Lecture 10

Inheritance, polymorphism, introspection

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
October 2nd, 2023

IDA, Linköping University

10.1

Content

Contents

10.2

1 Derived Classes

1.1 Introduction

Derived Classes

C++ has a relatively complex model for derivation/inheritance

- it allows a subclass to inherit from multiple base classes
  - *simple* inheritance — a single base class
  - *multiple* inheritance — two or more direct base classes
  - *repeated* inheritance — an indirect base class is inherited multiple times via multiple inheritance
  - multiple and repeated inheritance can lead to ambiguitiess 10.3

Availability of base class members

- Availability of members of a base class depends on how the inheritance was specified
  - **public** base class — **public**-members in base class become **public** in the inherited class (**protected** become **protected**)
  - **protected** base class — **public**-members in base class become **protected** in the derived class (**protected** become **protected**)
  - **private** base class — **public**-members in base class become **private** in the derived class (**protected** become **private**)
  - a class can appoints *friends* — a **friend** can access all members, even private 10.4

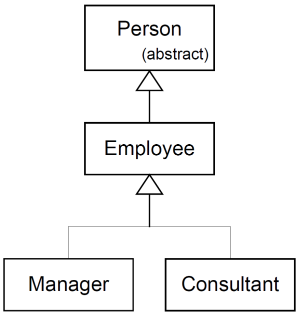
Polymorphic Behaviour

- A polymorphic behaviour refers to the ability for a type/variable to take multiple forms (ie Rectangle and Circle can be treated as a Shape).
- Polymorphic behaviour is decided by the programmer
  - object which should have a polymorphic behaviour should be referenced via pointers or references
  - only functions declared *virtual* can be dynamic and exhibit a polymorphic behaviour 10.5

1.2 A polymorphic class hierarchy

Person-Employee-Manager-Consultant — a polymorphic class hierarchy

- **Person** — a general representation of a person
  - has a name and person number
  - all employees common properties
  - no object of type **Person** should be instantiated it will be an *abstract* class
- **Employee** — employees in general
  - the date of recruitment, employee id, salary, section
  - more specialised categories of employees are derived from this class
  - those objects should be instantiable
- **Manager** — department heads
  - responsible for a department and its employees
- **Consultant** — temporary employees
  - no difference compared to a permanent employee except a distinction by type



10.6

Person

```
class Person
{
public:
    virtual ~Person() = default;

    virtual std::string str() const;
    virtual Person* clone() const = 0;

    std::string get_name() const;
    void set_name(const std::string&);
    CRN get_crn() const;
    void set_crn(const CRN&);

protected:
    Person(const std::string& name, const CRN& crn);
    Person(const Person&) = default;

private:
    Person& operator=(const Person&) = delete;

    std::string name_;
    CRN         crn_;
};
```

10.7

Constructor that takes name and person number

```
Person::Person(const std::string& name, const CRN& crn)
    : name_( name ), crn_( crn )
{}
```

- Ensures that a new person always has a name and person number
  - default constructor is not generated
  - only derived class can call the constructor — **protected**

10.8

Member functions `str()`

```
virtual std::string str() const;
```

Definition:

```
string Person::str() const
{
    return name_ + ' ' + crn_.str();
}
```

Dynamic call if *virtual* method and object refered by pointer or reference:

- Dynamic type determines which virtual function is called

  ```
  Person* p{ new Manager{name, crn, date, employment_number, salary, dept} };
  ```

  ```
  cout << p->str() << endl;
  ```

  - pointers `p` has *static type* `Person*`
  - expression `*p` has *dynamic type* `Manager`

    ```
    (*p).str()
    ```

    * `Manager::str()` is called — prefer to use the arrow operator

      ```
      p->str()
      ```

10.9

Member function `clone()`

```
virtual Person* clone() const = 0;
```

Polymorphic class sometimes needs a polymorphic copy function:

- to use polymorphic classes it often means you will need to allocate object dynamically and handle them via pointers
  - requires a polymorphic copy function `clone()`
  - each subclass needs to provide its own implementation of `clone()`

10.10

Subclass Employee

```
class Employee : public Person
{
public:
    Employee(const std::string& name,
             const CRN&         crn,
             const Date&        e_date,
             int                e_number,
             double             salary,
             int                dept = 0);

    ~Employee() = default;

    std::string str() const override;
    Employee* clone() const override; //return type covariant med Person*

    int get_department() const;
    Date get_employment_date() const;
    int get_employment_number() const;
    double get_salary() const;

protected:
    Employee(const Employee&) = default;
```

10.11

Subclass Employee (continue)

```
private:
    Employee& operator=(const Employee&) = delete;

    friend class Manager;
    void set_department(int dept);
    void set_salary(double salary);

    Date           e_date_;
    int            e_number_;
    double         salary_;
    int            dept_;
};
```

10.12

Public constructor for Employee

```
Employee::Employee(const string& name,
                   const CRN&    crn,
                   const Date&   e_date,
                   int           e_nbr,
                   double        salary,
                   int           dept)
    : Person(name, crn), e_date_(e_date), e_number_(e_nbr), salary_(salary),
      dept_(dept)
{}
```

- Person-subobject is initialised first
  - Person-constructor is first in the list of initialiser
  - call the corresponding constructor of person
- Employees own members are initialised in declaration order
  - write initialiser in same order
- a constructor should explicitly initialise all base class and non-static data member 10.13

Member functions `str()` implementation

```
string Employee::str() const
{
    return Person::str() + "␣(Employee)␣" + e_date_.str()
                         + '␣' + std::to_string(dept_);
}
```

- call `str()` for Person-subobject to generate the first part of the string
  - qualified name `Person::str()` is used to avoid recursion 10.14

Member function `clone()` implementation

```
Employee* clone() const
{
    return new Employee{ *this };
}
```

- Return a copy of the object on which `clone()` is called
- copy constructor is the natural operation to make a copy
  - in turn, it will also call the copy constructor for `Person`
- when the return type belongs to a polymorphic class hierarchy, we can adapt the returned type

```
Employee* p1{ new Employee{ name, crn, date, employment_nbr, salary} };

Employee* p2 = p1->clone(); // no need to do a type conversion
                            // clone() return Employee*

Person* p3 = p1->clone();   // implicit conversion o Person* - upcast
```

  - types are said to be *covariant*

10.15

Subclass Manager

```
class Manager : public Employee
{
public:
    Manager(const std::string& name,
            const CRN&         crn,
            const Date&        e_date,
            int                e_number,
            double             salary,
            int                dept);
~Manager() = default;

std::string str() const override;
Manager* clone() const override;

void add_department_member(Employee* ep) const;
void remove_department_member(int e_number) const;
void print_department_list(std::ostream&) const;
void raise_salary(double percent) const;

protected:
    Manager(const Manager&) = default;
private:
    Manager& operator=(const Manager&) = delete;

    // Manager does not own the Employee-object,
    // Manager should not delete Employee-object
    mutable std::map<int, Employee*> dept_members_;
};
```

10.16

Public constructor for Manager

```
Manager(const std::string& name,
        const CRN&         crn,
        const Date&        e_date,
        int                e_number,
        double             salary,
                           int dept)
    : Employee( name, crn, e_date, e_number, salary, dept )
{}
```

- All parameters are passed as argument to the direct base class Employee
- `dept_members` has a default constructor, no need to list it in the initialiser list

10.17

Member function `str()` and `clone()` implementation

```
string Manager::str() const
{
    return Person::str() + "␣(Manager)␣" + get_employment_date().str() + '␣'
                         + std::to_string(get_department());
}

Manager* clone() const
{
    return new Manager{ *this };
}
```

Assume we have forgotten to supply a `clone()` implementation for Manager:

- The *last override* would be `Employee::clone()`
- instead of a Manager `clone()` returns an Employee

10.18

Employees of a department are handled by a manager

```
void Manager::add_department_member(Employee* ep) const
{
    // Division of an employee is the same as the manager
    ep->set_department(get_department()); // need friendship

    // Add to the list of employees
    dept_members_.insert(make_pair(ep->get_employment_number(), ep));
}
```

- Manager must be **friend** of Employee to get access to the **private**-member `set_department` in this context

  - parameter ep is a pointer to Employee
  - only **public**-operation are allowed with ep, if Manager is not a friend of Employee
  - if ep was a pointer to Manager and set_department was **protected** then there would be no need for **friend**
- A member function in Manager
  - can use **private**-member of itself and other Manager pointer
  - can use **protected**-member (including inherited ones) of itself and other Manager pointer
  - can only access **public**-members of object of type Employee and Consultant, unless it is a **friend**

10.19

Subclass Consultant

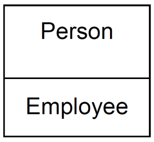
```
class Consultant final : public Employee // No subclass of Consultant allowed
{
public:
    using Employee::Employee;             // inherit constructor

    ~Consultant() = default;

    std::string str() const override;
    Consultant* clone() const override;

protected:
    Consultant(const Consultant&) = default;

private:
    Consultant& operator=(const Consultant&) = delete;
};
```

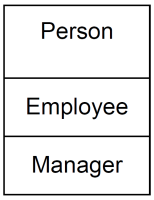
10.20

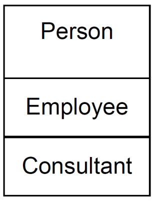
Initialisation and destruction of derived types

- an object of a derived type is composed of sub-items
  - a sub-item corresponds to the base-class
  - in addition to own members
- initialisation is top-down
  - base class is initialised before the sub-class
  - first constructor called is the constructor to the *most derived class*
- destructor order is reversed to the initialisation — bottom-up
  - data members of subclass are destroyed before the members of the base class
  - first called is the most derived class destructor
  - the data members of the class are destroyed in the reverse order of declaration
- it is important that the root class of the hierarchy has a virtual destructor

```
Person* p{ new Consultant(...) };
...
delete p;          // ~Person() or ~Consultant() ?
```







10.21

Use Person, Employee, Manager, Consultant

```
Person* pp= nullptr;      // can point to Employee-, Manager- or
                            // Consultant-object (Person is abstract)
Employee* pe= nullptr;    // can point to Employee-, Manager- or
                            // Consultant-object
Manager* pm= nullptr;     // can only point to a Manager-objekt
Consultant* pc= nullptr; // can only point to a Consultant-objekt

pm = new Manager(name, crn, date, employment_nbr, salary, 17);

pp = pm;                          // upcast is automatic
                                  // Manager* -> Person*

pm = dynamic_cast<Manager*>(pp); // downcast must be explicit
                                  // Person* -> Manager*

if (pm != nullptr)                // is it a Manager?
{
    pm->print_department_list(cout);
}
```

- polymorphic pointers can point to the object of corresponding type and subtype
- *upcast* is an automatic and safe conversion
- *downcast* must be explicit and checked
  - `print_department_list()` is specific to Manager and can only be used with a pointer of type `Manager*`

10.22

1.3 Introspection

Introspection

One way to determine an object type is to use **typeid**-operator — including `<typeinfo>`

```
if (typeid(*pp) == typeid(Manager)) ...
```

- can be used with *type name*, *object* and all kind of *expressions*
- a **typeid**-operator returns an object of type **`type_info`**
- type control can be done by comparing two **`type_info`**-objects

10.23

typeid-operator

typeid-operator:

`typeid``(*p)` returns a **`type_info`**-object for the type of object pointed by `p`
`typeid``(r)` return a **`type_info`**-object for the type of object referenced by `r`
`typeid``(T)` return a **`type_info`**-object for the type `T`
`typeid``(p)` is usually a mistake if `p` is a pointer — it returns **`type_info`**-object for a pointer type

`type_info`-operations:

== test if two **`type_info`**-objects are the same
!= test if two **`type_info`**-objects are different
`name()` return the “name” as a C-string

10.24

Introspection (continue)

Type control can also be done with **`dynamic_cast`**

- use of polymorphic pointers:

```
Manager* pm{ dynamic_cast<Manager*>(pp) };

if (pm != nullptr)
{
    pm->print_department_list(cout);
}
```

  - **`dynamic_cast`** returns `nullptr` if `pp` is *not* pointing to an object of type Manager or a subclass of Manager

- the use of polymorphic reference — `rp` assumed to have type `Person&`

```
dynamic_cast<Manager&>(rp).print_department_list(cout);
```

  - if `rp` is not an object of type Manager or a subtype of Manager, a **`bad_cast`** exception is cast
  - there is no "empty-reference-value" — a reference is always bound to a value

10.25

1.4 Dynamic type conversion

Dynamic type conversion

With the operator **`dynamic_cast`** we can convert a pointer or a reference:

`dynamic_cast``<T*>(p)`	converts pointer `p` to "pointer of `T`"
`dynamic_cast``<T&>(r)`	converts reference `r` to "reference of `T`"

- *downcast* from base class pointer to subclass pointer
- *upcast* is an automatic and safe conversion
- with multiple inheritance it is also possible to "*crosscast*"

10.26

2 Common C++ mistakes and pitfalls (continue)

Inheriting from the same class twice

- When inheriting from the same base class twice, the resulting structure contains the base class twice, and it triggers ambiguity
- A solution is virtual inheritance
- Use multi-inheritance, but avoid common ancestor

10.27

Virtual troubles

- Missing virtual in base class
- Missing virtual destructor
  - If a class has any virtual functions, it should most likely have a virtual destructor

10.28

Copy

```
class A {};
class B : public A {};

B b;
A a = b; // What is the type of a?
```

10.29

Array of polymorphic objects

```
class Employee
{
public:
    void raise_salary(double by_percent);
};
class Manager : public Employee
{
    // ...
};
void make_them_happy(Employee* e, int ne)
{  for (int i = 0; i < ne; i++)
      e[i].raise_salary(0.10);
}
int main()
{
    Employee e[20];
    Manager m[5];
    m[0] = Manager("Joe␣Bush", "Sales");
    // ...
    make_them_happy(e, 20);
    make_them_happy(m + 1, 4); // let's skip Joe
    return 0;
}
```

10.30

Array of polymorphic objects

- It compiles:

```
make_them_happy(m + 1, 4); // let's skip Joe
```

- But:
  - the array increment on `Employee*` and on `Manager*` is different:

```
Employee* em = m;
std::cout << em[1] << "␣" << m[1] << std::endl;
// -> 0x77124800 0x77124802
```

10.31

operator= does not call parent operator

- Constructors and destructors automatically call the base one
- **operator=** does not:

```
Manager& Manager::operator=(const Manager& b)
{
    if (this == &b) return *this;
    Employee::operator=(b);
    _dept = b._dept
    return *this;
}
```

10.32