائمة على الأحداث في C++:

- يقلل الترابط بين الأجزاء
- يدعم التوسع والتغيير الديناميكي
- يعزز قابلية الصيانة والاختبار

عند استخدامه بوعي، يُعد Observer حجر أساس في تصميم الأنظمة التفاعلية، الرسومية، والموزعة، ويُجسّد فلسفة OOP في فصل المسؤوليات والتفاعل المنظم.

٤.٥ نمط Strategy

مقدمة

يُعد نمط Strategy أحد أنماط السلوك (Behavioral Design Patterns) المهمة، ويقوم على تعريف عائلة من الخوارزميات، ثم تغليف كل خوارزمية بكائن مستقل وجعل هذه الخوارزميات قابلة للاستبدال. يسمح هذا النمط للعميل (Client) باختيار الخوارزمية المناسبة أثناء وقت التنفيذ (Runtime)، مما يحقق مرونة عالية ويفصل منطق الخوارزمية عن الكائن الذي يستخدمها.
يكون هذا النمط مفيداً بشكل خاص عندما:

- توجد عدة خوارزميات لتحقيق نفس الهدف
- قد يتغير اختيار الخوارزمية أثناء تشغيل البرنامج
- نرغب في تجنب التفرعات الشرطية المعقدة داخل الكائنات

١.٤.٥ نظرة عامة على نمط Strategy

١. التعريف

يعرّف نمط Strategy مجموعة من الخوارزميات، ويغلف كل واحدة منها داخل كائن مستقل، ويجعلها قابلة للتبديل دون التأثير على الكائن الذي يستخدمها. يسمح هذا بأن تتغير الخوارزميات بشكل مستقل عن العملاء.

٢. المكوّنات الأساسية

- Strategy: واجهة مجردة تمثل الخوارزمية.
- ConcreteStrategy: تطبيقات فعلية للواجهة، كل منها يقدّم خوارزمية مختلفة.
- Context: الكائن الذي يحتفظ بمؤشر إلى Strategy ويستخدمه لتنفيذ الخوارزمية.

الفكرة الجوهرية:

غيّر السلوك، لا الكائن.

٢.٤.٥ تنفيذ نمط Strategy في C++

مثال تطبيقي: نظام معالجة المدفوعات
لنفترض وجود نظام يدعم عدة وسائل دفع (بطاقة ائتمان، Bitcoin، PayPal). يسمح نمط Strategy باختيار وسيلة الدفع ديناميكيًا دون تغيير منطق المعالجة الأساسي.

١. تعريف واجهة Strategy

```cpp
#include <iostream>

class PaymentStrategy {
public:
    virtual void pay(double amount) const = 0;
    virtual ~PaymentStrategy() = default;
};
```

الدور:

- توحيد واجهة جميع طرق الدفع
- تمكين تعدد الأشكال (Polymorphism)

٢. تنفيذ الاستراتيجيات الفعلية

```cpp
// ConcreteStrategy 1: Credit Card Payment
class CreditCardPayment : public PaymentStrategy {
private:
    std::string name;
    std::string cardNumber;

public:
    CreditCardPayment(const std::string& name,
```

```
                        const std::string& cardNumber)
        : name(name), cardNumber(cardNumber) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount
                  << " using Credit Card. Card Number: "
                  << cardNumber << std::endl;
    }
};

// ConcreteStrategy 2: PayPal Payment
class PayPalPayment : public PaymentStrategy {
private:
    std::string email;

public:
    explicit PayPalPayment(const std::string& email)
        : email(email) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount
                  << " using PayPal. Email: "
                  << email << std::endl;
    }
};

// ConcreteStrategy 3: Bitcoin Payment
class BitcoinPayment : public PaymentStrategy {
private:
```

```
    std::string walletAddress;

public:
    explicit BitcoinPayment(const std::string& walletAddress)
        : walletAddress(walletAddress) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount
                  << " using Bitcoin. Wallet Address: "
                  << walletAddress << std::endl;
    }
};
```

ملاحظة تصميمية:

- كل خوارزمية معزولة في فئة مستقلة
- إضافة طريقة دفع جديدة لا تتطلب تعديل الكود الحالي

٣. تعريف كائن Context

```
class PaymentContext {
private:
    PaymentStrategy* strategy;

public:
    explicit PaymentContext(PaymentStrategy* strategy)
        : strategy(strategy) {}

    void setStrategy(PaymentStrategy* newStrategy) {
        strategy = newStrategy;
```

```
    }

    void executePayment(double amount) const {
        strategy->pay(amount);
    }
};
```

الدور:

- فصل منطق الاستخدام عن منطق التنفيذ
- السماح بتبديل الخوارزمية أثناء التشغيل

٤. استخدام نمط Strategy

```
int main() {
    CreditCardPayment creditCard(
        "Alice", "1234-5678-9012-3456");
    PayPalPayment paypal("alice@example.com");
    BitcoinPayment bitcoin(
        "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa");

    PaymentContext context(&creditCard);
    context.executePayment(100.0);   // Credit Card

    context.setStrategy(&paypal);
    context.executePayment(200.0);   // PayPal

    context.setStrategy(&bitcoin);
    context.executePayment(0.05);    // Bitcoin
```

```
    return 0;
}
```

٣.٤.٥ اعتبارات عملية

- المرونة: يمكن تغيير الخوارزمية أثناء التشغيل دون تعديل الكود.
- مبدأ **Open/Closed**: يمكن إضافة استراتيجيات جديدة دون تعديل Context أو الواجهات الحالية.
- التعقيد: يزيد عدد الأصناف، لكن المكاسب في القابلية للصيانة غالبًا تفوق هذا التعقيد.
- الاختبار: يمكن اختبار كل Strategy بشكل مستقل.

في Modern C++ يمكن تحسين التصميم باستخدام:

- std::unique_ptr للإدارة الملكية
- std::function للاستراتيجيات البسيطة

٤.٤.٥ الخلاصة

يسمح نمط Strategy ببناء أنظمة مرنة وقابلة للتوسع من خلال تغليف السلوكيات (الخوارزميات) داخل كائنات مستقلة.
أهم الفوائد:

- المرونة: اختيار وتبديل السلوك أثناء التشغيل.
- الوحدوية: فصل كل خوارزمية في صنف مستقل.
- قابلية الصيانة: إضافة سلوكيات جديدة دون كسر النظام.

باستخدام نمط Strategy بشكل مدروس، يمكن تصميم أنظمة C++ قادرة على التكيف مع التغيرات المستقبلية دون التضحية بالوضوح أو السلامة التصميمية.

٥.٥ نمط Command

مقدمة

يُعد نمط Command أحد أنماط السلوك (Behavioral Design Patterns)، ويقوم على تغليف الطلب (Request) داخل كائن مستقل. يسمح هذا التغليف بتمرير الطلبات ككائنات، وتخزينها في قوائم انتظار، وتسجيلها، بل وحتى دعم عمليات التراجع (Undo / Redo).

من خلال فصل الجهة التي تُصدر الأمر عن الجهة التي تنفذه، يعزز نمط Command المرونة، وقابلية التوسّع، ونظافة التصميم، خصوصًا في الأنظمة التي تتعامل مع أوامر متعددة أو قابلة للتغيير أثناء التنفيذ.

١.٥.٥ نظرة عامة على نمط Command

١. التعريف

يحوّل نمط Command الطلب إلى كائن مستقل يحتوي على جميع المعلومات اللازمة لتنفيذه. هذا الفصل يسمح بأن تختلف مسؤولية إصدار الأمر عن مسؤولية تنفيذه، وهو مناسب بشكل خاص لـ:

- تمرير العمليات كمعاملات
- جدولة الأوامر أو تسجيلها
- دعم التراجع عن العمليات

٢. المكوّنات الأساسية

- Command: واجهة تعرّف عملية تنفيذ الأمر.
- ConcreteCommand: تطبيق فعلي يربط الأمر بالمستقبِل .(Receiver)
- Client: ينشئ ويُهيئ كائنات الأوامر.
- Invoker: الكائن الذي يطلب تنفيذ الأمر.
- Receiver: الكائن الذي ينفّذ العملية الفعلية.

الفكرة الجوهرية:

حوّل الطلب إلى كائن، ثم دع التنفيذ يتم بشكل مستقل.

٢.٥.٥ تنفيذ نمط Command في C++

مثال تطبيقي: نظام تحكم عن بُعد
لنفترض وجود جهاز تحكم يمكنه تشغيل وإيقاف أجهزة مختلفة مثل الإضاءة والمروحة. يسمح نمط Command بتغليف هذه العمليات دون ربط جهاز التحكم بتفاصيل التنفيذ.

١. تعريف واجهة Command

```
class Command {
public:
    virtual void execute() = 0;
    virtual ~Command() = default;
};
```

الدور:

- توحيد واجهة تنفيذ جميع الأوامر
- تمكين تعدد الأشكال (Polymorphism)

٢. تنفيذ الأوامر الفعلية **(Concrete Commands)**

```
#include <iostream>

// Receiver classes
class Light {
public:
    void on()  { std::cout << "Light is ON\n"; }
```

```
    void off() { std::cout << "Light is OFF\n"; }
};

class Fan {
public:
    void start() { std::cout << "Fan is STARTED\n"; }
    void stop()  { std::cout << "Fan is STOPPED\n"; }
};

// ConcreteCommand for Light
class LightOnCommand : public Command {
    Light* light;
public:
    explicit LightOnCommand(Light* light) : light(light) {}
    void execute() override { light->on(); }
};

class LightOffCommand : public Command {
    Light* light;
public:
    explicit LightOffCommand(Light* light) : light(light) {}
    void execute() override { light->off(); }
};

// ConcreteCommand for Fan
class FanStartCommand : public Command {
    Fan* fan;
public:
    explicit FanStartCommand(Fan* fan) : fan(fan) {}
```

```
    void execute() override { fan->start(); }
};

class FanStopCommand : public Command {
    Fan* fan;
public:
    explicit FanStopCommand(Fan* fan) : fan(fan) {}
    void execute() override { fan->stop(); }
};
```

ملاحظة تصميمية:

- كل أمر مرتبط بمستقبِل محدد
- يمكن إضافة أوامر جديدة دون تعديل الكود الحالي

٣. تعريف كائن Invoker

```
class RemoteControl {
    Command* command{nullptr};
public:
    void setCommand(Command* newCommand) {
        command = newCommand;
    }

    void pressButton() {
        if (command) {
            command->execute();
        }
    }
};
```

الدور:

- لا يعرف تفاصيل التنفيذ
- يكتفي باستدعاء الأمر المعيّن

٤. استخدام نمط Command

```
int main() {
    Light light;
    Fan fan;

    LightOnCommand  lightOn(&light);
    LightOffCommand lightOff(&light);
    FanStartCommand fanStart(&fan);
    FanStopCommand  fanStop(&fan);

    RemoteControl remote;

    remote.setCommand(&lightOn);
    remote.pressButton();   // Light is ON

    remote.setCommand(&lightOff);
    remote.pressButton();   // Light is OFF

    remote.setCommand(&fanStart);
    remote.pressButton();   // Fan is STARTED

    remote.setCommand(&fanStop);
    remote.pressButton();   // Fan is STOPPED
```

```
    return 0;
}
```

٣.٥.٥ اعتبارات عملية

- الفصل :(Decoupling) فصل المرسِل عن المنفِّذ يسمح بتغيير السلوك دون التأثير على البنية العامة.
- المرونة: يدعم الجدولة، التسجيل، والتنفيذ الديناميكي للأوامر.
- دعم التراجع :(Undo) يمكن تخزين الأوامر المنفذة وإعادة عكسها.
- التعقيد: يضيف عدداً من الأصناف، لكنه يحسن التنظيم وقابلية الصيانة.

في C++ Modern يمكن تحسين التصميم عبر:

- استخدام std::unique_ptr لإدارة الملكية
- استخدام std::function للأوامر البسيطة

٤.٥.٥ الخلاصة

يقوم نمط Command بتغليف الطلبات داخل كائنات مستقلة، مما يوفر تصميمًا مرنًا وقابلاً للتوسّع والتنظيم.
أهم الفوائد:

- الفصل: استقلال المرسِل عن المنفِّذ.
- المرونة: تنفيذ الأوامر وإدارتها ديناميكيًا.
- التراجع: دعم Undo / Redo عند الحاجة.

- قابلية الصيانة: إضافة أوامر جديدة دون تعديل الكود القائم.

باستخدام نمط Command بشكل مدروس، يمكن بناء أنظمة C++ قادرة على التعامل مع سيناريوهات أوامر معقدة بدرجة عالية من الوضوح والمرونة والمتانة.

٦.٥ نمط Method Template

مقدمة

يُعد نمط Template Method أحد أنماط السلوك (Behavioral Design Patterns)، ويهدف إلى تعريف الهيكل العام (Skeleton) لخوارزمية ما داخل صنف أساسي، مع السماح للأصناف المشتقة بإعادة تعريف بعض الخطوات المحددة دون تغيير البنية العامة للخوارزمية.

يُستخدم هذا النمط عندما نرغب في:

- الحفاظ على تسلسل ثابت للخوارزمية
- السماح بتخصيص خطوات محددة فقط
- منع العبث بالبنية العامة للتنفيذ

١.٦.٥ نظرة عامة على نمط Template Method

١. التعريف

يوفر نمط Method Template هيكلاً عامًا لخوارزمية داخل صنف أساسي، ويتيح للأصناف المشتقة إعادة تعريف بعض الخطوات دون التأثير على تسلسل التنفيذ الكامل. بهذه الطريقة، يتم الجمع بين إعادة الاستخدام والمرونة.

٢. المكوّنات الأساسية

- AbstractClass: يعرّف الدالة القالب (Template Method) والخطوات العامة للخوارزمية، وقد يحتوي على عمليات مجردة.
- ConcreteClass: يطبّق العمليات المجردة، ويقدّم السلوك الخاص بكل خطوة.

الفكرة الجوهرية:

دع الهيكل ثابتًا، واسمح بتخصيص التفاصيل.

٢.٦.٥ تنفيذ نمط Template Method في C++

مثال تطبيقي: تحضير المشروبات
تحضير القهوة والشاي يتبع نفس الخطوات العامة: غلي الماء، التحضير، السكب في الكوب، ثم الإضافات. لكن تختلف تفاصيل التحضير والإضافات بين المشروبات.

١. تعريف الصنف الأساسي (Abstract Class)

```cpp
#include <iostream>

class Beverage {
public:
    // Template Method
    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    virtual ~Beverage() = default;

protected:
    virtual void brew() const = 0;          //
    virtual void addCondiments() const = 0; //

    void boilWater() const {
        std::cout << "Boiling water" << std::endl;
    }
```

```
    void pourInCup() const {
        std::cout << "Pouring into cup" << std::endl;
    }
};
```

التحليل:

- prepareRecipe تمثل الخوارزمية الكاملة
- لا يُسمح للأصناف المشتقة بتغيير ترتيب الخطوات
- يتم فرض تنفيذ الخطوات المجرّدة فقط

٣. تنفيذ الأصناف المشتقة (Concrete Classes)

```
// ConcreteClass: Coffee
class Coffee : public Beverage {
protected:
    void brew() const override {
        std::cout << "Dripping coffee through filter" << std::endl;
    }

    void addCondiments() const override {
        std::cout << "Adding sugar and milk" << std::endl;
    }
};

// ConcreteClass: Tea
class Tea : public Beverage {
protected:
    void brew() const override {
        std::cout << "Steeping the tea" << std::endl;
```

```
    }

    void addCondiments() const override {
        std::cout << "Adding lemon" << std::endl;
    }
};
```

المغزى التصميمي:

- كل صنف يخصص خطواته فقط
- الهيكل العام بقي محميًا داخل الصنف الأساسي

٣. استخدام نمط **Template Method**

```
int main() {
    Coffee coffee;
    Tea tea;

    std::cout << "Making coffee..." << std::endl;
    coffee.prepareRecipe();
    std::cout << std::endl;

    std::cout << "Making tea..." << std::endl;
    tea.prepareRecipe();

    return 0;
}
```

٣.٦.٥ اعتبارات عملية

- إعادة الاستخدام: تجميع الخطوات المشتركة في الصنف الأساسي يقلل التكرار ويُبسّط الصيانة.
- الاتساق: يضمن تنفيذ الخوارزمية بنفس الترتيب دائمًا.
- قابلية التوسّع: يمكن إضافة سلوك جديد عبر أصناف مشتقة دون تعديل الصنف الأساسي (Open/Closed Principle).
- فرض البنية: لا يمكن للأصناف المشتقة تغيير تدفق الخوارزمية، مما يمنع أخطاء تصميمية خطيرة.

في Modern C++ يمكن تعزيز هذا النمط باستخدام:

- `final` لمنع إعادة التعريف غير المقصود
- دوال `non-virtual` للخطوات الثابتة

٤.٦.٥ الخلاصة

يُعد نمط Method Template أداة تصميم قوية لتعريف خوارزميات ذات بنية ثابتة مع السماح بتخصيص جزئي للسلوك.
أهم الفوائد:

- إعادة الاستخدام: توحيد الخطوات المشتركة.
- الاتساق: ضمان تنفيذ موحّد للخوارزمية.
- المرونة: تخصيص السلوك عبر الأصناف المشتقة.
- الانضباط: فرض هيكل صارم يمنع كسر التصميم.

استخدام نمط Template Method يؤدي إلى شيفرة منظمة، قابلة للصيانة، وسهلة التوسعة، وهو نمط جوهري في تصميم الأنظمة الكبيرة باستخدام C++ الحديثة.

الفصل ٦: القوالب (Templates) وتعدد الأشكال الحديث

- Function Templates و Class Templates.
- Variadic Templates.
- تخصيص القوالب والتخصيص الجزئي Template Specialization & Partial Specialization.
- نمط CRTP (Curiously Recurring Template Pattern).

١.٦ قوالب الدوال وقوالب الأصناف

مقدمة

تُعد القوالب (Templates) حجر الأساس في البرمجة الحديثة بلغة C++، إذ توفّر آليات قوية للبرمجة العامة (Generic Programming). تسمح القوالب بكتابة الشيفرة بشكل مستقل عن نوع البيانات، مما يعزز إعادة الاستخدام (Reusability)، ويحافظ على أمان الأنواع (Type Safety). يستعرض هذا القسم قوالب الدوال وقوالب الأصناف، إضافةً إلى التخصيص والتحسينات الحديثة في C++.

١.١.٦ ما هي القوالب؟

تمكّن القوالب الدوال والأصناف من العمل مع أنواع عامة. وبذلك يستطيع المبرمج كتابة الشيفرة مرة واحدة واستخدامها مع أنواع بيانات متعددة، مما يجعلها عنصرًا جوهريًا في تعدد الأشكال الحديث (Modern Polymorphism) في C++.

١. قوالب الدوال **(Function Templates)**

تسمح قوالب الدوال بتعريف دوال عامة يمكن إعادة استخدامها مع أنواع متعددة.

مثال أساسي على قالب دالة

```cpp
#include <iostream>

// Function template to find the maximum of two values
template <typename T>
T maximum(T a, T b) {
   return (a > b) ? a : b;
}

int main() {
   std::cout << "Maximum of 3 and 5: " << maximum(3, 5) << std::endl;
   std::cout << "Maximum of 3.5 and 2.1: " << maximum(3.5, 2.1) << std::endl;
```

```
    std::cout << "Maximum of 'A' and 'B': " << maximum('A', 'B') << std::endl;
    return 0;
}
```

تعمل هذه الدالة مع أي نوع يدعم معاملات المقارنة.

٢. تخصيص قوالب الدوال **(Function Template Specialization)**

يسمح التخصيص بتوفير سلوك خاص لأنواع معيّنة دون التأثير على القالب العام.

مثال على تخصيص قالب دالة

```
#include <iostream>

// Primary template
template <typename T>
void print(T value) {
    std::cout << "Generic value: " << value << std::endl;
}

// Specialization for char* (C-style strings)
template <>
void print<char*>(char* value) {
    std::cout << "C-style string: " << value << std::endl;
}

int main() {
    print(10);
    print(3.14);
    char str[] = "Hello, World!";
    print(str);
    return 0;
```

```
}
```

في هذا المثال، تتصرف الدالة print بشكل مختلف عند التعامل مع سلاسل C-style مقارنةً بالأنواع الأخرى.

٣. قوالب الأصناف **(Class Templates)**

تتيح قوالب الأصناف إنشاء أصناف عامة تعمل مع أنواع متعددة.

مثال أساسي على قالب صنف

```
#include <iostream>

// Class template for a Box
template <typename T>
class Box {
private:
   T value;

public:
   Box(T v) : value(v) {}

   T getValue() const { return value; }
   void setValue(T v) { value = v; }
};

int main() {
   Box<int> intBox(123);
   Box<double> doubleBox(456.78);

   std::cout << "Integer Box contains: " << intBox.getValue() << std::endl;
   std::cout << "Double Box contains: " << doubleBox.getValue() << std::endl;
```

```
    return 0;
}
```

يستطيع الصنف Box احتواء أي نوع يتم تحديده عند الإنشاء.

٤. تخصيص قوالب الأصناف **(Class Template Specialization)**
يسمح التخصيص بتغيير سلوك الصنف لقيم أو أنواع معيّنة.

مثال على تخصيص قالب صنف

```
#include <iostream>

// Primary template
template <typename T>
class Storage {
public:
    void display() { std::cout << "Generic storage" << std::endl; }
};

// Specialization for int
template <>
class Storage<int> {
public:
    void display() { std::cout << "Storage for integers" << std::endl; }
};

int main() {
    Storage<double> genericStorage;
    Storage<int> intStorage;
```

```
    genericStorage.display(); // Generic storage
    intStorage.display();     // Storage for integers

    return 0;
}
```

يسمح هذا الأسلوب بأن يكون للصنف Storage<int> سلوك مختلف عن القالب العام.

٢.١.٦ تحسينات C++ الحديثة: المفاهيم والقيود

قدّمت C++20 مفهوم Concepts والقيود (Constraints) لفرض شروط واضحة على معاملات القوالب، مما يحسّن قابلية القراءة ورسائل الأخطاء أثناء الترجمة.

مثال على المفاهيم

```
#include <iostream>
#include <concepts>

// Define a concept
template <typename T>
concept Addable = requires(T a, T b) { { a + b } -> std::same_as<T>; };

// Function template constrained by concept
template <Addable T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << "Sum of 3 and 5: " << add(3, 5) << std::endl;
    std::cout << "Sum of 3.5 and 2.1: " << add(3.5, 2.1) << std::endl;
```

```
    // add(std::string("Hello"), 3); // Compile-time error
    return 0;
}
```

يفرض المفهوم Addable أنواعًا تدعم عامل الجمع فقط.

٣.١.٦ أفضل الممارسات

١. استخدم القوالب بحكمة: تجنّب الإفراط في القوالب حتى لا تتأثر قابلية القراءة.

٢. فضّل خصائص الأنواع: استخدم Type Traits بدل التخصيص المفرط.

٣. وثّق المتطلبات: وضّح بجلاء ما تتطلبه معاملات القوالب.

٤. استفد من المفاهيم: استخدم Concepts لفرض القيود وتحسين التشخيص.

٤.١.٦ الخلاصة

تُعد القوالب عنصرًا أساسيًا في البرمجة العامة ضمن C++ الحديثة. تمكّن قوالب الدوال وقوالب الأصناف من كتابة شيفرة مرنة، قابلة لإعادة الاستخدام، وآمنة من حيث الأنواع. كما أن التخصيص والمفاهيم يضيفان قوةً إضافية، تسمح بضبط السلوك وفرض القيود بدقة.
إتقان القوالب هو خطوة جوهرية لبناء برامج C++ قوية، قابلة للصيانة، ومتوافقة مع أساليب التصميم الحديثة.

٢.٦ القوالب المتغيرة في C++

القوالب المتغيرة (Variadic Templates)، التي قُدِّمت في C++11، تُعد ميزة قوية تتيح للدوال والأصناف التعامل مع عدد غير محدد من المعاملات. تُعزّز هذه الميزة المرونة وقابلية إعادة الاستخدام، وتُسهم مباشرةً في تعدد الأشكال وقت الترجمة (Compile-Time Polymorphism).

١.٢.٦ مقدمة في القوالب المتغيرة

- التعريف: تمكّن القوالب المتغيرة من تعريف دوال أو أصناف تستقبل عددًا غير محدود من معاملات القوالب. بخلاف القوالب التقليدية، لا يكون عدد المعاملات ثابتًا.

- صياغة القوالب المتغيرة: يُستخدم رمز الحذف (...) للإشارة إلى حزمة معاملات (Parameter Pack).

```
template<typename... Args>
void foo(Args... args) {
    // Implementation
}
```

هنا تُعد ...Args حزمة معاملات يمكن أن تحتوي على صفر أو أكثر من المعاملات.

- المفاهيم الأساسية:

١. توسيع حزم المعاملات: يتم باستخدام الاستدعاء التكراري أو تعبيرات الطي (Fold Expressions) ابتداءً من C++17.

٢. تخصيص القوالب: يمكن دمج القوالب المتغيرة مع التخصيص للتعامل مع حالات خاصة.

٣. تعدد الأشكال الحديث: تسمح القوالب المتغيرة بالتكيّف وقت الترجمة مع اختلاف الأنواع وعدد المعاملات.

٢.٢.٦ العمل مع القوالب المتغيرة

المثال 1: قالب دالة متغير بسيط

```cpp
#include <iostream>

// Base case
void print() {
    std::cout << "End of arguments." << std::endl;
}

// Recursive variadic template
template<typename T, typename... Args>
void print(T first, Args... args) {
    std::cout << first << std::endl;
    print(args...);
}

int main() {
    print(1, 2.5, "Hello", 'A');
    return 0;
}
```

الشرح:

- يتم طباعة المعامل الأول في كل خطوة، ثم تمرير بقية المعاملات بشكل تكراري حتى الوصول إلى الحالة الأساسية (print()).
- يدعم أي عدد وأي نوع من المعاملات.

الناتج:

```
1
2.5
Hello
A
End of arguments.
```

المثال 2: قالب صنف متغير (بنية **Tuple**)

```cpp
#include <iostream>

// Base case: Empty Tuple
template<typename... Values>
class Tuple;

// Recursive case: Tuple with at least one element
template<typename T, typename... Rest>
class Tuple<T, Rest...> {
public:
    T first;
    Tuple<Rest...> rest;

    Tuple(T f, Rest... r) : first(f), rest(r...) {}

    void print() {
        std::cout << first << " ";
        rest.print();
    }
};

// Specialization for empty Tuple
```

```
template<>
class Tuple<> {
public:
    void print() { std::cout << std::endl; }
};

int main() {
    Tuple<int, double, std::string> myTuple(42, 3.14, "Hello, World!");
    myTuple.print();
    return 0;
}
```

الشرح:

- يخزّن الصنف Tuple قيمًا متعددة بأنواع مختلفة باستخدام بنية تكرارية.
- تتعامل الحالة الأساسية مع Tuple الفارغ، بينما تعالج الحالة التكرارية بقية العناصر.

الناتج:

```
42 3.14 Hello, World!
```

٣.٢.٦ القوالب المتغيرة وتعبيرات الطي

قدّمت C++17 ما يُعرف بـ تعبيرات الطي (Fold Expressions)، التي تتيح تنفيذ عمليات مختصرة على حزم المعاملات.

المثال 3: الجمع باستخدام تعبير الطي

```
#include <iostream>

template<typename... Args>
```

```cpp
auto sum(Args... args) {
    return (args + ...);  // Sum all arguments
}

int main() {
    std::cout << "Sum: " << sum(1, 2, 3, 4, 5) << std::endl;
    return 0;
}
```

الشرح:

- التعبير (... + args) يطبّق عامل الجمع على جميع عناصر الحزمة.
- يُلغي الحاجة إلى الاستدعاء التكراري الصريح.

الناتج:

```
Sum: 15
```

٤.٢.٦ القوالب المتغيرة وتعدد الأشكال وقت الترجمة

تمكّن القوالب المتغيرة من تحقيق تعدد الأشكال وقت الترجمة (Compile-Time Polymorphism)، حيث تتكيّف الدوال والأصناف مع اختلاف الأنواع وعدد المعاملات دون أي كلفة وقت التشغيل.
مقارنة مع تعدد الأشكال وقت التشغيل:

- تعدد الأشكال وقت الترجمة:
  - يعتمد على القوالب؛ تُحسم الأنواع أثناء الترجمة.
  - أكثر كفاءة؛ لا يوجد ربط ديناميكي.
  - مثال: دالة print() التكرارية التي تتكيّف مع أنواع المعاملات.

- تعدد الأشكال وقت التشغيل:
  - يعتمد على الوراثة والدوال الافتراضية.
  - أكثر مرونة لكنه يضيف كلفة زمنية.

٥.٢.٦ الخلاصة

تُعد القوالب المتغيرة ركيزة أساسية في C++ الحديثة، إذ تتيح التعامل الآمن والمرن مع عدد غير محدود من المعاملات.
أهم الفوائد:

- حزم المعاملات: مرونة عالية في التعامل مع عدة معاملات.
- الاستدعاء التكراري والتخصيص: معالجة أنواع متعددة بكفاءة.
- تعبيرات الطي: تبسيط العمليات على الحزم.
- تعدد الأشكال وقت الترجمة: بديل فعّال وآمن لتعدد الأشكال وقت التشغيل.

باستخدام القوالب المتغيرة، يستطيع المبرمج بناء مكوّنات C++ عالية القابلية لإعادة الاستخدام، مرنة، وكفؤة، ومتوافقة مع أساليب التصميم الحديثة.

٣.٦ تخصيص القوالب والتخصيص الجزئي

١.٣.٦ مقدمة

تُعد القوالب (Templates) حجر الأساس في برمجة C++ الحديثة، إذ تُمكّن من البرمجة العامة (Generic Programming) وتحقق تعدد الأشكال وقت الترجمة (Compile-Time Polymorphism). وعلى عكس تعدد الأشكال وقت التشغيل المبني على الوراثة والدوال الافتراضية، تُحسم الأنواع والسلوكيات في القوالب أثناء الترجمة.

يأتي تخصيص القوالب (Template Specialization) والتخصيص الجزئي (Partial Specialization) كآليتين لتقييد أو تعديل سلوك القوالب لأنواع محددة أو لمجموعات جزئية من الأنواع، مما يسمح بمعالجة حالات خاصة لا يكون فيها القالب العام مناسبًا أو فعّالاً.

٢.٣.٦ ما هو تخصيص القوالب؟

- التعريف: يتيح تخصيص القوالب إعادة تعريف سلوك قالب عام لنوع معيّن أو مجموعة أنواع محددة، بحيث يتم تجاوز التنفيذ العام وتنفيذ منطق خاص بذلك النوع.
- متى نستخدم التخصيص؟: يُستخدم التخصيص عندما لا يكون القالب العام مناسبًا لنوع معيّن من حيث الأداء أو الدلالة، أو عندما يتطلب ذلك النوع سلوكًا مختلفًا كليًا.

المثال 1: تخصيص كامل للقالب

```
#include <iostream>

// General template
template<typename T>
class Calculator {
public:
   static void add(T a, T b) {
       std::cout << "General addition: " << a + b << std::endl;
```

```
    }
};

// Full specialization for 'char*'
template<>
class Calculator<char*> {
public:
    static void add(char* a, char* b) {
        std::cout << "String concatenation: "
                  << std::string(a) + b << std::endl;
    }
};

int main() {
    Calculator<int>::add(3, 4);
    Calculator<char*>::add("Hello ", "World!");
}
```

الشرح: يعالج القالب المتخصص النوع char* بطريقة مختلفة، حيث يقوم بدمج السلاسل النصية بدل إجراء عملية جمع عددية.

٣.٣.٦ التخصيص الجزئي للقوالب

- التعريف: يسمح التخصيص الجزئي بتعديل سلوك القالب لمجموعة فرعية من الأنواع، بينما يستمر القالب العام في التعامل مع بقية الحالات.
- حالات الاستخدام: يُعد مفيداً عندما يتطلب نمط معيّن من معاملات القالب سلوكًا خاصًا دون المساس بالتنفيذ العام.

المثال 2: تخصيص جزئي للمؤشرات

```cpp
#include <iostream>

// General template
template<typename T>
class Printer {
public:
    static void print(T value) {
        std::cout << "General print: " << value << std::endl;
    }
};

// Partial specialization for pointers
template<typename T>
class Printer<T*> {
public:
    static void print(T* value) {
        if (value)
            std::cout << "Pointer print: " << *value << std::endl;
        else
            std::cout << "Null pointer" << std::endl;
    }
};

int main() {
    int x = 42;
    Printer<int>::print(x);
    Printer<int*>::print(&x);
    Printer<int*>::print(nullptr);
}
```

الشرح: تُعامل الأنواع المؤشرية بشكل مختلف عن الأنواع العادية، مع الإبقاء على السلوك العام لبقية الأنواع.

٤.٣.٦ تخصيص القوالب مع قوالب الأصناف

- يمكن تخصيص قوالب الأصناف تخصيصًا كاملاً أو جزئيًا.
- يمكن أيضًا تخصيص بعض معاملات القالب عند وجود أكثر من معامل.

المثال 3: تخصيص بناءً على معاملات متعددة

```
#include <iostream>

// General template with two parameters
template<typename T1, typename T2>
class Pair {
public:
   static void print(T1 a, T2 b) {
       std::cout << "General Pair: "
                 << a << " and " << b << std::endl;
   }
};

// Partial specialization for identical types
template<typename T>
class Pair<T, T> {
public:
   static void print(T a, T b) {
       std::cout << "Same Type Pair: "
                 << a << " and " << b << std::endl;
```

```
    }
};

int main() {
    Pair<int, double>::print(1, 2.5);
    Pair<int, int>::print(3, 4);
}
```

الشرح: يعالج التخصيص الجزئي الحالة التي يكون فيها نوعا القالب متماثلين، ويطبّق منطقًا مختلفًا عنها مقارنةً بالحالة العامة.

٥.٣.٦ تخصيص الأنواع const و volatile

- يمكن أيضًا تخصيص القوالب للأنواع المقيّدة بـ const أو volatile.

المثال 4: تخصيص للأنواع الثابتة

```
#include <iostream>

// General template
template<typename T>
class Printer {
public:
    static void print(T value) {
        std::cout << "General print: "
                  << value << std::endl;
    }
};

// Partial specialization for const types
```

```
template<typename T>
class Printer<const T> {
public:
   static void print(const T value) {
       std::cout << "Const print: "
                 << value << std::endl;
   }
};

int main() {
   int x = 42;
   const int y = 84;
   Printer<int>::print(x);
   Printer<const int>::print(y);
}
```

الشرح: يسمح هذا التخصيص بالتعامل مع الكائنات غير القابلة للتعديل بصورة مختلفة عن الكائنات القابلة للتغيير.

٦.٣.٦ فوائد وتحديات تخصيص القوالب

الفوائد:

- تحسين الأداء عبر تخصيص السلوك لأنواع محددة.
- زيادة وضوح التصميم من خلال الفصل بين الحالات العامة والحالات الخاصة.

التحديات:

- زيادة التعقيد، مما يجعل القوالب أصعب في الفهم والتصحيح.
- الإفراط في التخصيص قد يؤدي إلى تضخم الشيفرة (Code Bloat).

٧.٣.٦ حالات استخدام عملية

- المكتبة القياسية: تستخدم مكتبة STL تخصيص القوالب داخليًا (مثل تحسينات std::vector).
- المكتبات المخصّصة: يمكن للمطورين بناء واجهات مرنة عبر تخصيص القوالب لأنواع أو أنماط معيّنة.

٨.٣.٦ الخلاصة

يُعد تخصيص القوالب والتخصيص الجزئي من الأدوات الأساسية في C++ الحديثة. فهما يتيحان الموازنة بين العمومية والتخصيص، وتحقيق شيفرة مرنة، قابلة للصيانة، وعالية الأداء، مع تحكم دقيق في سلوك القوالب وفق متطلبات التصميم.

٤.٦ نمط CRTP (Curiously Recurring Template Pattern)

١.٤.٦ مقدمة

تُعد القوالب (Templates) من الخصائص الجوهرية في لغة C++، إذ تُمكّن من كتابة شيفرة عامة غير مرتبطة بأنواع محددة، مما يعزز قابلية إعادة الاستخدام والمرونة التصميمية.
يُعد نمط CRTP (Curiously Recurring Template Pattern) أسلوبًا متقدمًا في استخدام القوالب، حيث ترث الفئة المشتقة من فئة قالبية مُنشأة باستخدام نوعها هي نفسها. أي أن: Derived ترث من Base<Derived>.
يوفر هذا النمط تعدد أشكال وقت الترجمة (Compile-Time Polymorphism) دون الحاجة إلى الدوال الافتراضية، وبالتالي دون كلفة زمنية وقت التشغيل.

٢.٤.٦ شرح نمط CRTP

- ما هو CRTP؟

هو نمط تصميم يعتمد على تمرير النوع المشتق إلى الفئة الأساسية كوسيط قالب، مما يسمح للفئة الأساسية باستدعاء دوال الفئة المشتقة أثناء الترجمة.

- فوائد CRTP:

- تعدد أشكال وقت الترجمة: يحقق سلوكًا مشابهًا للدوال الافتراضية دون كلفة وقت التشغيل.
- إعادة استخدام الشيفرة: يمكن للفئة الأساسية توفير سلوك مشترك يُعاد استخدامه عبر عدة فئات مشتقة.
- تحسين الأداء: يستطيع المترجم إجراء Inlining وتحسينات عدوانية لأن كل شيء معروف وقت الترجمة.

المثال 1: نمط CRTP الأساسي

```cpp
#include <iostream>

template<typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
    void implementation() {
        std::cout << "Base implementation\n";
    }
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Derived implementation\n";
    }
};

int main() {
    Derived d;
    d.interface();  // Outputs: Derived implementation
}
```

الشرح: تستخدم الفئة الأساسية static_cast لاستدعاء دالة implementation() من الفئة المشتقة، محققةً تعدد الأشكال دون دوال افتراضية.

٣.٤.٦ تعدد الأشكال الثابت باستخدام CRTP

- CRTP مقابل تعدد الأشكال وقت التشغيل:

- تعدد الأشكال وقت التشغيل يعتمد على الوراثة والدوال الافتراضية، ويترتب عليه كلفة البحث في جدول الدوال الافتراضية (vtable).
- تعدد الأشكال الثابت باستخدام CRTP يُحسم بالكامل وقت الترجمة، مما يلغي أي كلفة زمنية إضافية.

المثال 2: مقارنة CRTP بالدوال الافتراضية

```
#include <iostream>

// Runtime polymorphism
class BaseVirtual {
public:
   virtual void implementation() const {
       std::cout << "BaseVirtual implementation\n";
   }
   void interface() const { implementation(); }
};

class DerivedVirtual : public BaseVirtual {
public:
   void implementation() const override {
       std::cout << "DerivedVirtual implementation\n";
   }
};

// Static polymorphism (CRTP)
```

```cpp
template<typename Derived>
class BaseStatic {
public:
    void interface() const {
        static_cast<const Derived*>(this)->implementation();
    }
};

class DerivedStatic : public BaseStatic<DerivedStatic> {
public:
    void implementation() const {
        std::cout << "DerivedStatic implementation\n";
    }
};

int main() {
    BaseVirtual* v = new DerivedVirtual();
    v->interface();       // Runtime polymorphism

    DerivedStatic s;
    s.interface();        // Compile-time polymorphism (CRTP)

    delete v;
}
```

الشرح: يُظهر المثال كيف يحقق CRTP سلوكًا متعدد الأشكال دون الاعتماد على الدوال الافتراضية أو كلفة وقت التشغيل.

٦.٤.٤ CRTP لإعادة الاستخدام وMixins

- يمكن استخدام CRTP لبناء فئات Mixin تضيف سلوكًا للفئات المشتقة دون وراثة متعددة أو دوال افتراضية.

المثال 3: CRTP كنمط Mixin

```
#include <iostream>

template<typename Derived>
class Printable {
public:
   void print() const {
       std::cout
           << static_cast<const Derived*>(this)->getName()
           << std::endl;
   }
};

class Person : public Printable<Person> {
public:
   std::string getName() const { return "John Doe"; }
};

int main() {
   Person p;
   p.print();  // Outputs: John Doe
}
```

الشرح: تضيف الفئة Printable وظيفة الطباعة للفئة المشتقة دون معرفة تفاصيلها الداخلية.

٥.٤.٦ CRTP وتسلسل استدعاء الدوال (Method Chaining)

- يتيح CRTP إرجاع مرجع للفئة المشتقة، مما يسمح بتسلسل استدعاءات الدوال.

المثال 4: تسلسل الدوال باستخدام CRTP

```
#include <iostream>

template<typename Derived>
class Chainable {
public:
    Derived& doSomething() {
        std::cout << "Doing something...\n";
        return *static_cast<Derived*>(this);
    }
    Derived& doAnotherThing() {
        std::cout << "Doing another thing...\n";
        return *static_cast<Derived*>(this);
    }
};

class MyClass : public Chainable<MyClass> {
public:
    void finalAction() const {
        std::cout << "Final action\n";
    }
};

int main() {
    MyClass obj;
```

```
    obj.doSomething()
       .doAnotherThing()
       .finalAction();
}
```

الشرح: يعيد كل تابع مرجعًا للفئة المشتقة، مما يسمح بكتابة شيفرة متسلسلة وواضحة.

٦.٤.٦ CRTP وتحسين الأداء

- يتيح CRTP إجراء تحسينات قوية مثل Inlining وإزالة الاستدعاءات غير الضرورية.

المثال 5: كفاءة وقت الترجمة باستخدام CRTP

```
#include <iostream>

template<typename Derived>
class Base {
public:
    void execute() {
        static_cast<Derived*>(this)->run();
    }
};

class Optimized : public Base<Optimized> {
public:
    void run() const {
        std::cout << "Optimized execution\n";
    }
};
```

```
int main() {
   Optimized obj;
   obj.execute();  // Compile-time optimization
}
```

الشرح: يستطيع المترجم دمج الدوال وإزالة أي كلفة إضافية، مما يؤدي إلى تنفيذ عالي الأداء.

٦.٤.٧ قيود وتحديات نمط CRTP

- تعقيد الشيفرة: قد يكون CRTP صعب الفهم للمطورين غير الملمين بالبرمجة القالبية المتقدمة.
- مرونة أقل وقت التشغيل: لأن السلوك يُحسم وقت الترجمة، تقل القدرة على التغيير الديناميكي مقارنة بالدوال الافتراضية.

٦.٤.٨ الخلاصة

يوفر نمط CRTP تعدد أشكال ثابت، وإعادة استخدام فعالة للشيفرة، وتحسينات قوية وقت الترجمة، دون أي كلفة زمنية إضافية وقت التشغيل.
يُعد هذا النمط مثاليًا للتطبيقات الحساسة للأداء والأنظمة التي تتطلب ضمانات صارمة وقت الترجمة، ويُستخدم على نطاق واسع في مكتبات C++ الحديثة والتصميمات منخفضة المستوى.

الفصل ٧: إدارة الاستثناءات في البرمجة الكائنية

- إدارة الاستثناءات في البرمجة الكائنية (Object-Oriented Programming).
- مبدأ RAII (Resource Acquisition Is Initialization) في لغة C++.
- الكلمة المفتاحية noexcept ودورها في تصميم البرمجيات الكائنية.

١.٧ إدارة الاستثناءات في البرمجة الكائنية

١.١.٧ مقدمة

- ما هي إدارة الاستثناءات؟

إدارة الاستثناءات هي آلية للتعامل مع الأخطاء غير المتوقعة التي تحدث أثناء تنفيذ البرنامج، دون التسبب في انهيار التطبيق بالكامل.

- أهميتها في البرمجة الكائنية

في البرمجة الكائنية، تسمح إدارة الاستثناءات بفصل منطق معالجة الأخطاء عن منطق العمل الأساسي، مما يؤدي إلى شيفرة أكثر وضوحًا، وأسهل صيانة، وأكثر موثوقية.

٢.١.٧ أساسيات إدارة الاستثناءات

- ما هو الاستثناء؟

الاستثناء هو خطأ وقت التشغيل يقطع التدفق الطبيعي للبرنامج ويتطلب معالجة خاصة.

- أنواع الاستثناءات:

  - استثناءات قياسية: مثل std::exception في C++ أو System.Exception في C#.
  - استثناءات مخصصة: يقوم المطور بتعريفها للتعبير عن حالات خطأ محددة مرتبطة بمنطق التطبيق.

مثال: إدارة استثناء بسيطة في C++

```
#include <iostream>
#include <stdexcept>

int divide(int numerator, int denominator) {
    if (denominator == 0) {
```

```
        throw std::runtime_error("Division by zero!");
    }
    return numerator / denominator;
}

int main() {
    try {
        std::cout << "Result: " << divide(10, 0) << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }
    return 0;
}
```

الشرح: عند محاولة القسمة على الصفر، يتم رمي استثناء، ويتم التقاطه في كتلة catch بدون انهيار البرنامج.

٣.١.٧ المفاهيم الأساسية في إدارة الاستثناءات

- كتلة **try--catch**: تحتوي الشيفرة التي قد تولد استثناءات، وتوفر آلية لمعالجتها.
- رمي الاستثناءات: باستخدام الكلمة المفتاحية throw.
- التقاط الاستثناءات: باستخدام كتلة catch.
- تعدد كتل اللالتقاط: لمعالجة أنواع مختلفة من الاستثناءات.

مثال: استخدام عدة كتل catch

```
#include <iostream>
```

```
int main() {
    try {
        throw 10;
    } catch (int e) {
        std::cerr << "Integer exception caught: " << e << std::endl;
    } catch (...) {
        std::cerr << "Unknown exception caught" << std::endl;
    }
    return 0;
}
```

الشرح: يتم التقاط استثناء من نوع محدد، وفي حال حدوث أي استثناء آخر تتم معالجته باستخدام catch(...).

٤.١.٧ إدارة الاستثناءات في البرمجة الكائنية

- دور الاستثناءات في OOP: تجعل الشيفرة أكثر وضوحًا عبر فصل منطق الأخطاء عن منطق العمليات.
- تمرير الاستثناءات: يمكن تمرير الاستثناء عبر مكدس الاستدعاءات حتى مستوى أعلى قادر على معالجته.
- التغليف: تغليف منطق الاستثناءات داخل الكائنات يعزز المتانة والاعتمادية.

مثال: إدارة الاستثناءات داخل فئة

```
#include <iostream>
#include <stdexcept>

class Calculator {
public:
```

```
    int divide(int a, int b) {
        if (b == 0)
            throw std::invalid_argument("Cannot divide by zero.");
        return a / b;
    }
};

int main() {
    Calculator calc;
    try {
        std::cout << calc.divide(10, 0) << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

الشرح: الفئة تقوم برمي الاستثناء، بينما تُترك مسؤولية المعالجة للمستوى الأعلى.

٥.١.٧ أفضل الممارسات في إدارة الاستثناءات

- استخدام الاستثناءات للحالات الاستثنائية فقط، وليس للتحكم في التدفق الطبيعي.
- الاعتماد على مبدأ RAII لضمان تحرير الموارد تلقائيًا.
- تجنب التقاط الاستثناءات العامة دون سبب واضح.
- توثيق الاستثناءات التي قد ترميها الدوال.

مثال: **RAII** مع الاستثناءات

```cpp
#include <iostream>
#include <stdexcept>

class Resource {
public:
    Resource() { std::cout << "Acquiring resource\n"; }
    ~Resource() { std::cout << "Releasing resource\n"; }
    void doWork() {
        throw std::runtime_error("Error during work");
    }
};

int main() {
    try {
        Resource r;
        r.doWork();
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

الشرح: حتى عند وقوع الاستثناء، يتم استدعاء المدمّر تلقائيًا، مما يضمن تحرير المورد.

٦.١.٧ الاستثناءات المخصصة

- تسمح الاستثناءات المخصصة بالتعبير الدقيق عن حالات الخطأ.
- غالبًا ما ترث من std::exception.

مثال: استثناء مخصص

```cpp
#include <iostream>
#include <exception>

class DivisionByZeroException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Division by zero exception!";
    }
};

class Calculator {
public:
    int divide(int a, int b) {
        if (b == 0) throw DivisionByZeroException();
        return a / b;
    }
};

int main() {
    Calculator calc;
    try {
        std::cout << calc.divide(10, 0) << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

الشرح: الاستثناء المخصص يوفر رسالة واضحة تعكس طبيعة الخطأ بدقة.

٧.١.٧ مستويات أمان الاستثناءات

- الضمان الأساسي: يبقى البرنامج في حالة صالحة، حتى لو تغيرت بعض القيم.
- الضمان القوي: لا تحدث أي آثار جانبية عند وقوع الاستثناء.
- ضمان عدم الرمي: الدالة تضمن عدم رمي أي استثناء.

٨.١.٧ الخلاصة

- إدارة الاستثناءات عنصر أساسي لبناء برمجيات متينة وقابلة للصيانة.
- الاستخدام الصحيح للاستثناءات، مع RAII والاستثناءات المخصصة، يؤدي إلى تصميم احترافي وآمن في البرمجة الكائنية.

٢.٧ مبدأ RAII في لغة C++

١.٢.٧ مقدمة

يُعد مبدأ **Resource Acquisition Is Initialization (RAII)** أحد أهم الركائز في C++ الحديثة للإدارة الموارد مثل الذاكرة، ومقابض الملفات، واتصالات الشبكة، والمزامنات (mutexes). ويُستخدم هذا المبدأ على نطاق واسع في البرمجة الكائنية لضمان تحرير الموارد بشكل آمن، حتى في حال وقوع الاستثناءات.

يربط RAII عمر المورد بعمر الكائن، مما يجعل إدارة الموارد تلقائية، وقابلة للتنبؤ، وخالية من التسريبات.

٢.٢.٧ ما هو مبدأ RAII؟

- التعريف

  RAII هو نمط برمجي يتم فيه ربط امتلاك المورد بعمر الكائن. عند إنشاء الكائن يتم الاستحواذ على المورد، وعند خروج الكائن من النطاق يتم تحرير المورد تلقائيًا.

- المكونات الأساسية

  - الاستحواذ: يتم في المُنشئ (constructor).
  - التحرير: يتم في المُدمّر (destructor).

- أهميته مع الاستثناءات

  عند وقوع استثناء، قد يتجاوز البرنامج مسار التنفيذ الطبيعي. بدون RAII قد تُترك الموارد دون تحرير. أما مع RAII فيتم استدعاء المُدمّرات تلقائيًا، مما يمنع تسريب الموارد.

٣.٢.٧ فهم RAII من خلال مثال بسيط

مثال أساسي: إدارة الذاكرة باستخدام RAII

```cpp
#include <iostream>

class SmartPointer {
private:
    int* ptr;

public:
    SmartPointer(int* p = nullptr) : ptr(p) {
        std::cout << "Resource acquired\n";
    }

    ~SmartPointer() {
        delete ptr;
        std::cout << "Resource released\n";
    }

    int* get() const { return ptr; }
};

int main() {
    {
        SmartPointer sp(new int(42));
        std::cout << "Value: " << *sp.get() << std::endl;
    } // automatic release here
    return 0;
}
```

الشرح:

- يتم تخصيص الذاكرة في المُنشئ.

- يتم تحرير الذاكرة تلقائيًا في المُدمّر.
- لا حاجة إلى استدعاء delete يدويًا.

الناتج:

```
Resource acquired
Value: 42
Resource released
```

٤.٢.٧ RAII وأمان الاستثناءات

يرتبط مبدأ RAII ارتباطًا وثيقًا بأمان الاستثناءات (Exception Safety)، إذ يضمن تحرير الموارد حتى عند حدوث خطأ مفاجئ.

```cpp
#include <iostream>
#include <stdexcept>

class FileHandler {
private:
    FILE* file;

public:
    FileHandler(const char* filename) {
        file = fopen(filename, "w");
        if (!file)
            throw std::runtime_error("Failed to open file");
        std::cout << "File opened successfully\n";
    }
```

```
    ~FileHandler() {
        if (file) {
            fclose(file);
            std::cout << "File closed successfully\n";
        }
    }

    void write(const char* data) {
        fputs(data, file);
    }
};

int main() {
    try {
        FileHandler fh("example.txt");
        fh.write("Hello, World!");
        throw std::runtime_error("Unexpected error");
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

الشرح: حتى مع رمي استثناء، يتم إغلاق الملف تلقائيًا عند خروج الكائن من النطاق.

٥.٢.٧ RAII في البرمجة المتزامنة

يُعد RAII عنصرًا أساسيًا في البرمجة متعددة الخيوط، خاصة عند التعامل مع mutex.

```
#include <iostream>
```

```cpp
#include <thread>
#include <mutex>

std::mutex mtx;

void print(int id) {
    std::lock_guard<std::mutex> lock(mtx);
    std::cout << "Thread " << id << " is running\n";
}

int main() {
    std::thread t1(print, 1);
    std::thread t2(print, 2);
    t1.join();
    t2.join();
}
```

الشرح: std::lock_guard يقفل الـ mutex عند إنشائه، ويفك القفل تلقائيًا عند الخروج من النطاق، حتى في حال وقوع استثناء.

٦.٢.٧ RAII في C++ الحديثة: المؤشرات الذكية

المؤشرات الذكية هي تطبيق مباشر لمبدأ RAII.

```cpp
#include <iostream>
#include <memory>

void process(std::unique_ptr<int> ptr) {
    std::cout << "Value: " << *ptr << std::endl;
}
```

```
int main() {
    auto p = std::make_unique<int>(100);
    process(std::move(p));
}
```

الشرح:

- std::unique_ptr يضمن تحرير الذاكرة تلقائيًا.
- نقل الملكية باستخدام std::move يحافظ على قواعد RAII.

٧.٢.٧ الخلاصة

يربط مبدأ RAII إدارة الموارد بعمر الكائن، مما يجعل البرامج:

- أكثر أمانًا.
- مقاومة لتسريب الموارد.
- متوافقة مع الاستثناءات.

أهم النقاط:

- التحرير التلقائي للموارد.
- أمان قوي مع الاستثناءات.
- دعم واسع في مكتبة C++ القياسية.

٣.٧ المواصفة noexcept واستخدامها في البرمجة الكائنية

١.٣.٧ مقدمة

- مراجعة معالجة الاستثناءات
تُعد معالجة الاستثناءات عنصرًا جوهريًا في البرمجة الكائنية لضمان التعامل السليم مع الأخطاء وقت التنفيذ دون انهيار النظام أو تسريب الموارد.

- ما هي **noexcept**؟
الكلمة المفتاحية noexcept في C++ تُستخدم للإعلان صراحةً أن الدالة لن تقوم برمي أي استثناء.

- لماذا **noexcept** مهمة في OOP؟
لأنها:
  - تحسّن الأداء.
  - تتيح تحسينات ترجمة أعمق.
  - تضمن سلوكًا متوقعًا للكائنات، خاصة في المُدمّرات وعمليات النقل (move semantics).

٢.٣.٧ ما هي **noexcept** في C++؟

- التعريف
noexcept هي مواصفة لغوية تُخبر المترجم أن الدالة لا يمكن أن ترمي استثناءات. ويمكن استخدامها مع دوال المستخدم ودوال المكتبة القياسية.

- الصياغة

مثال أساسي على استخدام **noexcept**

```
#include <iostream>
```

```cpp
#include <stdexcept>

void safeFunction() noexcept {
    std::cout << "This function never throws\n";
}

void riskyFunction() {
    throw std::runtime_error("May throw");
}

int main() {
    safeFunction();
    try {
        riskyFunction();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}
```

الشرح:

- الدالة safeFunction مضمونة بعدم رمي استثناء.
- الدالة riskyFunction قد ترمي استثناء ويتم التعامل معه.

٣.٣.٧ المفاهيم الأساسية لـ noexcept

- ضمان استثناء ثابت
noexcept يوفر ضمانًا معروفًا وقت الترجمة وليس وقت التنفيذ فقط.

- الفرق بين الضمانات

  - الضمان التقليدي: يُفحص أثناء التنفيذ.
  - noexcept: يُستنتج أثناء الترجمة.

- تعبيرات **noexcept**

يمكن استخدام noexcept كتعبير منطقي.

مثال: تعبير **noexcept**

```cpp
#include <iostream>

void safe() noexcept {}
void unsafe() {}

int main() {
    std::cout << std::boolalpha;
    std::cout << noexcept(safe()) << std::endl;
    std::cout << noexcept(unsafe()) << std::endl;
}
```

الشرح:

- noexcept(safe()) تُرجع true.
- noexcept(unsafe()) تُرجع false.

٤.٣.٧ فوائد **noexcept** في البرمجة الكائنية

- تحسين الأداء

المترجم يمكنه:

- إزالة مسارات معالجة الاستثناءات.
- توليد شيفرة أسرع.

- ضمانات أقوى للكائنات
ضرورية بشكل خاص في:
  - المُدمّرات.
  - مُنشئات النقل.

مثال: **noexcept** مع **Move Semantics**

```cpp
#include <vector>

class NoexceptMove {
public:
    NoexceptMove() = default;
    NoexceptMove(const NoexceptMove&) = delete;
    NoexceptMove(NoexceptMove&&) noexcept = default;
};

int main() {
    std::vector<NoexceptMove> v;
    v.push_back(NoexceptMove());
}
```

الشرح: المكتبات القياسية تُفضل النقل فقط إذا كان noexcept، مما يجعل الحاويات أكثر كفاءة.

٥.٣.٧ المُدمّرات و noexcept

- المُدمّرات في C++ هي noexcept ضمنيًا.

- رمي استثناء داخل المُدمّر يؤدي إلى سلوك غير معرّف.

مثال: مُدمّر آمن وغير آمن

```
#include <iostream>
#include <stdexcept>

class Safe {
public:
    ~Safe() noexcept {
        std::cout << "Safe destructor\n";
    }
};

class Unsafe {
public:
    ~Unsafe() {
        throw std::runtime_error("Bad destructor");
    }
};
```

٦.٣.٧ رمي استثناء داخل دالة noexcept

السلوك: إذا تم رمي استثناء داخل دالة مُعلّمة بـ noexcept، فسيتم استدعاء std::terminate مباشرة.

```
#include <stdexcept>

void danger() noexcept {
```

```
    throw std::runtime_error("Fatal");
}

int main() {
    danger(); // std::terminate
}
```

النتيجة: انتهاء فوري للبرنامج.

٧.٣.٧ noexcept مع الوراثة

- عند إعادة تعريف دالة افتراضية، يجب الحفاظ على نفس مواصفة noexcept.

مثال: **noexcept** مع الدوال الافتراضية

```
#include <iostream>

class Base {
public:
    virtual void f() noexcept {
        std::cout << "Base\n";
    }
};

class Derived : public Base {
public:
    void f() noexcept override {
        std::cout << "Derived\n";
    }
};
```

٨.٣.٧ أفضل الممارسات

- استخدم noexcept فقط عندما يكون الضمان مؤكداً.
- مثالي لـ:
  - المُدمّرات.
  - مُنشئات النقل.
  - الدوال الخدمية الصغيرة.
- لا تُفرط في استخدامه.

٩.٣.٧ الخلاصة

الخلاصة: noexcept أداة تصميم قوية في البرمجة الكائنية، تجعل الشيفرة:

- أسرع.
- أكثر أمانًا.
- أسهل في التحليل والصيانة.

أهم النقاط:

- تحسينات أداء حقيقية.
- ضمانات قوية لإدارة الموارد.
- سلوك واضح ومتوقع للكائنات.

الفصل ٨: التكامل مع المكتبات الخارجية في C++

- استخدام مكتبات Boost في البرمجة الكائنية.
- توظيف إطار العمل Qt لتبسيط تطوير C++ بأسلوب كائني.

١.٨ استخدام مكتبات Boost في البرمجة الكائنية

١.١.٨ مقدمة

تلعب المكتبات الخارجية دورًا أساسيًا في تطوير تطبيقات C++ الحديثة، حيث تساهم في:

- تسريع عملية التطوير.
- تقليل الأخطاء.
- توفير حلول جاهزة ومجربة لمشكلات شائعة.

من أبرز هذه المكتبات **Boost**، وهي مجموعة ضخمة من مكتبات C++ المراجَعة بعناية، والمصممة لتكملة المكتبة القياسية.
توفر Boost أدوات قوية مثل:

- المؤشرات الذكية.
- التعابير النمطية.
- الخيوط المتعددة.
- التعامل مع الملفات.
- البرمجة غير المتزامنة.

وتندمج هذه الأدوات بسلاسة مع أسلوب البرمجة الكائنية (OOP) في Modern C++، مما يجعلها مثالية للمشاريع الكبيرة والمعقدة.

٢.١.٨ ما هي مكتبة Boost؟

Boost هي مجموعة من المكتبات القابلة لإعادة الاستخدام، تهدف إلى توسيع قدرات C++ بما يتجاوز ما توفره STL.
تغطي المكتبة نطاقًا واسعًا من الوظائف مثل:

- إدخال وإخراج الملفات.
- الشبكات.
- التعابير النمطية.
- إدارة الذاكرة المتقدمة.
- التزامن والخيوط.

العديد من مكونات Boost أصبحت لاحقًا جزءاً من معيار C++، مثل: std::shared_ptr و std::thread.

٣.١.٨ لماذا نستخدم Boost في OOP؟

دمج Boost في مشروع كائني يوفر مزايا مهمة:

- إعادة استخدام الشيفرة
  حلول جاهزة تقلل من إعادة اختراع الحلول.
- التصميم المعياري
  يمكن استخدام كل مكتبة بشكل مستقل، مما يتماشى مع التصميم الكائني النظيف.
- إدارة الذاكرة
  المؤشرات الذكية تقلل من التسريبات وتحسّن أمان الكائنات.

٤.١.٨ تثبيت مكتبة Boost

على نظام **Linux**:

```
sudo apt-get install libboost-all-dev
```

على نظام **Windows**:
يمكن تحميل النسخ الجاهزة أو البناء من المصدر وفق الإرشادات الرسمية.
الإدراج في المشروع:

```
#include <boost/shared_ptr.hpp>
```

٥.١.٨ استخدام Boost في التصميم الكائني

المؤشرات الذكية في **Boost**

إدارة الذاكرة عنصر محوري في OOP، خاصة عند التعامل مع كائنات ديناميكية.
توفر Boost مؤشرات ذكية مثل: boost::shared_ptr و boost::scoped_ptr.
مثال: استخدام **boost::shared_ptr**

```
#include <iostream>
#include <boost/shared_ptr.hpp>

class Book {
public:
    std::string title;

    Book(const std::string& t) : title(t) {
        std::cout << "Book created: " << title << std::endl;
    }

    ~Book() {
        std::cout << "Book destroyed: " << title << std::endl;
    }
```

```
    void display() const {
        std::cout << "Title: " << title << std::endl;
    }
};

class Library {
    boost::shared_ptr<Book> book;
public:
    Library(const boost::shared_ptr<Book>& b) : book(b) {}
    void show() const { book->display(); }
};

int main() {
    boost::shared_ptr<Book> b(new Book("Modern C++"));
    Library lib(b);
    lib.show();
}
```

الشرح:

- الذاكرة تُدار تلقائيًا.
- يُحذف الكائن عند انتهاء جميع المراجع.

التعابير النمطية باستخدام Boost.Regex

تسمح Boost.Regex بمعالجة النصوص المعقدة ضمن تصميم كائني منظم.

```
#include <boost/regex.hpp>
#include <string>
```

```
class EmailValidator {
public:
    static bool validate(const std::string& email) {
        boost::regex pattern(R"((\w+)(\.\w+)*@(\w+)\.(\w+))");
        return boost::regex_match(email, pattern);
    }
};
```

الشرح: تم احتواء منطق التحقق داخل صنف مستقل، مما يعزز مبدأ المسؤولية الواحدة.

Boost.Asio والبرمجة غير المتزامنة

توفر Boost.Asio دعمًا قويًا للعمليات غير المتزامنة، مثل:

- الشبكات.
- الإدخال والإخراج غير المحجوب.

ويمكن احتواء هذا السلوك داخل كائنات تحافظ على وضوح التصميم.

Boost.Filesystem وإدارة الملفات

توفر Boost.Filesystem واجهة موحدة لإدارة الملفات بشكل مستقل عن النظام.

```
#include <boost/filesystem.hpp>
#include <iostream>

class FileManager {
public:
    void createDir(const std::string& name) {
        boost::filesystem::create_directory(name);
```

```
    }

    void list(const std::string& dir) {
        for (auto& e : boost::filesystem::directory_iterator(dir)) {
            std::cout << e.path().string() << std::endl;
        }
    }
};
```

الشرح: تم دمج التعامل مع الملفات داخل صنف، بما يتوافق مع مبادئ OOP.

٦.١.٨ الخلاصة

تُعد مكتبات Boost مكملاً قوياً للغة C++، حيث توفر حلولاً جاهزة لإدارة الذاكرة، النصوص، الملفات، والبرمجة غير المتزامنة.
الفوائد الأساسية:

- تصميم أنظف وأكثر تنظيماً.
- شيفرة قابلة لإعادة الاستخدام.
- أداء أعلى واستقرار أفضل.

يؤدي دمج Boost مع التصميم الكائني إلى بناء أنظمة قوية، قابلة للصيانة، ومناسبة للتطبيقات الاحترافية طويلة العمر.

٢.٨ استخدام إطار العمل Qt في البرمجة الكائنية لتبسيط C++

١.٢.٨ مقدمة

تُعد لغة C++ من أقوى لغات البرمجة، حيث تمنح المطور تحكمًا كاملاً في:

- موارد النظام.
- إدارة الذاكرة.
- الأداء المنخفض المستوى.

لكن هذه القوة تأتي مصحوبة بدرجة عالية من التعقيد، خصوصًا عند تطوير التطبيقات الكبيرة أو الرسومية. هنا يأتي دور أطر العمل الخارجية مثل **Qt**، التي تهدف إلى تبسيط تطوير C++ دون التضحية بالكفاءة أو الأداء.
إطار العمل Qt هو إطار مفتوح المصدر ومتعدد المنصات، ويُعرف على نطاق واسع بتسهيل تطوير الواجهات الرسومية. إضافة إلى ذلك، يوفر Qt وحدات قوية للشبكات، الملفات، الخيوط، التزامن، وقواعد البيانات، مما يجعله مناسبًا للتطبيقات الرسومية وغير الرسومية على حد سواء.

٢.٢.٨ لماذا نستخدم Qt مع OOP في C++؟

دمج Qt مع التصميم الكائني يوفر مزايا جوهرية:

- التبسيط عبر التجريد
  يقوم Qt بإخفاء التفاصيل منخفضة المستوى مثل إدارة الأحداث، الذاكرة، ورسم الواجهات، مما يقلل العبء الذهني على المطور.
- دعم تعدد المنصات
  نفس الشيفرة تعمل على Windows، Linux، macOS دون تعديلات جوهرية.
- هيكلية معيارية كائنية
  يعتمد Qt على تصميم صنفي (Class-Based) ينسجم تمامًا مع مبادئ OOP، مما يسهل فصل الواجهة عن المنطق والخدمات الخلفية.

٣.٢.٨ إعداد بيئة Qt مع C++

خطوات الإعداد الأساسية:

١. تحميل وتثبيت Qt من الموقع الرسمي.

٢. استخدام بيئة التطوير Qt Creator.

٣. إنشاء مشروع جديد من نوع:

- Qt Widgets Application للتطبيقات الرسومية.
- Qt Console Application للتطبيقات غير الرسومية.

بعد ذلك يمكن إدراج وحدات Qt داخل الأصناف بسهولة.

٤.٢.٨ أمثلة عملية لاستخدام Qt بأسلوب كائني

مثال 1: تبسيط إنشاء الواجهات الرسومية

يتولى Qt إدارة النوافذ، حلقة الأحداث، والعناصر الرسومية تلقائيًا، مما يقلل الشيفرة النمطية في C++.

```
#include <QApplication>
#include <QPushButton>

class MyApp {
public:
    void run(int argc, char *argv[]) {
        QApplication app(argc, argv);

        QPushButton button("Click Me");
        button.resize(200, 100);
```

```
        button.show();

        app.exec();
    }
};

int main(int argc, char *argv[]) {
    MyApp app;
    app.run(argc, argv);
}
```

التبسيطات المحققة:

- QApplication يدير حلقة الأحداث.
- QPushButton يتكفل بإنشاء الزر.
- لا حاجة للتعامل اليدوي مع النظام الرسومي.

مثال 2: معالجة الأحداث بأسلوب كائني عبر الإشارات والمنافذ

يوفر Qt آلية **Signal-Slot** لفصل مصدر الحدث عن معالجته.

```
#include <QApplication>
#include <QPushButton>
#include <QObject>

class ButtonHandler : public QObject {
    Q_OBJECT
public slots:
    void onClick() {
```

```
        qDebug("Button clicked!");
    }
};

class MyApp {
public:
    void run(int argc, char *argv[]) {
        QApplication app(argc, argv);
        QPushButton button("Click Me");

        ButtonHandler handler;
        QObject::connect(&button, &QPushButton::clicked,
                         &handler, &ButtonHandler::onClick);

        button.show();
        app.exec();
    }
};
```

الفوائد:

- فصل منطق الحدث عن الواجهة.
- تصميم كائني نظيف وقابل للتوسعة.

مثال 3: الاحتواء والتقسيم المعياري

يمكن احتواء الواجهة بالكامل داخل صنف مستقل.

```
#include <QWidget>
#include <QPushButton>
```

```
class MainWindow : public QWidget {
public:
    MainWindow() {
        setWindowTitle("Modular Window");
        setFixedSize(300, 200);

        QPushButton *btn = new QPushButton("Close", this);
        btn->setGeometry(100, 100, 100, 40);
        connect(btn, &QPushButton::clicked, this, &QWidget::close);
    }
};
```

النتيجة:

- كود أوضح.
- إعادة استخدام أسهل.
- التزام صارم بمبادئ OOP.

مثال 4: تبسيط التعامل مع الملفات

يوفر Qt طبقات عالية المستوى للتعامل مع الملفات.

```
#include <QFile>
#include <QTextStream>
#include <QDebug>

class FileReader {
public:
```

```
    void read(const QString& path) {
        QFile file(path);
        if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
            qDebug() << "Cannot open file";
            return;
        }
        QTextStream in(&file);
        while (!in.atEnd()) {
            qDebug() << in.readLine();
        }
    }
};
```

المزايا:

- تقليل الشيفرة.
- إدارة أخطاء أسهل.
- واجهة برمجية موحدة عبر الأنظمة.

٥.٢.٨ مزايا استخدام Qt في المشاريع الكائنية

- شيفرة أنظف وأكثر وضوحًا.
- تسريع عملية التطوير.
- دعم كامل للتطبيقات متعددة المنصات.
- بنية قائمة على الأحداث.
- أدوات تطوير متكاملة لتصميم الواجهات والاختبار.

٦.٢.٨ الخلاصة

يُعد دمج Qt مع C++ نهجًا عمليًا يجمع بين:

- قوة الأداء.
- بساطة التطوير.
- وضوح التصميم الكائني.

يساعد Qt على تقليل التعقيد، وتحسين قابلية الصيانة، وتسريع بناء التطبيقات الواقعية.
وباستخدام تصميمه المتوافق مع OOP، يمكن بناء أنظمة قوية، قابلة للتوسع، ومناسبة للاستخدام الاحترافي طويل الأمد.

الفصل ٩: أفضل الممارسات في البرمجة الكائنية باستخدام C++

- مبادئ SOLID.
- مبدأ DRY (Don’t Repeat Yourself).
- مبدأ KISS (Keep It Simple, Stupid).
- قانون Law of Demeter.

١.٩ فهم وتطبيق مبادئ SOLID

١.١.٩ مقدمة

ترتكز البرمجة الكائنية (Object-Oriented Programming - OOP) على مفهوم الكائنات، وهي نُسخ (Instances) من الأصناف (Classes). إن التطبيق الصحيح لمبادئ OOP يؤدي إلى شيفرة:

- سهلة الصيانة.
- قابلة للتوسع.
- قوية ومقاومة للأخطاء.

من بين أهم الإرشادات المؤثرة في تصميم البرمجيات الكائنية تأتي مبادئ **SOLID**، والتي قدمها Robert C. Martin (Uncle Bob). تساعد هذه المبادئ في بناء أنظمة برمجية مرنة، مفهومة، وقابلة للتطوير على المدى الطويل.
يشير الاختصار SOLID إلى خمسة مبادئ أساسية:

١. **Single Responsibility Principle (SRP)**
٢. **Open/Closed Principle (OCP)**
٣. **Liskov Substitution Principle (LSP)**
٤. **Interface Segregation Principle (ISP)**
٥. **Dependency Inversion Principle (DIP)**

في هذا القسم، سنستعرض كل مبدأ مع أمثلة عملية بلغة C++ وأفضل الممارسات المرتبطة به.

٢.١.٩ مبدأ المسؤولية الواحدة (SRP)

التعريف: يجب أن يكون للصنف سبب واحد فقط للتغيير، أي أن تكون له مسؤولية واحدة واضحة.
الأهمية: يقلل هذا المبدأ من التعقيد، ويجعل الصنف أسهل للفهم والاختبار والصيانة.
مثال - انتهاك المبدأ:

```
class UserManager {
public:
    void addUser(const std::string& username) {
        // Add user
        logger.log("User added: " + username);
    }
private:
    Logger logger;
};
```

المشكلة: الصنف UserManager مسؤول عن إدارة المستخدمين وعن التسجيل (Logging) في الوقت نفسه.
تطبيق مبدأ **SRP**:

```
class UserLogger {
public:
    void log(const std::string& message) {
        // Log message
    }
};

class UserManager {
public:
    void addUser(const std::string& username) {
        // Add user
```

```
        userLogger.log("User added: " + username);
    }
private:
    UserLogger userLogger;
};
```

٣.١.٩ مبدأ الفتح/الإغلاق (OCP)

التعريف: يجب أن تكون الكيانات البرمجية مفتوحة للإضافة، ولكن مغلقة أمام التعديل.
الأهمية: يسمح هذا المبدأ بإضافة وظائف جديدة دون تعديل الشيفرة الحالية، مما يقلل احتمالية إدخال أخطاء.
مثال تطبيقي:

```
class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override { return width * height; }
private:
    double width, height;
};

class Circle : public Shape {
public:
    Circle(double r) : radius(r) {}
```

```
    double area() const override { return 3.14159 * radius * radius; }
private:
    double radius;
};
```

تطبيق **OCP**: يمكن إضافة أشكال جديدة دون تعديل الصنف Shape.

٤.١.٩ مبدأ استبدال ليسكوف (LSP)

التعريف: يجب أن تكون الكائنات من الأصناف المشتقة قابلة للاستبدال بكائنات الصنف الأساسي دون التأثير على صحة البرنامج.
الأهمية: يضمن هذا المبدأ إعادة الاستخدام الصحيحة والموثوقة عبر الوراثة.
مثال – انتهاك المبدأ:

```
class Bird {
public:
    virtual void fly() = 0;
};

class Sparrow : public Bird {
public:
    void fly() override { /* fly */ }
};

class Ostrich : public Bird {
public:
    void fly() override { throw std::logic_error("Cannot fly!"); }
};
```

المشكلة: ليس كل طائر يطير، مما يجعل الاستبدال غير صحيح.
تطبيق مبدأ **LSP**:

```
class Bird {
public:
    virtual void move() = 0;
};

class Sparrow : public Bird {
public:
    void move() override { /* fly */ }
};

class Ostrich : public Bird {
public:
    void move() override { /* run */ }
};
```

٥.١.٩ مبدأ فصل الواجهات (ISP)

التعريف: يجب ألا يُجبر العميل على الاعتماد على واجهات لا يستخدمها.
الأهمية: يمنع هذا المبدأ تضخم الواجهات ويجعل الأصناف أكثر وضوحًا.
مثال – انتهاك المبدأ:

```
class Worker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
};
```

تطبيق مبدأ **ISP**:

```
class Workable {
```

```
public:
    virtual void work() = 0;
};

class Eatable {
public:
    virtual void eat() = 0;
};

class HumanWorker : public Workable, public Eatable {
public:
    void work() override { /* work */ }
    void eat() override { /* eat */ }
};
```

٦.١.٩ مبدأ قلب الاعتماديات (DIP)

التعريف: يجب ألا تعتمد الوحدات عالية المستوى على الوحدات منخفضة المستوى، بل يجب أن يعتمد كلاهما على تجريدات.

الأهمية: يعزز هذا المبدأ الترابط الضعيف (Loose Coupling) وقابلية التوسع.

مثال - انتهاك المبدأ:

```
class FileManager {
public:
    void saveToFile(const std::string& data) {
        std::ofstream file("output.txt");
        file << data;
    }
};
```

تطبيق مبدأ **DIP**:

```
class IDataSaver {
public:
    virtual void save(const std::string& data) = 0;
};

class FileDataSaver : public IDataSaver {
public:
    void save(const std::string& data) override {
        std::ofstream file("output.txt");
        file << data;
    }
};

class DataManager {
public:
    DataManager(IDataSaver* saver) : dataSaver(saver) {}
    void saveData(const std::string& data) {
        dataSaver->save(data);
    }
private:
    IDataSaver* dataSaver;
};
```

٧.١.٩ الخلاصة

تُعد مبادئ SOLID حجر الأساس في بناء برمجيات C++ كائنية قوية وقابلة للصيانة. من خلال تطبيق:

- SRP
- OCP
- LSP
- ISP
- DIP

يمكن للمطورين:

- تحسين بنية النظام.
- تقليل الترابط غير الضروري.
- زيادة قابلية التوسع وإعادة الاستخدام.

إن الالتزام بهذه المبادئ في تصميم البرمجيات الكائنية باستخدام C++ يؤدي إلى أنظمة أكثر جودة واستدامة على المدى الطويل.

٢.٩ مبدأ DRY (Don't Repeat Yourself)

١.٢.٩ أهمية مبدأ DRY

يُعد مبدأ DRY (Don't Repeat Yourself) أحد أهم أفضل الممارسات في البرمجة الكائنية (OOP)، وخصوصًا في C++ الحديثة. يركّز هذا المبدأ على تقليل تكرار الشيفرة بهدف تحسين:

- قابلية الصيانة.
- وضوح التصميم.
- الكفاءة والاستقرار.

اتباع DRY يساعد المطورين على كتابة شيفرة أكثر تنظيمًا، قابلة لإعادة الاستخدام، وأقل عرضة للأخطاء أو التناقضات المنطقية.

٢.٢.٩ ما هو مبدأ DRY؟

يعني مبدأ DRY أن كل منطق أو معرفة في قاعدة الشيفرة يجب أن يكون له تمثيل واحد فقط موثوق. أي أن تكرار نفس المنطق في أماكن متعددة يُعد مؤشرًا على تصميم ضعيف.
الفوائد الأساسية:

- تقليل جهد الصيانة: التعديل يتم في مكان واحد فقط.
- تحسين الاتساق: وجود مصدر واحد للحقيقة يمنع السلوك غير المتناسق.
- تقليل الأخطاء: يقل احتمال ظهور أخطاء ناتجة عن نسيان تحديث أحد المواضع المكررة.

٣.٢.٩ الانتهاكات الشائعة لمبدأ DRY في C++

تشمل الانتهاكات الشائعة ما يلي:

- تكرار الدوال: دوال متعددة تؤدي نفس المهمة مع اختلافات طفيفة.
- النسخ واللصق: تكرار الشيفرة بين أصناف أو دوال مختلفة.
- القيم الصلبة **(Hardcoded Values)**: تكرار القيم العددية أو النصية بدل تعريفها مركزيًا.

٤.٢.٩ تطبيق مبدأ DRY في C++

ا. استخدام الدوال والقوالب **(Functions & Templates)**

يتم تجميع الشيفرة المتكررة في دالة واحدة أو قالب واحد يعالج عدة أنواع.

مثال (قبل تطبيق **DRY**):

```
int add_ints(int a, int b) { return a + b; }
double add_doubles(double a, double b) { return a + b; }
```

بعد تطبيق **DRY** باستخدام القوالب:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

ب. استخدام الوراثة وتعدد الأشكال

يمكن تجريد السلوك المشترك بين الأصناف باستخدام صنف أساسي وتحقيقه عبر الوراثة وتعدد الأشكال.

مثال (قبل **DRY**):

```
class Dog {
public:
    void speak() { std::cout << "Bark!\n"; }
```

```
};

class Cat {
public:
    void speak() { std::cout << "Meow!\n"; }
};
```

بعد تطبيق DRY:

```
class Animal {
public:
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    void speak() override { std::cout << "Bark!\n"; }
};

class Cat : public Animal {
public:
    void speak() override { std::cout << "Meow!\n"; }
};
```

ج. استخدام الثوابت والتعدادات

تجنب تكرار القيم الصلبة باستخدام ثوابت أو Enums.

مثال (قبل DRY):

```
if (status == 1) { std::cout << "Status is active\n"; }
if (status == 1) { /* do something else */ }
```

بعد تطبيق **DRY**:

```
constexpr int STATUS_ACTIVE = 1;

if (status == STATUS_ACTIVE) {
    std::cout << "Status is active\n";
}
if (status == STATUS_ACTIVE) {
    /* do something else */
}
```

د. تطبيق DRY في الاختبارات

ينطبق مبدأ DRY أيضًا على اختبارات الوحدات باستخدام Test Fixtures والاختبارات المُعلمة.

مثال (قبل **DRY**):

```
TEST(CalculatorTests, AddTest) {
    Calculator calc;
    EXPECT_EQ(calc.add(1, 2), 3);
}

TEST(CalculatorTests, SubtractTest) {
    Calculator calc;
    EXPECT_EQ(calc.subtract(5, 3), 2);
}
```

بعد تطبيق **DRY**:

```
class CalculatorTest : public ::testing::Test {
protected:
    Calculator calc;
};
```

```
TEST_F(CalculatorTest, AddTest) {
    EXPECT_EQ(calc.add(1, 2), 3);
}

TEST_F(CalculatorTest, SubtractTest) {
    EXPECT_EQ(calc.subtract(5, 3), 2);
}
```

٥.٢.٩ فوائد تطبيق DRY في C++

- سهولة الصيانة: التعديلات تتم في موضع واحد.
- وضوح الشيفرة: تقليل التكرار يجعل الشيفرة أسهل قراءة وفهمًا.
- تقليل الأخطاء: انخفاض احتمالية التناقضات المنطقية.
- إعادة الاستخدام: الأصناف، الدوال، والقوالب قابلة للاستخدام في أكثر من سياق.

٦.٢.٩ أدوات تساعد على اكتشاف التكرار

- **CPD (Copy-Paste Detector)**: لاكتشاف الشيفرة المكررة.
- **Clang-Tidy**: يحدد الأنماط المتكررة القابلة لإعادة الهيكلة.
- مراجعة الشيفرة: المراجعة الجماعية من أفضل الطرق لاكتشاف التكرار.

٧.٢.٩ الخلاصة

يُعد مبدأ DRY حجر أساس في البرمجة الكائنية الحديثة باستخدام C++. من خلال تجميع المنطق المتكرر في دوال، قوالب، وأصناف قابلة لإعادة الاستخدام، يتم بناء قاعدة شيفرة:

- أنظف.
- أسهل صيانة.
- أكثر قابلية للتوسع.

إن الالتزام بمبدأ DRY لا يقلل فقط من التكرار، بل يرفع جودة التصميم، ويقلل الأخطاء، ويعزز نجاح المشروع على المدى الطويل، مؤديًا إلى تطبيقات C++ أكثر قوة ومرونة.

٣.٩ مبدأ KISS (Keep It Simple, Stupid)

١.٣.٩ المقدمة

يشير مبدأ KISS إلى العبارة "Keep It Simple, Stupid``، ويؤكد على أهمية البساطة في تصميم البرمجيات وتنفيذها، مع تجنب التعقيد غير الضروري.
في البرمجة الكائنية باستخدام C++، تزداد أهمية هذا المبدأ بسبب القوة الكبيرة التي توفرها اللغة من خلال القوالب، الوراثة، وتعدد الأشكال، وهي ميزات قد تؤدي بسهولة إلى حلول معقدة وصعبة الفهم إذا أسيء استخدامها.
يشجع مبدأ KISS المطورين على بناء حلول مباشرة، واضحة، وسهلة الفهم، مما يؤدي إلى:

- تقليل الأخطاء.
- تسهيل الاختبار.
- تحسين قابلية الصيانة على المدى الطويل.

٢.٣.٩ فهم مبدأ KISS

- التعريف: صمّم ونفّذ البرمجيات بأبسط شكل ممكن، وتجنب التعقيد غير المبرر.
- أهميته: الشيفرة البسيطة أسهل في التصحيح، والتوسعة، والصيانة، بينما تؤدي التصاميم المعقدة إلى زيادة زمن التطوير وارتفاع احتمالية الأخطاء.

٣.٣.٩ تطبيق مبدأ KISS في البرمجة الكائنية باستخدام C++

١. تجنب الوراثة المفرطة والمعقدة

الوراثة أداة قوية، لكن الإفراط في استخدامها قد يؤدي إلى هياكل معقدة يصعب فهمها وصيانتها.

مثال: وراثة معقدة وغير واضحة

```
class Animal { public: virtual void move() = 0; };

class Mammal : public Animal {
public:
    void move() override { std::cout << "Walk\n"; }
};

class Bird : public Animal {
public:
    void move() override { std::cout << "Fly\n"; }
};

class Bat : public Mammal, public Bird {
    //    :
};
```

حل **KISS**: وراثة مبسطة

```
class Animal {
public:
    virtual void move() = 0;
};

class Bat : public Animal {
public:
    void move() override { std::cout << "Fly\n"; }
};
```

ب. الحفاظ على بساطة الدوال

يجب أن تقوم كل دالة بمسؤولية واحدة واضحة، حتى تبقى سهلة الفهم والصيانة.

مثال: دالة معقدة متعددة المسؤوليات

```
class Order {
public:
    void processOrder(bool isOnline, bool hasDiscount, bool isPriority) {
        if (isOnline) { /* process online */ }
        else { /* process in-store */ }

        if (hasDiscount) { /* apply discount */ }
        if (isPriority) { /* handle priority shipping */ }
    }
};
```

حل **KISS**: تقسيم المسؤوليات

```
class Order {
public:
    void processOnlineOrder() { /* process online */ }
    void processInStoreOrder() { /* process in-store */ }
    void applyDiscount() { /* apply discount */ }
    void handlePriorityShipping() { /* handle priority shipping */ }
};
```

ج. تجنب الإفراط في استخدام أنماط التصميم

أنماط التصميم وُجدت لتبسيط الحلول، لكن إساءة استخدامها قد تؤدي إلى تعقيد غير ضروري، مما يخالف مبدأ KISS.

مثال: استخدام غير مبرر لنمط **Singleton**

```
class Logger {
private:
    static Logger* instance;
```

```
    Logger() {}
public:
    static Logger* getInstance() {
        if (!instance) instance = new Logger();
        return instance;
    }
    void log(const std::string& message) {
        std::cout << message << std::endl;
    }
};
```

حل **KISS**: مسجل بسيط

```
class Logger {
public:
    void log(const std::string& message) {
        std::cout << message << std::endl;
    }
};
```

٤.٣.٩ هل ما زال مبدأ **KISS** صالحًا اليوم؟

نعم، وبقوة. لا يزال مبدأ KISS أساسيًا في تطوير C++ الحديثة للأسباب التالية:

١. التطوير الرشيق: الشيفرة البسيطة تسمح بتعديلات أسرع ودورات تطوير أقصر.

٢. العمل الجماعي: الشيفرة الواضحة أسهل للفهم من قِبل الفرق الكبيرة.

٣. قابلية التوسع: التصاميم البسيطة تقلل الدين التقني وتجعل الأنظمة أسهل في التوسع مستقبلاً.

٥.٣.٩ الخلاصة

يُعد مبدأ KISS مبدأً خالدًا وذو أهمية كبيرة في البرمجة الكائنية باستخدام C++. إن الحفاظ على بساطة الشيفرة يؤدي إلى:

- سهولة الصيانة.
- وضوح التصميم.
- قابلية التوسع على المدى الطويل.

من خلال تجنب الوراثة المعقدة، تقسيم الدوال حسب المسؤوليات، واستخدام أنماط التصميم بحكمة، يمكن للمطورين بناء أنظمة قوية، قابلة للصيانة، وقابلة للنمو بثقة واستقرار.

٤.٩ قانون ديميتر (The Law of Demeter)

١.٤.٩ المقدمة

يُعرف قانون ديميتر (Law of Demeter – LoD) أيضًا باسم مبدأ أقل معرفة (Principle of Least Knowledge)، وهو أحد المبادئ الأساسية في البرمجة الكائنية (OOP) لبناء برمجيات معيارية، قابلة للصيانة، وقابلة للتوسع.
يشجّع هذا القانون الكائنات على التفاعل فقط مع الكائنات القريبة منها (المتعاونين المباشرين)، ويُثني عن استخدام سلاسل طويلة من استدعاءات الدوال. تطبيق هذا المبدأ يقلل من الترابط (Coupling) بين الكائنات، ويؤدي إلى شيفرة أكثر مرونة وأسهل في الصيانة.
في هذا القسم، نستعرض قانون ديميتر وتطبيقه العملي في C++ مع أمثلة توضيحية.

٢.٤.٩ فهم قانون ديميتر

- التعريف: "Talk to friends, not strangers`` (تحدث مع الأصدقاء لا الغرباء). يجب على الكائن استدعاء الدوال فقط على:
  - نفسه
  - أعضائه الداخلية (المتغيرات العضوية)
  - الكائنات المُمرّرة له كوسائط
  - الكائنات التي ينشئها مباشرة
- الهدف: حصر التفاعل ضمن الاعتماديات المباشرة، وتجنّب سلاسل الاستدعاء العميقة التي تؤدي إلى ترابط قوي وصعوبة في الصيانة.

٣.٤.٩ أهمية قانون ديميتر

١. تحسين المعيارية: يقلل الاعتماد المتبادل بين المكونات، مما يسمح بتعديل كل جزء بشكل مستقل.

ب. تعزيز مبدأ التغليف: يضمن أن كل كائن يدير حالته الداخلية دون كشف تفاصيل غير ضرورية.

ج. تحسين قابلية الصيانة: تقليل الاعتماديات يقلل من احتمالية كسر الشيفرة عند التغيير.

د. تقليل تعقيد الشيفرة: تقصير سلاسل الاستدعاء يجعل الشيفرة أوضح وأسهل في التصحيح.

٤.٤.٩ أمثلة على مخالفة والالتزام بقانون ديميتر

مخالفة قانون ديميتر

```
class Engine {
public:
    class SparkPlug {
    public:
        void ignite() { std::cout << "Ignition!\n"; }
    };

    SparkPlug* getSparkPlug() { return &sparkPlug; }

private:
    SparkPlug sparkPlug;
};

class Car {
public:
    Engine* getEngine() { return &engine; }

private:
    Engine engine;
};
```

```
int main() {
    Car car;
    //     LoD:
    car.getEngine()->getSparkPlug()->ignite();
    return 0;
}
```

المشكلة: يعتمد Car على البنية الداخلية لـ Engine، مما يؤدي إلى ترابط قوي وصعوبة في التغيير لاحقًا.

الالتزام بقانون ديميتر

```
class Engine {
public:
    void igniteSparkPlug() {
        sparkPlug.ignite();
    }

private:
    class SparkPlug {
    public:
        void ignite() { std::cout << "Ignition!\n"; }
    };

    SparkPlug sparkPlug;
};

class Car {
public:
```

```
    void startEngine() {
        engine.igniteSparkPlug();
    }

private:
    Engine engine;
};

int main() {
    Car car;
    car.startEngine(); //        LoD
    return 0;
}
```

الحل: يتعامل Car فقط مع Engine، بينما يتكفل Engine بإدارة تفاصيله الداخلية، مما يقلل الترابط ويبسّط التصميم.

٥.٤.٩ أفضل الممارسات لتطبيق قانون ديميتر

١. تقليل سلاسل الاستدعاء: تجنب الاستدعاءات المتداخلة الطويلة.

```
//
user.getAddress().getCity().getZipCode();

//
user.getCityZipCode();
```

٢. تفويض المسؤوليات: دع كل كائن يتولى سلوكه بنفسه دون كشف تفاصيله الداخلية.

```
class Payment {
public:
```

```
    void process() {
        std::cout << "Payment processed!\n";
    }
};

class Order {
public:
    void processPayment() {
        payment.process();
    }
private:
    Payment payment;
};
```

٣. تجنب كسر التغليف: لا تكشف البنى الداخلية إلا عند الضرورة، ووفّر واجهات عالية المستوى للتعامل معها.

٤. حالات شائعة لمخالفة LoD:

- سوء استخدام حقن الاعتماديات: حقن كائنات أكثر من اللازم يؤدي إلى اعتماديات بعيدة.
- استخدام أنماط **Facade** أو **Mediator**: تساعد على مركزية التفاعل والالتزام بقانون ديميتر.

٦.٤.٩ الخلاصة

يُعد قانون ديميتر مبدأً أساسيًا في البرمجة الكائنية باستخدام C++ لبناء أنظمة معيارية، قابلة للصيانة، وقوية من الناحية المعمارية.
من خلال تقليل الاعتماديات المباشرة، وتجنّب سلاسل الاستدعاء العميقة، يعزز هذا القانون:

- مبدأ التغليف.
- وضوح الشيفرة.
- سهولة الاختبار والتوسعة.

الالتزام بقانون ديميتر يضمن شيفرة أكثر مرونة، وأقل عرضة للانكسار مع تطور الأنظمة وتعقّدها.

الفصل ١٠: اختبار الشيفرة الكائنية (Testing Object-Oriented Code)

- الاختبارات الوحدوية باستخدام **Google Test** و **Catch2**
- المحاكاة **(Mocking)** والتطوير بالاختبار أولاً **(TDD)**

١.١٠ الاختبارات الوحدوية باستخدام Google Test و Catch2

١.١.١٠ المقدمة

تُعد الاختبارات الوحدوية (Unit Testing) من الركائز الأساسية في تطوير البرمجيات، خصوصًا في البرمجة الكائنية (OOP) حيث تمثل الفئات (Classes) وتفاعلاتها جوهر التطبيق.
الهدف من الاختبار الوحدوي هو التأكد من أن كل فئة:

- تعمل بشكل صحيح بمعزل عن بقية النظام
- تحافظ على سلوكها المتوقع عند التعديل أو إعادة الهيكلة
- تكتشف الأخطاء مبكرًا قبل تراكمها

من أشهر أطر الاختبار في C++: **Google Test** و **Catch2**. يوفر كلاهما بيئة منظمة لاختبار الشيفرة الكائنية بكفاءة ووضوح.

٢.١.١٠ Google Test

نبذة عامة

Google Test هو إطار اختبار متكامل طورته شركة Google، ويتميز بما يلي:

- مجموعة غنية من أدوات التحقق (Assertions)
- دعم قوي للاختبارات الفئات المعقدة
- دعم Test Fixtures لاختبار الكائنات ذات الحالة

يُعد خيارًا ممتازًا للمشاريع الكبيرة أو طويلة الأمد.

إعداد Google Test

١. تثبيت الإطار:

```
git clone https://github.com/google/googletest.git
cd googletest
cmake .
make
```

٢. الربط مع المشروع باستخدام CMake:

```
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})
add_executable(your_test your_test.cpp)
target_link_libraries(your_test ${GTEST_LIBRARIES} pthread)
```

مثال عملي باستخدام Google Test

فئة بسيطة للاختبار:

```
class Calculator {
public:
    int add(int a, int b) { return a + b; }
    int subtract(int a, int b) { return a - b; }
};
```

اختبارات وحدوية:

```
#include <gtest/gtest.h>
#include "calculator.h"

TEST(CalculatorTest, Addition) {
```

```
    Calculator calc;
    EXPECT_EQ(calc.add(2, 3), 5);
    EXPECT_EQ(calc.add(-1, -1), -2);
}

TEST(CalculatorTest, Subtraction) {
    Calculator calc;
    EXPECT_EQ(calc.subtract(5, 3), 2);
    EXPECT_EQ(calc.subtract(0, 0), 0);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

ملاحظات مهمة:

- EXPECT_EQ تتحقق من تساوي القيم دون إيقاف الاختبار عند الفشل
- يتم تنظيم الاختبارات ضمن مجموعات واضحة

تشغيل الاختبارات:

```
./your_test
```

٣.١.١٠ Catch2

نبذة عامة

Catch2 إطار اختبار خفيف وسهل الاستخدام، ويتميز بأنه:

- Header-only (لا يتطلب بناء مكتبة خارجية)
- سريع الدمج مع المشاريع الصغيرة والمتوسطة
- صياغته بسيطة وسهلة القراءة

إعداد Catch2

- التثبيت باستخدام vcpkg:

```
./vcpkg install catch2
```

- تضمينه في ملف الاختبار:

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include "calculator.h"
```

مثال عملي باستخدام Catch2

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include "calculator.h"

TEST_CASE("Addition works correctly", "[calculator]") {
    Calculator calc;
    REQUIRE(calc.add(2, 3) == 5);
    REQUIRE(calc.add(-1, -1) == -2);
}

TEST_CASE("Subtraction works correctly", "[calculator]") {
```

```
    Calculator calc;
    REQUIRE(calc.subtract(5, 3) == 2);
    REQUIRE(calc.subtract(0, 0) == 0);
}
```

ملاحظات:

- REQUIRE توقف الاختبار فور فشل الشرط
- TEST_CASE تسمح بتنظيم الاختبارات باستخدام وسوم (Tags)

٤.١.١٠ مقارنة بين Google Test و Catch2

الميزة	Google Test	Catch2
التثبيت	مكتبة خارجية	Header-only
التعقيد	متقدم وغني بالميزات	بسيط وسريع
الدعم للاختبارات المعقدة	ممتاز	جيد
سهولة الدمج	متوسط	سهل جداً
المشاريع المناسبة	كبيرة ومؤسسية	صغيرة ومتوسطة

٥.١.١٠ الخلاصة

يوفر كل من Google Test و Catch2 حلولاً قوية لاختبار الشيفرة الكائنية في C++. الاختيار بينهما يعتمد على:

- حجم المشروع
- تعقيد الشيفرة
- متطلبات البنية الاختبارية

الالتزام بالاختبارات الوحدوية:

- يرفع موثوقية الشيفرة
- يسهل إعادة الهيكلة بثقة
- يقلل الأخطاء في المراحل المتقدمة

في البرمجة الكائنية الحديثة، الاختبار ليس خيارًا إضافيًا، بل جزء أساسي من التصميم الهندسي السليم.

٢.١٠ المحاكاة (Mocking) والتطوير بالاختبار أولاً (Test-Driven Development) في C++ الحديثة

١.٢.١٠ المقدمة

يُعد الاختبار جزءًا حاسمًا من عملية تطوير البرمجيات، إذ يضمن أن الشيفرة تتصرف بشكل صحيح وموثوق. في البرمجة الكائنية (OOP)، حيث تُبنى الأنظمة على كائنات مترابطة ذات علاقات واعتمادات معقدة، تصبح عملية الاختبار أكثر أهمية وحساسية.
تُعد كل من المحاكاة **(Mocking)** و التطوير بالاختبار أولاً **(TDD)** من التقنيات القوية التي تساعد على اختبار الشيفرة الكائنية في C++ بشكل منضبط، موثوق، وقابل للصيانة.

٢.٢.١٠ المحاكاة (Mocking)

التعريف

تعني المحاكاة إنشاء نسخ مبسطة من الكائنات تُسمّى كائنات وهمية (Mocks)، بحيث يمكن التحكم في سلوكها ومراقبته أثناء الاختبار.
تُستخدم هذه الكائنات بدلاً من الاعتمادات الحقيقية مثل قواعد البيانات أو الشبكات أو الخدمات الخارجية، مما يسمح بعزل الشيفرة قيد الاختبار وجعل النتائج أكثر توقعًا ودقة.

لماذا نستخدم المحاكاة؟

- العزل: فصل الشيفرة قيد الاختبار عن الأنظمة الخارجية لتجنب التأثيرات الجانبية.
- التحكم: محاكاة سلوكيات وسيناريوهات مختلفة للاعتمادات (نجاح، فشل، قيم محددة).
- قابلية الاختبار: تبسيط الاختبارات عبر تقليل التعقيد والاعتماد على مكونات خارجية.

مثال باستخدام GMock

```
#include <gtest/gtest.h>
#include "gmock/gmock.h"

class ExternalService {
public:
    virtual int fetchData() = 0;
};

class MyClass {
public:
    explicit MyClass(ExternalService* service) : service_(service) {}

    int processData() {
        int data = service_->fetchData();
        return data;
    }

private:
    ExternalService* service_;
};

TEST(MyClassTest, ProcessData) {
    class MockExternalService : public ExternalService {
    public:
        MOCK_METHOD(int, fetchData, ());
    };

    MockExternalService mockService;
```

```
    EXPECT_CALL(mockService, fetchData())
        .Times(1)
        .WillOnce(Return(42));

    MyClass myClass(&mockService);
    int result = myClass.processData();

    EXPECT_EQ(result, 42);
}
```

الشرح:

- تم إنشاء كائن وهمي Mock للفئة ExternalService.
- تم تحديد السلوك المتوقع للدالة fetchData().
- تم التأكد من أن processData() تُعيد القيمة الصحيحة دون الاعتماد على خدمة حقيقية.

٣.٢.١٠ التطوير بالاختبار أولاً (TDD)

التعريف

التطوير بالاختبار أولاً (Test-Driven Development) هو منهجية تطوير برمجيات تعتمد على كتابة الاختبارات قبل كتابة الشيفرة الفعلية.
يُجبر هذا الأسلوب المطور على التفكير في التصميم والسلوك المطلوب قبل التنفيذ، مما يؤدي إلى شيفرة:

- قابلة للاختبار
- واضحة المسؤوليات
- أقل تعقيداً

دورة TDD

١. الأحمر (Red): كتابة اختبار يفشل ويمثل السلوك المطلوب.

٢. الأخضر (Green): كتابة أبسط شيفرة ممكنة تجعل الاختبار ينجح.

٣. إعادة الهيكلة (Refactor): تحسين الشيفرة دون تغيير سلوكها.

مثال باستخدام Google Test

```
#include <gtest/gtest.h>

class Calculator {
public:
    int add(int a, int b) { return a + b; }
};

TEST(CalculatorTest, Add) {
    Calculator calculator;
    int result = calculator.add(2, 3);
    EXPECT_EQ(result, 5);
}
```

الشرح:

- تم أولاً كتابة اختبار يحدد السلوك المتوقع.
- ثم تم تنفيذ الدالة add() لتلبية الاختبار.
- يضمن هذا الأسلوب أن الشيفرة مكتوبة لتلبية متطلبات واضحة.

٤.٢.١٠ الدمج بين المحاكاة و **TDD**

نظرة عامة

تتكامل المحاكاة مع TDD بشكل طبيعي:

- TDD يحدد السلوك المطلوب بدقة.
- Mocking يعزل الاعتمادات أثناء الاختبار.

النتيجة هي شيفرة كائنية:

- سهلة الاختبار
- ضعيفة الترابط
- عالية الجودة

الفوائد الرئيسية

- اكتشاف الأخطاء في وقت مبكر.
- زيادة الثقة في صحة الشيفرة.
- تسهيل الصيانة وإعادة الهيكلة.
- تحسين التعاون بين أعضاء الفريق.

٥.٢.١٠ الخلاصة

يؤدي اعتماد المحاكاة (Mocking) والتطوير بالاختبار أولاً (TDD) في مشاريع C++ الحديثة إلى تحسين كبير في جودة البرمجيات وموثوقيتها وقابليتها للصيانة.
تُمكّن هذه التقنيات المطورين من:

- عزل الاعتمادات المعقدة
- اختبار السلوك بدقة
- بناء شيفرة كائنية قوية ومنضبطة

في الأنظمة الكائنية الاحترافية، الاختبار ليس مرحلة لاحقة، بل جزء أصيل من التصميم الجيد.

الفصل ١١: المتغيرات الثابتة والديناميكية في C++

- الكائنات الثابتة مقابل الكائنات الديناميكية.
- الذاكرة المكدسية (Stack) مقابل الذاكرة الكومية (Heap).

مقدمة

في لغة C++، تلعب طريقة تخصيص الذاكرة دورًا جوهريًا في سلوك البرنامج وأدائه واستقراره. يمكن تصنيف المتغيرات والكائنات بحسب طريقة تخصيص الذاكرة وعمرها الزمني إلى نوعين رئيسيين:

- متغيرات ثابتة **(Static)**
- متغيرات ديناميكية **(Dynamic)**

يؤثر هذا التصنيف بشكل مباشر على:

- عمر المتغير (Lifetime)
- مكان تخزينه في الذاكرة
- طريقة إدارته وتحريره
- كفاءة البرنامج وسلامته

يهدف هذا الفصل إلى توضيح الفروقات بين المتغيرات والكائنات الثابتة والديناميكية، مع شرح العلاقة بين Stack و Heap من خلال أمثلة عملية واضحة.

١.١١ المتغيرات الثابتة (Static Variables)

١.١.١١ التعريف

المتغيرات الثابتة هي متغيرات يتم تخصيص الذاكرة الخاصة بها وقت الترجمة (Compile Time)، ويستمر وجودها طوال فترة تنفيذ البرنامج.
تُستخدم هذه المتغيرات عندما نحتاج إلى:

- الاحتفاظ بالحالة بين استدعاءات الدوال

- مشاركة البيانات بين جميع كائنات الصنف

وتنقسم المتغيرات الثابتة إلى:

- متغيرات ثابتة محلية: تُعرّف داخل دالة وتحتفظ بقيمتها بين الاستدعاءات.
- متغيرات ثابتة على مستوى الصنف: تُعرّف داخل الصنف باستخدام الكلمة static وتُشارك بين جميع كائنات الصنف.

٢.١.١١ خصائص المتغيرات الثابتة

- تُخصّص الذاكرة مرة واحدة فقط.
- تبقى موجودة حتى انتهاء البرنامج.
- تُهيّأ مرة واحدة فقط.
- تحتفظ بقيمتها بين استدعاءات الدوال.
- تُهيّأ افتراضيًا بالقيمة صفر إذا لم تُهيّأ صراحة.

٣.١.١١ مثال عملي على المتغيرات الثابتة

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

void staticVarExample() {
    static int counter = 0;  //
    counter++;
    cout << "Counter: " << counter << endl;
}
```

```
int main() {
    staticVarExample();  // Counter: 1
    staticVarExample();  // Counter: 2
    staticVarExample();  // Counter: 3
    return 0;
}
```

الشرح:

- المتغير counter لا يُنشأ مع كل استدعاء للدالة.
- يحتفظ بقيمته السابقة بين الاستدعاءات.
- يعكس سلوك التخزين الثابت المرتبط بعمر البرنامج الكامل.

٢.١١ المتغيرات الديناميكية (Dynamic Variables)

١.٢.١١ التعريف

المتغيرات الديناميكية هي متغيرات يتم تخصيص ذاكرتها وقت التنفيذ (Run Time) باستخدام الكلمتين new و delete.
يكون عمر هذه المتغيرات تحت تحكم المبرمج، وليس مرتبطًا بنطاق (Scope) الدالة.

٢.٢.١١ خصائص المتغيرات الديناميكية

- يتم تخصيصها على الذاكرة الكومية (Heap).
- عمرها يستمر حتى يتم تحريرها يدويًا.
- أكثر مرونة ولكن أكثر خطورة إذا أسيء استخدامها.
- قد تؤدي إلى تسرب الذاكرة (Memory Leaks) إذا لم تُحرر.

٣.٢.١١ مثال على المتغيرات الديناميكية

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

int main() {
    int* ptr = new int(10);  //
    cout << "Value: " << *ptr << endl;

    delete ptr;              //
    ptr = nullptr;
    return 0;
}
```

الشرح:

- تم تخصيص الذاكرة أثناء التنفيذ.
- يتحمل المبرمج مسؤولية تحرير الذاكرة.
- نسيان delete يؤدي إلى تسرب الذاكرة.

٣.١١ الكائنات الثابتة مقابل الكائنات الديناميكية

١.٣.١١ مقارنة مفاهيمية

- الكائن الثابت:
  - يُنشأ تلقائيًا.
  - يُدمَّر تلقائيًا.
  - عمره مرتبط بالنطاق.

- الكائن الديناميكي:
  - يُنشأ باستخدام new.
  - يُدمَّر باستخدام delete.
  - عمره مستقل عن النطاق.

```
class Example {
public:
    Example() { cout << "Constructed\n"; }
    ~Example() { cout << "Destroyed\n"; }
};

int main() {
    Example a;                  //
    Example* b = new Example;   //
    delete b;
    return 0;
}
```

٤.١١ الذاكرة المكدسية مقابل الذاكرة الكومية

١.٤.١١ الذاكرة المكدسية (Stack)

- سريعة جدًا في التخصيص والتحرير.
- تُدار تلقائيًا.
- حجمها محدود.
- تُستخدم للمتغيرات المحلية والكائنات الثابتة.

٢.٤.١١ الذاكرة الكومية (Heap)

- مرنة في الحجم.
- أبطأ من Stack.
- تتطلب إدارة يدوية أو ذكية.
- تُستخدم للكائنات الديناميكية.

٥.١١ أفضل الممارسات

- فضّل المتغيرات الثابتة كلما أمكن.
- استخدم الكائنات الديناميكية فقط عند الحاجة.
- اعتمد على مؤشرات ذكية (Smart Pointers) بدل new/delete.
- افهم الفرق بين العمر الزمني والنطاق.

٦.١١ الخلاصة

الفهم العميق للفروقات بين المتغيرات الثابتة والديناميكية وبين Stack و Heap هو أساس كتابة برامج C++ قوية وآمنة.
الاختيار الصحيح لطريقة التخصيص:

- يُحسّن الأداء
- يقلل الأخطاء
- يمنع تسرب الذاكرة

وفي البرمجة الكائنية الاحترافية، إدارة الذاكرة ليست خياراً ثانوياً، بل جزء جوهري من التصميم السليم.

٧.١١ المتغيرات الديناميكية (Dynamic Variables)

١.٧.١١ التعريف

المتغيرات الديناميكية هي متغيرات يتم تخصيص الذاكرة الخاصة بها أثناء وقت التنفيذ (Run Time) باستخدام المعامل new، ويجب تحرير هذه الذاكرة يدويًا باستخدام المعامل delete.
وعلى عكس المتغيرات الثابتة، فإن عمر المتغيرات الديناميكية لا يرتبط بالنطاق (Scope) وإنما يتحكم به المبرمج بشكل مباشر، مما يوفر مرونة عالية في إدارة الذاكرة.

٢.٧.١١ خصائص المتغيرات الديناميكية

- يتم تخصيص الذاكرة على الذاكرة الكومية (Heap).
- عمر المتغير يُحدد صراحةً بواسطة المبرمج.
- يجب تحرير الذاكرة يدويًا باستخدام delete.
- نسيان تحرير الذاكرة يؤدي إلى تسرب الذاكرة (Memory Leak).
- مناسبة لتخصيص كتل ذاكرة كبيرة أو ذات حجم غير معروف وقت الترجمة.

٣.٧.١١ مثال عملي على المتغيرات الديناميكية

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

int main() {
    int* dynamicVar = new int;    //             Heap
    *dynamicVar = 10;

    cout << "Dynamic Variable Value: " << *dynamicVar << endl;
```

```
    delete dynamicVar;          //
    dynamicVar = nullptr;       //
    return 0;
}
```

الشرح:

- المتغير dynamicVar هو مؤشر إلى قيمة صحيحة تم تخصيصها ديناميكيًا على Heap.
- يتم تعيين القيمة يدويًا بعد التخصيص.
- يجب على المبرمج تحرير الذاكرة باستخدام delete.
- تعيين المؤشر إلى nullptr بعد الحذف يمنع أخطاء المؤشرات المتدلية (Dangling Pointers).

٤.٧.١١ متى نستخدم المتغيرات الديناميكية؟

- عندما لا يكون حجم البيانات معروفًا وقت الترجمة.
- عند الحاجة إلى كائنات يتجاوز عمرها نطاق الدالة.
- في هياكل البيانات الديناميكية مثل:
  - القوائم المرتبطة
  - الأشجار
  - الرسوم البيانية

٥.٧.١١ ملاحظات تصميمية مهمة

- لا يُنصح باستخدام new و delete مباشرة في الكود الحديث.

- يُفضل الاعتماد على:
  - std::unique_ptr
  - std::shared_ptr

لتطبيق مبدأ RAII وضمان الأمان.

٦.٧.١١ الخلاصة

المتغيرات الديناميكية توفر مرونة عالية في إدارة الذاكرة، لكنها تأتي مع مسؤولية كبيرة. الاستخدام غير الصحيح يؤدي إلى مشاكل خطيرة مثل تسرب الذاكرة وانهيار البرامج.
في البرمجة الكائنية الاحترافية بلغة C++، يجب استخدام المتغيرات الديناميكية بحذر، وفهم عميق لعمر الكائن وإدارة الموارد، مع تفضيل الأدوات الحديثة التي توفر الأمان والاستقرار.

٨.١١ الكائنات الثابتة مقابل الكائنات الديناميكية (Static vs Dynamic Objects)

١.٨.١١ الكائنات الثابتة (Static Objects)

التعريف
الكائنات الثابتة هي كائنات يتم تخصيص الذاكرة الخاصة بها وقت الترجمة (Compile Time) أو في منطقة الذاكرة الثابتة، ويمكن أن تكون:

- كائنات عامة (Global Objects)
- كائنات معرفة داخل دالة باستخدام الكلمة المفتاحية static
- أعضاء ساكنة (Static Members) داخل الأصناف

خصائص الكائنات الثابتة

- عمرها يمتد طوال مدة تنفيذ البرنامج.
- يتم إنشاؤها وتهيئتها مرة واحدة فقط.
- إذا كانت عضوًا ساكنًا في صنف، فهي مشتركة بين جميع كائنات الصنف.
- يتم استدعاء المُدمِّر (Destructor) تلقائيًا عند انتهاء البرنامج.

مثال عملي على الكائنات الثابتة

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

class MyClass {
```

```
public:
    MyClass() { cout << "Constructor called" << endl; }
    ~MyClass() { cout << "Destructor called" << endl; }
};

void createStaticObject() {
    static MyClass obj;  //
}

int main() {
    createStaticObject();  // Constructor called
    createStaticObject();  //
    return 0;
}
// Destructor called
```

الشرح:

- الكائن obj يتم إنشاؤه مرة واحدة فقط.
- استدعاء الدالة مرة أخرى لا يؤدي إلى إعادة إنشاء الكائن.
- يتم تدمير الكائن تلقائيًا عند انتهاء البرنامج.

٢.٨.١١ الكائنات الديناميكية (Dynamic Objects)

التعريف

الكائنات الديناميكية هي كائنات يتم إنشاؤها أثناء وقت التنفيذ (Run Time) باستخدام المعامل new، ويجب تحريرها يدويًا باستخدام delete.

تُستخدم هذه الكائنات عندما لا يكون عددها أو حجمها معروفًا وقت الترجمة.

خصائص الكائنات الديناميكية

- يتم تخصيصها على الذاكرة الكومية (Heap).
- عمرها لا يرتبط بنطاق الدالة.
- يجب تحريرها يدويًا باستخدام delete.
- توفر مرونة عالية في إدارة الذاكرة.

مثال عملي على الكائنات الديناميكية

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

class MyClass {
public:
    MyClass() { cout << "Constructor called" << endl; }
    ~MyClass() { cout << "Destructor called" << endl; }
};

int main() {
    MyClass* obj = new MyClass();  //
    delete obj;                    //
    obj = nullptr;
    return 0;
}
```

الشرح:

- يتم إنشاء الكائن باستخدام new.

- لا يتم تدمير الكائن تلقائيًا عند الخروج من النطاق.
- يجب على المبرمج استدعاء delete صراحة.

٣.٨.١١ الفروقات الجوهرية بين الكائنات الثابتة والديناميكية

الخاصية	كائنات ثابتة	كائنات ديناميكية
مكان الذاكرة	ذاكرة ثابتة	Heap
عمر الكائن	طوال مدة البرنامج	حتى يتم حذفه يدويًا
التهيئة	مرة واحدة فقط	يدويًا بواسطة المبرمج
إدارة الذاكرة	تلقائية	يدوية
المرونة	محدودة	عالية جدًا

٤.٨.١١ متى نستخدم كل نوع؟

استخدم الكائنات الثابتة عندما:

- يكون عدد الكائنات وحجمها معروفًا وقت الترجمة.
- تكون الأولوية للأداء والاستقرار.
- تحتاج إلى كائنات تعيش طوال مدة البرنامج.

استخدم الكائنات الديناميكية عندما:

- يكون عدد الكائنات غير معروف مسبقًا.
- تحتاج إلى إدارة عمر الكائن بدقة.
- تعمل مع هياكل بيانات ديناميكية مثل القوائم والأشجار.

٥.٨.١١ الخلاصة

فهم الفرق بين الكائنات الثابتة والديناميكية في C++ يُعد من الأساسيات الجوهرية لكتابة برامج فعالة وآمنة.
الكائنات الثابتة توفر بساطة وأمانًا تلقائيًا، بينما الكائنات الديناميكية تمنح مرونة وتحكمًا عاليًا، لكنها تتطلب انضباطًا صارمًا في إدارة الذاكرة.
المبرمج المحترف يختار النوع المناسب بناءً على طبيعة المشكلة، الأداء المطلوب، ومتطلبات إدارة الموارد.

٩.١١ ذاكرة المكدس مقابل ذاكرة الكومة (Stack vs Heap Memory)

المقدمة
تُعد إدارة الذاكرة بكفاءة عاملاً حاسمًا في أداء البرامج واستقرارها، خصوصًا في لغة C++ التي تمنح المبرمج تحكمًا مباشرًا في الذاكرة.
لفهم كيفية عمل البرامج بشكل صحيح، يجب التمييز بين:

- ذاكرة المكدس (Stack)
- ذاكرة الكومة (Heap)

كما يجب إدراك العلاقة بين هذه الذاكرات وبين المتغيرات الثابتة والديناميكية. يستعرض هذا القسم هذه المفاهيم بشكل منظم مع أمثلة عملية توضيحية.

١.٩.١١ تقسيم الذاكرة في برنامج C++

تنقسم الذاكرة في برنامج C++ نموذجي إلى عدة أجزاء رئيسية:

- مقطع التعليمات **(Text Segment)** يحتوي على الشيفرة التنفيذية للبرنامج.
- مقطع البيانات **(Data Segment)** يحتوي على المتغيرات العامة والثابتة.
  - بيانات مهيأة (Initialized Data)
  - بيانات غير مهيأة (BSS)
- ذاكرة المكدس **(Stack)** لتخزين إطارات استدعاء الدوال والمتغيرات المحلية.
- ذاكرة الكومة **(Heap)** لتخصيص الذاكرة الديناميكية أثناء التنفيذ.

فهم هذا التقسيم ضروري لأن:

- المتغيرات الثابتة توجد في مقطع البيانات
- المتغيرات المحلية توجد في المكدس
- المتغيرات الديناميكية توجد في الكومة

٢.٩.١١ ذاكرة المكدس (Stack Memory)

التعريف
ذاكرة المكدس هي مساحة ذاكرة تُدار تلقائيًا، تُستخدم لتخزين:

- المتغيرات المحلية
- معاملات الدوال
- معلومات استدعاء الدوال

تنمو هذه الذاكرة وتتناقص تلقائيًا مع دخول الدوال وخروجها.

المتغيرات المحلية على المكدس

- يتم إنشاؤها عند استدعاء الدالة.
- يتم تدميرها تلقائيًا عند الخروج من الدالة.
- إدارتها تتم بالكامل بواسطة المترجم.
- الوصول إليها سريع جدًا.

مثال عملي

```cpp
#include <iostream>

void func() {
    int x = 10; //
    std::cout << "x = " << x << std::endl;
} //        x

int main() {
    func();
    return 0;
}
```

الشرح:

- المتغير x يُنشأ عند استدعاء الدالة.
- يتم حذفه تلقائيًا عند خروج الدالة.

مشكلة شائعة: فيضان المكدس **(Stack Overflow)**
يحدث عند:

- الاستدعاء التكراري العميق (Deep Recursion)
- إنشاء مصفوفات محلية كبيرة الحجم

٣.٩.١١ ذاكرة الكومة **(Heap Memory)**

التعريف
ذاكرة الكومة هي مساحة ذاكرة تُستخدم للتخصيص الديناميكي أثناء وقت التنفيذ، ويتم التحكم بها يدويًا باستخدام:

- new
- delete

المتغيرات الديناميكية على الكومة

- يتم تخصيصها أثناء التنفيذ.
- عمرها لا يرتبط بنطاق دالة.
- يجب تحريرها يدوياً.
- أكثر مرونة ولكن أبطأ من المكدس.

مثال عملي

```
#include <iostream>

int main() {
    int* ptr = new int;
    *ptr = 20;
    std::cout << "Value = " << *ptr << std::endl;

    delete ptr; //
    ptr = nullptr;
    return 0;
}
```

الشرح:

- يتم تخصيص الذاكرة على الكومة.
- عدم استخدام delete يؤدي إلى تسريب ذاكرة.

مشكلة شائعة: تسريب الذاكرة **(Memory Leak)**
يحدث عندما:

- تُفقد الإشارة إلى الذاكرة دون تحريرها.
- تستمر الذاكرة محجوزة حتى نهاية البرنامج.

٤.٩.١١ مقارنة بين المكدس والكومة

الخاصية	المكدس	الكومة
آلية التخصيص	تلقائية	يدوية
العمر	مرتبط بالنطاق	حتى يتم الحذف
سرعة الوصول	سريعة جداً	أبطأ نسبيًا
حجم الذاكرة	محدود	كبير
الاستخدام الشائع	متغيرات محلية	هياكل ديناميكية
المخاطر	فيضان المكدس	تسريب الذاكرة

٥.٩.١١ العلاقة بين المتغيرات الثابتة والديناميكية

المتغيرات الثابتة

- تُخزَّن في مقطع البيانات.
- لا توجد في المكدس ولا في الكومة.
- عمرها يمتد طوال تنفيذ البرنامج.

```
#include <iostream>

void func() {
    static int counter = 0;
    counter++;
    std::cout << counter << std::endl;
}

int main() {
    func();
```

```
    func();
    func();
}
```

المتغيرات الديناميكية

- تُنشأ على الكومة.
- تُستخدم للبيانات ذات الحجم المتغير.

```
#include <iostream>

int main() {
    int* arr = new int[5];
    for (int i = 0; i < 5; ++i) {
        arr[i] = i * 10;
        std::cout << arr[i] << " ";
    }
    delete[] arr;
    return 0;
}
```

٦.٩.١١ متى نستخدم المكدس أو الكومة؟

استخدم المكدس عندما:

- يكون الحجم معروفًا وصغيرًا.
- يكون الأداء عاملاً حاسمًا.

استخدم الكومة عندما:

- يكون الحجم غير معروف مسبقًا.
- تحتاج البيانات إلى عمر أطول من الدالة.

٧.٩.١١ أفضل الممارسات

- تجنب التخصيص اليدوي قدر الإمكان.
- استخدم المؤشرات الذكية: std::unique_ptr و std::shared_ptr.
- تجنب الاستدعاء التكراري العميق.

٨.٩.١١ الخلاصة

فهم الفرق بين المكدس والكومة، وعلاقة ذلك بالمتغيرات الثابتة والديناميكية، يُعد حجر أساس في البرمجة الاحترافية بلغة C++.
المكدس يوفر سرعة وبساطة، بينما تمنح الكومة مرونة عالية، لكنها تتطلب انضباطًا صارمًا في إدارة الذاكرة.
المبرمج المحترف هو من يُحسن الاختيار بينهما وفق متطلبات التصميم والأداء والأمان.

الفصل ١٢: التحديات والمزالق الشائعة في البرمجة كائنية التوجه

- مشكلات الوراثة المتعددة.
- مشكلة الماسة وحلّها باستخدام الوراثة الافتراضية.

١.١٢ مشكلات الوراثة المتعددة في C++

المقدمة

تُعد الوراثة المتعددة (Multiple Inheritance) من أكثر ميزات C++ قوةً وتعقيداً في آنٍ واحد. فهي تسمح للصنف الواحد بالوراثة من أكثر من صنف أساس، مما يتيح إعادة استخدام الشيفرة ودمج السلوكيات المختلفة.

لكن هذه القوة تأتي بثمن؛ إذ تُدخل الوراثة المتعددة تعقيدات تصميمية، وغموضاً في الاستدعاءات، ومشكلات صيانة قد تصبح خطيرة في الأنظمة الكبيرة.

يستعرض هذا القسم أبرز المشكلات الشائعة في الوراثة المتعددة، مع أمثلة عملية وحلول مدروسة.

١.١.١٢ ما هي الوراثة المتعددة؟

في البرمجة كائنية التوجه، تعني الوراثة المتعددة أن يرث صنف واحد من أكثر من صنف أساس، على عكس الوراثة الأحادية التي تسمح بوراثة صنف واحد فقط.

الصيغة العامة:

```
class Base1 { /* ... */ };
class Base2 { /* ... */ };

class Derived : public Base1, public Base2 {
    /*      Base1   Base2 */
};
```

رغم أن هذا الأسلوب يتيح دمج الوظائف، إلا أنه يفتح الباب لمجموعة من المشكلات التي يجب فهمها جيداً.

٢.١.١٢ المشكلات الشائعة في الوراثة المتعددة

مشكلة الماسة (الغموض)

تحدث مشكلة الماسة عندما يرث صنف من صنفين، وهذان الصنفان يشتركان في صنف أساس واحد، مما يؤدي إلى وجود نسختين من الصنف الأساسي.
مثال يوضح المشكلة:

```
#include <iostream>

class Animal {
public:
    void eat() { std::cout << "Animal is eating\n"; }
};

class Mammal : public Animal {};
class Bird   : public Animal {};

class Bat : public Mammal, public Bird {};

int main() {
    Bat bat;
    // bat.eat(); // :           Animal
}
```

المشكلة:

- الصنف Bat يحتوي على نسختين من Animal.
- المترجم لا يستطيع تحديد أي دالة eat() يجب استدعاؤها.

الحل: الوراثة الافتراضية

```
class Mammal : public virtual Animal {};
class Bird   : public virtual Animal {};

class Bat : public Mammal, public Bird {};

int main() {
    Bat bat;
    bat.eat(); //
}
```

الشرح: الوراثة الافتراضية تضمن وجود نسخة واحدة فقط من الصنف الأساسي داخل شجرة الوراثة.

زيادة التعقيد البنيوي

كلما زاد عدد الأصناف الموروثة، زادت صعوبة فهم العلاقات بينها، وزادت تكلفة الصيانة والتعديل.

```
class Printer {
public:
    void print() { std::cout << "Printing\n"; }
};

class Scanner {
public:
    void scan() { std::cout << "Scanning\n"; }
};

class MultiFunctionDevice : public Printer, public Scanner {};
```

المشكلة:

- إضافة المزيد من الوراثة تجعل التسلسل معقداً.
- صعوبة تتبع مصدر السلوكيات.

تعارض الأسماء

عند احتواء الأصناف الأساسية على دوال أو أعضاء تحمل نفس الاسم، يحدث غموض عند الاستدعاء.

```
class Printer {
public:
    void powerOn() { std::cout << "Printer on\n"; }
};

class Scanner {
public:
    void powerOn() { std::cout << "Scanner on\n"; }
};

class MultiFunctionDevice : public Printer, public Scanner {};

int main() {
    MultiFunctionDevice mfd;
    // mfd.powerOn(); //
}
```

الحل: استخدام محدد النطاق

```
class MultiFunctionDevice : public Printer, public Scanner {
public:
    void powerOnAll() {
        Printer::powerOn();
        Scanner::powerOn();
    }
};
```

```
int main() {
    MultiFunctionDevice mfd;
    mfd.powerOnAll();
}
```

تعقيد البُناة (Constructors)

في الوراثة المتعددة، يجب على الصنف المشتق تهيئة جميع الأصناف الأساسية، مما يزيد التعقيد خاصة عند اختلاف البُناة.

```
class Base1 {
public:
    Base1(int x) { std::cout << "Base1\n"; }
};

class Base2 {
public:
    Base2(int y) { std::cout << "Base2\n"; }
};

class Derived : public Base1, public Base2 {
public:
    Derived(int x, int y)
        : Base1(x), Base2(y) {
        std::cout << "Derived\n";
    }
};
```

ملاحظة: رغم أن C++11 قدّمت تفويض البُناة داخل الصنف الواحد، إلا أن تهيئة الأصناف الأساسية تبقى إلزامية وصريحة.

مشكلة الصنف الأساسي الهش

أي تعديل في صنف أساس قد يؤدي إلى كسر أصناف مشتقة بشكل غير متوقع.

```
class Base1 {
public:
    virtual void print() const {
        std::cout << "Base1\n";
    }
};

class Base2 {
public:
    virtual void print() const {
        std::cout << "Base2\n";
    }
};

class Derived : public Base1, public Base2 {
public:
    void print() const override {
        Base1::print();
    }
};
```

المشكلة:

- تغيير أحد الأصناف الأساسية قد يؤثر على السلوك النهائي.

٣.١.١٢ متى تُستخدم الوراثة المتعددة؟

تُستخدم الوراثة المتعددة بشكل آمن نسبيًا في حالات محددة:

- وراثة الواجهات: أصناف مجردة تحتوي على دوال افتراضية خالصة فقط.
- دمج سلوكيات مستقلة: دون مشاركة حالة داخلية.

توصية هندسية:

- فضّل التركيب **(Composition)** على الوراثة.
- استخدم الوراثة المتعددة بحذر شديد.

٤.١.١٢ الخلاصة

الوراثة المتعددة في C++ سلاح ذو حدين: تمنح مرونة عالية، لكنها تُدخل تعقيدًا تصميميًا خطيرًا إذا أسيء استخدامها.
أهم النقاط:

- مشكلة الماسة تُحل باستخدام الوراثة الافتراضية.
- تعارض الأسماء والغموض شائعان.
- تعقيد البُناة والصيانة يزداد مع تعدد الوراثة.
- التركيب غالبًا خيار أكثر أمانًا.

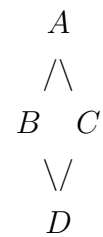
المبرمج المحترف لا يتجنب الوراثة المتعددة كليًا، بل يستخدمها بوعيٍ هندسي وفي أضيق الحدود الممكنة.

٢.١٢ مشكلة الماسة وحلّها باستخدام الوراثة الافتراضية

تُوفّر البرمجة كائنية التوجه (OOP) مرونة عالية في تصميم البرمجيات وإعادة استخدامها، لكن بعض تراكيب الوراثة المعقّدة قد تؤدي إلى مشكلات تصميمية خطيرة. من أبرز هذه المشكلات ما يُعرف بـ مشكلة الماسة **(Diamond Problem)**.
يشرح هذا القسم ماهية مشكلة الماسة، وأسبابها، وتأثيرها على الشيفرة، وكيفية حلّها بشكل صحيح باستخدام الوراثة الافتراضية في C++.

١.٢.١٢ ما هي مشكلة الماسة؟

تحدث مشكلة الماسة عندما يرث صنفٌ ما من صنفين، وهذان الصنفان يشتركان في صنف أساس واحد، مما يؤدي إلى بنية وراثة على شكل ماسة.



الشرح:

- الصنف D يرث من B و C.
- الصنفان B و C يرثان من الصنف A.
- نتيجة لذلك، قد يرث D نسختين من A.

هذا التكرار يؤدي إلى غموض ومشكلات في استدعاء الدوال وإدارة الحالة الداخلية.

٢.٢.١٢ المشكلات الناتجة عن مشكلة الماسة

١. الغموض (Ambiguity) لا يستطيع المترجم تحديد أي نسخة من الدالة أو المتغير الموروث يجب استخدامها.

٢. تعدد نسخ الصنف الأساسي وجود أكثر من نسخة من الصنف A يزيد استهلاك الذاكرة وقد يؤدي إلى تناقض في الحالة الداخلية.

٣. تعقيد التصميم هياكل الوراثة الماسية تجعل الشيفرة أصعب فهمًا وأكثر عرضة للأخطاء والصيانة المكلفة.

٣.٢.١٢ حل مشكلة الماسة باستخدام الوراثة الافتراضية

الوراثة الافتراضية هي آلية في C++ تضمن وجود نسخة واحدة فقط من الصنف الأساسي المشترك، حتى لو تمّت وراثته عبر عدة مسارات.
الفكرة الأساسية:
عند التصريح بأن الوراثة من الصنف الأساسي هي virtual، يضمن المترجم إنشاء نسخة واحدة مشتركة من ذلك الصنف داخل شجرة الوراثة.

٤.٢.١٢ مثال تطبيقي في C++

```
#include <iostream>

class A {
public:
    A() { std::cout << "A Constructor\n"; }
    virtual void show() { std::cout << "Class A\n"; }
};

class B : virtual public A {
```

```
public:
    B() { std::cout << "B Constructor\n"; }
    void show() override { std::cout << "Class B\n"; }
};

class C : virtual public A {
public:
    C() { std::cout << "C Constructor\n"; }
    void show() override { std::cout << "Class C\n"; }
};

class D : public B, public C {
public:
    D() { std::cout << "D Constructor\n"; }
};

int main() {
    D obj;
    obj.show();
    return 0;
}
```

الناتج

```
A Constructor
B Constructor
C Constructor
D Constructor
Class C
```

الشرح

١. الوراثة الافتراضية كل من B و C يرثان A بوراثة افتراضية، مما يضمن وجود نسخة واحدة فقط من A.

٢. ترتيب استدعاء البُناة يتم استدعاء باني A أولاً، ثم B، ثم C، وأخيرًا D.

٣. حسم الدوال الافتراضية عند استدعاء ()show، يتم استخدام نسخة C لأنها آخر صنف يوفّر إعادة تعريف للدالة في مسار الوراثة.

٥.٢.١٢ الخلاصة

تُعد مشكلة الماسة من أخطر التحديات في الوراثة المتعددة، إذ تؤدي إلى غموض، وتكرار غير ضروري، وتعقيد في بنية الشيفرة.
الوراثة الافتراضية تُقدّم حلاً رسميًا وفعالاً لهذه المشكلة من خلال مشاركة نسخة واحدة من الصنف الأساسي المشترك.
فهم هذه الآلية واستخدامها بحذر يسمح ببناء هياكل وراثة معقّدة بأمان واستقرار، ويُجنّب الكثير من الأخطاء الخفية في تصميم البرمجيات الكائنية في C++.

الفصل ١٣: البرمجة الكائنية عالية الأداء

تهدف البرمجة عالية الأداء في C++ إلى الاستفادة القصوى من قدرات العتاد مع الحفاظ على بنية كائنية منظمة وقابلة للصيانة. ورغم أن OOP تُسهّل التصميم وتحسين القابلية للتوسعة، إلا أنها قد تُدخل كلفة زمنية إضافية ناتجة عن استدعاءات الدوال، والتجريد الزائد، والإدارة غير الكفؤة للذاكرة.

يتناول هذا الفصل أهم الممارسات العملية لتحسين الأداء في البرمجة الكائنية، مع التركيز على:

- التحسين باستخدام .Inlining
- تجنب التكرار غير الضروري.
- الإدارة الكفؤة للذاكرة.

١.١٣ تحسين الأداء باستخدام Inlining

في الأنظمة الحسّاسة للأداء مثل محركات الألعاب، والأنظمة الزمنية الحقيقية، والبرمجيات واسعة النطاق، يصبح كل جزء من الميكروثانية مهمًا. استدعاء الدوال، خاصة في التصاميم الكائنية، قد يفرض كلفة زمنية ناتجة عن: تمرير الوسائط، والقفز بين العناوين، وإدارة المكدس.
تُعد تقنية Inlining إحدى أهم وسائل تقليل هذه الكلفة دون التضحية بوضوح التصميم.

١.١.١٣ ما هو Inlining؟

الـ Inlining هو تحسين يقوم فيه المترجم باستبدال استدعاء الدالة بمحتوى الدالة نفسه داخل موقع الاستدعاء.
بدلاً من:

- القفز إلى عنوان الدالة
- تنفيذها
- العودة إلى المستدعي

يتم دمج الكود مباشرة في مكان الاستخدام.
تكون هذه التقنية فعّالة بشكل خاص مع الدوال:

- الصغيرة الحجم
- كثيرة الاستدعاء
- البسيطة منطقيًا

مثل: getters، setters، ودوال المرافق الخفيفة.

٢.١.١٣ فوائد Inlining

١. تقليل كلفة استدعاء الدوال يتم حذف كلفة القفز وإدارة المكدس بالكامل.

٢. تحسين أداء الذاكرة المخبئية (Cache) تقليل القفزات بين الدوال يُحسّن من توطين التعليمات داخل ذاكرة التعليمات.

٣. فتح المجال لتحسينات إضافية يسمح للمترجم بإزالة الحسابات المكررة وتحسين الترتيب الداخلي للتعليمات.

٤. تقليل الضغط على Stack Call وهو عامل بالغ الأهمية في الأنظمة الزمنية الحقيقية.

٣.١.١٣ سلبيات Inlining

١. تضخم الشيفرة التنفيذية Code) (Bloat تكرار كود الدالة في عدة مواقع قد يزيد حجم الملف التنفيذي.

٢. تعقيد التتبع أثناء التصحيح اختفاء حدود الدوال قد يجعل تتبع الأخطاء أصعب.

٣. فائدة محدودة مع الدوال الكبيرة الدوال المعقّدة لا تستفيد كثيرًا من الدمج، مقابل زيادة حجم الشيفرة.

٤. قيود مع الدوال الافتراضية آلية الربط الديناميكي تمنع غالبًا عملية الدمج.

٤.١.١٣ استخدام الكلمة المفتاحية inline في C++

في C++، تُستخدم الكلمة المفتاحية inline كتلميح للمترجم، وليس أمرًا إلزاميًا. القرار النهائي يعود للمترجم بناءً على التحليل الأمثل.

مثال: دمج دالة بسيطة

```cpp
#include <iostream>

inline int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 10;
    int result = add(x, y);
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

Inlining في البرمجة الكائنية

```cpp
class Rectangle {
private:
    int width, height;

public:
    inline int getWidth()  const { return width; }
    inline int getHeight() const { return height; }

    inline void setWidth(int w)  { width  = w; }
    inline void setHeight(int h) { height = h; }

    inline int area() const {
```

```
        return width * height;
    }
};
```

هذا الأسلوب شائع جداً في الأصناف خفيفة الوزن (Value Types).

Inlining داخل الحلقات

```
inline int multiply(int a, int b) {
    return a * b;
}

int main() {
    int result = 0;
    for (int i = 0; i < 1'000'000; ++i) {
        result += multiply(i, 2);
    }
    return 0;
}
```

في الحلقات ذات التكرار العالي، تتراكم كلفة الاستدعاء، ويكون الدمج ذا أثر ملموس.

٥.١.١٣ متى يجب تجنب Inlining؟

١. الدوال الكبيرة أو المعقّدة

٢. الدوال ية

٣. الدوال اللافتراضية متعددة الأشكال

في هذه الحالات، قد تكون كلفة التضخم أكبر من الفائدة الزمنية.

٦.١.١٣ الخلاصة

تُعد تقنية Inlining أداة فعّالة لتحسين الأداء في البرمجة الكائنية بلغة C++، خصوصًا مع الدوال الصغيرة وكثيرة الاستدعاء.
لكن استخدامها العشوائي قد يؤدي إلى تضخم الشيفرة وصعوبة الصيانة. النهج الصحيح يعتمد على:

- الفهم العميق لبنية البرنامج
- القياس الفعلي للأداء (Profiling)
- اتخاذ قرارات مدروسة مبنية على البيانات

بهذا الأسلوب، يمكن الجمع بين قوة التصميم الكائني وكفاءة الأداء المنخفض المستوى دون تناقض.

٢.١٣ تجنّب التكرار غير الضروري

في البرمجة الكائنية (OOP)، يُعدّ التخلص من التكرار غير الضروري أمرًا جوهريًا لكتابة شيفرة قابلة للصيانة، وعالية الأداء، وواضحة البنية. التكرار الزائد لا يؤدي فقط إلى تعقيد الصيانة، بل قد يسبب: اختناقات في الأداء، وزيادة في استهلاك الذاكرة، وارتفاع احتمال ظهور الأخطاء.
يُعد مبدأ DRY (Don't Repeat Yourself) من الركائز الأساسية لتقليل التكرار وبناء برمجيات مُحسّنة ومرنة. يستعرض هذا القسم أهم الأساليب العملية في C++ لتجنّب التكرار، مدعومة بأمثلة تطبيقية واضحة.

١.٢.١٣ مشكلة التكرار في البرمجة الكائنية

يظهر التكرار طبيعيًا في التصاميم الكائنية بسبب الأنماط المتكررة وتشابه البُنى. لكن التكرار المفرط يؤدي إلى عدة مشكلات، منها:

- مشكلات في الأداء والذاكرة: تكرار الشيفرة يزيد حجم البرنامج وقد يقلل كفاءة التنفيذ في المشاريع الكبيرة.
- ازدواجية المنطق البرمجي: صيانة منطق مكرر في عدة مواضع تزيد احتمال التناقض والأخطاء.
- ضعف القابلية للصيانة: أي تعديل يتطلب تغييرات متكررة في عدة أماكن، وهو أمر مرهق ومعرّض للأخطاء.

٢.٢.١٣ تطبيق مبدأ DRY

يهدف مبدأ DRY إلى ضمان وجود كل وظيفة أو معرفة في موضع واحد فقط داخل النظام. في C++، يمكن تحقيق ذلك بعدة تقنيات عملية.

أهم التقنيات لتجنّب التكرار في C++

١. إعادة استخدام الدوال والأساليب

يتم تغليف المنطق المتكرر داخل دوال أو أساليب قابلة لإعادة الاستخدام.

مثال:

```
class Shape {
public:
    double calculateArea(double length, double width) {
        return length * width;
    }
};

int main() {
    Shape rectangle;
    double area1 = rectangle.calculateArea(5.0, 10.0);
    double area2 = rectangle.calculateArea(7.0, 12.0);
    return 0;
}
```

يسمح هذا الأسلوب بتوحيد المنطق وتجنّب تكراره في أكثر من موضع.

٢. استخدام الوراثة وتعدد الأشكال

توفّر الأصناف الأساسية سلوكًا مشتركًا يمكن للأصناف المشتقة إعادة استخدامه مع تخصيصه عند الحاجة.

مثال:

```
class Animal {
public:
    virtual void speak() const {
```

```
        std::cout << "Animal sound\n";
    }
};

class Dog : public Animal {
public:
    void speak() const override {
        std::cout << "Bark\n";
    }
};

class Cat : public Animal {
public:
    void speak() const override {
        std::cout << "Meow\n";
    }
};

void makeAnimalSpeak(const Animal& animal) {
    animal.speak();
}

int main() {
    Dog dog;
    Cat cat;
    makeAnimalSpeak(dog);
    makeAnimalSpeak(cat);
    return 0;
}
```

بهذا الأسلوب، يتم التخلص من التكرار مع الحفاظ على مرونة السلوك.

٣. استخدام القوالب (Templates) لتعميم الشيفرة

تمكّن القوالب من كتابة دالة أو صنف واحد يعمل مع أنواع بيانات متعددة، بدل تكرار نفس الشيفرة.

مثال:

```
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    int intMax = findMax(5, 10);
    double doubleMax = findMax(3.5, 7.8);
    return 0;
}
```

تُغني القوالب عن كتابة دوال منفصلة لكل نوع بيانات.

٤. استخدام خوارزميات المكتبة القياسية (STL)

توفر مكتبة STL خوارزميات جاهزة وفعالة تقلل الحاجة إلى إعادة كتابة المنطق الشائع.

مثال:

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    std::vector<int> numbers = {5, 3, 8, 6, 1};
```

```
    std::sort(numbers.begin(), numbers.end());
    std::for_each(numbers.begin(), numbers.end(),
                  [](int n) { std::cout << n << " "; });
    return 0;
}
```

هذا النهج يقلل التكرار ويعتمد على حلول مجرّبة وعالية الأداء.

٥. نمط CRTP (Curiously Recurring Template Pattern)

يسمح CRTP بتحقيق تعدد الأشكال وقت الترجمة بدون كلفة الدوال الافتراضية.

مثال:

```
template <typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class DerivedClass : public Base<DerivedClass> {
public:
    void implementation() {
        std::cout << "DerivedClass implementation\n";
    }
};

int main() {
    DerivedClass obj;
    obj.interface();
```

```
    return 0;
}
```

يوفّر هذا الأسلوب سلوكًا متعدد الأشكال مع كفاءة أعلى.

٦. تقليل التكرار باستخدام الماكرو (بحذر)

يمكن للماكرو تقليل الشيفرة النمطية، لكن يُفضّل القوالب في C++ الحديثة.

مثال:

```
#define GENERATE_GETTER_SETTER(Type, VarName) \
    Type get##VarName() const { return VarName; } \
    void set##VarName(Type value) { VarName = value; }

class Person {
private:
    std::string name;
    int age;
public:
    GENERATE_GETTER_SETTER(std::string, Name)
    GENERATE_GETTER_SETTER(int, Age)
};

int main() {
    Person p;
    p.setName("John");
    p.setAge(30);
    std::cout << p.getName()
              << " is "
              << p.getAge()
              << " years old.\n";
```

```
    return 0;
}
```

يقلل هذا الأسلوب من التكرار الشكلي، لكن يجب استخدامه بحذر.

٣.٢.١٣ الخلاصة

يُعد تجنّب التكرار غير الضروري أحد الأعمدة الأساسية في البرمجة الكائنية عالية الأداء. الاعتماد على:

- إعادة استخدام الدوال
- الوراثة وتعدد الأشكال
- القوالب
- خوارزميات STL
- أنماط التصميم مثل CRTP

يؤدي إلى شيفرة: أكثر وضوحًا، وأقل أخطاءً، وأعلى قابلية للصيانة، وأفضل أداءً على المدى الطويل.
الالتزام بمبدأ DRY ليس مجرد تحسين شكلي، بل هو قرار هندسي يرفع جودة النظام ويضمن استدامته.

٣.١٣ الإدارة الفعّالة للذاكرة

في التطبيقات عالية الأداء، خصوصًا في برمجة الأنظمة والبرمجيات القريبة من العتاد، تُعد الإدارة الفعّالة للذاكرة عاملاً حاسمًا في الاستقرار والأداء. ورغم أن البرمجة الكائنية (OOP) توفّر بنية وتنظيمًا ممتازين، إلا أن سوء إدارة الذاكرة قد يؤدي إلى: تسرّبات، تجزئة، وانخفاض تدريجي في الأداء.
تمنح C++ تحكمًا مباشرًا ودقيقًا بالذاكرة، وهو سلاح ذو حدّين: قوة كبيرة مقابل مسؤولية عالية. يستعرض هذا القسم الاستراتيجيات الحديثة لإدارة الذاكرة بكفاءة في C++ OOP، مدعومة بأمثلة عملية واضحة.

١.٣.١٣ أهمية الإدارة الفعّالة للذاكرة

تضمن الإدارة السليمة للذاكرة أن البرامج:

- تحقّق أعلى أداء ممكن: تقليل عمليات الحجز والتحرير المتكررة أمر بالغ الأهمية في الأنظمة الزمنية الحقيقية والتطبيقات الحسّاسة للأداء.
- تمنع تسرّب الذاكرة: التعامل الصحيح مع الموارد يمنع نفاد الذاكرة والانهيارات غير المتوقعة.
- تحافظ على استقرار النظام: الاستخدام الأمثل للذاكرة ينعكس مباشرة على القابلية للتوسّع والموثوقية طويلة الأمد.

قد تؤدي OOP إلى كلفة إضافية بسبب: سلاسل وراثة عميقة، كائنات كبيرة، وحجوزات متكررة. لذا فإن دمج OOP مع تقنيات إدارة الذاكرة الحديثة هو السبيل لتحقيق الأداء العالي دون التضحية بالتصميم الجيد.

٢.٣.١٣ تقنيات إدارة الذاكرة في C++

١. الإدارة اليدوية للذاكرة باستخدام المؤشرات

يتيح استخدام new وdelete تحكمًا مباشرًا بالذاكرة، لكنه يحمل مخاطر كبيرة مثل: تسرّب الذاكرة، مؤشرات معلّقة، والتحرير المزدوج.

مثال:

```
class Car {
public:
    Car() { std::cout << "Car created\n"; }
    ~Car() { std::cout << "Car destroyed\n"; }
};

int main() {
    Car* myCar = new Car();   //
    //
    delete myCar;             //
    return 0;
}
```

رغم المرونة، فإن نسيان delete يؤدي مباشرة إلى تسرّب في الذاكرة.

٢. المؤشرات الذكية (Smart Pointers)

قدّمت C++11 المؤشرات الذكية كحل آمن لإدارة الذاكرة تلقائيًا.

- std::unique_ptr ملكية حصرية؛ يتم تحرير الكائن عند خروج المؤشر من النطاق.

```
#include <memory>

class Engine {
public:
    Engine() { std::cout << "Engine created\n"; }
    ~Engine() { std::cout << "Engine destroyed\n"; }
```

```
};

int main() {
    std::unique_ptr<Engine> engine =
        std::make_unique<Engine>();
    return 0; //
}
```

- std::shared_ptr ملكية مشتركة؛ يُحرَّر الكائن عند اختفاء آخر مرجع.

```
#include <memory>

class Driver {
public:
    Driver() { std::cout << "Driver created\n"; }
    ~Driver() { std::cout << "Driver destroyed\n"; }
};

int main() {
    std::shared_ptr<Driver> d1 =
        std::make_shared<Driver>();
    std::shared_ptr<Driver> d2 = d1;
    return 0;
}
```

- std::weak_ptr مرجع غير مالك، يُستخدم لتجنّب الاعتمادات الدائرية.

```
#include <memory>

class Passenger {
public:
    Passenger() { std::cout << "Passenger created\n"; }
```

```
    ~Passenger() { std::cout << "Passenger destroyed\n"; }
};

int main() {
    std::shared_ptr<Passenger> p =
        std::make_shared<Passenger>();
    std::weak_ptr<Passenger> wp = p;

    if (auto sp = wp.lock()) {
        std::cout << "Passenger still exists\n";
    }
    return 0;
}
```

٣. مبدأ RAII (Resource Acquisition Is Initialization)

يربط مبدأ RAII اكتساب الموارد بالبناء (Constructor) وتحريرها بالهدم (Destructor)، سواء كانت: ذاكرة، ملفات، أقفال، أو موارد نظام.

```
class File {
    FILE* file;
public:
    File(const char* filename) {
        file = fopen(filename, "r");
        if (!file)
            throw std::runtime_error("Open failed");
    }
    ~File() {
        if (file) fclose(file);
    }
```

```
};

int main() {
    try {
        File f("example.txt");
    } catch (const std::exception& e) {
        std::cerr << e.what() << '\n';
    }
    return 0;
}
```

يضمن RAII تنظيف الموارد حتى في وجود الاستثناءات.

٤. تجميع الذاكرة (Memory Pooling) والمخصّصات المخصّصة

في الأنظمة الزمنية الحقيقية وألعاب الفيديو، يُفضَّل الحجز المسبق للذاكرة لتقليل كلفة الحجز المتكرر.

```
template<typename T>
class MemoryPool {
public:
    T* allocate() { return new T(); }
    void deallocate(T* ptr) { delete ptr; }
};

int main() {
    MemoryPool<int> pool;
    int* a = pool.allocate();
    *a = 10;
    std::cout << *a << std::endl;
    pool.deallocate(a);
```

```
    return 0;
}
```

رغم بساطة المثال، إلا أن المفهوم أساسي في الأنظمة عالية الأداء.

٥. تجنّب تسرّب الذاكرة

أفضل الممارسات تشمل:

- استخدام المؤشرات الذكية
- الالتزام بـ RAII
- تعريف مُهدّمات صحيحة
- استخدام أدوات تحليل مثل: Valgrind وAddressSanitizer

٦. تحسين استخدام الذاكرة باستخدام Semantics Move

قدّمت C++11 دلالات النقل لتفادي النسخ المكلف للكائنات الكبيرة، واستبداله بنقل الملكية.

```
#include <vector>
#include <iostream>

class BigData {
    std::vector<int> data;
public:
    BigData(size_t size) : data(size) {}

    BigData(BigData&& other) noexcept
        : data(std::move(other.data)) {
        std::cout << "Data moved\n";
    }
```

```
    BigData& operator=(BigData&& other) noexcept {
        if (this != &other)
            data = std::move(other.data);
        return *this;
    }
};

int main() {
    BigData d1(1'000'000);
    BigData d2 = std::move(d1);
    return 0;
}
```

تُعد هذه التقنية حاسمة عند التعامل مع حاويات كبيرة وموارد ثقيلة.

٣.٣.١٣ الخلاصة

تمثّل الإدارة الفعّالة للذاكرة أحد أعمدة البرمجة عالية الأداء في C++ OOP. من خلال:

- المؤشرات الذكية
- مبدأ RAII
- تجميع الذاكرة
- دلالات النقل

يمكن بناء أنظمة: آمنة، قابلة للتوسّع، وعالية الكفاءة، دون التفريط بمزايا التصميم الكائني.
الإدارة الصحيحة للذاكرة ليست خياراً إضافياً، بل شرط أساسي لبناء برمجيات احترافية تتحمّل ضغط الواقع وتعقيد الأنظمة الحديثة.

الفصل ١٤: البرمجة متعددة الخيوط في البرمجة الكائنية

- أمان الخيوط (Thread Safety).
- الكائنات المشتركة الآمنة للخيوط.
- الأقفال والمُعَاوِق (Mutex & Locks) في البرمجة الكائنية.

١.١٤ أمان الخيوط (Thread Safety)

تُعد البرمجة متعددة الخيوط أحد الأعمدة الأساسية في تطوير البرمجيات الحديثة، حيث تتيح للتطبيقات تنفيذ عدة عمليات بشكل متزامن. في سياق البرمجة الكائنية (OOP)، يصبح ضمان أمان الخيوط أمرًا بالغ الأهمية عند وصول عدة خيوط إلى موارد مشتركة.
يشير أمان الخيوط إلى قدرة البرنامج على العمل بسلوك صحيح ومتوقّع حتى في حال القراءة أو التعديل المتزامن للبيانات المشتركة.
يستعرض هذا القسم مفهوم أمان الخيوط في C++ OOP، أهميته، أبرز مخاطره، وأهم الأساليب العملية لتحقيقه.

١.١.١٤ لماذا يُعد أمان الخيوط أمرًا بالغ الأهمية؟

عند غياب المزامنة الصحيحة، قد يؤدي الوصول المتزامن إلى الموارد المشتركة إلى مشكلات خطيرة، منها:

- حالات السباق **(Race Conditions):** يعتمد سلوك البرنامج على ترتيب تنفيذ الخيوط، مما يؤدي إلى نتائج غير متوقعة وغير قابلة للتكرار.
- تلف البيانات **(Data Corruption):** التعديلات المتزامنة دون تنسيق قد تضع الكائنات في حالات غير متناسقة.
- الاختناقات :(Deadlocks) تنتظر خيوط متعددة تحرير موارد يحتجزها بعضها البعض، مما يؤدي إلى توقف البرنامج نهائيًا.

يتطلّب تحقيق أمان الخيوط تنظيم الوصول إلى البيانات المشتركة بطريقة تضمن الاتساق وقابلية التنبؤ.

٢.١.١٤ مثال على حالة سباق (Race Condition)

```cpp
#include <iostream>
```

```cpp
#include <thread>

class Counter {
public:
    int count = 0;

    void increment() {
        for (int i = 0; i < 100000; ++i) {
            count++;
        }
    }
};

int main() {
    Counter counter;

    std::thread t1(&Counter::increment, &counter);
    std::thread t2(&Counter::increment, &counter);

    t1.join();
    t2.join();

    std::cout << "Final count: "
              << counter.count << std::endl;
    return 0;
}
```

في هذا المثال، يقوم خيطان بزيادة المتغير count في الوقت نفسه. النتيجة المتوقعة هي 200000، لكن بسبب غياب المزامنة، قد تكون النتيجة أقل أو غير ثابتة، وهي حالة سباق كلاسيكية.

٣.١.١٤ تقنيات ضمان أمان الخيوط

توجد عدة تقنيات لضمان أمان الخيوط في تطبيقات OOP متعددة الخيوط:

١. المُعَاوِق **(Mutex – Mutual Exclusion)**

يضمن std::mutex أن يتم الوصول إلى مورد مشترك من خيط واحد فقط في كل مرة.

```
#include <iostream>
#include <thread>
#include <mutex>

class Counter {
public:
    int count = 0;
    std::mutex mtx;

    void increment() {
        for (int i = 0; i < 100000; ++i) {
            std::lock_guard<std::mutex> lock(mtx);
            count++;
        }
    }
};

int main() {
    Counter counter;

    std::thread t1(&Counter::increment, &counter);
    std::thread t2(&Counter::increment, &counter);
```

```
    t1.join();
    t2.join();

    std::cout << "Final count: "
              << counter.count << std::endl;
    return 0;
}
```

تقوم std::lock_guard بقفل المُعَاوِق تلقائيًا عند إنشائها وتحريره عند الخروج من النطاق، مما يمنع الأخطاء الشائعة.

٢. المتغيرات الذرّية **(Atomic Variables)**

توفّر std::atomic عمليات آمنة للخيوط دون الحاجة إلى أقفال صريحة، وهي مناسبة للعمليات البسيطة.

```
#include <iostream>
#include <thread>
#include <atomic>

class Counter {
public:
    std::atomic<int> count{0};

    void increment() {
        for (int i = 0; i < 100000; ++i) {
            count++;  //
        }
    }
};
```

```
int main() {
    Counter counter;

    std::thread t1(&Counter::increment, &counter);
    std::thread t2(&Counter::increment, &counter);

    t1.join();
    t2.join();

    std::cout << "Final count: "
              << counter.count << std::endl;
    return 0;
}
```

تُعد المتغيرات الذرّية أسرع من الأقفال في الحالات البسيطة، لكنها محدودة الاستخدام ولا تصلح للمنطق المعقّد.

٣. الكائنات والبنى الآمنة للخيوط

استخدام بُنى بيانات مصمّمة للعمل المتزامن يقلّل الحاجة إلى المزامنة اليدوية. توفّر مكتبات مثل Boost و Intel TBB مثل هذه البنى.

مثال: Singleton آمن للخيوط

```
#include <iostream>
#include <mutex>
#include <memory>

class Singleton {
private:
    static std::unique_ptr<Singleton> instance;
```

```
    static std::mutex mtx;

    Singleton() {}

public:
    static Singleton* getInstance() {
        std::lock_guard<std::mutex> lock(mtx);
        if (!instance) {
            instance.reset(new Singleton());
        }
        return instance.get();
    }

    void display() {
        std::cout << "Singleton Instance\n";
    }
};

std::unique_ptr<Singleton> Singleton::instance = nullptr;
std::mutex Singleton::mtx;

int main() {
    Singleton* s1 = Singleton::getInstance();
    Singleton* s2 = Singleton::getInstance();

    s1->display();
    s2->display();
    return 0;
}
```

يضمن المُعَاوِق إنشاء نسخة واحدة فقط حتى في وجود عدة خيوط.

٤. تجنّب الحالة المشتركة قدر الإمكان

كلما قلّت البيانات المشتركة، قلّت الحاجة إلى المزامنة. يمكن:

- استخدام نسخ مستقلة لكل خيط
- الاعتماد على التخزين المحلي للخيوط
- تمرير البيانات بدل مشاركتها

وهو غالبًا الحل الأبسط والأكثر كفاءة.

٤.١.١٤ الخلاصة

يُعد أمان الخيوط عنصرًا أساسيًا لبناء تطبيقات OOP متعددة الخيوط تتميّز بالموثوقية وقابلية التوسّع.
من خلال:

- استخدام std::mutex والأقفال
- الاستفادة من std::atomic
- توظيف بُنى بيانات آمنة للخيوط
- تقليل الحالة المشتركة

يمكن بناء برامج تعمل بشكل صحيح حتى تحت ضغط التوازي العالي.
التصميم الجيد في البرمجة متعددة الخيوط لا يقل أهمية عن آليات المزامنة نفسها، وهو ما يميّز البرمجيات الاحترافية عن الحلول الهشّة.

١٤.٢ الكائنات المشتركة الآمنة للخيوط

في البرمجة الكائنية متعددة الخيوط (Multithreaded OOP)، غالبًا ما تحتاج عدة خيوط إلى الوصول إلى نفس الكائنات لتحسين الأداء والاستجابة. لكن هذا التشارك يُدخل تحديات خطيرة تتعلق بسلامة البيانات وصحة السلوك البرمجي.
يستعرض هذا القسم مفهوم الكائنات المشتركة والمخاطر المرتبطة بها، ثم يقدّم أهم الاستراتيجيات العملية لبناء كائنات مشتركة آمنة للخيوط في لغة C++.

١٤.٢.١ ما المقصود بالكائنات المشتركة؟

الكائن المشترك هو كائن يتم الوصول إليه من أكثر من خيط في الوقت نفسه. وعند غياب المزامنة الصحيحة، قد تظهر مشكلات خطيرة، مثل:

- حالات السباق **(Race Conditions)**: تؤدي التحديثات المتزامنة إلى نتائج غير متوقعة.
- تلف البيانات **(Data Corruption)**: القراءة أثناء التعديل قد تترك الكائن في حالة غير متسقة.
- الاختناقات :(Deadlocks) انتظار الخيوط لبعضها البعض يؤدي إلى تجميد النظام.

الإدارة الصحيحة للكائنات المشتركة هي ما يضمن سلوكًا متوقّعًا وصحيحًا للبرنامج.

١٤.٢.٢ استراتيجيات بناء كائنات مشتركة آمنة للخيوط

توجد عدة تقنيات لضمان أمان الخيوط عند التعامل مع الكائنات المشتركة:

١. استخدام المُعَاوِق (Mutex) لحماية الكائن المشترك

يسمح المُعَاوِق لخيط واحد فقط بالدخول إلى المقطع الحرج في كل مرة.

```
#include <iostream>
#include <thread>
```

```cpp
#include <mutex>

class SharedObject {
public:
    int value;
    std::mutex mtx;

    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        ++value;
    }
};

void threadFunction(SharedObject& obj) {
    for (int i = 0; i < 100000; ++i)
        obj.increment();
}

int main() {
    SharedObject obj{0};

    std::thread t1(threadFunction, std::ref(obj));
    std::thread t2(threadFunction, std::ref(obj));

    t1.join();
    t2.join();

    std::cout << "Final Value: "
              << obj.value << std::endl;
```

```
    return 0;
}
```

يضمن std::lock_guard قفل المُعَاوِق وفتحه تلقائيًا، مما يقلّل الأخطاء البرمجية.

٢. العمليات الذرّية **(Atomic Operations)**

توفّر المتغيرات الذرّية عمليات آمنة للخيوط دون الحاجة إلى أقفال، وهي مناسبة للأنواع البسيطة.

```
#include <iostream>
#include <thread>
#include <atomic>

class SharedObject {
public:
    std::atomic<int> value;

    void increment() {
        ++value;
    }
};

void threadFunction(SharedObject& obj) {
    for (int i = 0; i < 100000; ++i)
        obj.increment();
}

int main() {
    SharedObject obj{0};
```

```
    std::thread t1(threadFunction, std::ref(obj));
    std::thread t2(threadFunction, std::ref(obj));

    t1.join();
    t2.join();

    std::cout << "Final Value: "
              << obj.value.load() << std::endl;
    return 0;
}
```

العمليات الذرّية أسرع من الأقفال، لكنها محدودة الاستخدام ولا تصلح للحالات المعقّدة.

٣. الحاويات الآمنة للخيوط

عند مشاركة مجموعات من البيانات مثل القوائم أو المتجهات، يجب حماية العمليات عليها.

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

class SharedObject {
public:
    std::vector<int> vec;
    std::mutex mtx;

    void addValue(int val) {
        std::lock_guard<std::mutex> lock(mtx);
        vec.push_back(val);
    }
```

```
    void print() {
        std::lock_guard<std::mutex> lock(mtx);
        for (int v : vec)
            std::cout << v << " ";
        std::cout << std::endl;
    }
};

void threadFunction(SharedObject& obj) {
    for (int i = 0; i < 5; ++i)
        obj.addValue(i);
}

int main() {
    SharedObject obj;

    std::thread t1(threadFunction, std::ref(obj));
    std::thread t2(threadFunction, std::ref(obj));

    t1.join();
    t2.join();

    obj.print();
    return 0;
}
```

يمكن أيضًا الاستعانة بمكتبات مثل Boost و Intel TBB لتوفير حاويات متزامنة جاهزة.

٤. الملكية المشتركة باستخدام **Smart Pointers**

تُستخدم std::shared_ptr لإدارة ملكية كائن مشترك بين عدة خيوط بشكل آمن.

```
#include <iostream>
#include <memory>
#include <thread>

void threadFunction(std::shared_ptr<int> ptr) {
    (*ptr)++;
}

int main() {
    auto sharedObj = std::make_shared<int>(0);

    std::thread t1(threadFunction, sharedObj);
    std::thread t2(threadFunction, sharedObj);

    t1.join();
    t2.join();

    std::cout << "Final Value: "
              << *sharedObj << std::endl;
    return 0;
}
```

لكن يجب الانتباه إلى أن shared_ptr يحمي إدارة العمر فقط، ولا يحمي الوصول المتزامن للبيانات الداخلية.

٥. أقفال القراءة والكتابة **(Read-Write Locks)**

عند كثرة عمليات القراءة وقلة الكتابة، تُعد std::shared_mutex حلاً أكثر كفاءة.

```
#include <iostream>
```

```cpp
#include <thread>
#include <shared_mutex>

class SharedObject {
public:
    int value;
    std::shared_mutex rw_mtx;

    void readValue() {
        std::shared_lock<std::shared_mutex> lock(rw_mtx);
        std::cout << "Read Value: "
                  << value << std::endl;
    }

    void writeValue(int newVal) {
        std::unique_lock<std::shared_mutex> lock(rw_mtx);
        value = newVal;
    }
};

void reader(SharedObject& obj) {
    obj.readValue();
}

void writer(SharedObject& obj, int v) {
    obj.writeValue(v);
}

int main() {
```

```
    SharedObject obj{42};

    std::thread r1(reader, std::ref(obj));
    std::thread r2(reader, std::ref(obj));
    std::thread w(writer, std::ref(obj), 100);

    r1.join();
    r2.join();
    w.join();
    return 0;
}
```

يسمح هذا النمط بعدة قراءات متزامنة مع ضمان حصرية الكتابة.

٣.٢.١٤ الخلاصة

إدارة الكائنات المشتركة في البرمجة الكائنية متعددة الخيوط تتطلب وعيًا عميقًا بمخاطر التوازي.
باستخدام:

- المُعَاوِق (Mutex)
- المتغيرات الذرّية
- الحاويات الآمنة للخيوط
- مؤشرات الملكية الذكية
- أقفال القراءة والكتابة

يمكن بناء أنظمة آمنة، فعّالة، وقابلة للتوسّع.
الاختيار الصحيح لآلية المزامنة يعتمد على نمط الوصول للبيانات ومتطلبات الأداء، وهو ما يميّز التصميم الاحترافي في البرمجيات متعددة الخيوط.

٣.١٤ المُعَاوِق (Mutex) والأقفال في البرمجة الكائنية

تُعد البرمجة متعددة الخيوط (Multithreading) أحد الأعمدة الأساسية في البرمجيات الحديثة، حيث تسمح بتنفيذ عدة مهام بالتوازي واستغلال أنوية المعالج بكفاءة. لكن هذا التوازي يُدخل تحديًا خطيرًا: الوصول المتزامن إلى الموارد المشتركة.
في البرمجة الكائنية (OOP)، تمثل الكائنات المشتركة نقاطًا حرجة قد تؤدي إلى حالات سباق، تلف البيانات، أو سلوك غير متوقع. هنا يأتي دور المُعَاوِق (Mutex) والأقفال (Locks) كآليات أساسية لضمان سلامة الخيوط.
يستعرض هذا القسم مفهوم المُعَاوِق، أنواع الأقفال في C++، طرق استخدامها داخل الكائنات، وأفضل الممارسات في التصميم الكائني المتعدد الخيوط.

١.٣.١٤ ما هو المُعَاوِق ؟(Mutex)

المُعَاوِق (Mutual Exclusion) هو أداة مزامنة تضمن أن خيطًا واحدًا فقط يمكنه الوصول إلى مقطع حرج في لحظة معينة.
الفكرة الأساسية:

- القفل :(Lock) حجز المورد ومنع بقية الخيوط من استخدامه.
- الفتح :(Unlock) تحرير المورد والسماح لغيره بالدخول.
- الحجب :(Blocking) الخيوط الأخرى تنتظر حتى يُفتح المُعَاوِق.

بدون المُعَاوِق، قد يتداخل تنفيذ الخيوط بشكل غير متوقع، مؤديًا إلى أخطاء يصعب تتبعها.

٢.٣.١٤ ما المقصود بالأقفال ؟(Locks)

الأقفال هي طبقة أعلى فوق المُعَاوِق، تسهّل إدارته وتعتمد غالبًا على مبدأ .RAII
في C++، الأكثر شيوعًا هما:

- std::lock_guard: قفل بسيط، يُغلق المُعَاوِق عند الإنشاء ويفتحه تلقائيًا عند الخروج من النطاق.
- std::unique_lock: قفل أكثر مرونة، يدعم القفل المؤجل، الفتح اليدوي، والأقفال المؤقتة.

هذه الأدوات تقلّل الأخطاء وتمنع نسيان فتح المُعَاوِق.

٣.٣.١٤ لماذا نستخدم المُعَاوِق والأقفال؟

استخدام المُعَاوِق والأقفال يساعد على:

١. منع حالات السباق عند الوصول للبيانات المشتركة.

٢. الحفاظ على اتساق الحالة الداخلية للكائنات.

٣. تقليل مخاطر الأعطال غير المتوقعة.

لكن سوء الاستخدام قد يؤدي إلى اختناقات أو انخفاض الأداء، لذا يجب استخدامها بحذر.

٤.٣.١٤ استخدام المُعَاوِق داخل الكائنات

في التصميم الكائني، يُفضّل تغليف المُعَاوِق داخل الكائن لحماية حالته الداخلية.

استخدام std::lock_guard

```
#include <iostream>
#include <thread>
#include <mutex>

class Counter {
private:
```

```
    int count;
    std::mutex mtx;

public:
    Counter() : count(0) {}

    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        ++count;
        std::cout << "Count after increment: "
                  << count << "\n";
    }

    int getCount() {
        std::lock_guard<std::mutex> lock(mtx);
        return count;
    }
};

void task(Counter& c) {
    for (int i = 0; i < 5; ++i)
        c.increment();
}

int main() {
    Counter counter;
    std::thread t1(task, std::ref(counter));
    std::thread t2(task, std::ref(counter));
```

```
    t1.join();
    t2.join();

    std::cout << "Final count: "
              << counter.getCount() << "\n";
    return 0;
}
```

هذا الأسلوب هو الأكثر أمانًا وبساطة، ويُعد الخيار الافتراضي في معظم الحالات.

استخدام std::unique_lock

```
#include <iostream>
#include <thread>
#include <mutex>

class Counter {
private:
    int count;
    std::mutex mtx;

public:
    Counter() : count(0) {}

    void increment() {
        std::unique_lock<std::mutex> lock(mtx);
        ++count;
        std::cout << "Count after increment: "
                  << count << "\n";
        lock.unlock(); //
```

```
    }

    int getCount() {
        std::unique_lock<std::mutex> lock(mtx);
        return count;
    }
};
```

مزايا std::unique_lock:

- القفل المؤجل
- الفتح اليدوي
- دعم الأقفال الزمنية

مثال: القفل المؤجل

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

class SharedResource {
private:
    int data;
    std::mutex mtx;

public:
    SharedResource() : data(0) {}
```

```
    void update() {
        std::unique_lock<std::mutex> lock(mtx, std::defer_lock);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lock.lock();
        ++data;
        std::cout << "Data updated: "
                  << data << "\n";
    }
};
```

هذا الأسلوب مفيد عند الحاجة لتأخير القفل حتى نقطة محددة.

٥.٣.١٤ أخطاء شائعة يجب تجنبها

- الاختناقات :(Deadlocks) تجنّبها بقفل المُعَاوِق بنفس الترتيب دائمًا.
- الإفراط في القفل: قلّل حجم المقطع الحرج لتحسين الأداء.
- الإدارة اليدوية للأقفال: استخدم RAII لتجنّب نسيان الفتح.

٦.٣.١٤ المُعَاوِق وأنماط التصميم

يُستخدم المُعَاوِق في أنماط تصميم متعددة، أشهرها Singleton الآمن للخيوط.

```
#include <iostream>
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
```

```
    static std::mutex mtx;
    Singleton() {}

public:
    static Singleton* getInstance() {
        std::lock_guard<std::mutex> lock(mtx);
        if (!instance)
            instance = new Singleton();
        return instance;
    }

    void show() {
        std::cout << "Singleton accessed\n";
    }
};

Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

٧.٣.١٤ الخلاصة

المُعَاوِق والأقفال هي حجر الأساس لبناء برمجيات كائنية متعددة الخيوط آمنة وموثوقة.
باستخدام:

- std::mutex
- std::lock_guard
- std::unique_lock

ومع الالتزام بمبدأ RAII وتصميم المقاطع الحرجة بعناية، يمكنك بناء أنظمة فعّالة، قابلة للتوسّع، وخالية من أخطاء التزامن القاتلة.

الملاحق

صُمِّمت هذه الملاحق لتكون مرجعًا عمليًا يُستخدم أثناء التصميم ومراجعة الشيفرة واتخاذ القرار في أنظمة Object-Oriented Programming (OOP) المبنية بـModern C++ (C++20/23). لا تكرر هذه الملاحق محتوى الفصول، بل تُحوِّله إلى أدوات: قوائم تحقق، جداول قرار، ومعايير تقييم قابلة للتطبيق فورًا.

ملحق أ: قائمة التحقق الهندسية لتصميم الكائنات

أ-1: قبل كتابة class

استخدم هذه الأسئلة لتثبيت حدود الكائن قبل كتابة أي شيفرة:

- المسؤولية: هل يمكن تلخيص مسؤولية الكائن بجملة واحدة دقيقة؟ هل توجد مسؤولية ثانية مخفية؟
- حدود المجال: هل هذا الكائن يمثل مفهومًا حقيقيًا في المجال أم هو تجميع تقني مؤقت؟
- حالات الصحة: ما هي الحالات الصالحة/غير الصالحة للكائن؟ وما الذي يمنع تكوين حالة غير صالحة؟
- الملكية والعمر: ما الذي يملكه الكائن؟ وما الذي يستعيره؟ وما الذي يشير إليه فقط؟
- التبعيات: ما أقل عدد ممكن من التبعيات؟ وهل يمكن عكس الاعتماد بدلاً من تثبيته؟

- الواجهة: ما أقل واجهة تُنجز الهدف دون كشف الداخل؟ ما الذي يجب ألا يكون عاماً؟

أ-2: عند تصميم الواجهة API

قائمة تحقق مركّزة على جودة الواجهة:

- التعبير عن النية: هل أسماء الدوال تعكس النية لا الآلية؟ هل الواجهة تمنع الاستخدام الخاطئ؟
- الثبات: هل الواجهة قابلة للاستمرار دون تغييرات متكررة؟ هل تتجنب المعاملات الاختيارية المشوشة؟
- الإدخال/الإخراج: هل أنواع المعاملات تعبّر عن القيود (ثبات/ملكية/عدم إمكانية الفراغ)؟
- الاستثناءات/الأخطاء: هل سياسة الأخطاء واضحة ومتماسكة عبر الواجهة؟
- الزمن: هل كلفة الاستدعاء متوقعة؟ هل تمنع الواجهة عمليات مكلفة دون إشعار؟

أ-3: بعد اكتمال التصميم (مراجعة ذاتية)

- هل الكائن صغير بما يكفي لتُفهم فكرته بسرعة؟
- هل يوجد تسريب تفاصيل (أنواع/مؤشرات/حاويات) لا ينبغي أن يراها المستخدم؟
- هل يمكن كتابة اختبارات للكائن دون إعدادات ضخمة؟
- هل كل دالة عامة لها سبب قوي؟ هل هناك دوال يمكن جعلها private أو نقلها خارج الكائن؟
- هل يتحمل الكائن التغيير المتوقع دون كسر واسع؟

أ-4: إشارات تحذير تصميمية (Design Smells)

إذا ظهرت علامتان أو أكثر، فغالبًا هناك خلل حدودي:

- كائن يملك أكثر من سبب للتغيير.
- دوال عامة كثيرة تُستخدم نادرًا (واجهة "موسوعية").
- وجود سلاسل استدعاءات طويلة للوصول إلى معلومة بسيطة.
- كثرة الاعتماد على get/set بدل أفعال مجال واضحة.
- كائن لا يمكن إنشاؤه إلا عبر تهيئة كبيرة غير طبيعية.

ملحق ب: جدول اتخاذ القرار — الوراثة أم التركيب؟

ب-1: قاعدة ذهبية

لا تستخدم الوراثة إلا إذا كنت تحتاج تعدد أشكال ديناميكي حقيقي (runtime polymorphism) وكانت العلاقة is-a صحيحة على مستوى المجال وليست مجرد تشابه تنفيذ.

ب-2: جدول قرار سريع

السؤال	إذا كانت الإجابة نعم	إذا كانت الإجابة لا
هل العلاقة is-a حقيقية في المجال؟	الوراثة ممكنة	التركيب أفضل
هل ستستفيد من الاستبدال الكامل للأب بالابن دائمًا؟	الوراثة ممكنة	التركيب/واجهات أفضل
هل تحتاج الإضافة عبر plugin وقت التشغيل؟	واجهة مجردة + وراثة	قوالب/تركيب غالبًا
هل تغيّر الأب سيكسر الأبناء كثيرًا؟	هذا تحذير قوي	التركيب يقلل الانفجار
هل تريد إعادة استخدام تنفيذ فقط؟	لا تستخدم الوراثة	استخدم تركيب/مساعدات/سياسات

ب-3: حالات تُمنع فيها الوراثة (عمليًا)

- لا توجد حاجة حقيقية لتعدد أشكال ديناميكي.
- "وراثة للتجميع" (الوراثة لتوفير حقول مشتركة فقط).
- هرم عميق (أكثر من مستويين) بلا ضرورة مجال قوية.
- الفئة الأب ليست مصممة للوراثة (لا عقد واضح، لا virtual destructor حيث يلزم).

ب-4: حالات تُفضَّل فيها الوراثة

- لديك عقد سلوك ثابت وواضح (واجهة) وتحتاج عدة تطبيقات وقت التشغيل.

- تريد حقن سلوك عبر polymorphic interface في نظام قابل للتوسعة.
- حدود الاعتماد واضحة (العميل يعتمد على واجهة مجردة لا على تطبيق).

ملحق ج: تعدد الأشكال في **Modern C++** — ساكن أم ديناميكي؟

ج-1: مصفوفة اختيار

المعيار	ديناميكي (**virtual**)	ساكن (قوالب/**concepts**)
التوسعة وقت التشغيل	ممتاز	ضعيف/غير مناسب
الأداء	كلفة استدعاء غير مباشرة	غالبًا أسرع (إزالة استدعاءات)
زمن الترجمة وحجم الثنائيات	أقل غالبًا	قد يزيد (توليد نسخ)
ثبات ABI	أسهل عادةً	أصعب إن كثر التوليد
وضوح القيود	عقد سلوكي نصي	قيود قابلة للتحقق (concepts)

ج-2: قواعد عملية للاختيار

- اختر التعدد الديناميكي عندما: الإضافة تتم وقت التشغيل، أو تُحمّل إضافات، أو تتوقع تطبيقات جديدة دون إعادة بناء كل شيء.
- اختر التعدد الساكن عندما: الأداء حرج، أو تريد التحقق من القيود وقت الترجمة، أو السلوك ثابت ضمن حدود ترجمة واحدة.
- لا تستخدم ديناميكيًا فقط لتجنب القوالب، ولا تستخدم ساكنًا فقط "لأنه أحدث"؛ اختر وفق متطلبات التوسعة والحدود.

ج-3: فخ شائع

عندما تُستخدم الوراثة فقط للاستبدال if/else داخل كائن واحد، غالبًا المشكلة ليست في تعدد الأشكال بل في حدود المسؤولية أو في تصميم البيانات.

ملحق د: إدارة العمر والملكية في تصميم OOP

د-1: مفردات الملكية (يجب أن تظهر في التصميم)

عرّف كل علاقة على أنها واحدة من التالي، ولا تتركها "مبهمة":

- يملك (**owns**): الكائن مسؤول عن إنشاء المورد وتدميره.
- يستعير (**borrows**): يستخدم مورداً يملكه غيره دون تمديد عمره.
- يراقب (**observes**): يشير إلى كائن قد يختفي ويحتاج تحقّقًا قبل الاستخدام.
- يشترك (**shares**): عمر المورد مرتبط بعدد المالكين (حالة تتطلب انضباطًا).

د-2: قواعد قرار للمؤشرات الذكية

- unique_ptr: خيار افتراضي للملكية المنفردة. إذا لم يوجد سبب قوي لغيره، استخدمه.
- shared_ptr: لا يستخدم إلا عند وجود مشاركة عمر حقيقية لا يمكن تمثيلها بوضوح عبر ملكية منفردة.
- weak_ptr: يُستخدم لكسر الدورات المرجعية أو لتمثيل "مراقبة" دون تمديد العمر.
- المؤشرات الخام (raw pointers): مناسبة للتمثيل غير المالك (non-owning) عند وضوح العمر، ويُفضّل دعمها بعقود واجهة واضحة.

د-3: أخطاء ملكية تقتل التصميم لاحقًا

- تمرير shared_ptr عبر كل طبقات النظام "للأمان" (ينتهي بتشابك عمر غير مفهوم).
- إخفاء الملكية داخل get() وإرجاع مؤشر خام دون توضيح عمره.
- دورات shared_ptr بين كائنين متعاونين دون weak_ptr.
- تدمير موارد في أماكن غير متوقعة (غياب RAII الفعلي).

د-4: قاعدة **RAII** كقاعدة تصميم

أي مورد (ملف/قفل/مقبس/ذاكرة/معاملة) يجب أن يكون عمره ممسوكًا بكائن نطاقي: إن لم تستطع تحديد "من يغلق/يفك/ينهي" بشكل قطعي، فالتصميم غير مكتمل.

ملحق هـ: واجهات **OOP** الجيدة مقابل الواجهات الخطرة

هـ-1: صفات الواجهة الجيدة

- تمنع الخطأ بدل أن تعتمد على "تعليمات الاستخدام".
- تعبّر عن القيود بالأنواع لا بالتعليقات.
- تُظهر السياسة: هل الدالة تُلقي استثناء؟ هل تُرجع نتيجة/خطأ؟
- تقلل حالات الاستخدام غير المقصودة (أقل طرق، أوضح أسماء).
- تقرأ مثل عقد: ما الذي تتطلبه؟ ما الذي تضمنه؟

هـ-2: علامات واجهة خطرة

- دوال عامة تُقبل "أي شيء" ثم تفشل لاحقًا (غياب التحقق البنيوي).
- واجهة تعتمد على ترتيب استدعاءات معيّن غير مُعلن (حالة داخلية حساسة).
- كثرة setX() التي تسمح بحالات غير صالحة بين الاستدعاءات.
- إرجاع مراجع/مؤشرات إلى بيانات داخلية بلا توضيح لعمرها أو ثباتها.

هـ-3: قاعدة عملية

إذا كان الاستخدام الصحيح يتطلب قراءة الكود الداخلي أو معرفة "عرف الفريق"، فالواجهة ليست واجهة بل تسريب تنفيذ.

ملحق و: أنماط تصميم صالحة في Modern C++ وكيف تُستعمل بوعي

و-1: أنماط غالبًا ما تزال مفيدة (مع ضبط)

- **Factory**: عندما تريد فصل الإنشاء عن الاستخدام، خصوصًا مع تعدد تطبيقات وقت التشغيل.
- **Strategy**: عندما تتغير الخوارزمية دون تغيير العميل، ويمكن تمثيلها بديناميكي أو ساكن حسب السياق.
- **Observer**: مفيد للأحداث، لكن يحتاج ضبطًا لمنع تسريبات التبعية والعمر.
- **Decorator**: مفيد لإضافة سلوك تدريجي، مع الانتباه لتضخم الطبقات.

و-2: أنماط تضعف الحاجة لها في Modern C++

- أنماط تُستخدم لتعويض غياب ميزات لغة قديمة: كثير منها يُستبدل اليوم بـlambdas، RAII، والقوالب.
- القاعدة: إن كان النمط لا يضيف وضوحًا ولا يعالج قيدًا حقيقيًا، فهو عبء.

و-3: اختبار ضرورة النمط

اسأل قبل إدخال أي نمط:

- ما المشكلة المحددة التي يحلها؟
- ماذا يحدث إن أزلناه؟ هل تتدهور القابلية للصيانة فعلاً؟
- هل سيزيد التبعيات والسطح العام للمنظومة؟

ملحق ز: أخطاء OOP القاتلة في C++ التي لا تظهر فورًا

ز-1: أخطاء تصميمية تتفاقم مع الزمن

- هرم وراثة متضخم: ينهار عند أول تعديل في الأب.
- عقود غير مكتوبة: "الكل يعرف كيف يُستخدم" ثم يتغير الفريق وتتضاعف الأخطاء.
- كائنات سِرّية الحالة: سلوكها يعتمد على ترتيب استدعاءات غير واضح.
- ترابط ناعم قاتل: كائنات تبدو مستقلة لكنها تتشارك افتراضات خفية.

ز-2: أخطاء أداء مرتبطة بـOOP

- استخدام تعدد أشكال ديناميكي في مسار حرج دون قياس.
- تجزئة الذاكرة بسبب إنشاء كائنات صغيرة كثيرة على الهيب بلا ضرورة.
- تصميم يعتمد على مؤشرات ونداءات افتراضية بدل بيانات متجاورة عند الحاجة لمعالجة كثيفة.

ز-3: قاعدة مراجعة

أي تصميم لا يوضح "من يملك ماذا" و"ما عقد الواجهة" سيكلف لاحقًا أكثر مما يوفره اليوم.

ملحق ح: نموذج مراجعة تصميم كائني قبل اعتماد الشيفرة (Design Review)

ح-1: أسئلة اعتماد عامة (نعم/لا)

- هل هناك خريطة مسؤوليات واضحة لكل مكوّن؟
- هل الواجهات معزولة عن التفاصيل الداخلية؟
- هل الملكية والعمر موثّقان ضمن الواجهة (بالأنواع أو بالمعنى)؟
- هل فشل إحدى الوحدات لا يجرّ فشلاً متسلسلاً بلا حدود؟
- هل يمكن اختبار كل وحدة بمعزل دون بيئة ضخمة؟

ح-2: مراجعة التبعيات

- هل تعتمد الطبقات العليا على تجريدات لا على تطبيقات؟
- هل هناك تبعيات دائرية؟
- هل يمكن استبدال تطبيق دون تغيير العميل؟

ح-3: مراجعة سياسة الأخطاء

- هل هناك أسلوب موحد للإبلاغ عن الخطأ؟
- هل حالات الفشل المهمة تُعاد إلى العميل بشكل يمكن التعامل معه؟
- هل الاستثناءات تُستخدم حيث تكون مناسبة حقًا أم كحل عام؟

ح-4: قرار نهائي

يُعتمد التصميم فقط إذا كانت الإجابات على الأسئلة الحرجة بنعم، وكانت التبعيات والملكية وسياسة الأخطاء واضحة ومتماسكة.

ملحق ط: متى لا يكون OOP هو الحل الأفضل؟ (اختياري)

ط-1: مؤشرات عملية لتفضيل أساليب أخرى

- لديك معالجة كثيفة على كميات كبيرة من البيانات حيث التجاور في الذاكرة حرج.
- الغالب هو التحويل/التجميع/التصفية على بيانات أكثر من كونه "سلوك كائنات".
- تعدد أشكال كائني يضيف كلفة دون فائدة توسعة حقيقية.

ط-2: مزج ناضج بدل صراع

- استخدم OOP لتعريف الحدود، السياسات، العقود، وإدارة الموارد.
- استخدم أساليب موجهة للبيانات في المسارات الحسابية الحرجة حيث يلزم.
- الهدف ليس "أسلوب واحد"، بل نظام واضح، قابل للصيانة، ويمكن قياس أدائه.

ملاحظة ختامية:
القيمة الحقيقية لهذه الملاحق أنها تُحوِّل مبادئ OOP في Modern C++ إلى أدوات قرار ومراجعة. عند استخدامك لها باستمرار أثناء التصميم والمراجعة، ستصبح جودة بنية النظام نتيجة طبيعية لا جهدًا استثنائيًا.

المراجع

تجمع هذه المراجع المصادر الأساسية والمعاصرة التي استند إليها هذا الكتاب في عرض مفاهيم البرمجة كائنية التوجه (Object-Oriented Programming) كما تُمارَس في Modern C++ (C++20/23). وقد جرى اختيارها بعناية لتوازن بين الأسس النظرية، والتطبيق العملي، والتوجّه الهندسي الحديث للغة.

المراجع الأساسية في C++ وOOP

الكتب

١. Meyers Scott — **"Effective Modern C++"**

- مرجع أساسي لمفاهيم C++11–C++20، يركّز على أفضل الممارسات، المؤشرات الذكية، تعبيرات lambda، والتوازي.

٢. Stroustrup Bjarne — **"The C++ Programming Language, 4th Edition"**

- المرجع الرسمي الشامل للغة، يغطي Modern C++ حتى C++17 مع إرشادات ما تزال صالحة ومؤثرة في سياق C++23.

٣. Nesteruk Dmitri — **"Design Patterns in Modern C++"** (2022)

- تطبيق عملي لأنماط التصميم الكلاسيكية باستخدام ميزات Modern C++ مثل RAII، المؤشرات الذكية، lambdas، وconcepts.

٤. **"Clean C++: Sustainable Software Development Patterns and Best Practices"** — Roth Stephan (2021)

- معالجة حديثة لمبادئ الشيفرة النظيفة في C++17–C++23 مع تركيز واضح على OOP والبرمجة العامة.

٥. **"C++ Primer, 6th Edition"** — Moo E. Barbara Lajoie, Jos e Lippman, B. Stanley

- مقدمة شاملة ومحدّثة تغطي القوالب، OOP، وميزات C++17/20 بأسلوب تدريجي ومنهجي.

المراجع المتقدمة وميزات Modern C++

١. **"C++20 for Programmers"** — Deitel Harvey Deitel, Paul (2022)

- تغطية شاملة لمفاهيم C++20 بما في ذلك concepts، ranges، coroutines، modules، وميزات المكتبة القياسية الحديثة.

٢. **"C++ Templates: The Complete Guide, 2nd Edition"** — David Josuttis, Nicolai Gregor Douglas Vandevoorde, (2021)

- المرجع الأعمق لفهم القوالب، البرمجة القالبية المتقدمة، ودمج concepts في تصميم Modern C++.

٣. **"Concurrency in Action, 2nd Edition"** — Williams Anthony (2020)

- مرجع عملي للتوازي، الذرّيات، المزامنة، وميزات التوازي الحديثة في C++.

٤. **"Functional Programming in Modern C++"** — uki Ivan (2021)

- يوضّح كيف تتكامل الأساليب الوظيفية مع OOP باستخدام lambdas، ranges، وconstexpr.

البرمجة كائنية التوجه وتصميم البرمجيات

١. **"Object-Oriented Design Heuristics"** — Riel J. Arthur (إصدار محدّث 2020)

- قواعد تصميم كائني عملية ما تزال صالحة عند مواءمتها مع ممارسات Modern C++.

٢. **"Agile Principles, Patterns, and Practices in C++"** — Micah Martin, C. Robert Martin (إصدار محدّث 2022)

- عرض تطبيقي لمبادئ OOP والأنماط الرشيقة في سياق C++ الحديث.

٣. **"Modern C++ Design Patterns Cookbook"** — Sarcar Vaskaran (2021)

- أمثلة تطبيقية لأنماط التصميم باستخدام ميزات Modern C++.

إدارة الذاكرة، الأداء، والتحسين

١. **"C++ High Performance, 2nd Edition"** — Sehr Viktor Andrist, Bj rn (2021)

- يناقش تحسين الذاكرة، هياكل البيانات الملائمة للذاكرة المخبئية، وممارسات الأداء في Modern C++.

٢. **"Pro C++17: With Modern C++ Techniques"** — C. Suresh Solter, A. Nicolas Sivathanu (2020)

- تركيز على كتابة شيفرة فعالة من حيث الأداء باستخدام تقنيات C++17–C++20.

المراجع الإلكترونية والتوثيق الرسمي

١. **cppreference.com**

- المرجع الأشمل لتوثيق معايير C++20/23، المكتبة القياسية، والأمثلة التوضيحية.

٢. **isocpp.org**

- الموقع الرسمي لمعايير C++، ويحتوي على مقالات، تحديثات، وروابط مقترحات WG21.

٣. **C++23 Draft Standard and WG21 Papers**

- المسودات الرسمية ومقترحات لجنة ISO C++ التي توضّح التوجهات المستقبلية للغة.

المجتمع التقني ومنصات التعلّم

١. Turner Jason — **C++ Weekly**

- مقاطع قصيرة تركز على أفضل الممارسات، الأداء، وميزات C++23.

٢. **CppCon Talks**

- محاضرات متقدمة تغطي التصميم، التوازي، الأداء، وتطورات Modern C++.

٣. **(Pluralsight, Udemy, Coursera) Modern C++ Courses**

- دورات حديثة تغطي ميزات C++20/23 وتطبيقاتها العملية.

الأدوات والمكتبات الداعمة

١. **Boost C++ Libraries**

- مكتبات عالية الجودة، مراجَعة نظيريًا، تدعم تطوير أنظمة Modern C++.

٢. **Google Test** و **Google Mock**

- أطر اختبار وحدات حديثة متوافقة مع C++20/23.

٣. **Valgrind, AddressSanitizer, ThreadSanitizer**

- أدوات أساسية لاكتشاف أخطاء الذاكرة والتوازي في مشاريع C++.

٤. **Clang-Tidy, CppCheck, LLVM Analyzer**

- أدوات تحليل ساكن لتحسين جودة الشيفرة والالتزام بأفضل الممارسات.

كيفية استخدام هذه المراجع

- الدعم العلمي: تُستخدم هذه المصادر لتعزيز الموثوقية والدقة في الشرح والتحليل.
- القراءة المتعمقة: يُنصح بالرجوع إلى المراجع المتخصصة عند التوسع في موضوع بعينه.
- المرجعية المستمرة: صُمّمت هذه القائمة لتكون مرجعًا يُعاد استخدامه خلال الممارسة المهنية، لا لمرة واحدة فقط.

إن اعتماد هذه المراجع يضمن بقاء هذا الكتاب متوافقًا مع أحدث توجهات Modern C++، ويضعه ضمن الإطار الأكاديمي والمهني الصحيح لتعليم وتطبيق OOP على مستوى احترافي.