سلسلة تعلم البرمجة بلغة C++ الحديثة

Learn Modern C++ Programming Course

إعداد المهندس أحمد الديب

#36: Derived Classes P1

Derived Classes

- A concept (idea, notion, etc.) does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with other concepts.
- Implementation inheritance: to save implementation effort by sharing facilities provided by a base class. (run-time polymorphism or dynamic polymorphism)
- Interface inheritance: to allow different derived classes to be used interchangeably through the interface provided by a common base class. (compile-time polymorphism or static polymorphism)

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Derived Classes

```
struct Employee {
  std::string name;
  std::string title;
  // ...
};
```

```
struct Manager {
  Employee employee;
  std::vector<Employee*> list;  // people managed
  int level;
  // ...
};
```

Nothing that tells the compiler and other tools that Manager is also an Employee.

We could either use explicit type conversion on a Manager* or put the address of the employee member onto a list of employees. However, both solutions are inelegant and can be quite obscure.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Derived Classes

```
struct Employee {
  std::string name;
  std::string title;
  // ...
};


struct Manager : public Employee {
  std::vector<Employee*> list;  // people managed
  int level;
  // ...
};


int main() {
  //
  Employee ramy{"Ramy", "SW Engineer"};
  Employee hady{"Hady", "SW Engineer"};
  Manager fady{"Fady", "SW Manager"};


  fady.list.push_back(&ramy);
  fady.list.push_back(&hady);


  std::vector<Employee*> all{&ramy, &hady, &fady};  // ok
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Now, Manager is a subtype of Employee, so that a Manager can be used wherever an Employee is acceptable.

Derived Classes

```
void fun(Manager man, Employee emp) {
  Employee* pemp{&man};                          // ok
  Manager* pman{&emp};                           // error
  Manager* pman{static_cast<Manager*>(pemp)};    // ok
}
```

In general, if a class Derived has a public base class Base, then a Derived* can be assigned to a variable of type Base* without the use of explicit type conversion. The opposite conversion, from Base* to Derived*, must be explicit.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Member Functions

```
class Employee {
 public:
  Employee(std::string name, std::string title) : _name{name},
_title{title} {}
  void print() const;


 private:
  std::string _name;
  std::string _title;
};


class Manager : public Employee {
 public:
  Manager(std::string name, std::string title, int level)
      : Employee(name, title), _level{level} {}
  void add_managed(Employee* managed) { list.push_back(managed); }
  void print() const;


 private:
  std::vector<Employee*> list;  // people managed
  int _level;
};
```

A member of a derived class can use the public – and protected – members of a base class as if they were declared in the derived class itself.

The concept of a private member would be rendered meaningless by allowing a programmer to gain access to the private part of a class simply by deriving a new class from it.

```
void Manager::print() const {
  std::cout << "Manager: " << '\n';
  Employee::print();
  std::cout << "Managing: " << '\n';
  for (auto&& emp : list) {
    emp->print();
  }
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Type Fields

Given a pointer of type Base*, to which derived type does the object pointed to really belong? There are four fundamental solutions:

1. Ensure that only objects of a single type are pointed to.
2. Place a type field in the base class for the functions to inspect.
3. Use dynamic_cast.
4. Use virtual functions.

```
void print(Employee* emp) { emp->print(); }

int main() {
  //
  Employee ramy{"Ramy", "SW Engineer"};
  Employee hady{"Hady", "SW Engineer"};
  Manager fady{"Fady", "SW Manager", 2};

  fady.add_managed(&ramy);
  fady.add_managed(&hady);

  print(&ramy);
  print(&fady);
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Type Fields

```
class Employee {
 public:
  enum EmployeeType { manager, employee };
  EmployeeType type;
  Employee(std::string name, std::string title)
      : type{employee}, _name{name}, _title{title} {}
  void print() const;


 private:
  std::string _name;
  std::string _title;
};


class Manager : public Employee {
 public:
  Manager(std::string name, std::string title, int level)
      : Employee(name, title), _level{level} {
    type = manager;
  }
  void add_managed(Employee* managed) { list.push_back(managed); }
  void print() const;


 private:
  std::vector<Employee*> list;  // people managed
  int _level;
};
```

```
void print(Employee* emp) {
  switch (emp->type) {
    case Employee::employee:
      emp->print();
      break;
    case Employee::manager: {
      const Manager* man = static_cast<const Manager*>(emp);
      man->print();
      break;
    }
    default:
      break;
  }
}
```

This works fine, especially in a small program maintained by a single person. However, it has a fundamental weakness in that it depends on the programmer manipulating types in a way that cannot be checked by the compiler.

Furthermore, any addition of a new kind of Employee involves a change to all the key functions in a system — the ones containing the tests on the type field. In other words, use of a type field is an error-prone technique that leads to maintenance problems.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Thank you