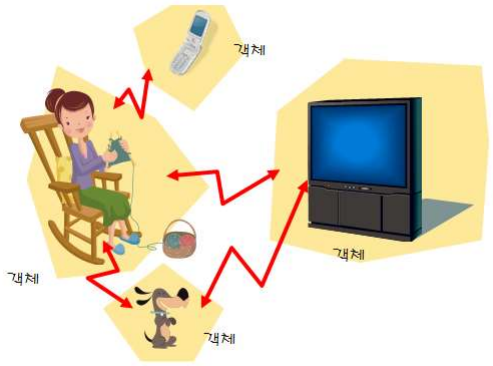
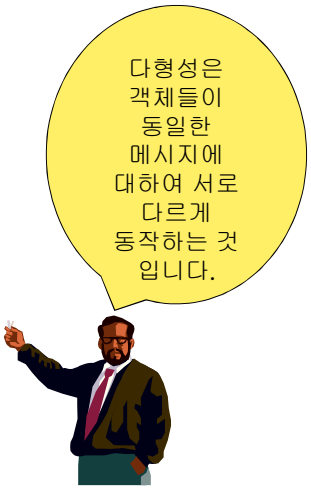
C++ Espresso

제9장 다형성



이번 장에서 학습할 내용

- 다형성
- 가상 함수
- 순수 가상 함수



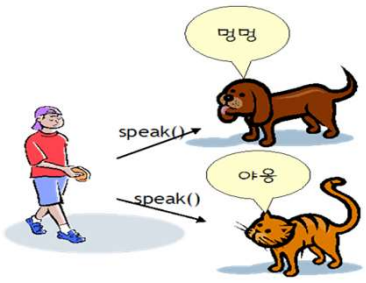
C 에서 형변환

```
int x = 10, y;
float a = 5.5, b;

y = a;              cout << y << endl;
          // warning C4244: 'float'에서 'int'(으)로 변환하면서 데이터가 손실...

y = (int)a;         cout << y << endl;          // OK

b = x;              cout << b << endl;
          // warning C4244: 'int'에서 'float'(으)로 변환하면서 데이터가 손실...

b = (float)x;       cout << b << endl;          // OK
```

- but, 객체에서의 형 변환은 좀 복잡

다형성이란?

- 다형성(polymorphism)이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 서로 다른 동작을 하는 것
  - 똑같은 speak()호출해도 개 객체 speak() 호출하면 멍멍, 고양이 객체 speak() 하면 야옹
- 다형성은 객체 지향 기법에서 하나의 코드로 다양한 타입의 객체를 처리하는 중요한 기술이다.



객체 포인터의 형변환

- 먼저 객체 포인터의 형변환을 살펴보자.

객체 포인터의 형변환

상향 형변환(upcasting):
- 자식 클래스 타입을 부모 클래스타입으로 변환 (자식이 부모로 위장)

하향 형변환(downcasting):
- 부모 클래스 타입(상향 형변환한 자식, 부모로 위장한 자식)을 자식 클래스타입으로 변환

상속과 객체 포인터

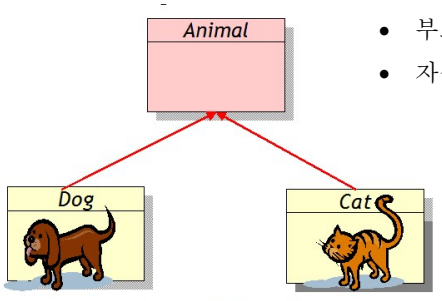


그림 9.2 상속 계층도

- 부모 가리키는 포인터는 자식 가리킬 수 있음.
- 자식 가리키는 포인터는 부모 가리킬 수 없음.

Animal 타입(부모) 포인터로 Dog 객체(자식)를 참조하니 틀린 거 같지만 올바른 문장!!

```
Dog a;
Animal *pa = &a;  // OK
pa = new Dog();   // OK!
```

```
Animal a;
Dog *ps = &a;  // Error
```

```
// C 에서 포인터 사용
int a, *pa;
pa = &a;
pa = new int;
```

도형 예제 - 다음 쪽 위한 class

```
class Shape {
protected:
        int x, y; // 좌상단 좌표

public:
        void setOrigin(int x, int y){
                this->x = x;
                this->y = y;
        }
        void draw() {
                cout <<"Shape Draw";

        }
};
```

```
class Rectangle : public Shape {
private:
        int width, height;
public:
        void setWidth(int w) {
                width = w;
        }
        void setHeight(int h) {
                height = h;
        }
        void draw() {
                cout << "Rectangle Draw";
        }
};
```

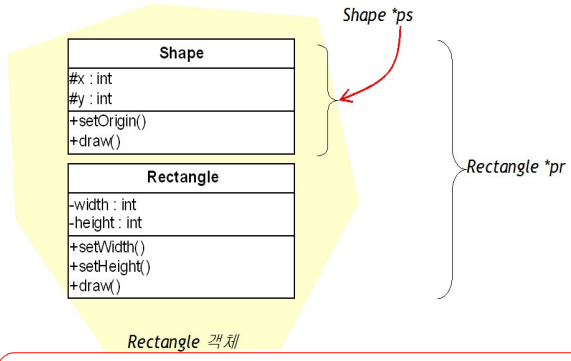
상향 형변환

- Shape *ps = new Rectangle(); // OK!
- ps->setOrigin(10, 10); // OK!
- ps->setWidth(100); // Error

상향 형변환

- 자식을 부모로 형변환 (자식이 부모로 위장)
  - ☞ 부모 클래스(type) 포인터는
  - ☞ 자식 객체를 가리킬 수는 있어도
  - ☞ 부모로 위장했기에
  - ☞ 자식 객체의 멤버 접근 못함
- 부모 type인 ps 이용하여 자식 멤버함수 사용하려면 ?

→ 부모 포인터 ps 를 자식으로 "하향식 형 변환" 해야함.



Rectangle 객체를 Shape 포인터로 가리키면 Shape에 정의된 부분밖에 가리키지 못한다.

- 부모로 위장했으니 부모 역할만 가능

하향 형변환

- Shape *ps = new Rectangle(); // 상향 형변환
  - 여기서 부모 포인터 ps를 통하여 자식 클래스 Rectangle의 멤버에 접근하려면?

→ 부모 클래스 가리키는 포인터 ps 를 자식 클래스 가리키는 포인터로 변환 해야 함. (아래의 2 가지 방법 가능)

1. Rectangle *pr = (Rectangle *) ps; **// 부모를 자식으로 변환**
   pr->setWidth(100); // 자식의 함수

2. ((Rectangle *) ps)->setWidth(100); **// 부모를 자식으로 변환**

하향 형변환

예제
생성자 없어 완성 예제 아님

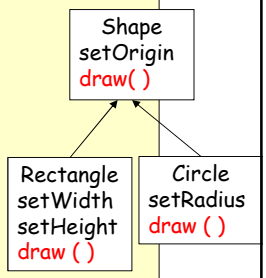
```
#include <iostream>
using namespace std;

// 일반적인 도형을 나타내는 부모클래스
class Shape {
protected:
  int x, y;

public:
  void setOrigin(int x, int y){
            this->x = x;
            this->y = y;
  }
  void draw() {
            cout <<"Shape Draw" << endl;
  }
};
```

```
class Rectangle : public Shape {
private:
  int width, height;
public:
  void setWidth(int w) {
            width = w;
  }
  void setHeight(int h) {
            height = h;
  }
  void draw() {
            cout << "Rectangle Draw"<< endl;
  }
};
```

```
class Circle : public Shape {
private:
  int radius;

public:
  void setRadius(int r) {
            radius = r;
  }
  void draw() {
            cout << "Circle Draw"<< endl;
  }
};
```

예제

```
int main(){
    // 부모 클래스 포인터로 자식 클래스 객체 가리킴
    Shape *ps = new Rectangle();        // OK!

    // ps 는 부모 클래스 포인터, 부모 멤버함수 접근 가능
    ps->setOrigin(10, 10);
    ps->draw();         // Shape 의 draw 호출

    // ps 는 부모 클래스 포인터, 자식 멤버함수 접근 불가능
    // 자식 멤버함수 접근 위해 자식 클래스 포인터로 변경

    ((Rectangle *)ps)->draw();              // Rectangle의 draw 호출
    ((Rectangle *)ps)->setWidth(100);   // Rectangle의 setWidth() 호출
    delete ps;
}
```




```
Shape Draw
Rectangle Draw
계속하려면 아무 키나 누르십시오 . . .
```

부모 자식간의 형변환

- 자식을 부모로 형변환 후 다시 자식으로 형변환 → 자주 사용

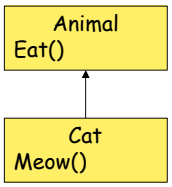
```
Shape* ps = new Rectangle();
Rectangle* p = (Rectangle*)ps;
```

- 단순히 부모를 자식으로 형변환 → error 혹은 사용 안함

```
Rectangle* ps1 = new Shape();       // Error

Shape * ps2 = new Shape();          // 부모 객체 생성
Rectangle* p2 = (Rectangle*)ps2;    // 부모를 자식으로 형 변환
                                    // 문법적 오류는 없으나 실행시 오류 발생
```

C++, C# 다형성 동일한 내용

- C++, C# 모두 자식이 부모로 위장가능
- 자식이 부모로 위장하면 부모 멤버만 사용 가능
- 자식 멤버를 사용하려면 자식으로 형변환 해야 함.
- C# 구문




```
Animal x = new Cat( );
x.Eat( );          // 부모함수 사용 가능
x.Meow( );         // 오류, 부모로 위장하여 자식 함수 Meow() 사용 못함

Cat y = (Cat) x;// 자식으로 형변환
y.Meow( );                    // 자식함수 가능
((Cat) x).Meow( )             // 가능
```

함수의 매개 변수

- 부모 타입의 매개변수는 부모 뿐만 아니라 자식도 받을 수 있다.
- 함수의 매개 변수는 자식 클래스보다는 부모 클래스 타입으로 선언하는 것이 좋다.
  - Circle, Rectangle 를 각각 받는 함수 2개 만드는 것 보다
  - Shape 를 받는 함수 1개만 만들면 자식인 Circle, Rectangle 를 모두 받을 수 있음

```
void move(Shape& s, int sx, int sy)  // 평행 이동하는 일반 함수
{
        s.setOrigin(sx, sy); // Shape(부모) 의 멤버함수만 호출 가능
}                            // 이후에 배울 가상함수 사용하면 자식 멤버 사용 가능
int main() {
        Rectangle r;
        move(r, 0, 0);          // move 함수에서 Shape &s = r; 로 해석

        Circle c;
        move(c, 10, 10);        // move 함수에서 Shape &s = c; 로 해석
        return 0;
}
```

- Shape 의 자식인 모든 도형을 받을 수 있다.

(참고) 멤버함수는 어디에 ?

- 멤버변수는 각 객체에 존재 → 당연, 각 객체마다 멤버 값이 다름
- 멤버함수도 각 객체에 존재 ? -
  - 지금까지 클래스의 멤버함수는 객체생성시 객체 안에 있다고 설명 ← 쉽게 설명하기 위하여…
- but, 실제로는 객체 밖에 존재
  - 멤버함수가 각 객체 안에 있으면 같은 코드의 함수들이 각 객체마다 존재 → 중복된 내용이 다수 존재, 비효율적
  - 멤버함수는 메모리에 하나만 존재하고
  - 각 객체는 멤버함수에 대한 주소를 가지고 있음 → 함수 호출시 해당 주소에 가서 함수 수행
    - cf) 컴퓨터구조 call xxxx(번지)

(참고) this pointer

```
class Car{
    int x;
priblic
    sum(int a, int b){
        x = a+b;
    }
};
```

```
class Car{
    int x;
priblic
    sum(Car *this, int a, int b){
        this->x = a+b;
    }
};
```

- 모든 멤버함수는 1st 매개변수로 호출한 객체의 포인터를 받음
- 모든 멤버함수의 매개변수에서 *this 를 전달하기에 생략하여 작성 → (숨어있음)
- 참고) 파이썬 경우 함수 매개변수로 항상 self(*this 와 동일 개념) 전달

기말 과제

- 과제 2 : 프로그램 분석 / 기능 추가(??) → 10점
  - 중간고사 이후에 부여
  - 분석 : 제공 프로그램 1, 프로그램 2, 프로그램 3, → 1에 추가, 변경 비슷 내용
  - 부교재(열혈 C++ 프로그래밍) 참고

가상 함수(중요)

- 다음과 같은 상속 계층도를 가정하여 보자.

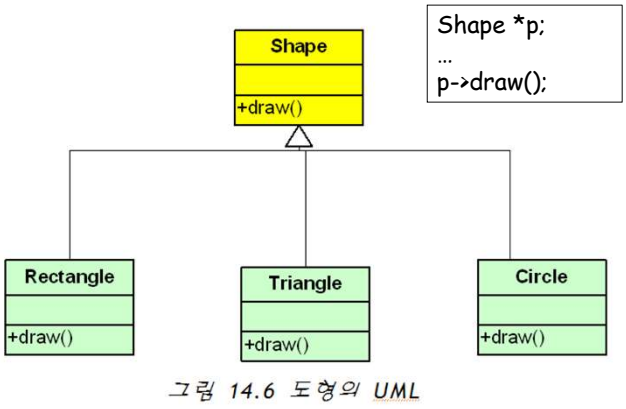


그림 14.6 도형의 UML

예제

```
class Shape {
        void draw() { cout << "Shape Draw" << endl; }
}
class Rectangle : public Shape {
        void draw() { cout << "Rect Draw" << endl; }
}
int main()
{
        Shape *ps = new Rectangle();// OK!
        ps->draw();                          // 어떤 draw()가 호출되는가?
}
```

ps는 부모 Shape 포인터이므로
부모 Shape의 draw()가 호출

```
Shape Draw
```

가상 함수

Shape *ps = new Rectangle(); // 자식이 부모로 형변환

- 자식 객체가 부모 객체로(Shape)으로 위장(형 변환)했어도 draw( ) 를 호출하면
  - 포인터가 가리키는 실제 객체가 사각형 객체이면 사각형을 그리는 draw()가 호출되고

    ps->draw(); // 사각형의 draw() 호출

  - 포인터가 가리키는 실제 객체가 원 객체이면 원을 그리는 draw()가 호출된다면 좋을 것이다

    ps->draw(); // 원의 draw() 호출

  - 위 두 문장은 같은 형식이지만 ps 가 가리키는 실제 객체에 따라 다른 함수가 호출

-> draw()를 가상 함수로 작성하면 가능

가상 함수

```
class Shape {
protected:
  int x, y;

public:
  void setOrigin(int x, int y){
            this->x = x;
            this->y = y;
  }                                  가상 함수 정의
  virtual void draw() {
            cout <<"Shape Draw" << endl;
  }
};

class Circle : public Shape {
private:
  int radius;
public:
  void setRadius(int r) {
            radius = r;
  }                                  재정의
  void draw() {
            cout << "Circle Draw"<< endl;
  }
};
```

```
class Rectangle : public Shape {
private:
  int width, height;

public:
  void setWidth(int w) {
            width = w;
  }
  void setHeight(int h) {
            height = h;
  }                                  재정의
  void draw() {
            cout << "Rectangle Draw" << endl;
  }
};

// 부모 클래스에서 가상함수로 선언한 함수가
// 자식 클래스에 존재하면 그 함수도 가상함수가
// 된다. → 원형 동일해야 함.
```

시험에 나옴

가상 함수

```
int main() {
  Shape *ps = new Rectangle();  // OK!
  ps->draw();
  delete ps;

  Shape *ps1 = new Circle();    // OK!
  ps1->draw();
  delete ps1;
  return 0;
}
```

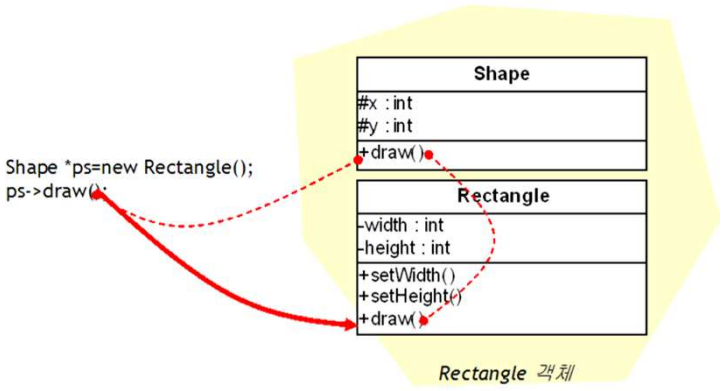
```
Rectangle Draw
Circle Draw
계속하려면 아무 키나 누르십시오 . . .
```

- ps 는 Shape(부모) 포인터 → 자식의 멤버 호출 불가
- 단, **가상함수를 호출하는 경우는 포인터가 가리키는 객체를 파악** → Rectangle 객체 → Rectangle 객체의 함수 호출

- 일반적으로 부모 포인터는 자식 멤버함수 이용 불가능
  - ➢ 하향 형 변환 해야 사용 가능
- **가상함수를 사용하면**, 형 변환 하지 않아도 부모 포인터는 자식 멤버함수 이용 가능

동적 바인딩

- 컴파일 단계에서 모든 바인딩이 완료되는 것을 정적 바인딩(static binding)이라고 한다.
- 반대로 바인딩이 실행 시까지 연기되고 실행 시간에 실제 호출되는 함수를 결정하는 것을 동적 바인딩(dynamic binding), 또는 지연 바인딩(late binding)이라고 한다. → 가상함수



정적 바인딩과 동적 바인딩

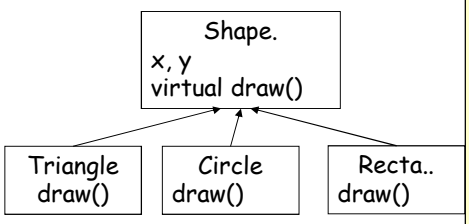
바인딩의 종류	특징	속도	대상
정적 바인딩 (static binding)	컴파일 시간에 호출 함수가 결정된다.	빠르다	일반 함수
동적 바인딩 (dynamic binding)	실행 시간에 호출 함수가 결정된다.	늦다	가상 함수

예 제

```
#include <iostream>
using namespace std;

class Shape {
protected:
  int x, y;  // 원점
public:
 virtual void draw() {
            cout <<"Shape Draw";
  }
};
class Circle : public Shape {
 private:
   int radius;
 public:
   void draw() {
            cout << "Circle Draw" <<
endl;
   }
};
```

```
class Rectangle : public Shape {
private:
  int width, height;

public:
  void draw() {
            cout << "Rectangle Draw" << endl;
  }
};

class Triangle: public Shape {
private:
  int base, height;
public:
  void draw() {
            cout << "Triangle Draw" << endl;
  }
};
```

예 제




```
int main() {
  Shape *arrayOfShapes[3];

  arrayOfShapes[0] = new Rectangle();
  arrayOfShapes[1] = new Triangle();
  arrayOfShapes[2] = new Circle();

  for (int i = 0; i < 3; i++)
           arrayOfShapes[i]->draw();
  return 0;
}
```

```
Rectangle Draw
Triangle Draw
Circle Draw
```

다형성의 장점

- 새로운 도형이 추가되어도 main()의 루프는 변경할 필요가 없다.

```
class Parallelogram : public Shape     // 평행사변형
{
public:
    void draw(){
        cout << "Parallelogram Draw" << endl;
    }
};

int main() {
  Shape *arrayOfShapes[4];

  arrayOfShapes[0] = new Rectangle();
  arrayOfShapes[1] = new Triangle();
  arrayOfShapes[2] = new Circle();
  arrayOfShapes[3] = new Parallelogram();
  for (int i = 0; i < 4; i++) {
    arrayOfShapes[i]->draw();
  }
}
```

예제(책은 결과 오류)

```
#include <iostream>
using namespace std;

class Animal  {
public:
  Animal() { cout <<"Animal 생성자" << endl; }
  ~Animal() { cout <<"Animal 소멸자" << endl; }
  virtual void speak() {
            cout <<"Animal speak()" << endl;
  }
};

class Dog : public Animal {
public:
  Dog() { cout <<"Dog 생성자" << endl; }
  ~Dog() { cout <<"Dog 소멸자" << endl; }
  void speak() { cout <<"멍멍" << endl; }
};
```

```
class Cat : public Animal {
public:
  Cat() { cout <<"Cat 생성자" << endl; }
  ~Cat() { cout <<"Cat 소멸자" << endl; }
  void speak() { cout <<"야옹" << endl; }
};

int main() {
  Animal *a1 = new Dog();
  a1->speak();

  Animal *a2 = new Cat();
  a2->speak();
  delete a1;
  delete a2;
  return 0;
}
```

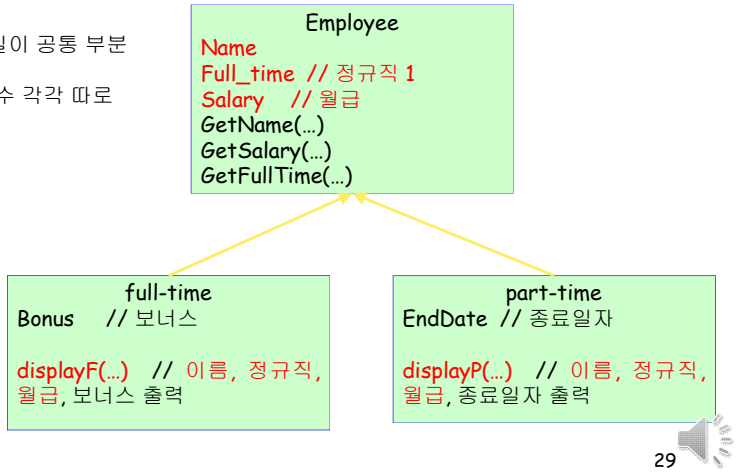
```
Animal 생성자
Dog 생성자
멍멍
Animal 생성자
Cat 생성자
야옹
Animal 소멸자
Animal 소멸자
(a1, a2 는 Animal 이므로)
```

소멸자 제대로 작동 않는 문제 발생
→ 이후에는 가상함수 사용하여 해결

가상함수 예제(8장 내용)

- 등장 객체를 파악하여 각각을 클래스로 작성 → full time, part time
- 각 클래스의 공통 부분을 클래스로 구성하고 부모 클래스로 설정

?? display 하는 일이 공통 부분 존재
→ 8장 에서는 함수 각각 따로 작성



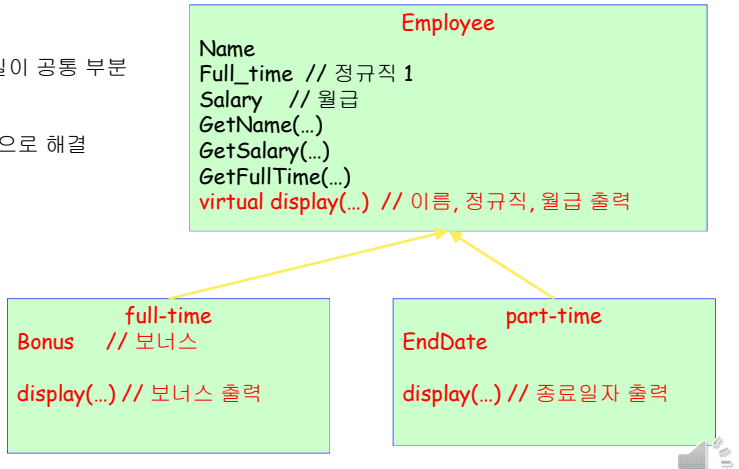
29

가상함수 예제 (9장 내용)

- 등장 객체를 파악하여 각각을 클래스로 작성 → full time, part time
- 각 클래스의 공통 부분을 클래스로 구성하고 부모 클래스로 설정

?? display 하는 일이 공통 부분 존재

→ 가상함수 사용으로 해결



30

가상함수 예제

- full-time, part-time 직원(가상 함수 변경시 출력)

```
int main() {
        // 이름, 정규직(1)/비정규직(0), 월급, 보너스
  Employee *a = new FullTime("김철수", 1, 200, 120);

        // 비정규직 보너스 없음. 대신 계약 종료 일자
  Employee *b = new PartTime("김현빈", 0, 150, "181230");


  // 부모, 자식 모든 속성을 출력, 함수이름 동일
  a->display();      // 가상함수  .... (1)
  b->display();      // .... (2)
        // (1)은 이름, 정규직(1)/비정규직(0), 월급, 보너스 출력
        // (2)는 이름, 정규직(1)/비정규직(0), 월급, 종료일자 출력
  return 0;
}
```

31

- 각각을 클래스로 구현

```
class Employee {
    string name;
    int salary, full_time;

public:
    Employee(string n, int s, int f)
    : name(n), salary(s), full_time(f) { }

    string getName() const { return name; }
    int getSalary() const { return salary; }
    bool getFullTime() const
    { return full_time; }

    virtual void display() const;
};
void Employee::display() const {
    cout << name << salary << full_time << endl;
}
```

Employee
Name
Full_time // 정규직 1
Salary // 월급
getName(…)
getSalary(…)
getFullTime(…)
virtual display(…)

32

```
class FullTime : public Employee {
    int bonus;
public:
    FullTime(string n, int f, int s, int b)
          : Employee(n,  f, s), bonus(b) { }

    void display() const;
};

void FullTime ::display() const {          // 부모 것 호출하고 자신 것 수행
    Employee::display();       // 부모의 것(멤버 변수)는 부모에게 출력 요청
    cout << bonus << endl;     // 자신의 것만 출력
}
```

full-time
Bonus // 보너스

display(…) // 보너스 출력

33

```
class PartTime : public Employee {
    string EndDate;
public:
    PartTime(string n, int f, int s, string e)
          : Employee(n,  f, s) { EndDate = e; }

    void display() const;
};

void PartTime ::display() const {
    Employee::display();       // 부모의 것 (멤버 변수)는 부모에게 출력 요청
    cout << EndDate << endl;   // 자신의 것만 출력
}
```

part-time
EndDate

display(…) // 종료일자 출력

34

참조자와 가상함수 예제

```
class Animal  {
public:
   virtual void speak() {
            cout <<"Animal speak()" << endl;
   }
};

class Dog : public Animal {
public:
   void speak() { cout <<"멍멍" << endl; }
};

class Cat : public Animal {
public:
   void speak() { cout <<"야옹" << endl; }
};
```

```
int main()
{
     Dog d;
     Animal &a1 = d;
     a1.speak();

     Cat c;
     Animal &a2 = c;
     a2.speak();
     return 0;
}
```

```
멍멍
야옹
```

- 참조자인 경우에는 다형성이 동작될 것인가?
- 참조자도 포인터와 동일하게 적용된다.

참고(가상함수)

```
// 이전 가상함수 예제는 포인터, 참조자
// 사용하여 객체 가리킴 → 가상함수 작용
// 이 예제는 가상함수 작용 안함.

class Animal  {
public:
   virtual void speak() {
            cout <<"Animal speak()" << endl;
   }
};

class Dog : public Animal {
public:
   void speak() { cout <<"멍멍" << endl; }
};
```

```
int main()
{
     Dog d;
     Animal a1 = d;
     a1.speak();

     return 0;
}
```

```
Animal speak()
```

- 가상함수는 포인터, 참조자인 경우만 사용 가능

참조자와 가상함수 예제

```
class Shape {
protected:
    int x, y;
public:
    void setOrigin(int x, int y) {
        this->x = x;   this->y = y;
    }
    virtual void draw() {
        cout << "Shape Draw " << endl;
    }
};
class Rectangle : public Shape {
private:
    int width, height;
public:
    void setWidth(int w) { width = w;    }
    void setHeight(int h) {height = h;   }
    void draw() { cout << "Rect Draw" << endl; }
};
```

```
class Circle : public Shape {
private:
    int radius;
public:
    void setRadius(int r) { radius = r;    }
    void draw() { cout << "Circle Draw" << endl;
    }
};
// 평행 이동하는 일반 함수
void move(Shape& s, int sx, int sy) {
    s.setOrigin(sx, sy);// 부모의 멤버 호출
    s.draw();           // 자식의 멤버 호출
}                       // 참조자는 가상함수 사용
int main(){
    Rectangle r;
    move(r, 0, 0);
    Circle c;
    move(c, 10, 10);
    return 0;
}
```

가상함수 사용하지 않는 경우

```
int main(){

    Dog* p1 = new Dog();
    p1->speak();

    Cat* p2 = new Cat();
    p2->speak();  // p1, p2 와 같이 이름이 다른  변수
    return 0;     // 여러 개 사용해야함. → 불편
}
```

- 함수 재정의 했으나 가상함수 사용하지 않은 경우

- 결과

멍멍
야옹

```
class Animal {
public:
    void speak() {      // 가상함수 아님
        cout << "Animal speak()" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() { cout << "멍멍" << endl; }
};

class Cat : public Animal {
public:
    void speak() { cout << "야옹" << endl; }
};
```

- 일반 함수 : 정적 바인딩
- 가상 함수 : 동적 바인딩

가상함수 사용하는 경우

```
int main() {

    Animal* a = new Dog();
    a->speak();

    a = new Cat();
    a->speak();
    return 0;
}
```

- 가상함수 사용한 경우, main 이 간단

```
멍멍
야옹
```

```
class Animal {
public:
    virtual void speak() {
        cout << "Animal speak()" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() { cout << "멍멍" << endl; }
};

class Cat : public Animal {
public:
    void speak() { cout << "야옹" << endl; }
};
```

재정의: 부모 클래스에 있는 상속받은 멤버 함수를 다시 정의하는 것 → overriding

report 1

- **제출처 변경 : DOOR 과제**
- **상속과 가상함수 사용**
- **사용하는 객체들과 속성, 동작**
  - Point, Line, Circle => 이들을 class로 구성
- class 간 관계
  - Point 클래스 ← has-a 관계 → Line, Circle 클래스
  - Line, Circle 클래스 공통점 모아 부모 클래스(Shape 클래스) 구성
    - Shape 클래스 ← is-a 관계 → Line, Circle 클래스
- 멤버변수는 모두 private

report 1

- Line, Circle => 이들을 class로 구성
  - 속성 : 시작점, 끝점(좌상단점, 우하단점 좌표)
    - 원은 사각형의 내접원으로 그릴 수 있음.
  - 동작 1 => 생성자 : 시작점, 끝점 좌표를 인자로 받아 속성에 저장
  - 동작 2 → Draw(…)
    : Line class 는 속성(시작점, 끝점) 출력, 직선 그린다 출력
    : Circle class 는 속성(시작점, 끝점) 출력, 원 그린다 출력
- **Draw( ) 함수들**
  - **자식들의 Draw() 함수 내용이 (일부 동일 + 일부 다른 일) → 동일 한 내용을 부모에서 처리, 틀린 내용은 자식에서 처리**
- **제출 : oop_91_학번.txt**

report 1

```
void main( ) {
        Circle a(1, 1, 5, 5);  // 좌상단점, 우하단점 좌표
        Line b(5, 5, 9, 9);    // 좌상단점, 우하단점 좌표

        a.Draw( );             // "원 그린다" 좌상단/우하단점 좌표 출력
        b.Draw( );             // "직선 그린다" 좌상단/우하단점 좌표 출력

        Shape *p;
        p = new Line(10, 10, 100, 100);
        p->Draw( );            // "직선 그린다" 출력, 좌상단/우하단점 좌표 출력
        p = new Circle(100, 100, 200, 200);
        p->Draw( );            // "원 그린다" 출력, 좌상단/우하단점 좌표 출력
        delete p;
}
```

가상 소멸자

- 다형성을 사용하는 과정에서 소멸자는 가상함수로 작성해야 함.
  - 다형성을 사용하는 과정에서 소멸자를 virtual로 해주지 않으면 문제가 발생
  - (예제) String 클래스의 멤버(문자열)를 상속받아서 문자열 앞뒤에 헤더를 붙여 출력하는 MyString 이라는 클래스를 정의하여 보자.

부모멤버(문자열)

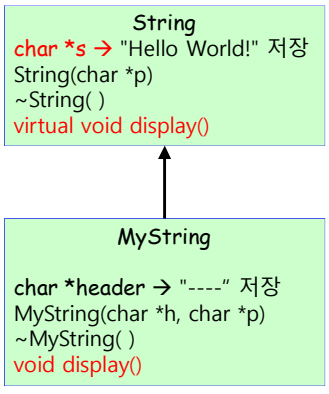
부모 멤버 상속 받고 부모의 문자열 앞뒤에 헤더("--") 붙여 출력

String 객체
(부모, 문자열 저장)

MyString 객체
(자식)

가상 소멸자

```
int main(){
    String *p = new MyString("----", "Hello World!");

    p->display(); // 가상함수 호출
    // ----Hello World!----

    delete p;     // p 는 부모 포인터이므로
                  // 소멸자는 부모 것 호출
    return 0;
}
```



가상 소멸자 필요성

```
class String {
   char *s;
public:
   String(char *p){
      cout << "String() 생성자" << endl;
      s = new char[strlen(p)+1];
      strcpy(s, p);
   }
   ~String(){
      cout << "String() 소멸자" << endl;
      delete[] s;
   }
   virtual void display()  {
      cout << s;
   }
};
```

```
class MyString : public String {
   char *header;
public:
   MyString(char *h, char *p) : String(p){
      cout << "MyString() 생성자" << endl;
      header = new char[strlen(h)+1];
      strcpy(header, h);
   }
   ~MyString(){
      cout << "MyString() 소멸자" << endl;
      delete[] header;
   }
   void display() {
      cout << header;  // 헤더출력
      String::display();    // 부모함수 호출
      cout << header << endl;  // 헤더출력
   }
};
```

```
int main(){
      String *p = new MyString("----", "Hello World!");

      p->display(); // 가상함수 호출
      delete p;      //  p 는 부모 포인터이므로
                     // 소멸자는 부모 것 호출
      return 0;
}
```

```
String() 생성자
MyString() 생성자
----Hello World!----
String() 소멸자
```

- p 지칭 객체 멤버 중에서

s (Hello world) 공간 → 없어짐
header(----) 공간 → 안 없어짐

가상 소멸자

- 어떻게 하여야 자식 클래스의 **MyString** 소멸자 호출되게 할 수 있는가?
- **String** 클래스(부모 클래스)의 소멸자를 **virtual**로 선언하면 된다.
- 소멸자는 모두 가상함수로 선언(부모, 자식 모두)

자식 소멸자 호출하면
부모 소멸자 자동 호출

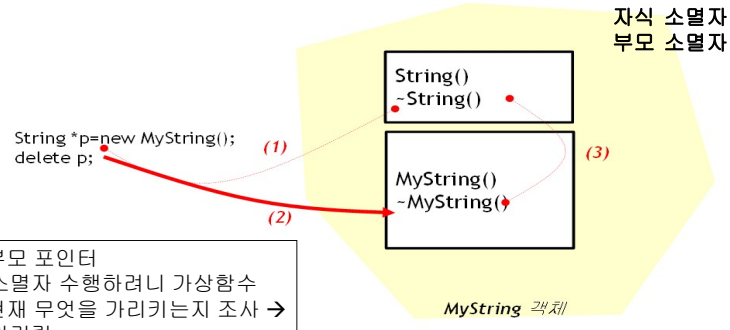


그림 9.10 가상 소멸자

- **p** 는 부모 포인터
- 부모 소멸자 수행하려니 가상함수
- **p** 가 현재 무엇을 가리키는지 조사 → 자식 가리킴
- 자식의 소멸자 호출
- 부모 소멸자 호출 (부모 것 제거)
- 자식의 소멸자 수행, 자식 것 제거

가상 소멸자

```
class String {
      ...
      virtual ~String(){
            cout << "String() 소멸자" << endl;
            delete[] s;
      }
};
class MyString : public String {
      ...
      virtual ~MyString() {          // 자식 소멸자는 부모 소멸자와 이름 달라도 virtual 없어도 무방
            cout << "MyString() 소멸자" << endl;
            delete[] header;
      }
};
int main() {
      ...// 앞과동일
}
```

```
String() 생성자
MyString() 생성자
----Hello World!----
MyString() 소멸자
String() 소멸자
```

소멸자 문제

```
class String {
  char *s;
public:
  String(char *p){
            cout << "String() 생성자" << endl;
            s = new char[strlen(p)+1];
            strcpy(s, p);
  }
  virtual ~String(){
            cout << "String() 소멸자" << endl;
            delete[] s;
  }
  virtual void display()  {
            cout << s;
  }
};
```

```
class MyString : public String {
  char *header;
public:
  MyString(char *h, char *p) : String(p){
            cout << "MyString() 생성자" << endl;
            header = new char[strlen(h)+1];
            strcpy(header, h);
  }
  ~MyString(){
            cout << "MyString() 소멸자" << endl;
            delete[] header;
  }
  void display() {
            cout << header;      // 헤더출력
            String::display();   // 부모함수 호출
            cout << header << endl;  // 헤더출력
  }
};
```

- 교재 내용 컴파일 오류 발생
- C++ 버전 업으로 규칙 강화로 인하여 예전에는 오류 아닌 내용 오류 발생

참고) 이전 코드 오류 수정 1, 문자열 전달

- 이전 코드 컴파일시 오류
  - MyString 생성자
    - MyString(char *h, char *p); // 생성자
  - Main 에서
    - String* p = new MyString("----", "Hello World!");
  - error C2664: 'MyString::MyString(const MyString &)':
    인수 1을(를) 'const char [5]'에서 'char *'(으)로 변환할 수 없습니다.
  - 무슨 오류 ?? → error code(C2664) 를 searching --> 인자 전달시 type 문제

참고) 오류 수정 1, 문자열 전달

```
void prn(char* a) {
    cout << a << endl;
}

int main() {
    char str[] = "abcd";
    prn(str);     // ok
    return 0;
}

// 결과 abcd
```

```
void prn(char* a) {
    cout << a << endl;
}

int main() {

    prn("abcd"); // err
    return 0;
}

// 컴파일 오류
```

error C2664: 'void prn(char *)': 인수 1을(를) 'const char [5]'에서 'char *'(으)로 변환할 수 없습니다.

참고) 오류 수정 1, 문자열 전달

- 리터럴(literal) : "문자 그대로의"라는 뜻
  - 프로그래밍 언어에서 리터럴은 무엇을 의미?
  - 아래 코드에서 다음과 같은 것들을 '리터럴'이라 부른다.

```
char ch = 'A';              // A를 리터럴이라 한다
string str = "Hello"        // 문자열 Hello를 리터럴이라 한다
```

- 즉, 이와 같이 '변하지 않는, 고정된 값'이 **리터럴** 또는 **리터럴 상수**라고 불린다. → 상수
- 상수를 인자로 함수 호출시 그 상수는 호출되는 함수 내에서 변하면 안된다.
  → 상수를 받는 매개변수는 const 로 지정해야 함

참고) 오류 수정 1, 문자열 전달

```
void prn(char* a) {
    cout << a << endl;
}

int main() {
    char str[] = "abcd";
    prn(str);
    return 0;
}

// 결과 abcd
```

2) 배열 시작 주소를 받음

1) str 에 저장된 배열 시작 주소 전달

```
void prn(char* a) {
    cout << a << endl;
}

int main() {

    prn("abcd"); // err
    return 0;
}

// 컴파일 오류
```

```
void prn(const char* a) {
    cout << a << endl;
}

int main() {

    prn("abcd"); // ok
    return 0;
}

// 결과 abcd
```

- 오류 원인 : C++ 에서 자료형을 엄격하게 조사하여 발생하는 오류
  - C++ 은 문자열 상수(리터럴)의 자료형을 "char* " 아니라 " const char* " 로 처리
  - "abcd" 리터럴(문자열 상수)를 전달 → 함수에서 char *a(포인터 변수)로 받음 → 오류
    - 포인터 변수로 받으면 함수 내에서 문자열 내용이 변할 수 있음
  - 인자 "abcd" 전달시 "abcd"는 상수(리터럴) → "abcd" 는 값이 변하면 안됨 → const 로 받아야 함.

참고) 오류 수정 1, 문자열 전달

- 다음과 같이 수정하여 해결

➢ main 에서
String *p = new MyString("----", "Hello World!");

➢ String 생성자
String(const char* p) { ... }

➢ MyString 생성자
MyString(const char* h, const char* p) : String(p) { ... }

참고) 오류 수정 2, 문자열 복사

- 수정 후 컴파일시 또 오류

```
String(const char* p) {   // MyString 도 동일
    cout << "String() 생성자" << endl;
    s = new char[strlen(p) + 1];
    strcpy(s, p);      // err
}
```

- error C4996: 'strcpy': This function or variable may be unsafe. Consider using strcpy_s instead ....

참고) 오류 수정 2, 문자열 복사

- strcpy 대신 strcpy_s 사용
  - strcpy_s(dst, dst 길이, src)

```
String(const char* p) {
    cout << "String() 생성자" << endl;
    s = new char[strlen(p) + 1];
    strcpy_s(s, strlen(s)+1, p);
}

MyString(const char* h, const char* p) : String(p) {
    cout << "MyString() 생성자" << endl;
    header = new char[strlen(h) + 1];
    strcpy_s(header, strlen(h) + 1, h);
}
```

순수 가상 함수

- 순수 가상 함수(pure virtual function): 함수 헤더(원형)만 존재하고 함수의 몸체는 없는 함수 → 하는 일이 없는 가상함수

```
virtual    반환형      함수이름(매개변수  리스트) = 0;
```

- (예) virtual void draw() = 0;

- 추상 클래스(abstract class): 순수 가상 함수를 가지고 있는 클래스
  - 가상함수 작성시 자식의 함수에서 하는 일이 모두 다른 경우 부모의 함수를 순수 가상함수로 작성
  - 도형을 실제 그리는 코드 작성시 Shape 자식인 Rectangle, Circle, Triangle 등 의 가상함수 draw( ) 는
    - 각 도형을 그리는 의미(의미만 동일)
    - 실제 도형 그릴 때 각자 자신 도형을 그리는 내용으로 구성(공통 내용 없음)
    - Shape 의 draw() 를 순수 가상함수로 작성

순수 가상 함수의 예

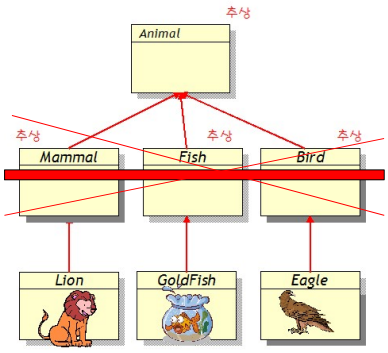
```
class Shape {
protected:
   int x, y;
public:       ...
   virtual void draw() = 0;
};

class Rectangle : public Shape {
private:
   int width, height;
 public:
   void draw() {
      cout << "사각형 실제 그리는 내용" << endl;
   }
};
class Circle : public Shape {
private:
   int r;
 public:
   void draw() {
      cout <<  " 원 실제 그리는 내용" << endl;
   }
};
```

```
int main() {
   Shape *ps = new Rectangle();    // OK!
   ps->draw();   // 사각형을 그린다.

   ps = new Circle();     // OK!
   ps->draw();   // 원을 그린다
   delete ps;

   return 0;
}
```

Shape 의 draw( ) 는 실제 사용할 일이 없음.

추상 클래스

- 추상 클래스(abstract class): 순수 가상 함수를 가지고 있는 클래스
- 추상 클래스는 추상적인 개념을 표현하는데 적당하다.



- 자식 클래스 함수들
  - ✓ move()
  - ✓ eat()
  - ✓ speak() ← 물고기 ?
- 함수 실제 내용이 세 동물에서 모두 다름
- 함수들 이름은 동일하게 작성,
- 순수 가상함수를 부모에 작성

예 제

```
class Animal {
        virtual void move() = 0;  // 자식은 각자 다른 방식으로 이동
        virtual void eat() = 0;     // 자식은 각자 다른걸 먹음
        virtual void speak() = 0; // 자식은 각자 다르게 짖음
};

class Lion : public Animal {
        void move(){
                cout << "사자의 move() << endl;
        }
        void eat(){
                cout << "사자의 eat() << endl;
        }
        void speak(){
                cout << "사자의 speak() << endl;
        }
};
```

공통 부분이 없으니 없는 것을 공통으로 부모로 올림

예 제

```
class GoldFish : public Animal {
        void move(){
                cout << " GoldFish 의 move() << endl;
        }
        void eat(){
                cout << " GoldFish 의 eat() << endl;
        }
        void speak(){
                cout << " GoldFish 의 speak() << endl;
        }
};

class Eagle : public Animal {
        void move(){
                cout << " Eagle 의 move() << endl;
        }
        void eat(){
                cout << " Eagle 의 eat() << endl;
        }
        void speak(){
                cout << " Eagle 의 speak() << endl;
        }
};
```

추상클래스는 객체 생성 불가

```
class Shape {
public:
    virtual void Draw() = 0;
};

class Rect : public Shape {
public:
    virtual void Draw() {..... } // 구현
};

int main() {
    Shape  s; // error, 추상 클래스

    Rect r; // ok
}
```

추상 클래스 특징

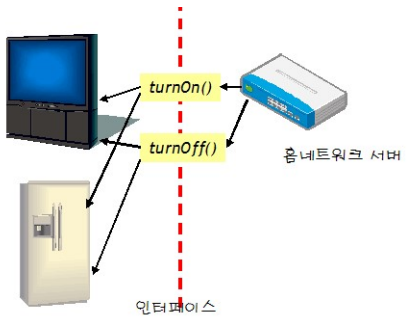
- 객체를 생성할 수 없다.

추상 클래스의 설계 의도

- 기존의 class 코드 사용시 부모 함수가 순수 가상 함수로 작성되면
- 자식 클래스에게 해당 가상함수 내용(본체)을 반드시 작성하여 사용하라고 지시하는 것

추상 클래스를 인터페이스로

- 추상 클래스는 객체들 사이에 상호 작용하기 위한 인터페이스 정의하는 용도로 사용 가능
  - 홈네트워킹 시스템 구성시 TV, 냉장고 등을 같은 이름의 함수 turnOn( ), turnOff( ) 로 구동시키려 함.
  - 이들 가전제품의 부모 클래스(RC class)를 추상 클래스로 만들고
    - 그 안에 순수 가상함수 turnOn(..), turnOff(..) 을 작성
  - 각 제품(자식 클래스)은 자신의 방식으로 전원 키고 끄는 작업하도록
    - 각 제품(자식 클래스) 은 제품 구조에 따라 각각 turnOn(..), turnOff(..) 작성
  - 홈 네트워킹 시스템은 turnOn( ), turnOff( ) 함수만 호출하여 구동




```
RC *p;      // 부모 클래스
            // 홈 네트워킹 시스템
p = new Television();
P->turnOn(); // TV 켜짐

p = new Refrigerator();
P->turnOn(); // 냉장고 켜짐
```

예 제

```
class RemoteControl {
        // 순수가상함수정의
        virtual void turnON() = 0;     // 가전제품을켠다.
        virtual void turnOFF() = 0;    // 가전제품을끈다.
}

class Television : public RemoteControl {
        void turnON()
        {
                // 실제로TV의전원을켜기위한코드가들어간다.
                ...
        }
        void turnOFF()
        {
                // 실제로TV의전원을끄기위한코드가들어간다.
                ...
        }
}
```

예 제

```
 class Refrigerator : public RemoteControl {
        void turnON() {
                // 실제로 냉장고의전원을켜기위한코드가들어간다.
                ...
        }
        void turnOFF() {
                // 실제로 냉장고의전원을끄기위한코드가들어간다.
        }
}

int main(){
        RemoteControl *pt = new Television();
        pt->turnOn();
        pt->turnOff();

        RemoteControl *pr = new Refrigerator();
        pr->turnOn();
        pr->turnOff();

        delete pt;          delete pr;          return 0;
}
```

report 2

- **상속, 가상함수를 사용하여 2 입력 and, or, xor gate를 구현,**
  - 프로그램에서 and, or, xor 연산자는 각각 &&, ||, ^ 사용
  - ANDGate, ORGate, XORGate 객체 상태(멤버변수, protected)
    : x, y -> 입력변수 저장
    : z -> 연산결과 저장
  - ANDGate, ORGate, XORGate 동작(멤버함수, **public**)
    1) 생성자 -> 인자 없음, 입력/출력 멤버 변수를 모두 false로 저장
    2) 입력(x, y) 지정 -> void inputSet(bool xx, bool yy)
    : xx 를 멤버변수 x 에 저장, yy 를 멤버변수 y 에 저장
    3) 입력에 따른 and, or, xor 연산 수행 -> op( )
    : 각 gate에서 멤버변수 (x, y) 사용하여 연산 수행 후 결과를 z 에 저장
    : 입력 / 결과 출력
- 특징
  - **부모 멤버변수가 protected → 자식이 사용 가능**
  - **자식들의 op() 함수 내용이 완전히 다른 경우 → 순수 가상 함수 사용**

main

```
void main() {
        Gate *p;
        p = new ANDGate();
        p->inputSet(true, false);
        p->op();
        delete p;

        p = new ORGate();
        p->inputSet(true, false);
        p->op();
        delete p;

        p = new XORGate();
        p->inputSet(true, false);
        p->op();
        delete p;
}
```