가상함수와 추상클래스

Review : 템플릿과 STL

- 템플릿은 함수나 클래스 코드를 찍어내듯이 생산할 수 있도록 일반화(generic)시키는 도구
  - 함수 템플릿
    - 코드는 같지만 자료형이 다른 함수를 자동으로 생성. template <typename T> void func (T & a, T & b);
  - 클래스 템플릿
    - 데이터 멤버의 자료형만 다른 클래스를 생성. template <typename T> class Stack{ ... }
- STL(Standard Template Library)
  - 템플릿으로 작성된 템플릿 클래스와 함수 라이브러리
  - pair, tuple, vector, map, set, stack, queue, deque
  -
- algorithm
  - include <algorithm>
  - sort(), reverse(), rotate(), random_shuffle(), count(), count_if(), binary_search()
- 람다
  - [캡쳐리스트](매개변수리스트) -> 리턴타입 { 함수코드 }

학습 목표

- 함수 오버라이딩과 동적 바인딩을 이해한다.
- 가상 함수를 이해한다.
- 다형성을 이해한다.
- 다형성을 적용한 프로그램을 구현할 수 있다.
- 추상 클래스의 목적을 이해하고 프로그램에 활용할 수 있다.

학습 목차

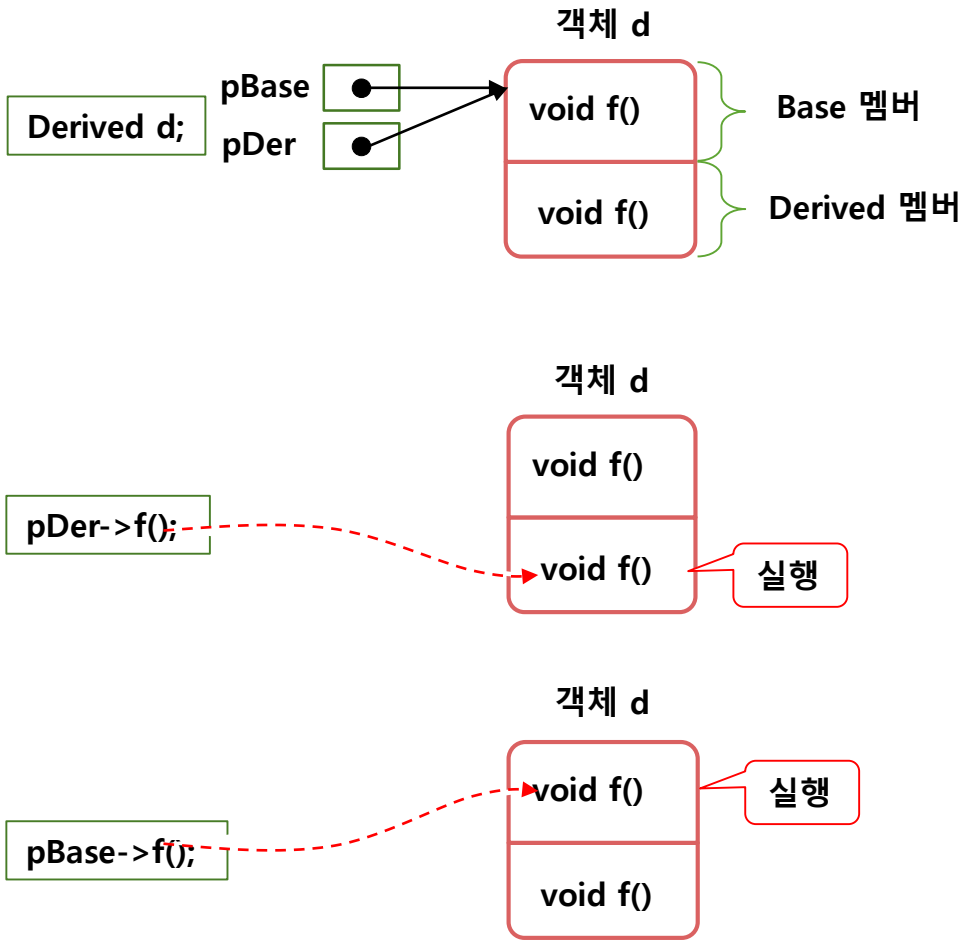
- 상속 관계에서 함수 재정의
- 가상 함수와 오버라이딩
- 런타임 타입 정보와 동적 자료형 변환
- 추상 클래스

- 다형성 구현 관련 포인트
  - 가상 함수, 동적 바인딩, 함수 오버라이딩

상속 관계에서 함수 재정의

- 정적 바인딩 : 함수 재정의(컴파일 시간 다형성)

```
class Base {
public:
   void f() { cout << "Base::f() called" << endl; }
};

class Derived : public Base {
public:   //함수 재정의
   void f() { cout << "Derived::f() called" << endl; }
};

void main() {
   Derived d, *pDer;
   pDer = &d;
   pDer->f(); //Derived::f() 호출
   //pDer->Base::f() .... Base::f() 호출

   Base* pBase;
   pBase = pDer; //업캐스팅
   pBase->f(); //Base::f() 호출
}
```



다형성

- 다형성은 같은 이름을 갖는 여러 형태의 함수를 클래스 별로 만들 수 있게 해주는 기능.

- 다형성의 예 : printArea()
  - 삼각형 객체의 printArea()는 삼각형의 넓이를,
  - 사각형 객체의 printArea()는 사각형의 넓이를 구하게 함.

- C++ 다형성 요소
  - 포인터
    - 기본 클래스를 가리키는 포인터를 사용 해 기본 클래스와 파생 클래스 모두를 가리키게 함.
  - 호환 객체
    - 상속을 하는 객체
  - 가상 함수

가상 함수와 오버라이딩

```
class Base {
public:
    virtual void f(); // f()는 가상 함수
};
```

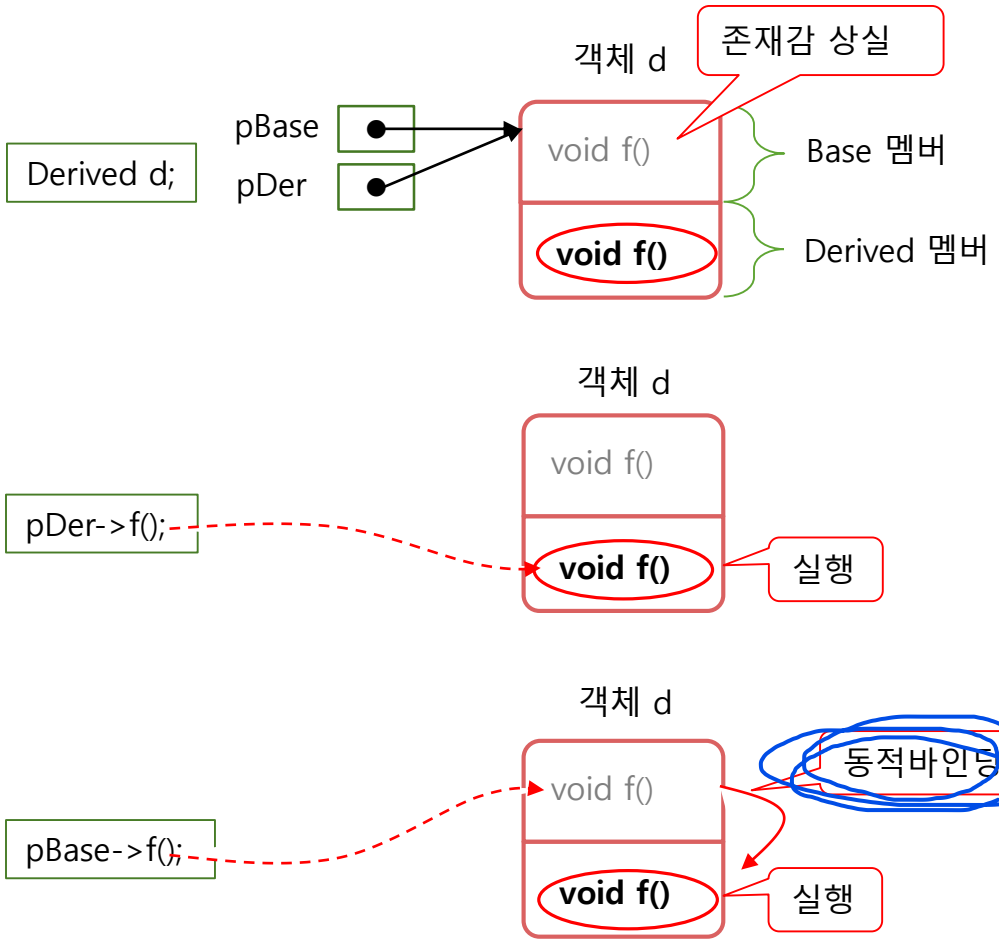
- 가상 함수(virtual function)
  - virtual 키워드로 선언된 멤버 함수.
  - virtual 키워드의 의미
    - 동적 바인딩 지시어 : 컴파일러에게 함수에 대한 호출 바인딩을 실행시간까지 미루도록 지시.

- 함수 오버라이딩(function overriding)
  - 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언.
    - 기본 클래스에 있는 가상 함수의 존재감을 상실 시킴.
    - 파생 클래스에서 오버라이딩 한 함수가 호출되도록 동적 바인딩 : 실행시간 다형성
    - 함수 재정의라고도 부름 : 다형성의 한 종류

- 오버라이딩의 목적
  - 파생 클래스에서 구현할 함수 인터페이스 제공(파생 클래스의 다형성)

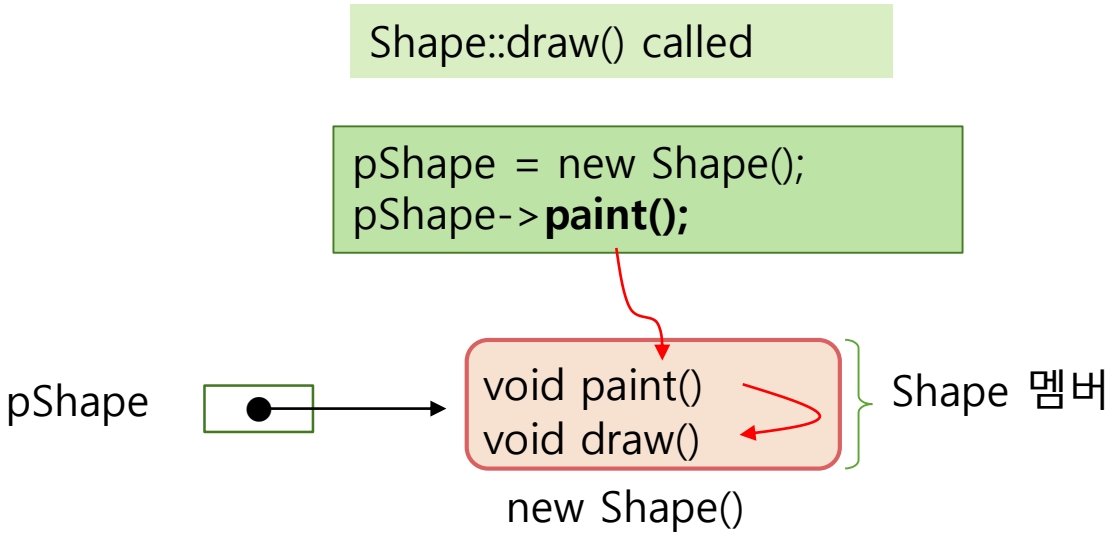
오버라이딩과 가상 함수 호출

```
class Base {
public:  //가상 함수 선언
   virtual void f() { cout << "Base::f() called" << endl; }
};

class Derived : public Base {
public:
  //오버라이딩, virtual 생략 가능, 동적 바인딩
   virtual void f() { cout << "Derived::f() called" << endl; }
};

int main() {
   Derived d, *pDer;
   pDer = &d;
   pDer->f();       //Derived::f() 호출

   Base *pBase = pDer; //업 캐스팅
   pBase->f();      //동적 바인딩 발생!! Derived::f() 실행
}
```



동적 바인딩

- 동적 바인딩
  - 파생 클래스에 대하여 기본 클래스에 대한 포인터로 가상 함수를 호출하는 경우,
  - 객체 내에 오버라이딩 한 파생 클래스의 함수를 찾아 실행되는 것.
  - 실행 중에 이루어짐.
  - 실행시간 바인딩, 런타임 바인딩, 늦은 바인딩으로 불림




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

class Shape {
public:
   void paint() {
      draw();
   }
   virtual void draw() {
      cout << "Shape::draw() called" << endl;
   }
};

int main() {
   Shape *pShape = new Shape();
   pShape->paint();
   delete pShape;
}
```

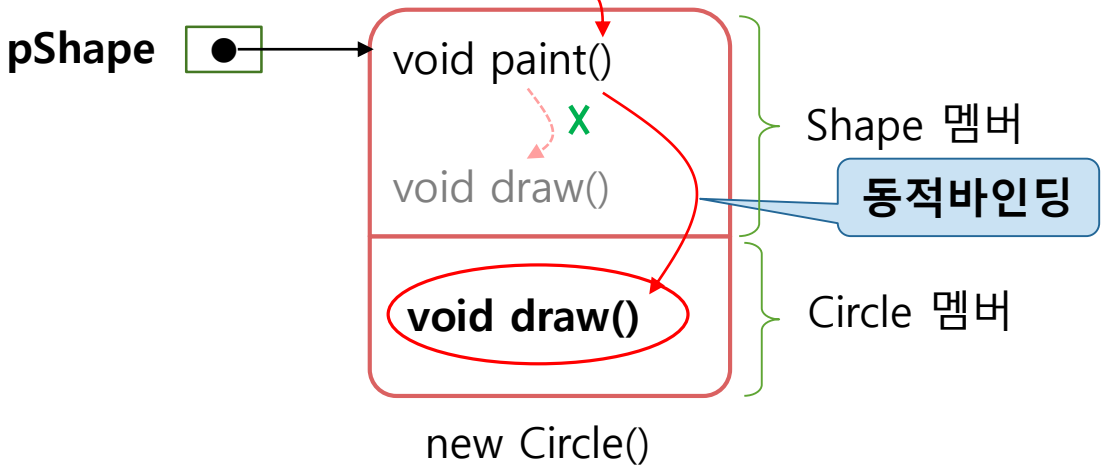
오버라이딩된 함수를 호출하는 동적 바인딩

```
class Shape {
public:
   void paint() {
      draw();
   }
   virtual void draw() {
      cout << "Shape::draw() called" << endl;
   }
};

class Circle : public Shape {
public:
   virtual void draw() {
      cout << "Circle::draw() called" << endl;
   }
};
int main() {
   Shape *pShape = new Circle(); // 업캐스팅
   pShape->paint();
   delete pShape;
}
```

기본 클래스에서 파생 클래스의 함수 호출

Circle::draw() called

pShape = new Circle();
pShape->**paint();**



override와 final 지시어

```
class Shape {
    public:
        void paint() { draw(); }
         virtual void draw();
};
class Circle : public Shape {
    public:
    void draw() override;
};                          //override는 파생 클래스 원형 뒤에 작성, 기본 클래스가 virtual이 아닐 경우 컴파일 오류 발생

int main() {
    Shape *pShape = new Circle();  // 업캐스팅
    pShape->paint();
    delete pShape;
}
```

❖ final 지시어

- 가상 함수 뒤에 사용 - 파생 클래스에서 오버라이딩 금지
- 클래스 이름 뒤에 사용 - 상속 금지

오버라이딩의 성공 조건

- 가상 함수 이름, 매개 변수 타입과 개수, 리턴 타입이 모두 일치

```
class Base {
public:
   virtual void fail();
   virtual void success();
   virtual void g(int);
};

class Derived : public Base {
public:
   virtual int fail();              //오버라이딩 실패. 리턴 타입이 다름
   void success();                  //오버라이딩 성공, virtual 생략 가능
   virtual void g(int, double);     //오버로딩 사례. 정상 컴파일
};
```

- 오버라이딩 시 virtual 지시어 생략 가능
  - 가상 함수의 virtual 지시어는 상속됨, 파생 클래스에서 virtual 생략 가능

- 가상 함수의 접근 지정
  - private, protected, public 중 자유롭게 지정 가능

오버로딩, 함수 재정의, 오버라이딩 비교

비교 요소	오버로딩	함수 재정의(가상 함수가 아닌 멤버에 대해)	오버라이딩
정의	매개 변수 타입이나 개수가 다르지만, 이름이 같은 함수들이 중복 작성되는 것	기본 클래스의 멤버 함수를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 완벽히 같은 원형으로 재작성하는 것	기본 클래스의 가상 함수를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 완벽히 같은 원형으로 재작성하는 것
존재	클래스의 멤버들 사이, 외부 함수들 사이, 그리고 기본 클래스와 파생 클래스 사이에 존재 가능	상속 관계	상속 관계
목적	이름이 같은 여러 개의 함수를 중복 작성하여 사용의 편의성 향상	기본 클래스의 멤버 함수와 별도로 파생 클래스에서 필요하여 재작성	기본 클래스에 구현된 가상 함수를 무시하고, 파생 클래스에서 새로운 기능으로 재작성하고자 함
바인딩	정적 바인딩. 컴파일 시에 중복된 함수들의 호출 구분	정적 바인딩. 컴파일 시에 함수의 호출 구분	동적 바인딩. 실행 시간에 오버라이딩된 함수를 찾아 실행
객체 지향 특성	컴파일 시간 다형성	컴파일 시간 다형성	실행 시간 다형성

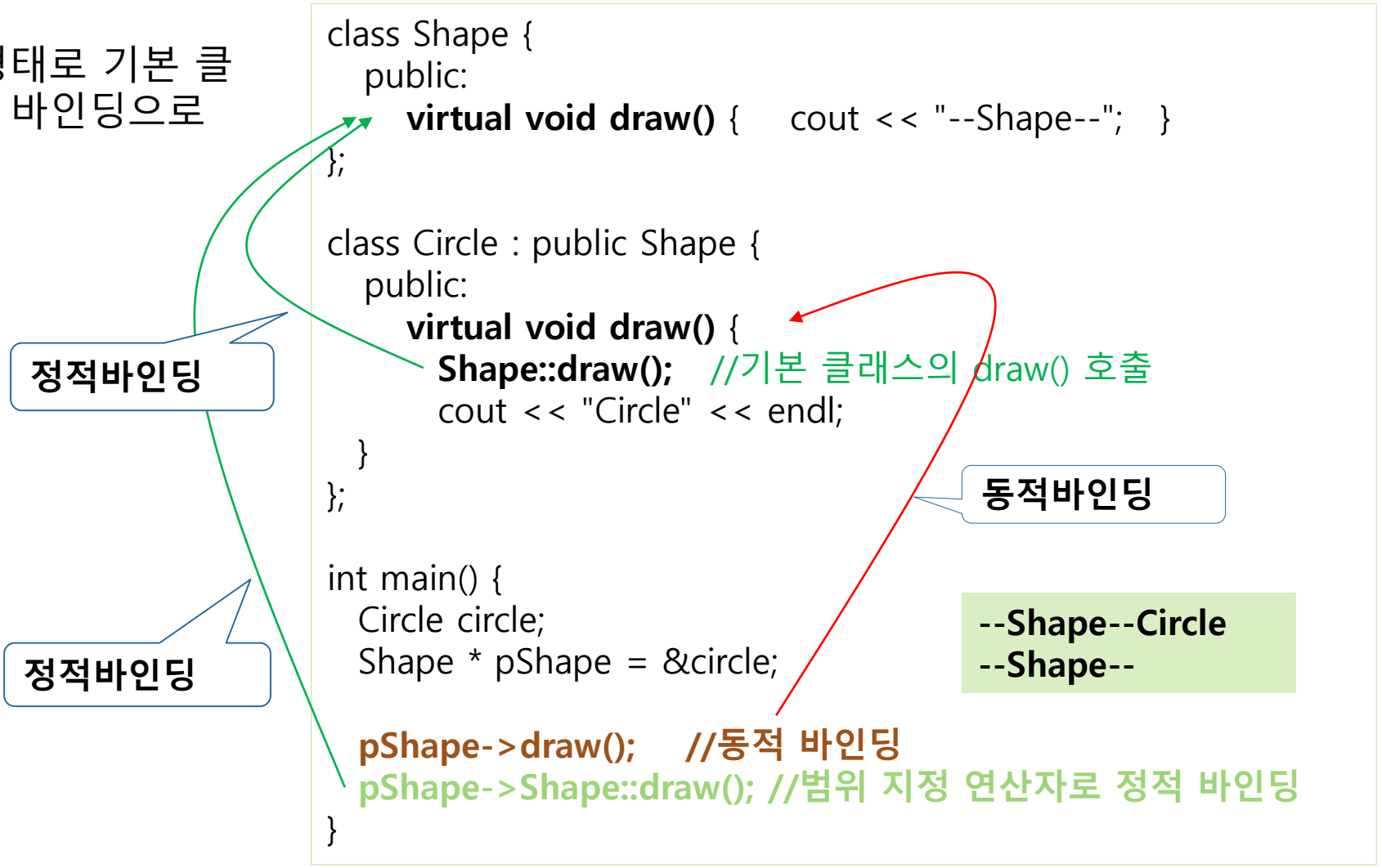
상속이 반복되는 경우의 가상 함수 호출

[실행 결과]
GrandDerived::f() called
GrandDerived::f() called
GrandDerived::f() called

```
class Base {
   public:
      virtual void f() { cout << "Base::f() called" << endl; }
};

class Derived : public Base {
   public:
      void f() { cout << "Derived::f() called" << endl; }
};

class GrandDerived  : public Derived {
   public:
      void f() { cout << "GrandDerived::f() called" << endl; }
};
int main() {
   GrandDerived g;
   Base *bp;
   Derived *dp;
   GrandDerived *gp;
   bp = dp = gp = &g;
   bp->f();   dp->f();   gp->f();  //동적 바인딩에 의해 모두 GrandDerived의 함수 f() 호출
}
```

정적 바인딩

- 범위 지정 연산자(::)
  - 정적 바인딩 지시
  - 기본클래스::가상함수() 형태로 기본 클래스의 가상 함수를 정적 바인딩으로 호출



가상 소멸자

- 소멸자를 virtual 키워드로 선언
- 소멸자 호출 시 동적 바인딩 발생

```
class Base {
public:
   ~Base();
};

class Derived: public Base {
public:
   ~Derived();
};

int main() {
   Base *p = new Derived();
   delete p; //~Base() 소멸자만 실행
}
```

❶

소멸자가 가상 함수가 아닌 경우

```
class Base {
public:
   virtual ~Base();
};

class Derived: public Base {
public:
   virtual ~Derived();
};

int main() {
   Base *p = new Derived();
   delete p;
}
```

동적 바인딩

❷

❸

❶

파생 클래스의 소멸자가 자신의 코드 실행 후, 기본 클래스의 소멸자를 호출하도록 컴파일 됨

❶ ~Base() 소멸자 호출
❷ ~Derived() 실행
❸ ~Base() 실행

가상 소멸자 경우

가상 소멸자 예

virtual 차이!

- 소멸자를 가상함수로 선언
  - ✓ 파생클래스와 기본 클래스의 소멸자를 모두 실행
  - ✓ 기본 클래스의 소멸자를 가상함수로 선언하여 정상적인 소멸 과정이 진행되도록 하는 것이 좋음

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

class Base {
   public:
      virtual ~Base() { cout << "~Base()" << endl; }
};

class Derived: public Base {
   public:
      virtual ~Derived() { cout << "~Derived()" << endl; }
};

int main() {
   Derived *dp  = new Derived();
   Base *bp = new Derived();

   delete dp;    //Derived의 포인터로 소멸, virtual 유무에 상관없이 파생 소멸자, 기본 소멸자 모두 실행.
   delete bp;    //만약 virtual이 아니면 Base의 소멸자만 실행.
}
```

~Derived()
~Base()
delete dp;

~Derived()
~Base()
delete bp;

런타임 타입 정보와 동적 자료형 변환

- 런타임 타입 정보
  - 클래스가 계층 구조를 가질 때, 객체가 어떤 객체인지 확인하거나 객체의 자료형을 변경해야 하는 경우
    - 런타임 타입 정보(RTTI, Run-Time Type Information)를 이용

  - typeid 연산자
    - 런타임 때 객체의 자료형 확인
    - <typeinfo> 헤더에 있는 type_info 클래스 사용
    - type_info 클래스는 생성자, 소멸자, 복사 생성자가 없음
    - type_info 객체는 typeid라는 연산자를 호출해서 생성

- 동적 자료형 변환
  - 베이스 클래스에 대한 포인터에 파생 클래스의 객체 대입- 업 캐스트(upcast)

```
Shape *pShape = new Circle();  // 업캐스팅
```

  - 파생 클래스에 대한 포인터에 베이스 클래스의 객체 대입 – 다운 캐스트 (downcast)
    - dynamic_cast 연산자 사용 – virtual 함수가 없으면 오류

```
Circle *pCircle = dynamic_cast<Circle *>(pShape); //다운캐스팅
```

런타임 타입 정보와 동적 자료형 변환 예

```
#include <typeinfo>
class Shape {
public:
    void paint() { draw(); }
    virtual void draw() { cout << "Shape::draw() called" << endl; }
};
class Circle : public Shape {
public:
    virtual void draw() { cout << "Circle::draw() called" << endl; }
    void write() { cout << "Circle::write()" << endl; }
};
int main() {
    Shape *pShape = new Circle();  //업캐스팅

    if (typeid(*pShape) == typeid(Circle)) { //type_info 객체에 ==, != 연산자를 사용하여 자료형 비교
        //name() : type_info 객체의 자료형 반환, 컴파일러 마다 결과가 다름
        cout << "typeid(*pShape).name() : " << typeid(*pShape).name() << endl;

        Circle *pCircle = dynamic_cast<Circle *>(pShape); //다운캐스팅
        pCircle->write();
    }
    cout << "typeid(pShape).name() : " << typeid(pShape).name()<< endl;
   delete pShape;
}
```

visual studio 2019

```
typeid(*pShape).name() : class Circle
Circle::write()
typeid(pShape).name() : class Shape *
```

visual studio code

```
typeid(*pShape).name() : 6Circle
Circle::write()
typeid(pShape).name() : P5Shape
```

가상 함수와 오버라이딩 활용

다형성의 실현
- draw() 가상 함수를 가진 기본 클래스 Shape
- 오버라이딩을 통해 Circle, Rect, Line 클래스에서 자신만의 draw() 구현

```
class Shape {
protected:
   virtual void draw() { }
};
```

가상 함수 선언.
파생 클래스에서 재정의할 함수에 대한 **인터페이스** 역할

```
class Circle : public Shape {
protected:
   virtual void draw() {
      // Circle을 그린다.
   }
};
```

오버라이딩,
다형성 실현

```
class Rect : public Shape {
protected:
   virtual void draw() {
      // Rect을 그린다.
   }
};
```

```
class Line : public Shape {
protected:
   virtual void draw() {
      // Line을 그린다.
   }
};
```

```
void paint( Shape* p) {
   p->draw();  //p가 가리키는 객체에 오버라이딩된 draw() 호출
}

paint( new Circle() );   //Circle을 그린다.
paint( new Rect() );     //Rect을 그린다.
paint( new Line() );     //Line을 그린다.
```

순수 가상 함수

- 기본 클래스의 가상 함수 목적
  - 파생 클래스에서 재정의할 함수를 알려주는 역할
    - 실행할 코드를 작성할 목적이 아님
  - *기본 클래스의 가상 함수를 굳이 구현할 필요가 있을까?*

- 순수 가상 함수
  - pure virtual function
  - 함수의 코드가 없고 선언만 있는 가상 멤버 함수
  - 선언 방법
    - 멤버 함수의 원형=0;으로 선언

```
class Shape {
public:
    virtual void draw()=0; // 순수 가상 함수 선언
};
```

추상 클래스

- 추상 클래스 : 최소한 하나의 순수 가상 함수를 가진 클래스

```
class Shape {    //Shape은 추상 클래스
        Shape *next;
 public:
        void paint() {     draw();    }
        virtual void draw() = 0; //순수 가상 함수
};
void Shape::paint() {
   draw();    //순수 가상 함수라도 호출은 할 수 있다.
}
```

- 추상 클래스의 특징
  - 온전한 클래스가 아니므로 객체 생성 불가능

```
Shape shape;   //컴파일 오류
Shape *p = new Shape(); //컴파일 오류
```

error C2259: 'Shape' : 추상 클래스를 인스턴스화할 수 없습니다.

  - 추상 클래스의 포인터는 선언 가능

```
Shape *p;
```

추상 클래스 사용 목적

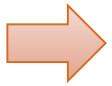
- 추상 클래스의 목적
  - 인스턴스를 생성할 목적 아님
  - 상속에서 기본 클래스의 역할을 하기 위함
    - 순수 가상 함수를 통해 파생 클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할.
      - 추상 클래스의 모든 멤버 함수를 순수 가상 함수로 선언할 필요는 없음.

추상 클래스와 상속

- 추상 클래스의 상속
  - 추상 클래스를 단순 상속하면 자동 추상 클래스
- 추상 클래스의 구현
  - 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩
    - 파생 클래스는 추상 클래스가 아님

```
class Shape { //추상 클래스
public:
   virtual void draw() = 0;
};

class Circle : public Shape { //추상클래스
public:
   string toString() { return "Circle 객체"; }
};

Shape shape; //객체 생성 오류
Circle waffle; //객체 생성 오류
```

추상 클래스의 단순 상속



```
class Shape { //추상 클래스
public:
   virtual void draw() = 0;
};
```

추상 클래스의 구현

```
class Circle : public Shape { //추상 클래스 아님
public:
    virtual void draw() { //순수 가상 함수 오버라이딩
      cout << "Circle";
   }
   string toString() { return "Circle 객체"; }
};

Shape shape;    //객체 생성 오류
Circle waffle;  //정상적인 객체 생성
```

추상 클래스 예(1)

다음 추상 클래스 Calculator를 상속받아 GoodCalc 클래스를 구현하세요.

```
class Calculator {
public:
   virtual int add(int a, int b) = 0;          //두 정수의 합 리턴
   virtual int subtract(int a, int b) = 0;     //두 정수의 차 리턴
   virtual double average(int a [], int size) = 0; //배열 a의 평균 리턴. size는 배열의 크기
};
```

```
class GoodCalc : public Calculator {
public:
  //순수 가상 함수 구현
   int add(int a, int b) { return a + b; }

   int subtract(int a, int b) { return a - b; }

   double average(int a [], int size) {
      double sum = 0;
      for(int i=0; i<size; i++)
         sum += a[i];
      return sum/size;
   }
};
```

```
int main() {
   int a[] = {1,2,3,4,5};
   Calculator *p = new GoodCalc();

   cout << p->add(2, 3) << endl;
   cout << p->subtract(2, 3) << endl;
   cout << p->average(a, 5) << endl;

   delete p;
}
```

```
5
-1
3
```

추상 클래스 예(2)

다음 코드와 실행 결과를 참고하여 추상 클래스 Calculator를 상속받는 Adder와 Subractor 클래스를 구현하세요.

```
class Calculator {
    int a, b;
    void input() {
        cout << "정수 2 개를 입력하세요>> ";  cin >> a >> b;
    }
public:
    virtual int calc(int a, int b) = 0; // 두 정수의 합 리턴
    void run() {
        input();
        cout << "계산된 값은 " << calc(a, b) << endl;
    }
};
int main() {
    Calculator *cp = new Adder();
    cp->run();
    delete cp;

    cp = new Subtractor();
    cp->run();
    delete cp;
    return 0;
}
```

```
class Adder : public Calculator {
    // 순수 가상 함수 구현
    int calc(int a, int b) {
        return a + b;
    }
};

class Subtractor : public Calculator {
    // 순수 가상 함수 구현
    int calc(int a, int b) {
        return a - b;
    }
};
```

학습 정리

- 가상 함수(virtual function)
  - virtual 키워드로 선언된 멤버 함수.
  - virtual 키워드의 의미는 컴파일러에게 함수에 대한 호출 바인딩을 실행 시간까지 미루도록 지시하는 것.
  - 객체 내에 오버라이딩 한 파생 클래스의 함수를 찾아 실행

- 소멸자를 가상함수로 선언
  - 파생클래스와 기본 클래스의 소멸자를 모두 실행.
  - 기본 클래스의 소멸자를 가상함수로 선언하여 정상적인 소멸 과정이 진행되도록 하는 것이 좋음

- 추상 클래스
  - 순수 가상 함수를 통해 파생 클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할.
  - 순수 가상 함수 : 함수의 코드가 없고 선언만 있는 가상 멤버 함수, 멤버 함수의 원형=0;으로 선언.

Q & A

- “가상함수와 추상클래스”에 대한 학습이 모두 끝났습니다.

- 새로운 내용이 많았습니다. 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.

- cpp_10_가상함수와추상클래스_ex.pdf 에 확인 학습 문제들을 담았습니다.
- 이론 학습을 완료한 후 확인 학습 문제들로 학습 내용을 점검 하시기 바랍니다.

- 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.

- 수고하셨습니다.^^