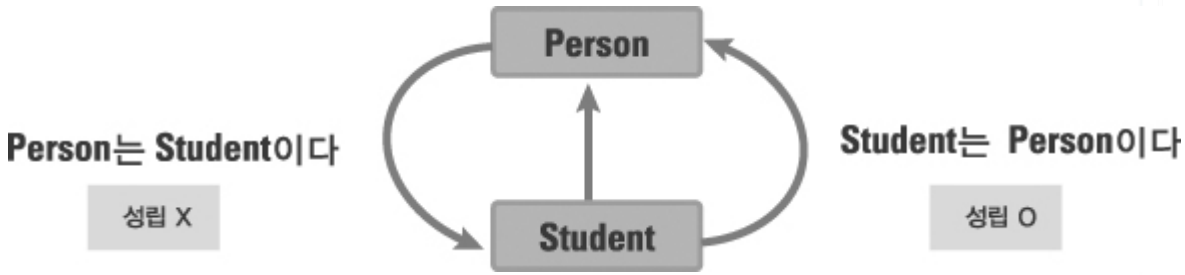
8.상속과 다형성 (polymorphism)

- 상속된 객체와 포인터/참조자의 관계
- 정적 바인딩과 동적 바인딩
- virtual 소멸자

Jong Hyuk Park

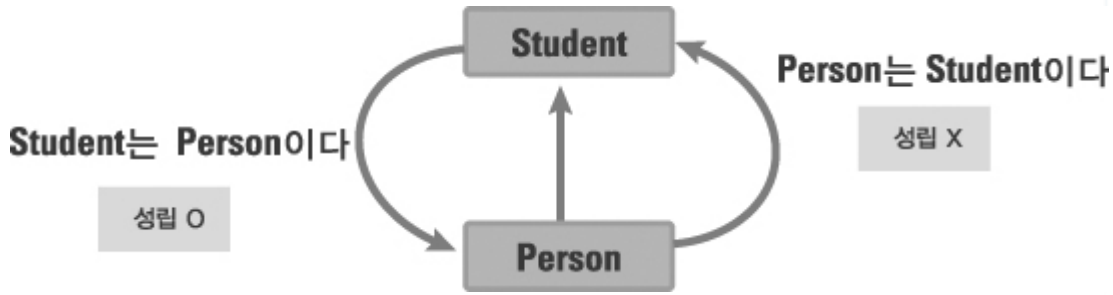
상속의 조건

❖ public 상속은 is-a 관계가 성립되도록 하자.



상속의 조건

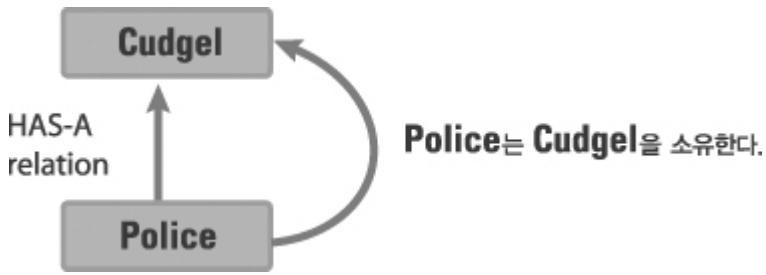
❖ 잘못된 상속의 예



상속의 조건

❖HAS-A(소유) 관계에 의한 상속!

- 경찰은 몽둥이를 소유한다
- The Police have a cudgel.
- hasa1.cpp



상속의 조건

❖HAS-A에 의한 상속 그리고 대안!

- 포함 관계를 통해서 소유 관계를 표현
- 객체 멤버에 의한 포함 관계의 형성
- 객체 포인터 멤버에 의한 포함 관계의 형성
- hasa2.cpp, hasa3.cpp

Police 객체는 Cudgel 객체를 포함한다.

상속의 조건

```
/*    hasa2.cpp */
class Cudgel   //몽둥이
{
public:
      void Swing(){ cout<<"Swing a cudgel!"<<endl; }
};
class Police //몽둥이를 소유하는 경찰
{
      Cudgel cud;
public:
      void UseWeapon(){ cud.Swing(); }
};

int main()
{
      Police pol;
      pol.UseWeapon();
      return 0;
}
```

상속의 조건

```
/*    hasa3.cpp */
class Cudgel  //몽둥이
{
public:
      void Swing(){ cout<<"Swing a cudgel!"<<endl; }
};
class Police //몽둥이를 소유하는 경찰
{
      Cudgel* cud;
public:
      Police(){ cud=new Cudgel; }
      ~Police(){ delete cud; }
      void UseWeapon(){ cud->Swing(); }
};
int main()
{
      Police pol;
      pol.UseWeapon();
      return 0;
}
```

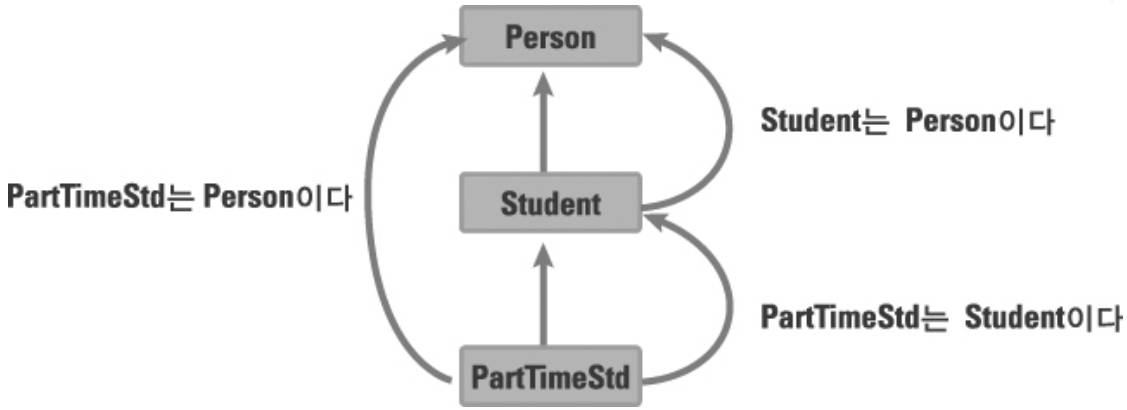
상속된 객체와 포인터/참조자의 관계

Jong Hyuk Park

상속된 객체와 포인터 관계

❖ 객체 포인터

- 객체의 주소 값을 저장할 수 있는 포인터



상속된 객체의 포인터,참조자

❖ 객체의 포인터(참조자)

- AAA 클래스의 포인터는 AAA 객체의 주소 뿐만 아니라 AAA 클래스를 상속하는 파생 클래스의 객체의 주소 값도 저장 가능
- AAA 클래스의 참조자는 AAA 객체 뿐만 아니라 AAA 클래스를 상속하는 파생 클래스의 객체 도 참조 가능

❖ 객체 포인터(참조자)의 권한

- 포인터를 통해서 접근할 수 있는 객체 멤버의 영역
- AAA 클래스의 객체 포인터는 AAA 클래스의 멤버와 AAA 클래스가 상속받은 부모 클래스의 멤버만 접근 가능
- AAA 클래스의 참조자는 AAA 클래스의 멤버와 AAA 클래스가 상속받은 부모 클래스의 멤버만 접근 가능

객체의 포인터 예

```
#include <iostream>
using std::endl;
using std::cout;
class Person
{
public:
	void Sleep(){
	cout<<"Sleep"<<endl;
	}
};
class Student : public Person
{
public:
	void Study(){
	cout<<"Study"<<endl;
	}
};
class PartTimeStd : public Student
{
public:
	void Work(){
	cout<<"Work"<<endl;
	}
};
```

```
int main(void)
{
	Person* p1=new Person;
	Person* p2=new Student;
	Person* p3=new PartTimeStd;

	p1->Sleep();
	p2->Sleep();
	p3->Sleep();

	return 0;
}
```

```
int main(void)
{
	Person* p1=new PartTimeStd;
	Student* p2=new PartTimeStd;
	PartTimeStd * p3=new PartTimeStd;

	p1->Sleep();
	p2->Sleep();
	p3->Sleep();

	return 0;
}
```

객체 포인터의 권한 예

```
#include <iostream>
using std::endl;
using std::cout;

class Person
{
public:
	void Sleep(){
		cout<<"Sleep"<<endl;
	}
};

class Student : public Person
{
public:
	void Study(){
		cout<<"Study"<<endl;
	}
};

class PartTimeStd : public Student
{
public:
	void Work(){
		cout<<"Work"<<endl;
	}
};
```

```
int main(void)
{
	Person* p3=new PartTimeStd;

	p3->Sleep();
	p3->Study();    // ! 에러
	p3->Work();     // ! 에러

	return 0;
}
```

객체의 참조자 예

```
#include <iostream>
using std::endl;
using std::cout;
class Person
{
public:
	void Sleep(){
	cout<<"Sleep"<<endl;
	}
};
class Student : public Person
{
public:
	void Study(){
	cout<<"Study"<<endl;
	}
};
class PartTimeStd : public Student
{
public:
	void Work(){
	cout<<"Work"<<endl;
	}
};
```

```
int main(void)
{
	PartTimeStd   p;
	Student& ref1=p;
	 Person & ref2=p;

	p.Sleep();
	ref1.Sleep();
	ref2.Sleep();

	return 0;
}
```

객체 참조자의 권한 예

```
#include <iostream>
using std::endl;
using std::cout;

class Person
{
public:
	void Sleep(){
		cout<<"Sleep"<<endl;
	}
};

class Student : public Person
{
public:
	void Study(){
		cout<<"Study"<<endl;
	}
};

class PartTimeStd : public Student
{
public:
	void Work(){
		cout<<"Work"<<endl;
	}
};
```

```
int main(void)
{
	PartTimeStd p;
	Person& ref=p;

	ref.Sleep();
	ref.Study();    // ! 에러
	ref.Work();     // ! 에러

	return 0;
}
```

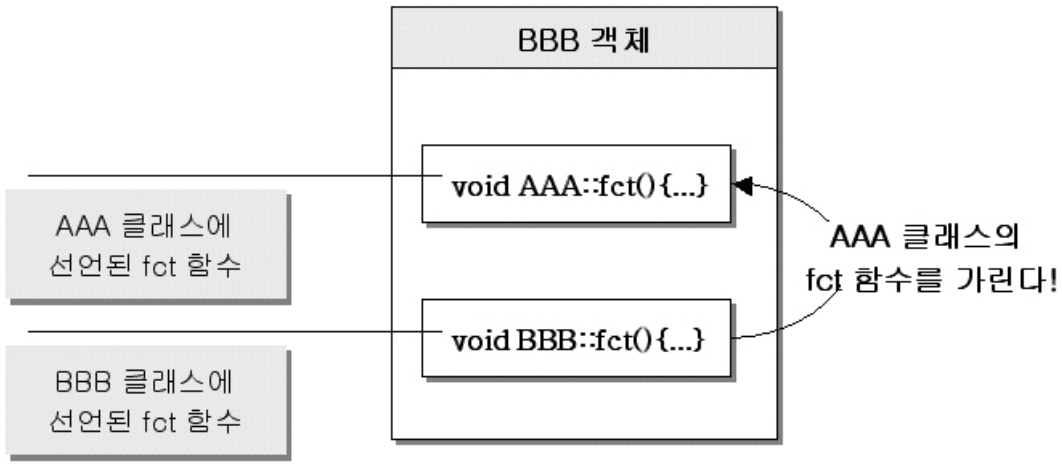
정적 바인딩과 동적 바인딩

Jong Hyuk Park

오버라이딩(overriding)

❖ 오버라이딩(Overriding)의 이해

- Base 클래스에 선언된 멤버와 같은 형태의 멤버를 Derived 클래스에서 선언
- Base 클래스의 멤버를 가리는 효과!
- 보는 시야(Pointer)에 따라서 달라지는 효과!
- Overriding1.cpp, Overriding2.cpp



오버라이딩(overriding)

❖ 베이스 클래스에서 선언된 함수를 파생 클래스에서 다시 선언

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        void fct(){
        cout<<"AAA"<<endl;
        }
};

class BBB : public AAA
{
public:
        void fct(){
        cout<<"BBB"<<endl;
        }
};
```

```
int main(void)
{
        BBB b;

        b.fct();

        return 0;
}
```

```
BBB
```

오버라이딩 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        void fct(){
        cout<<"AAA"<<endl;
        }
};

class BBB : public AAA
{
public:
        void fct(){
        cout<<"BBB"<<endl;
        }
};
```

```
int main(void)
{
        BBB* b=new BBB;
        b->fct();

        AAA* a=b;
        a->fct();

        delete b;
        return 0;
}
```

```
BBB
AAA
```

가상함수(virtual function)

- **가상함수**는 베이스 클래스 내에서 정의된 멤버함수를 파생 클래스에서 재정의하고자 할 때 사용
  - 베이스 클래스의 멤버함수와 같은 이름을 갖는 함수를 파생 클래스에서 재정의함으로써 각 클래스마다 고유의 기능을 갖도록 변경할 때 이용
  - 파생 클래스에서 재정의되는 가상함수는 함수 중복과 달리 베이스 클래스와 함수의 반환형, 인수의 갯수, 형이 같아야 함

- 가상함수를 정의하기 위해서는 가장 먼저 기술되는 상위 클래스(베이스 클래스)의 멤버함수 앞에 **virtual**이라는 키워드로 기술

가상함수 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        virtual void fct(){  // 가상함수
        cout<<"AAA"<<endl;
        }
};

class BBB : public AAA
{
public:
        void fct(){
        cout<<"BBB"<<endl;
        }
};
```

```
int main(void)
{
        BBB* b=new BBB;
        b->fct();

        AAA* a=b;
        a->fct();

        delete b;
        return 0;
}
```

```
BBB
BBB
```

가상함수 특성 상속 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        virtual void fct(){  // 가상함수
        cout<<"AAA"<<endl;
        }
};
class BBB : public AAA
{
public:
        void fct(){     // virtual void fct()
        cout<<"BBB"<<endl;
        }
};
class CCC : public BBB
{
public:
        void fct(){
        cout<<"CCC"<<endl;
        }
};
```

```
int main(void)
{
        BBB* b=new BBB;
        b->fct();

        AAA* a=b;
        a->fct();

        delete b;
        return 0;
}
```

```
CCC
CCC
```

바인딩(binding)과 다형성

- 바인딩
  - 정적 바인딩(static binding)
    - 컴파일 시(compile-time) 호출되는 함수를 결정
  - 동적 바인딩(dynamic binding)
    - 실행 시(run-time) 호출되는 함수를 결정

- **다형성(polymorphism)**
  - 같은 모습의 형태가 다른 특성
  - a->fct() 예
    - a라는 포인터(모습)가 가리키는 대상에 따라 호출되는 함수(형태)가 다름
  - 함수 오버로딩, 동적 바인딩 등이 다형성의 예

바인딩 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        virtual void fct(){  // 가상함수
        cout<<"AAA"<<endl;
        }
};

class BBB : public AAA
{
public:
        void fct(){
        cout<<"BBB"<<endl;
        }
};
```

```
int main(void)
{
        BBB b;
        b.fct();    // 정적 바인딩

        AAA* a=new BBB;
        a->fct();   // 동적 바인딩

        delete b;
        return 0;
}`
```

```
BBB
BBB
```

오버라이딩된 함수 호출 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
        virtual void fct(){
        cout<<"AAA"<<endl;
        }
};

class BBB : public AAA
{
public:
        void fct(){
        AAA::fct();  // 방법 1.
        cout<<"BBB"<<endl;
        }
};
```

```
int main(void)
{
        AAA* a=new BBB;

        cout<<"첫 번째 시도"<<endl;
        a->fct();

        cout<<"두 번째 시도"<<endl;
        a->AAA::fct(); // 방법 2.

        return 0;
}
```

```
첫 번째 시도
AAA
BBB
두 번째 시도
AAA
```

순수 가상함수

❖ 순수 가상함수(pure virtual function)

- 베이스 클래스에서는 어떤 동작도 정의되지 않고 함수의 선언만을 하는 가상함수
- 순수 가상함수를 선언하고 파생 클래스에서 이 가상함수를 중복 정의하지 않으면 컴파일 시에 에러가 발생
- 하나 이상의 멤버가 순수 가상함수인 클래스를 추상 클래스(abstract class)라 함
  - 완성된 클래스가 아니기 때문에 객체화되지 않는 클래스
- 베이스 클래스에서 다음과 같은 형식으로 선언

```
virtual   자료형 함수명(인수 리스트) = 0;
```

순수 가상 함수 예 (1)

```
#include <iostream>
using std::endl;
using std::cout;
class Date {              // 베이스 클래스
protected:`
      int year,month,day;
public:
      Date(int y,int m,int d)
            {  year = y; month = m; day = d; }
      virtual void print() = 0;  // 순수 가상함수
};
class Adate : public Date {
                              // 파생 클래스 Adate
public:
       Adate(int y,int m,int d) : Date(y,m,d)
             { /* no operation */ }
       void print()                    // 가상함수
           { cout << year << '.' << month << '.' << day << ".₩n"; }
};
class Bdate : public Date {
                    // 파생 클래스 Bdate
public:
       Bdate(int y,int m,int d) : Date(y,m,d)
           { /* no operation */ }
       void print();           // 가상함수
};
```

```
void Bdate::print()
{
   static char *mn[] = {
       "Jan.", "Feb.", "Mar.", "Apr.", "May",
       "June","July", "Aug.", "Sep.", "Oct.",
       "Nov.","Dec." };
   cout << mn[month-1] << ' ' << day
          << ' ' << year << '₩n';
}

int main()
{
   Adate a(1994,6,1);
   Bdate b(1945,8,15);
   Date &r1 = a, &r2 = b;  // 참조자
   r1.print();
   r2.print();
   return 0;
}
```

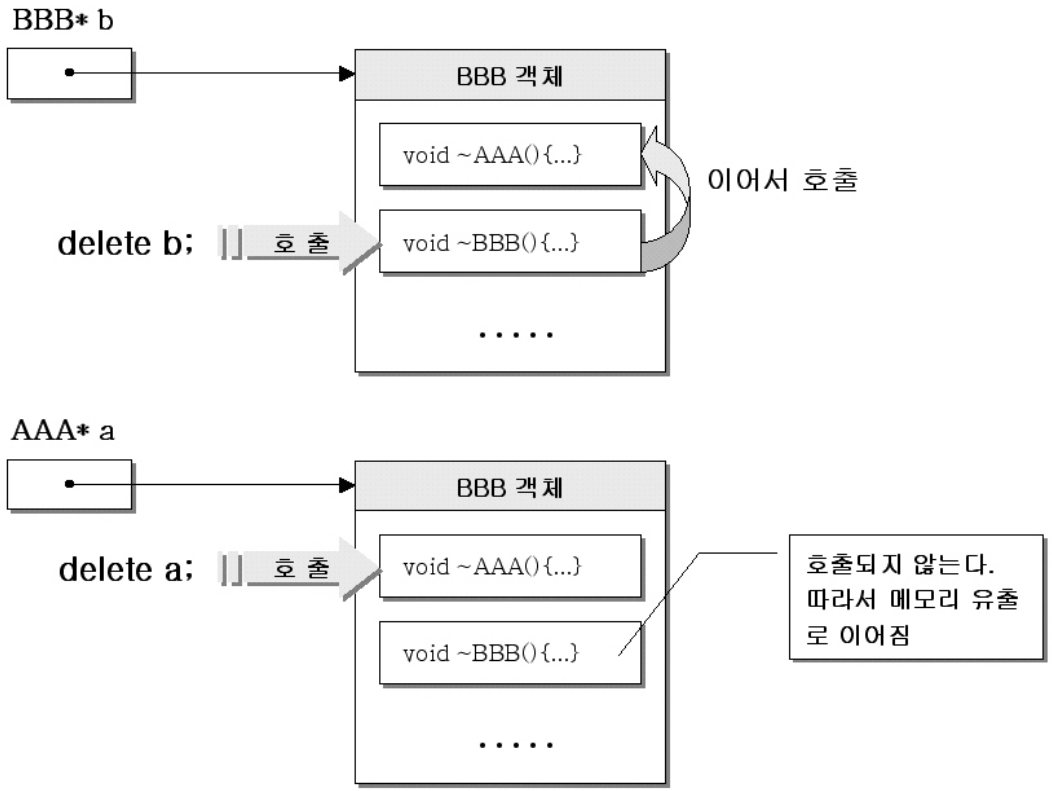
```
1994.6.1.
Aug. 15 1945
```

virtual 소멸자

Jong Hyuk Park

Virtual 소멸자의 필요성

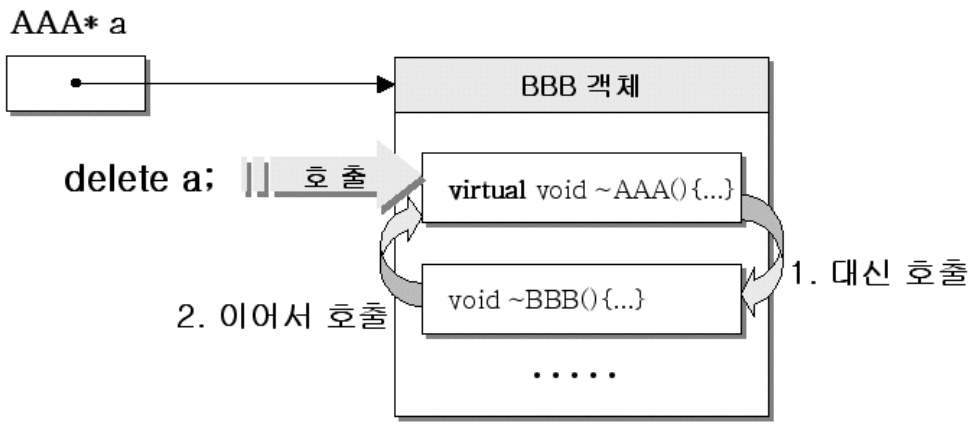
❖ 상속하고 있는 클래스 객체 소멸 문제점



Virtual 소멸자의 필요성

❖ virtual 소멸자

```
virtual ~AAA(){
cout<<"~AAA() call!"<<endl;
delete []str1;
}
```



virtual 소멸자

- 파생클래스를 가리키는 베이스클래스의 포인터가 가리키는 객체의 소멸 시에는 파생 클래스의 소멸자를 호출하지 않음

- virtual 소멸자
  - 객체 소멸 시 베이스클래스 뿐만 아니라 파생클래스의 소멸자도 호출
  - 소멸자 앞에 virtual 키워드

virtual 소멸자 필요성 예

```
#include <iostream>
using std::endl;
using std::cout;
class AAA
{
        char* str1;
public:
        AAA(char* _str1){
                str1= new char[strlen(_str1)+1];
                strcpy(str1, _str1);
        }
        ~AAA(){
                cout<<"~AAA() call!"<<endl;
                delete []str1;
        }
        virtual void ShowString(){
                cout<<str1<<' ';
        }
};
class BBB : public AAA
{
        char* str2;
public:
        BBB(char* _str1, char* _str2) : AAA(_str1){
                str2= new char[strlen(_str2)+1];
                strcpy(str2, _str2);
        }
        ~BBB(){
        cout<<"~BBB() call!"<<endl;
        delete []str2;
        }
        virtual void ShowString(){
        AAA::ShowString();
        cout<<str2<<endl;
        }
};
int main()
{
        AAA * a=new BBB("Good", "evening");
        BBB * b=new BBB("Good", "morning");
        a->ShowString();
        b->ShowString();
        cout<<"-----객체 소멸 직전----"<<endl;
        delete a;   // AAA 소멸자만 호출
        delete b;   // BBB,AAA 소멸자 호출
        return 0;
}
```

```
Good evening
Good morning
-----객체 소멸 직전----
~AAA() call!
~BBB() call!
~AAA() call!
```

virtual 소멸자 예

```
#include <iostream>
using std::endl;
using std::cout;
class AAA
{
        char* str1;
public:
        AAA(char* _str1){
                str1= new char[strlen(_str1)+1];
                strcpy(str1, _str1);
        }
        virtual ~AAA(){
                cout<<"~AAA() call!"<<endl;
                delete []str1;
        }
        virtual void ShowString(){
                cout<<str1<<' ';
        }
};
class BBB : public AAA
{
        char* str2;
public:
        BBB(char* _str1, char* _str2) : AAA(_str1){
                str2= new char[strlen(_str2)+1];
                strcpy(str2, _str2);
        }
        ~BBB(){
        cout<<"~BBB() call!"<<endl;
        delete []str2;
        }
        virtual void ShowString(){
        AAA::ShowString();
        cout<<str2<<endl;
        }
};
int main()
{
        AAA * a=new BBB("Good", "evening");
        BBB * b=new BBB("Good", "morning");
        a->ShowString();
        b->ShowString();
        cout<<"-----객체 소멸 직전----"<<endl;
        delete a;   // BBB, AAA 소멸자만 호출
        delete b;   // BBB,AAA 소멸자 호출
        return 0;
}
```

```
Good evening
Good morning
-----객체 소멸 직전----
~BBB() call!
~AAA() call!
~BBB() call!
~AAA() call!
```

프로젝트 과제

❖ 8-8절의 은행계좌 관리 과제 프로그램에 다음이 추가되었다.

- Account 클래스를 상속 받는 두 개의 새로운 계좌 도입
  - 신용계좌, FaithAccount: 우수고객 계좌로 계좌입금 시 1% 이자가 추가
  - 기부계좌, ContriAccount: 입금금액의 1% 금액을 사회기부

❖ 아래의 계좌를 추가하라.

- "적금계좌, InstallAccount"
  - 적금계좌의 고객은 반드시 일반계좌를 가지고 있어야 함
  - 계좌 id는 일반계좌와 같고, Account 에 적금계좌 여부 표시변수 도입
- "적금이체" 동작 추가
  - 일반계좌에서 지정된 금액을 적금계좌로 이체


Q & A
Jong Hyuk Park