Наследяване

Видове наследяване. Параметри на функции (указатели и референции). Конструктори и деструктори при наследяване. Копиране и move семантики при наследяване. Примери.

д-р Тодор Цонков
todort@uni-sofia.bg

Какво е наследяването?

Наследяването (inheritance) в C++ е механизъм, чрез който един клас (наречен derived/производен) може да придобие свойства и поведение на друг клас (наречен base/базов).

Това позволява:
1)Реализиране на концепцията "is-a" (например: Student е Person).

2)Използване на полиморфизъм по време на изпълнение на програмата

— различни класове могат да бъдат третирани като базов клас, ако имат виртуални функции.

Повторна употреба на код и по-добра организация на проекти.

Пример:

```
struct Base {
    int x;
    void show() { std::print("Base x =
{}\n", x); }
};
struct Derived : Base {
    int y;
};
```

Ползи от наследяването

```
struct Animal {
    void eat() { std::print("Eating...\n"); }
};

struct Dog : Animal {
    void bark() { std::print("Barking...\n");
}
};

int main() {
    Dog d;
    d.eat();   // наследен метод от
Animal
    d.bark();  // собствен метод
}
```

Наследяването дефинира логическа връзка: “Производният клас е вид от базовия клас”.

Пример: Dog is-an Animal, Cat is-an Animal.

Обобщение:
Тази концепция е основата за полиморфизъм и безопасно използване на обекти от различни производни класове чрез базов интерфейс.

ключова дума protected

protected е спецификатор за достъп (access specifier) в C++, който определя видимостта на членове (данни и функции) в клас.

Той означава:

Членът е достъпен:

✔ вътре в самия клас
✔ в производните (наследяващи) класове
✘ НЕ е достъпен от външен код

```
class Person {
protected:
    int age;
public:
    Person(int a) : age(a) {}
};

class Student : public Person {
public:
    Student(int a) : Person(a) {}
    void print_age() const {
        std::print("Age: {}\n", age); // OK – protected е достъпен в наследника
    }
};
```

protected - продължение

protected моделира следната идея:
„Това е вътрешна част от абстракцията, но позволявам на наследниците да я използват.“
Тоест: Не искаме външният свят да пипа вътрешното състояние
Но искаме разширяване чрез наследяване

Подходящо когато:
1)Проектираме базов клас за наследяване
2)Искаме да дадем „инструменти“ на наследниците
3)Имаме частична имплементация

Модерният дизайн често предпочита:
private + protected/public функции или композиция вместо наследяване

```
class Base {
private:
   int value;          // state – private
protected:
   void set_value(int v);  // controlled
access
};
```

Видове наследяване

Public inheritance — най-често; „is-a"; публичните членове на базата остават публични.

Protected / Private inheritance — контрол на видимостта: публичното/защитеното на базата става защитено/частно.

Multiple inheritance — наследяване от няколко базови класа.
Virtual inheritance — използва се при ромбовидни (diamond) графи, за да се избегне дублиране на обща база.

```
class B { public: int x; };

class D_public : public B { /* x остава public */ };

class D_prot   : protected B { /* x става protected */ };

class D_priv   : private B { /* x става private */ };
```

Когато се прави API за потребители се използва public inheritance. Protected/private се използват рядко (за реализация, не за интерфейс).

Наследяване - продължение

При наследяване може да се добавя всякаква функционалност в класа, който наследява базовия. В C++ не е възможно да се премахне или ограничи функционалност от базов клас, освен чрез промяна на изходния код. Въпреки това, в производен клас е възможно да се скрие функционалност, която съществува в базовия клас, така че тя да не може да се достъпи чрез производния клас. Това може да се направи просто чрез промяна на съответния спецификатор за достъп.

```
class Base
{
public:
    int m_value{};
};
class Derived : public Base
{
private:
    using Base::m_value;
public:
    Derived(int value) : Base { value }
    {
    }
};
```

Какво е object slicing?

Имаме следните възможност за функции:

```
void f_by_value(Person p);
void f_by_ref(Person& p);
void f_by_const_ref(const Person& p);
void f_by_ptr(Person* p);
void f_by_rvalue_ref(Person&& p);
```

Object slicing възниква, когато обект от наследен (derived) клас се копира или подава по стойност като обект от базов (base) клас.

В резултат се „отрязва“ частта на наследения клас — губят се данните и поведението, дефинирани в него.

Кога се случва най-често?

Подаване на параметри по стойност (Base b)
Връщане на обект по стойност като базов тип
Съхранение в контейнери от базов тип по стойност (std::vector<Base>)

Пример за object slicing

```
struct Base {
    int x = 1;
};
struct Derived : Base {
    int y = 2;
};
void print(Base b) {   // ❌ по стойност
→ slicing
    std::cout << b.x << "\n";
}
int main() {
    Derived d;
    print(d);         // y се губи
}
```

```
std::vector<Base> v;
v.push_back(Derived{});  // slicing
```

Решение:

```
std::vector<std::unique_ptr<Base>> v;
v.push_back(std::make_unique<Derive
d>());
```

Никога да не се приема базов клас по стойност

Да се ползва:

Base&

Base*

std::unique_ptr<Base>

std::shared_ptr<Base>

Пример за object slicing

Какво реално се случва slicing?
При: Base b = d;
се извиква copy constructor на Base:
Base(const Base& other);

Ключов момент
Copy constructor-ът на Base приема const Base&
Няма достъп до Derived частта и копираме само Base под-обекта

Как да забраним slicing?

```
class Base {
public:
    Base() = default;
    Base(const Base&) = delete;   // забранява копиране
    Base& operator=(const Base&) = delete;
};
```

Ред на конструкторите при наследяване

В C++ конструкторите при наследяване се извикват от базовия клас към производния.

Общият ред е:
Конструкторите на базовите класове
Конструкторите на член-данните (member variables) – в реда, в който са декларирани в класа
Конструкторът на самия производен клас

Важно: Редът на член-данните е този, в който са декларирани в класа, а не в initializer list-а.

```
struct Member {
    Member() {
        std::cout << "Member ctor\n";
    }
};
struct Base {
    Base() {
        std::cout << "Base ctor\n";
    }
};
struct Derived : Base {
    Member m;   // член-данна
    Derived() {
        std::cout << "Derived ctor\n";
    }
}; //Base ctor, Member ctor, Derived ctor
```

Копиране при наследяване

Когато се копира обект от производен клас, C++:
първо копира базовата част след това копира членовете на производния клас
Това гарантира коректна инициализация на обекта.

При извикване на copy constructor на Derived:
автоматично се извиква:

Base(const Base&) освен ако изрично не е указано друго

```
1)Derived d2 = d1;

2)Derived::Derived(const Derived& other)
   : Base(other),    // задължително първо
     member1(other.member1),
     member2(other.member2)
{}
```

1) и 2) са еквивалентни

```
Derived d;
Base b = d;   // object slicing
```

Копира се само Base частта

Производната информация се губи
Copy ctor на Derived не се извиква

Move семантики при наследяване

Move конструкторът следва същия ред като copy:
Base(Base&&)
Derived(Derived&&)

Но вместо копиране ресурсите се прехвърлят, а оригиналният обект остава във валидно, но неопределено състояние

Move ctor на Derived става deleted, ако:
базовият клас няма move ctor
или някой член не е movable
или copy ctor е дефиниран, но move не е

1) и 2) са еквивалентни

1)Derived d2 = std::move(d1);

```
2)Derived::Derived(Derived&& other)
    : Base(std::move(other)),
      member1(std::move(other.member1))
{}
```

Copy: Създава независим обект, По-бавен, Винаги безопасен
Move: Прехвърля ресурси, По-бърз, Изисква внимателен дизайн.

Open/Closed принцип от SOLID

Open/Closed Principle гласи:

“Software entities (classes, modules, functions) should be open for extension, but closed for modification.”

С други думи:

Open for extension: Можем да добавяме ново поведение или функционалност.

Closed for modification: Не трябва да променяме съществуващия код, който вече работи, защото това може да счупи нещо.

```
#include <string>
#include <print>

// Базов клас за извличане на съобщения
class MessagePrinter {
public:
    void print_message() const {
        std::print("Base message\n");
    }
};
```

Нека добавим нова функционалност чрез FancyMessagePrinter.

Open/Closed продължение

```
// Имаме нужда от нов тип съобщение
class FancyMessagePrinter : public
MessagePrinter {
public:
  void print_message() const {
     std::println("Fancy message"); //
разширение на функционалността
  }
};
int main() {
  MessagePrinter base;
  base.print_message();  // Base message
  FancyMessagePrinter fancy;
  fancy.print_message(); // Fancy message
}
```

Closed for modification: Не сме променяли MessagePrinter – старият код работи както преди.

```
class AdvancedPrinter : public
MessagePrinter {
public:
void print_fancy_uppercase() const {
          std::print("FANCY UPPERCASE
     MESSAGE\n");
}
};
```

Добавяме нова функционалност без да пипаме класа.

Какво е std::any?

// std::any е контейнер, който може да държи стойност от произволен тип, без да знаеш типа при компилация.
= „кутия за каквото и да е“ + runtime проверка на типа
std::any a -> може да съдържа:int, std::string
,MyWeirdType, буквално всичко, което е copyable (C++17)
Да се използва за колекции от произволни типове, които не са известни по време на компилация.
Ако типовете са известни - std::variant.

```
std::any a = 42;
int x = std::any_cast<int>(a);  // OK
std::string s = std::any_cast<std::string>(a);
// throws std::bad_any_cast

if (auto p = std::any_cast<int>(&a)) {
   // p е int*
}
```

Явява се като съвременен вариант на типа void* от езика C като гарантира type safety, ownership и да се ползва само когато НЕ знаем нищо за типа данни.
В Java и C# всички класове наследяват класа Object.

Сравнение със C# и Java

C++: единично и множествено наследяване на класове.
Java, C#: само единично наследяване на класове, множествено чрез интерфейси.

C++: базовите конструктори се извикват автоматично
Java/C#: базовите конструктори се извикват автоматично чрез super()/base()

C++: нова декларация със същото име в производния клас скрива базовата функция.
Java: методите се override-ват, полетата се shadow-ват.

```
class MessagePrinter
{
public void PrintMessage()
{Console.WriteLine("Base message");}
}

// Наследяване и разширение без virtual
class FancyMessagePrinter : MessagePrinter
{
// Същото име на метод, скрива базовия
public new void PrintMessage()
{Console.WriteLine("Fancy message");}
}
```

Пример с класа Student

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

// ================================
// 1. Базов клас: Person
// ================================
class Person {
protected:
    std::string name;
    int age;

public:
    // Конструктор
    Person(const std::string& n, int a) : name(n), age(a) {
        std::cout << "Person constructor: " << name << "\n";
    }

    // Деструктор
    ~Person() {
        std::cout << "Person destructor: " << name << "\n";
    }

    void show() const {
        std::cout << "Name: " << name << ", Age: " << age << "\n";
    }
};

// ================================
// 2. Наследяване: Student (public)
// ================================
class Student : public Person {
private:
    int grade;

public:
    Student(const std::string& n, int a, int g)
        : Person(n, a), grade(g) {
        std::cout << "Student constructor: " << name << "\n";
    }

    ~Student() {
        std::cout << "Student destructor: " << name << "\n";
    }

    void showGrade() const {
        std::cout << name << "'s grade: " << grade << "\n";
    }

    // Функции с параметри
    void setGradeByRef(int& g) {
        grade = g;
    }

    void setGradeByPtr(int* g) {
        if (g) grade = *g;
    }
};
```

Пример с класа Student

```cpp
// ================================
// 3. Наследяване: Teacher (protected)
// ================================
class Teacher : protected Person {
private:
    std::string subject;

public:
    Teacher(const std::string& n, int a, const std::string& sub)
        : Person(n, a), subject(sub) {}

    void showTeacher() const {
        std::cout << "Teacher " << name << ", Age " << age
                  << ", Subject: " << subject << "\n";
    }
};

// ================================
// 4. Наследяване: Assistant (private)
// ================================
class Assistant : private Person {
public:
    Assistant(const std::string& n, int a) : Person(n, a) {}

    void showAssistant() const {
        std::cout << "Assistant: " << name << ", Age: " << age << "\n";
    }
};

// ================================
// 5. main
// ================================
int main() {
    Student s1("Alice", 20, 10);
    s1.show();
    s1.showGrade();

    int newGrade = 12;
    s1.setGradeByRef(newGrade);
    s1.showGrade();

    int anotherGrade = 15;
    s1.setGradeByPtr(&anotherGrade);
    s1.showGrade();

    Teacher t1("Bob", 40, "Math");
    t1.showTeacher();

    Assistant a1("Charlie", 25);
    a1.showAssistant();

    return 0;
}
```

Обяснение на примера

Видове наследяване:

Student : public Person → наследява публично, членовете на Person запазват видимостта си.

Teacher : protected Person → protected наследяване; членовете на базовия клас стават protected.

Assistant : private Person → private наследяване; членовете на базовия клас стават private.

Функции с параметри:

setGradeByRef(int& g) → променливата се подава по референция.

setGradeByPtr(int* g) → променливата се подава по указател.

Конструктори и деструктори:

Всяка класа извиква конструктора на базовия клас чрез : Person(...).

Деструкторите се извикват в обратен ред на наследяване.

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 24