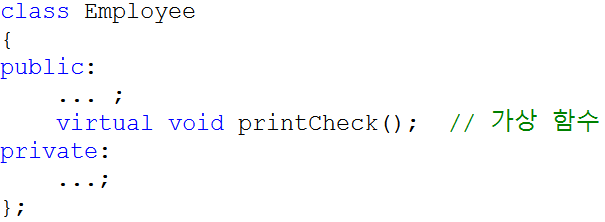
15장 다형성과 가상 함수

15.1 가상 함수의 기초

* 가상 함수(virtual function)

- 가상 함수는 **상속관계**에서 기반 클래스에 선언된 함수를 파생 클래스에서 재정의한 함수이다. (함수 오버라이딩(overriding))

* + - 상위 클래스의 멤버 함수를 무시한다.

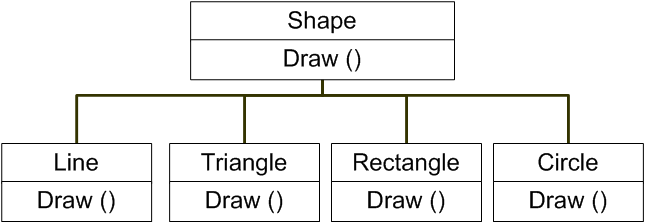


- 가상 함수는 해당 함수가 실행되기전에 구체적으로 정의되지 않기 때문에 가상이라는 이름이 붙는다.

- 가상 함수는 사후 바인딩(late binding), **동적 바인딩(dynamic binding)**, **다형성(polymorphism)**이라고도 한다.

* + - 다형성은 동일한 메시지를 전달하더라도 객체에 따라 서로 다른 동작을 하는 것을 말한다. 하나의 객체에서 다른 것을 실행할 수 있다.
    - 가상 함수는 객체지향 언어의 다형성의 한 종류이다.

- 예시: Shape-Draw()하게 되면, 하위 객체에 맞게 각 Draw()를 실행한다.

[](https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjMkvzY_53UAhWHj5QKHTofAKMQjRwIBw&url=https://stackoverflow.com/questions/8824359/why-use-virtual-functions&psig=AFQjCNEQsYofkthxXgbnzp7i_4Lb3BPhqg&ust=1496452889341459)

* 조건과 실행

- 가상 함수는 컴파일러에게 “나는 함수가 어떻게 구현되는지 모른다(진짜가 아니다). 프로그램이 실행될 때까지 기다려라. 그리고 객체(인스턴스)로부터 구현된 것을 실행하라.”라고 말하는 것이다.

* + - 하나의 코드로 다양한 형태의 객체를 처리한다.

- 구현 및 실행 절차

* + - [조건] 다형성은 상향 형변환(upcasting)의 **객체 포인터**를 통해 이루어진다.
    - [조건] 기반 클래스와 상속 클래스에 해당 함수가 재정의 되어 있어야 한다.
    - 가상 함수가 호출되면, 상속 클래스가 있는지 확인한다. 그리고 상속 클래스가 없으면, 그 함수 자체가 호출되고, 상속 클래스가 있으면, 상속된 클래스의 가상 함수가 호출된다. (함수 호출이 실행 시간까지 연기된다.)

- 가상 함수의 속성은 상속된다.

- 단점: 가상 함수는 더 많은 저장 공간을 사용하고 실행 시간이 느리다.

예제: 가상 함수

ex015\_01.cpp

|  |
| --- |
| /\* 15.1 가상 함수 기초  \* 가상 함수  \*/  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setNetPay(double newNetPay);  virtual void printCheck() ; // 가상 함수  private:  string Name;  string Ssn;  double NetPay;  };  //---Hourly클래스 인터페이스  class HourlyEmployee :public Employee  {  public:  HourlyEmployee();  HourlyEmployee(const string& theName, const string& theSsn, double theRate, double theHours);  double getRate() const;  double getHours() const;  void printCheck(); // 가상 함수 재정의, virtual void printCheck();  private:  double Rate;  double Hours;  };  // SalariedEmployee 클래스 인터페이스  class SalariedEmployee :public Employee  {  public:  SalariedEmployee();  SalariedEmployee(const string& theName, const string& theSsn, double theSalary);  double getSalary() const;  void printCheck(); // 가상 함수 재정의,virtual void printCheck();  private:  double Salary;  };  // main 함수  int main() {    Employee e1("Kim", "0123"), e2;  HourlyEmployee h1("Hong", "0329", 10, 20), h2;  cout << "<h1의 정보 출력>\n";  h1.printCheck();  cout << endl;  SalariedEmployee s1("Lee", "0508", 500), s2;  cout << "<s1의 정보 출력>\n";  s1.printCheck();  cout << endl;  // 동일 형변환  // h2 = h1; // 동일한 타입을 할당시키는 것  HourlyEmployee\* ph = &h2; // 정보의 손실이 없다.  ph->printCheck();  cout << endl;  // 상향 형변환  // e2 = h1; // 기반 클래스 타입으로 할당하는 것  Employee\* pe = &s2; // 정보의 손실이 있다.  pe->printCheck();  cout << endl;  // 가상 함수  pe=&h1;  pe->printCheck();  cout << endl;  return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  // 가상 함수  void Employee::printCheck()  {  cout << "Your check is not determined yet.";  }  //---HourlyEmployee 클래스 구현  HourlyEmployee::HourlyEmployee() : Employee(), Rate(0), Hours(0){  }  HourlyEmployee::HourlyEmployee(const string& theName, const string& theNumber, double theRate, double theHours)  : Employee(theName, theNumber) , Rate(theRate), Hours(theHours)  {  }    double HourlyEmployee::getRate() const  {  return Rate;  }  double HourlyEmployee::getHours() const  {  return Hours;  }  // 가상 함수 정의  void HourlyEmployee::printCheck()  {  setNetPay(Hours\*Rate);  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  cout << "Name: " << getName() << endl;  cout << "Number: " << getSsn() << endl;  cout << "You are a hourly employee." << endl;  cout << "You worked for " << Hours << " with the rate" << endl;  cout << "NetPay: $" << getNetPay() << endl;  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  }  // SalariedEmployee 클래스 구현  SalariedEmployee::SalariedEmployee() :Employee(), Salary(0)  {  }  SalariedEmployee::SalariedEmployee(const string& theName, const string& theSsn, double theSalary):Employee(theName, theSsn), Salary(theSalary)  {    }  double SalariedEmployee::getSalary() const  {  return Salary;  }  void SalariedEmployee::printCheck()  {  setNetPay(Salary);  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  cout << "Name: " << getName() << endl;  cout << "Number: " << getSsn() << endl;  cout << "You are a salaried employee." << endl;  cout << "Your weekly salary is " << getNetPay() << endl;  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  } |

* 순수 가상 함수와 추상 클래스

- 순수 가상 함수: 가상 함수의 정의를 가지지 않는 함수이다.

* + - 함수 선언에 virtual과 '=0'을 추가한다.
    - 어떤 종류의 멤버도 순수 가상 함수로 만들 수 있다.



- 추상 클래스: 하나 이상의 순수 가상 함수를 가지는 클래스이다.

* + - 다른 클래스를 파생하기 위한 기반 클래스로만 사용된다.
    - 추상 클래스의 객체를 생성할 수 없다.
    - 순수 가상 함수의 구현은 파생 클래스에서 한다.
    - 상속을 위한 인터페이스(프레임워크)를 제공한다.
* 예제: 순수 가상 함수

ex015\_02.cpp

|  |
| --- |
| /\* 15.1 가상 함수 기초  \* 순수 가상 함수  추상 클래스\*/  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스 --> 추상 클래스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setNetPay(double newNetPay);  virtual void printCheck() = 0 ; // 순수 가상 함수  private:  string Name;  string Ssn;  double NetPay;  };  //---Hourly클래스 인터페이스  class HourlyEmployee :public Employee  {  public:  HourlyEmployee();  HourlyEmployee(const string& theName, const string& theSsn, double theRate, double theHours);  double getRate() const;  double getHours() const;  void printCheck(); // 가상 함수 재정의  private:  double Rate;  double Hours;  };  // main 함수  int main() {    //Employee e1, e2;  //cout << "<e1의 정보 출력>\n";  //e1.printCheck();  //cout << endl;  HourlyEmployee h1("Hong", "0329", 10, 20), h2;  cout << "<h1의 정보 출력>\n";  h1.printCheck(); cout << endl;  cout << "<h2의 정보출력>\n";  h2.printCheck(); cout << endl;  return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  // 순수 가상 함수는 정의가 없다.  //void Employee::printCheck()  //{  // cout << "Your check is not determined yet.";  //}  //---HourlyEmployee 클래스 구현  HourlyEmployee::HourlyEmployee() : Employee(), Rate(0), Hours(0)  {  }  HourlyEmployee::HourlyEmployee(const string& theName, const string& theNumber, double theRate, double theHours)  : Employee(theName, theNumber) , Rate(theRate), Hours(theHours)  {  }    double HourlyEmployee::getRate() const  {  return Rate;  }  double HourlyEmployee::getHours() const  {  return Hours;  }  // 가상 함수 정의  void HourlyEmployee::printCheck()  {  setNetPay(Hours\*Rate);  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  cout << "Name: " << getName() << endl;  cout << "Number: " << getSsn() << endl;  cout << "You are a hourly employee." << endl;  cout << "You worked for " << Hours << " with the rate" << endl;  cout << "NetPay: $" << getNetPay() << endl;  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  } |

15.2 포인터와 가상 함수

- 파생 클래스의 객체를 기반 클래스의 변수(매개 변수)에 할당할 수 있다(코드 재사용).

upcasting(상향 형변환)

- 파생 클래스에서 기반 기반 클래스로 변환한다.

* + - 파생 클래스의 멤버 변수 값이 없어지는 슬라이싱(slicing) 문제가 발생한다.

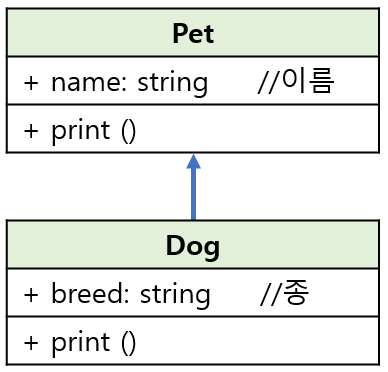
- 슬라이싱 문제 해결 방법

* + - 가상 함수, 포인터, 동적 변수를 사용한다.

downcasting(하향 형변환)

- 기반 클래스에서 파생 클래스로 변환한다.

* + - 아주 위험하며, 잘 못된 결과가 발생할 수도 있다.
    - 가상 함수, 포인터와 dynamic\_cast로 변환은 가능하지만 추천 하지 않는다.
* 예제: upcasting, downcasting

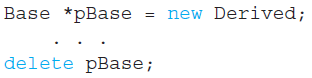


ex015\_03.cpp

|  |
| --- |
| /\*15.2 포인터와 가상 함수  upcasting(확대 변환)  슬라이싱 문제 해결 방법  downcasting(축소 변환)  \*/  #include<iostream>  #include<string>  using namespace std;  //--- Pet과 Dog 클래스 인터페이스  class Pet {  public:  string name;  virtual void print() const;  };  class Dog : public Pet {  public:  string breed;  virtual void print() const;  };  //---Pet과 Dog 클래스 구현  void Pet::print() const {  cout << "name: " << name << endl;  }  void Dog::print() const {  cout << "name: " << name << endl;  cout << "breed: " << breed << endl;  }  int main()  {  //upcasting과 slicing 문제  Pet vpet;  Dog vdog;  vdog.name = "Tiny";  vdog.breed = "Great Dane";  vpet = vdog;  vpet.print(); // Pet 클래스에서 호출하면, 파생클래스의 정보가 손실  // slicing 문제 해결  Pet\* ppet;  Dog\* pdog;  pdog = new Dog;  pdog->name = "Tiny";  pdog->breed = "Great Dane";  ppet = pdog; // 상향 형변환 포인터  ppet->print();  pdog->print();  // cout << ppet->name << endl;  // cout << ppet->breed << endl;  //The following, which accesses member varibales directly  //rather than via virtual functions would produce an error:  //cout << "name: " << ppet->name << " breed: "  //It generates an error message saying  //class Pet has no member named breed.  // downcasting 문제  //Pet\* ppet;  //Dog\* pdog;  //ppet = new Pet;  //ppet->name = "Happy";  //  //pdog = dynamic\_cast<Dog\*>(ppet); // 위험하다.  //pdog->print();  return 0;  } |

가상 함수 소멸자(가상 소멸자)

- 포인터에 의한 상향 형변환(upcasting) 시



* + - 일반적인 소멸자는 메모리의 낭비를 초래한다.

- (해결방번) 소멸자를 가상 함수로 만든다.

* + - 상속을 받고 클래스의 다형성을 이용하는 기반 클래스의 소멸자를 가상 함수로 만든다.

예제: 가상 함수 소멸자

ex015\_04.cpp

|  |
| --- |
| /\*15.2 포인터와 가상 함수  가상 함수 소멸자  \*/  #include<iostream>  using namespace std;  //--- Base와 Derived 클래스 인터페이스  class Base {  public:  Base() { cout << "Base() 생성" << endl; }  virtual ~Base() { cout << "~Base() 소멸" << endl; } // 일반 소멸자--->가상 소멸자  };  class Derived : public Base {  public:  Derived() { cout << "Derived() 생성" << endl; }  ~Derived() { cout << "~Derived() 소멸" << endl; }  };  //---main 함수  int main()  {  // 파생 클래스의 상향 형변환에서  // 기반 클래스의 일반 소멸자 사용  //Base\* pb;  //pb = new Derived;  //delete pb;  // 기반 클래스의 가상 소멸자 사용  Base\* pb;  pb = new Derived;  delete pb;  return 0;  } |