상속과 다형성

박 종 혁 교수

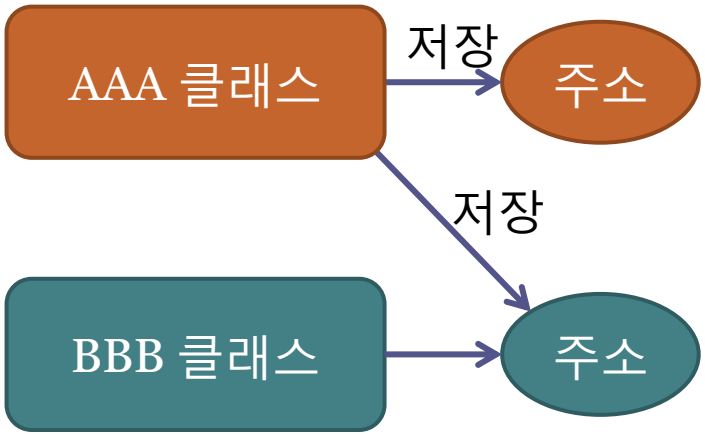
UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

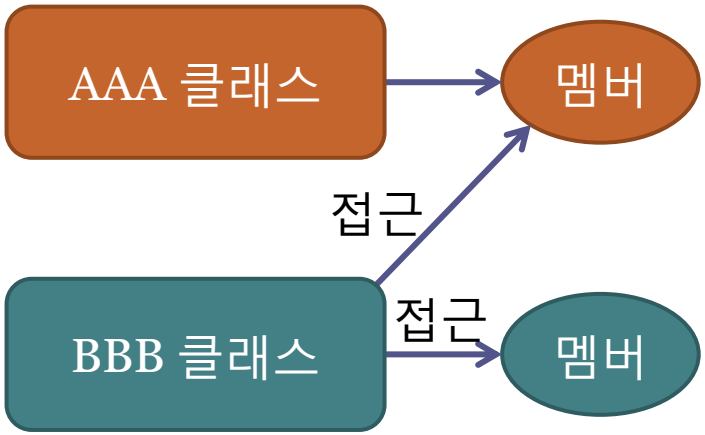
객체 포인터의 참조관계

상속된 객체와 포인터 관계



- 객체 포인터
  - 객체의 주소 값을 저장할 수 있는 포인터
  - AAA 클래스의 포인터는 AAA 객체의 주소 뿐만 아니라 AAA 클래스를 상속하는 Derived 클래스의 객체의 주소 값도 저장 가능
  - AAA 클래스의 참조자는 AAA 객체 뿐만 아니라 AAA 클래스를 상속하는 Derived 클래스의 객체 도 참조 가능

상속된 객체의 포인터,참조자



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

상속된 객체와 참조 관계

- 객체의 레퍼런스
  - 객체를 참조할 수 있는 레퍼런스
  - 클래스 포인터의 특성과 일치

- 객체 레퍼런스의 권한
  - 객체를 참조하는 레퍼런스의 권한
  - 클래스 포인터의 권한과 일치

객체의 주소 값을 저장하는 객체 포인터 변수

" C++에서, **AAA형 포인터 변수는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 가리킬 수 있다**(객체의 주소 값을 저장할 수 있다)."

```
class Student : public Person
{
     . . . . .
};

class PartTimeStudent : public Student
{
     . . . . .
};
```

Person * ptr=new Student();

Person * ptr=new PartTimeStudent();

Student * ptr=new PartTimeStudent();

Ex) ObejctPointer.cpp

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

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

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

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

int main(void)
{
	Person * ptr1=new Student();
	Person * ptr2=new PartTimeStudent();
	Student * ptr3=new PartTimeStudent();
	ptr1->Sleep();
	ptr2->Sleep();
	ptr3->Study();
	delete ptr1; delete ptr2; delete ptr3;
	return 0;
}
```

유도 클래스의 객체도 가리키는 포인터!

IS-A 관계

"학생(Student)은 사람(Person)의 일종이다."

"근로학생(PartTimeStudent)은 학생(Student)의 일종이다."

"근로학생(PartTimeStudent)은 사람(Person)의 일종이다."

유도 클래스 객체를 기초 클래스 객체로 바라볼 수 있는 근거

"학생(Student)은 사람(Person)이다."

"근로학생(PartTimeStudent)은 학생(Student)이다."

"근로학생(PartTimeStudent)은 사람(Person)이다."

문제의 제시를 위한 시나리오의 도입

```
class PermanentWorker
{
private:
    char name[100];
    int salary;     // 매달 지불해야 하는 급여액
public:
    PermanentWorker(char* name, int money)
        : salary(money)
    {
        strcpy(this->name, name);
    }
    int GetPay() const
    {
        return salary;
    }
    void ShowSalaryInfo() const
    {
        cout<<"name: "<<name<<endl;
        cout<<"salary: "<<GetPay()<<endl<<endl;
    }
};
```

PermanentWorker는 정규직을 표현해 놓은 클래스이다.

```
class EmployeeHandler
{
private:
    PermanentWorker* empList[50];
    int empNum;
public:
    EmployeeHandler() : empNum(0)
    { }
    void AddEmployee(PermanentWorker* emp)
    {
        empList[empNum++]=emp;
    }
    void ShowAllSalaryInfo() const
    {
        for(int i=0; i<empNum; i++)
            empList[i]->ShowSalaryInfo();
    }
    void ShowTotalSalary() const
    {
        int sum=0;
        for(int i=0; i<empNum; i++)
            sum+=empList[i]->GetPay();
        cout<<"salary sum: "<<sum<<endl;
    }
    ~EmployeeHandler()
    {
        for(int i=0; i<empNum; i++)
            delete empList[i];
    }
};
```

프로그램 전체 기능의 처리를, 프로그램의 흐름을 담당하는 클래스를 가리켜 컨트롤 클래스라 한다.

EmployeeHandler의 경우 컨트롤 클래스에 해당한다.

신규 직원 등록 시

전체 급여정보 출력

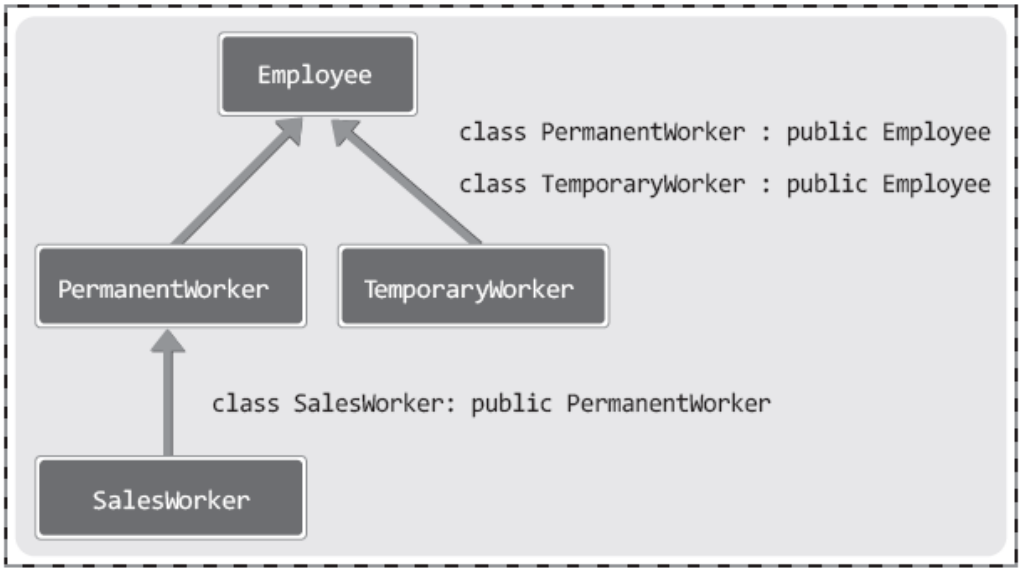
급여 합계 정보 출력

오렌지미디어 급여관리 확장성 문제 1차 해결

- 고용인 Employee
- 정규직 PermanentWorker
- 영업직 SalesWorker
- 임시직 TemporaryWorker

"정규직, 영업직, 임시직 모두 고용의 한 형태이다(고용인이다)."

"영업직은 정규직의 일종이다."



모든 클래스의 객체를 Employee 클래스의 객체로 간주(처리)할 수 있는 기반을 마련.

컨트롤 클래스 입장에서는 모든 객체를 Employee 객체로 간주해도 문제가 되지 않는다!

EmployeeHandler의 첫 번째 수정

```
class EmployeeHandler
{
private:
    Employee* empList[50];
    int empNum;
public:
    EmployeeHandler() : empNum(0)
    { }
    void AddEmployee(Employee* emp)
    {
        empList[empNum++]=emp;
    }
    void ShowAllSalaryInfo() const
    {
     /* for(int i=0; i<empNum; i++)
            empList[i]->ShowSalaryInfo();    */
    }
    void ShowTotalSalary() const
    {
        int sum=0;
     /*  for(int i=0; i<empNum; i++)
                sum+=empList[i]->GetPay();      */
        cout<<"salary sum: "<<sum<<endl;
    }
    ~EmployeeHandler()
    {
        for(int i=0; i<empNum; i++)
            delete empList[i];
    }
};
```

```
class Employee
{
private:
    char name[100];
public:
    Employee(char * name)
    {
        strcpy(this->name, name);
    }
    void ShowYourName() const
    {
        cout<<"name: "<<name<<endl;
    }
};
```

```
class PermanentWorker : public Employee
{
private:
    int salary;     // 월 급여
public:
    PermanentWorker(char* name, int money)
        : Employee(name), salary(money)
    { }
    int GetPay() const
    {
        return salary;
    }
    void ShowSalaryInfo() const
    {
        ShowYourName();
        cout<<"salary: "<<GetPay()<<endl<<endl
    }
};
```

왼쪽의 EmployeeHandler 클래스는 Employee 객체를 처리하는 컨트롤 클래스로 변경되었다.

Ex) EmployeeManager2.cpp

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

class Employee
{
private:
	char name[100];
public:
	Employee(char * name)
	{
		strcpy(this->name, name);
	}
	void ShowYourName() const
	{
		cout<<"name: "<<name<<endl;
	}
};

class PermanentWorker : public Employee
{
private:
	int salary;
public:
	PermanentWorker(char* name, int money)
		: Employee(name), salary(money)
	{ }
	int GetPay() const
	{
		return salary;
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class EmployeeHandler
{
private:
	Employee* empList[50];
	int empNum;
public:
	EmployeeHandler() : empNum(0)
	{ }
	void AddEmployee(Employee* emp)
	{
		empList[empNum++]=emp;
	}
	void ShowAllSalaryInfo() const
	{
		/*
		for(int i=0; i<empNum; i++)
			empList[i]->ShowSalaryInfo();
		*/
	}
	void ShowTotalSalary() const
	{
		int sum=0;
		/*
		for(int i=0; i<empNum; i++)
			sum+=empList[i]->GetPay();
		*/
		cout<<"salary sum: "<<sum<<endl;
	}
	~EmployeeHandler()
	{
		for(int i=0; i<empNum; i++)
			delete empList[i];
	}
};
```

```
int main(void)
{
		EmployeeHandler handler;

		// 직원 등록
		handler.AddEmployee(new
PermanentWorker("KIM", 1000));
		handler.AddEmployee(new
PermanentWorker("LEE", 1500));
		handler.AddEmployee(new
PermanentWorker("JUN", 2000));

		handler.ShowAllSalaryInfo();


		handler.ShowTotalSalary();
		return 0;
}
```

임시직: TemporaryWorker

```
class TemporaryWorker : public Employee
{
private:
    int workTime;      // 이 달에 일한 시간의 합계
    int payPerHour;    // 시간당 급여
public:
    TemporaryWorker(char * name, int pay)
            : Employee(name), workTime(0), payPerHour(pay)
    { }
    void AddWorkTime(int time)   // 일한 시간의 추가
    {
        workTime+=time;
    }
    int GetPay() const    // 이 달의 급여
    {
        return workTime*payPerHour;
    }
    void ShowSalaryInfo() const
    {
        ShowYourName();
        cout<<"salary: "<<GetPay()<<endl<<endl;
    }
};
```

- 임시직 급여

  '시간당 급여 × 일한 시간' 의 형태

영업직: SalesWorker

```
class SalesWorker : public PermanentWorker
{
private:
    int salesResult;      // 월 판매실적
    double bonusRatio;    // 상여금 비율
public:
    SalesWorker(char * name, int money, double ratio)
        : PermanentWorker(name, money), salesResult(0), bonusRatio(ratio)
    { }
    void AddSalesResult(int value)
    {
        salesResult+=value;
    }
    int GetPay() const
    {
        return PermanentWorker::GetPay()    // PermanentWorker의 GetPay 함수 호출
                    + (int)(salesResult*bonusRatio);
    }
    void ShowSalaryInfo() const
    {
        ShowYourName();
        cout<<"salary: "<<GetPay()<<endl<<endl;    // SalesWorker의 GetPay 함수가 호출됨
    }
};
```

- 영업직 급여

'기본급여(월 기본급여) + 인센티브' 의 형태

PermanentWorker 클래스의 GetPay 함수를 오버라이딩!

PermanentWorker 클래스의 ShowSalaryInfo 함수 오버라이딩!

함수 오버라이딩(Function Overriding)

> *"어! **PermanentWorker** 클래스에도 **GetPay** 함수와 **ShowSalaryInfo** 함수가 있는데, 유도 클래스인 **SalewWorker** 클래스에서도 동일한 이름과 형태로 두 함수를 정의하였네."*

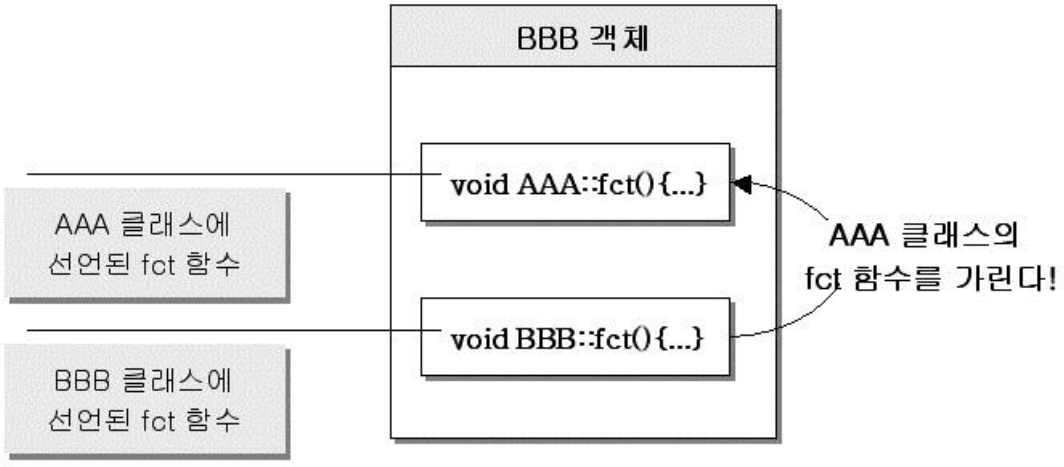
- '함수 오버라이딩(function overriding)
  - 기초 클래스에서 정의한 함수를 유도 클래스에서 동일한 이름과 형태로 함수를 정의하여 사용
    - 주의 : 함수 오버로딩과 혼동 금지
  - 함수 오버라이딩 되면, 오버라이딩 된 기초 클래스의 함수는 오버라이딩을 한 유도 클래스의 함수에 가려짐
  - 위의 SalesWorker 클래스 내에서 GetPay 함수를 호출하면 SalesWorker 클래스에 정의된 GetPay 함수가 호출됨

함수 오버라이딩 vs 함수 오버로딩

- 기초 클래스와 동일한 이름의 함수를 유도 클래스에서 정의한다고 해서 무조건 함수 오버라이딩이 되는 것은 아님.
- 함수 오버로딩
  - 매개변수의 자료형 및 개수가 다를 경우
  - 전달되는 인자에 따라서 호출되는 함수가 결정
  - 함수 오버로딩은 상속의 관계어서도 구성 가능

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

Overriding1.cpp

```
#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

Overriding2.cpp

```
#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
```

Ex) EmployeeManager3.cpp

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

class Employee
{
private:
	char name[100];
public:
	Employee(char * name)
	{
		strcpy(this->name, name);
	}
	void ShowYourName() const
	{
		cout<<"name: "<<name<<endl;
	}
};

class PermanentWorker : public Employee
{
private:
	int salary;
public:
	PermanentWorker(char * name, int money)
		: Employee(name), salary(money)
	{ }
	int GetPay() const
	{
		return salary;
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class TemporaryWorker : public Employee
{
private:
	int workTime;
	int payPerHour;
public:
	TemporaryWorker(char * name, int pay)
		: Employee(name), workTime(0), payPerHour(pay)
	{ }
	void AddWorkTime(int time)
	{
		workTime+=time;
	}
	int GetPay() const
	{
		return workTime*payPerHour;
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class SalesWorker : public PermanentWorker
{
private:
	int salesResult;	// 월 판매실적
	double bonusRatio;	// 상여금 비율
public:
	SalesWorker(char * name, int money, double ratio)
		: PermanentWorker(name, money), salesResult(0), bonusRatio(ratio)
	{ }
	void AddSalesResult(int value)
	{
		salesResult+=value;
	}
	int GetPay() const
	{
		return PermanentWorker::GetPay()	// PermanentWorker의 GetPay 함
	수 호출
		+
	(int)(salesResult*bonusRatio);
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class EmployeeHandler
{
private:
	Employee* empList[50];
	int empNum;
public:
	EmployeeHandler() : empNum(0)
	{ }
	void AddEmployee(Employee* emp)
	{
		empList[empNum++]=emp;
	}
	void ShowAllSalaryInfo() const
	{
		/*
		for(int i=0; i<empNum; i++)
			empList[i]->ShowSalaryInfo();
		*/
	}
	void ShowTotalSalary() const
	{
		int sum=0;
		/*
		for(int i=0; i<empNum; i++)
			sum+=empList[i]->GetPay();
		*/
		cout<<"salary sum: "<<sum<<endl;
	}
	~EmployeeHandler()
	{
		for(int i=0; i<empNum; i++)
			delete empList[i];
	}
};
```

```
int main(void)
{
	// 직원관리를 목적으로 설계된 컨트롤 클래스의 객체생성
	EmployeeHandler handler;

	// 정규직 등록
	handler.AddEmployee(new PermanentWorker("KIM", 1000));
	handler.AddEmployee(new PermanentWorker("LEE", 1500));

	// 임시직 등록
	TemporaryWorker * alba=new TemporaryWorker("Jung", 700);
	alba->AddWorkTime(5);			// 5시간 일한결과 등록
	handler.AddEmployee(alba);

	// 영업직 등록
	SalesWorker * seller=new SalesWorker("Hong", 1000, 0.1);
	seller->AddSalesResult(7000);		// 영업실적 7000
	handler.AddEmployee(seller);

	// 이번 달에 지불해야 할 급여의 정보
	handler.ShowAllSalaryInfo();

	// 이번 달에 지불해야 할 급여의 총합
	handler.ShowTotalSalary();
	return 0;
}
```

가상함수(Virtual Function)

가상함수(virtual function)

- **가상함수**는 베이스클래스 내에서 정의된 멤버함수를 파생클래스에서 재정의하고자 할때 사용
  - 베이스클래스의 멤버함수와 같은 이름을 갖는 함수를 파생클래스에서 재정의함으로써 각 클래스마다 고유의 기능을 갖도록 변경할때 이용
  - 파생클래스에서 재정의되는 가상함수는 함수중복과 달리 베이스클래스와 **함수의 반환형, 인수의 갯수, 형이 같아야함**
- 가상함수를 정의하기 위해서는 가장 먼저 기술되는 상위 클래스(베이스클래스)의 멤버 함수 앞에 **virtual**이라는 키워드로 기술

가상함수 예

Overriding3.cpp

```
#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

Overriding4.cpp

```
#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
```

기초 클래스의 포인터로 객체를 참조하면,

C++ 컴파일러는 포인터 연산의 가능성 여부를 판단할 때, **포인터의 자료형을 기준으로 판단하지, 실제 가리키는 객체의 자료형을 기준으로 판단하지 않는다.**

```
class Base
{
public:
    void BaseFunc() { cout<<"Base Function"<<endl; }
};

class Derived : public Base
{
public:
    void DerivedFunc() { cout<<"Derived Function"<<endl; }
};
```

```
int main(void)
{
    Base * bptr=new Derived();    // 컴파일 OK!
    bptr->DerivedFunc();          // 컴파일 Error!
    . . . .
}
```

```
int main(void)
{
    Base * bptr=new Derived();    // 컴파일 OK!
    Derived * dptr=bptr;          // 컴파일 Error!
    . . . .
}
```

```
int main(void)
{
    Derived * dptr=new Derived();    // 컴파일 OK!
    Base * bptr=dptr;          // 컴파일 OK!
    . . . .
}
```

앞서 한 이야기의 복습

"C++ 컴파일러는 포인터를 이용한 연산의 가능성 여부를 판단할 때, 포인터의 자료형을 기준으로 판단하지, 실제 가리키는 객체의 자료형을 기준으로 판단하지 않는다." 따라서 포인터 형에 해당하는 클래스의 멤버에만 접근이 가능하다.

```
class First
{
public:
    void FirstFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void SecondFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void ThirdFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    tptr->FirstFunc();    (○)
    tptr->SecondFunc();   (○)
    tptr->ThirdFunc();    (○)

    sptr->FirstFunc();    (○)
    sptr->SecondFunc();   (○)
    sptr->ThirdFunc();    (×)

    fptr->FirstFunc();    (○)
    fptr->SecondFunc();   (×)
    fptr->ThirdFunc();    (×)
    . . . .
}
```

예제 EmployeeManager2.cpp와 EmployeeManager3.cpp의 주석처리 부분에서 컴파일 에러가 발생하는 이유는?

함수의 오버라이딩과 포인터 형

```
class First
{
public:
    void MyFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void MyFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void MyFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}
```

```
FirstFunc
SecondFunc
ThirdFunc
```

실행결과

함수를 호출할 때 사용이 된 포인터의 형에 따라서 호출되는 함수가 결정된다!

포인터의 형에 정의된 함수가 호출된다.

가상함수(Virtual Function)

```
class First
{
public:
    virtual void MyFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    virtual void MyFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    virtual void MyFunc() { cout<<"ThirdFunc"<<endl; }
};
```

오버라이딩 된 함수가 virtual이면
오버라이딩 한 함수도 자동 virtual

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}
```

```
ThirdFunc
ThirdFunc
ThirdFunc
```

실행결과

포인터의 형에 상관 없이 포인터가 가리키는 객체의 마지막 오버라이딩 함수를 호출한다.

현 상황에서의 EmployeeManager 클래스는 모든 객체를 Employee 객체로 간주한다. 따라서 호출하는 함수도 Employee 객체의 멤버함수이다! 바로 이러한 문제의 해결책이 위의 예제에 있다!

급여관리 확장성 문제의 해결과 상속의 이유

```
class Employee
{
private:
    char name[100];
public:
    Employee(char * name)
    {
        strcpy(this->name, name);
    }
    void ShowYourName() const
    {
        cout<<"name: "<<name<<endl;
    }
    virtual int GetPay() const
    {
        return 0;
    }
    virtual void ShowSalaryInfo() const
    { }
};
```

GetPay 함수와 ShowSalaryInfo 함수를 Virtual로 선언하였으므로, EmpolyeeHandler가 호출하는 함수는 Employee 클래스의 멤버함수일지라도 실제 호출되는 함수는 각 포인터가 가리키는 객체의 마지막 오버라이딩 함수이다!

이렇듯 상속은 연관된 일련의 클래스들에 대해 공통의 규약을 적용할 수 있게 해 준다!

순수 가상함수

- 순수 가상함수(pure virtual function)
  - 베이스 클래스에서는 어떤 동작도 정의되지 않고 함수의 선언 만을 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
```

순수 가상함수와 추상 클래스

```
class Employee
{
private:
    char name[100];
public:
    Employee(char * name) { . . . . }
    void ShowYourName() const { . . . . }
    virtual int GetPay() const
    {
        return 0;
    }
    virtual void ShowSalaryInfo() const
    { }
};
```

몸체가 정의되지 않은 함수를 가리켜 순수 가상함수라 하며, 하나 이상의 순수 가상함수를 멤버로 두어서 객체 생성이 불가능한 클래스를 가리켜 추상 클래스라 한다.

오버라이딩의 관계를 목적으로 정의된 함수들!
따라서 몸체부분의 정의는 의미가 없다!

```
virtual int GetPay() const = 0;
virtual void ShowSalaryInfo() const = 0;
```

순수 가상함수로 대체 가능!

Ex) EmployeeManager5.cpp

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

class Employee
{
private:
	char name[100];
public:
	Employee(char * name)
	{
		strcpy(this->name, name);
	}
	void ShowYourName() const
	{
		cout<<"name: "<<name<<endl;
	}
	virtual int GetPay() const = 0;
	virtual void ShowSalaryInfo() const = 0;
};

class PermanentWorker : public Employee
{
private:
	int salary;
public:
	PermanentWorker(char * name, int money)
		: Employee(name), salary(money)
	{ }
	int GetPay() const
	{
		return salary;
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class TemporaryWorker : public Employee
{
private:
	int workTime;
	int payPerHour;
public:
	TemporaryWorker(char * name, int pay)
		: Employee(name), workTime(0), payPerHour(pay)
	{ }
	void AddWorkTime(int time)
	{
		workTime+=time;
	}
	int GetPay() const
	{
		return workTime*payPerHour;
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class SalesWorker : public PermanentWorker
{
private:
	int salesResult;	// 월 판매실적
	double bonusRatio;	// 상여금 비율
public:
	SalesWorker(char * name, int money, double ratio)
		: PermanentWorker(name, money), salesResult(0), bonusRatio(ratio)
	{ }
	void AddSalesResult(int value)
	{
		salesResult+=value;
	}
	int GetPay() const
	{
		return PermanentWorker::GetPay()
+
	(int)(salesResult*bonusRatio);
	}
	void ShowSalaryInfo() const
	{
		ShowYourName();
		cout<<"salary: "<<GetPay()<<endl<<endl;
	}
};
```

```
class EmployeeHandler
{
private:
	Employee* empList[50];
	int empNum;
public:
	EmployeeHandler() : empNum(0)
	{ }
	void AddEmployee(Employee* emp)
	{
		empList[empNum++]=emp;
	}
	void ShowAllSalaryInfo() const
	{

		for(int i=0; i<empNum; i++)
			empList[i]->ShowSalaryInfo();

	}
	void ShowTotalSalary() const
	{
		int sum=0;

		for(int i=0; i<empNum; i++)
			sum+=empList[i]->GetPay();

		cout<<"salary sum: "<<sum<<endl;
	}
	~EmployeeHandler()
	{
		for(int i=0; i<empNum; i++)
			delete empList[i];
	}
};
```

```
int main(void)
{
    // 직원관리를 목적으로 설계된 컨트롤 클래스의 객체생성
    EmployeeHandler handler;

    // 정규직 등록
    handler.AddEmployee(new PermanentWorker("KIM", 1000));
    handler.AddEmployee(new PermanentWorker("LEE", 1500));

    // 임시직 등록
   TemporaryWorker * alba=new TemporaryWorker("Jung", 700);
    alba->AddWorkTime(5);                    // 5시간 일한결과 등록
    handler.AddEmployee(alba);

    // 영업직 등록
    SalesWorker * seller=new SalesWorker("Hong", 1000, 0.1);
    seller->AddSalesResult(7000);            // 영업실적 7000
    handler.AddEmployee(seller);

    // 이번 달에 지불해야 할 급여의 정보
    handler.ShowAllSalaryInfo();

    // 이번 달에 지불해야 할 급여의 총합
    handler.ShowTotalSalary();
    return 0;
}
```

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

다형성(Polymorphism)

```
class First
{
public:
    virtual void SimpleFunc() { cout<<"First"<<endl; }
};

class Second: public First
{
public:
    virtual void SimpleFunc() { cout<<"Second"<<endl; }
};


int main(void)
{
    First * ptr=new First();
    ptr->SimpleFunc();    // 아래에 동일한 문장이 존재한다.
    delete ptr;
    ptr=new Second();
    ptr->SimpleFunc();    // 위에 동일한 문장이 존재한다.
    delete ptr;
    return 0;
}
```

지금까지 공부한 가상함수와 관련된 내용을 가리켜 '다형성'이라 한다!

다형성은 동질이상의 의미를 갖는다.

모습은 같은데 형태는 다르다.

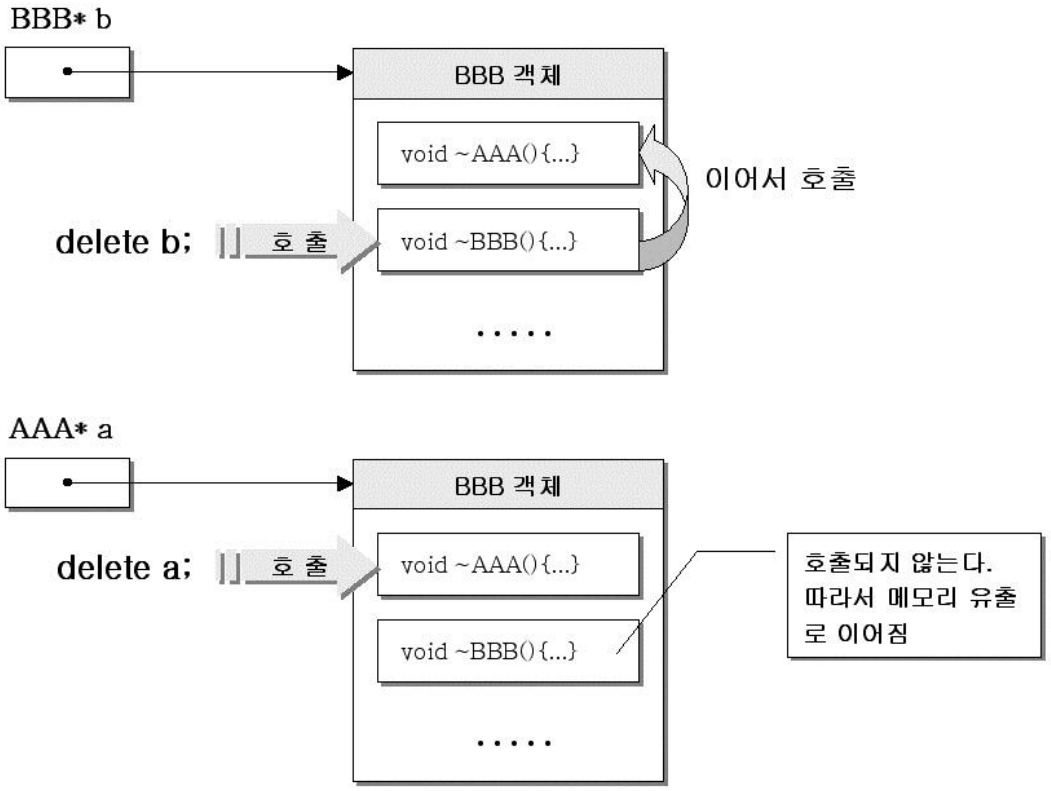
문장은 같은데 결과는 다르다!

ptr->Simplefunc 함수의 호출이 다형성의 예!

가상 소멸자와 참조자의 참조 가능성

Virtual 소멸자의 필요성

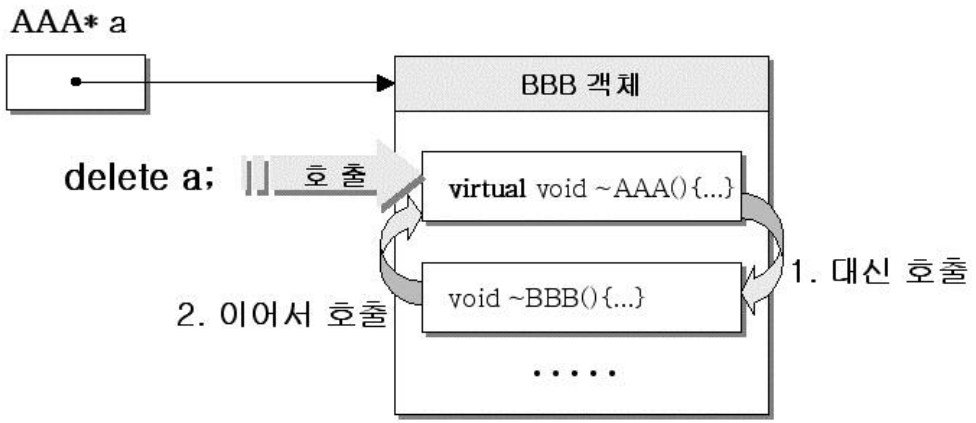
- 상속하고 있는 클래스 객체 소멸 문제점



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!
```

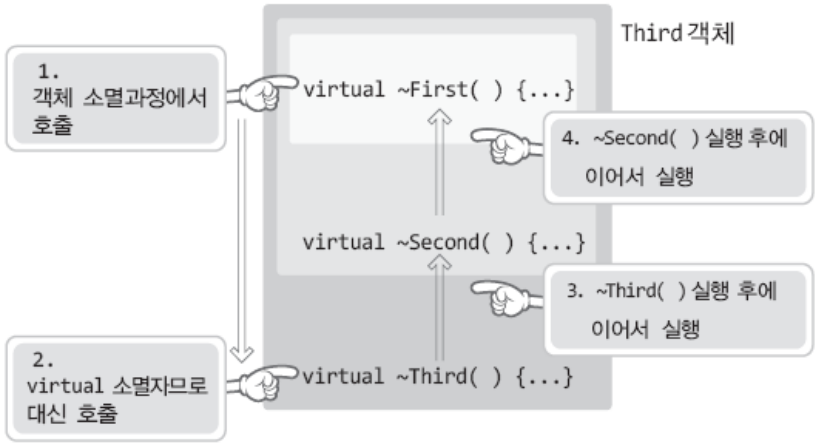
가상 소멸자(Virtual Destructor)

```
class First
{
    . . . .
public:
    virtual ~First() { . . . . }
};

class Second: public First
{
    . . . .
public:
    virtual ~Second() { . . . . }
};

class Third: public Second
{
    . . . .
public:
    virtual ~Third() { . . . . }
};
```

```
int main(void)
{
    First * ptr=new Third();
    delete ptr;
    . . . .
}
```



▶ [그림 08-3: 가상 소멸자의 호출과정]

소멸자를 가상으로 선언함으로써 각각의 생성자 내에서 할당한 메모리 공간을 효율적으로 해제할 수 있다.

참조자의 참조 가능성

"C++에서, AAA형 포인터 변수는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 가리킬 수 있다(객체의 주소 값을 저장할 수 있다)."

"C++에서, AAA형 참조자는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 참조할 수 있다."

```
class First
{
public:
    void FirstFunc() { cout<<"FirstFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"First's SimpleFunc()"<<endl; }
};

class Second: public First
{
public:
    void SecondFunc() { cout<<"SecondFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"Second's SimpleFunc()"<<endl; }
};

class Third: public Second
{
public:
    void ThirdFunc() { cout<<"ThirdFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"Third's SimpleFunc()"<<endl; }
};
```

```
int main(void)
{
    Third obj;
    obj.FirstFunc();
    obj.SecondFunc();
    obj.ThirdFunc();
    obj.SimpleFunc();

    Second & sref=obj;
    sref.FirstFunc();
    sref.SecondFunc();
    sref.SimpleFunc();

    First & fref=obj;
    fref.FirstFunc();
    fref.SimpleFunc();
    return 0;
}
```

실행결과

```
FirstFunc()
SecondFunc()
ThirdFunc()
Third's SimpleFunc()
FirstFunc()
SecondFunc()
Third's SimpleFunc()
FirstFunc()
Third's SimpleFunc()
```

Q & A