Наследяване

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

Видове наследяване. Параметри на функции (указатели и референции). Конструктори и деструктори при наследяване. Копиране при наследяване. Move семантики при наследяване. Пример с човек, студент и преподавател.

Синтаксис

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

Използване на полиморфизъм – различни класове могат да бъдат третирани като базов клас, ако имат виртуални функции.

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

```cpp
struct Person {
    std::string name;
    int age;
    void introduce() const { std::cout << name << ", age
" << age << std::endl; }
};

struct Student : public Person {
    int grade;
    void introduce() const { std::cout << name << ", age
" << age << ", grade " << grade << std::endl; }
};

int main() {
    Person p{"Ivan", 30};
    Student s{"Maria", 20, 12};
    p.introduce(); // Person::introduce
    s.introduce(); // Student::introduce
    Person* ptr = &s;
    ptr->introduce(); // Person::introduce
}
```

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.

Синтаксис

```
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).

Пример с клас Student

```
struct Person {
    std::string name;
    int age;
    virtual ~Person() = default;
    virtual void introduce() const { std::cout << name <<
", age " << age; }
};

struct Student : public Person {
    int grade;
    void introduce() const override {
      Person::introduce();
      std::cout << ", grade " << grade;
    }
};

struct Teacher : public Person {
    std::string subject;
      void introduce() const override {
          Person::introduce();
          std::cout << ", teaches " << subject;
      }
};
```

Object slicing

Преговор

```
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);
```

Дефиниция

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

```
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
}
```

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.

Пример

```
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
```

Какво е 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

Пример

```
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

Пример

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.

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)

```
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*
}
```

Препоръки

Да се използва за колекции от произволни типове, които не са известни по време на компилация.
Ако типовете са известни - std::variant.

Пример:

```
std::unordered_map<std::string, std::any>
config;

config["timeout"] = 500;
config["host"] = std::string("localhost");
config["debug"] = true;
```

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.

Пример с клас Person

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

class Person {
protected:
    std::string name_;

public:
    Person(std::string name)
        : name_(std::move(name)) {
        std::cout << "Person ctor\n";
    }

    Person(const Person& other)
        : name_(other.name_) {
        std::cout << "Person copy ctor\n";
    }

    Person(Person&& other) noexcept
        : name_(std::move(other.name_)) {
        std::cout << "Person move ctor\n";
    }

    ~Person() {
        std::cout << "Person dtor\n";
    }
};
```

Пример с клас Student

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

class Student : public Person {
    std::vector<int> grades_;

public:
    Student(std::string name, std::vector<int> grades)
        : Person(std::move(name)),
          grades_(std::move(grades)) {
        std::cout << "Student ctor\n";
    }

    Student(const Student& other)
        : Person(other),                // ! base ce
копира
          grades_(other.grades_) {
        std::cout << "Student copy ctor\n";
    }

    Student(Student&& other) noexcept
        : Person(std::move(other)),     // ! base ce
move-ва
          grades_(std::move(other.grades_)) {
        std::cout << "Student move ctor\n";
    }

    ~Student() {
        std::cout << "Student dtor\n";
    }
};
```

Пример с клас Teacher

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

class Teacher : private Person {
    std::string subject_;

public:
    Teacher(std::string name, std::string subject)
        : Person(std::move(name)),
          subject_(std::move(subject)) {
        std::cout << "Teacher ctor\n";
    }

    using Person::name_; // selective exposure
};


//Student is-a Person → public
//Teacher uses Person internally → private
```

Пример на Java

```java
public class Teacher {
    private Person person;  // композиция вместо private наследяване
    private String subject;

    public Teacher(String name, String subject) {
        this.person = new Person(name);
        this.subject = subject;
        System.out.println("Teacher ctor");
    }

    // getter за името, ако е нужно
    public String getName() {
        return person.getName();
    }

    public String getSubject() {
        return subject;
    }

    // setter-и, ако е необходимо
    public void setSubject(String subject) {
        this.subject = subject;
    }
}

В Java няма private inheritance, затова ползваме композиция (private Person person).

Copy / move семантика в Java се реализира с copy constructor или клониране (clone()), защото Java няма move семантика като C++.
```

Въпроси?

Благодаря за вниманието!