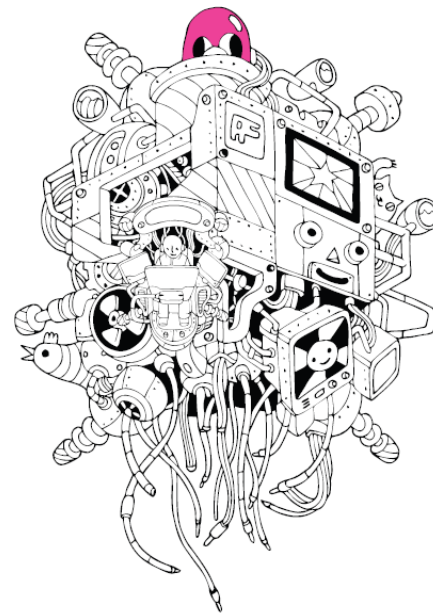


# 윤성우의 열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 08. 상속과 다형성

# 윤성우의 열혈 C++ 프로그래밍



## Chapter 08-1. 객체 포인터의 참조관계

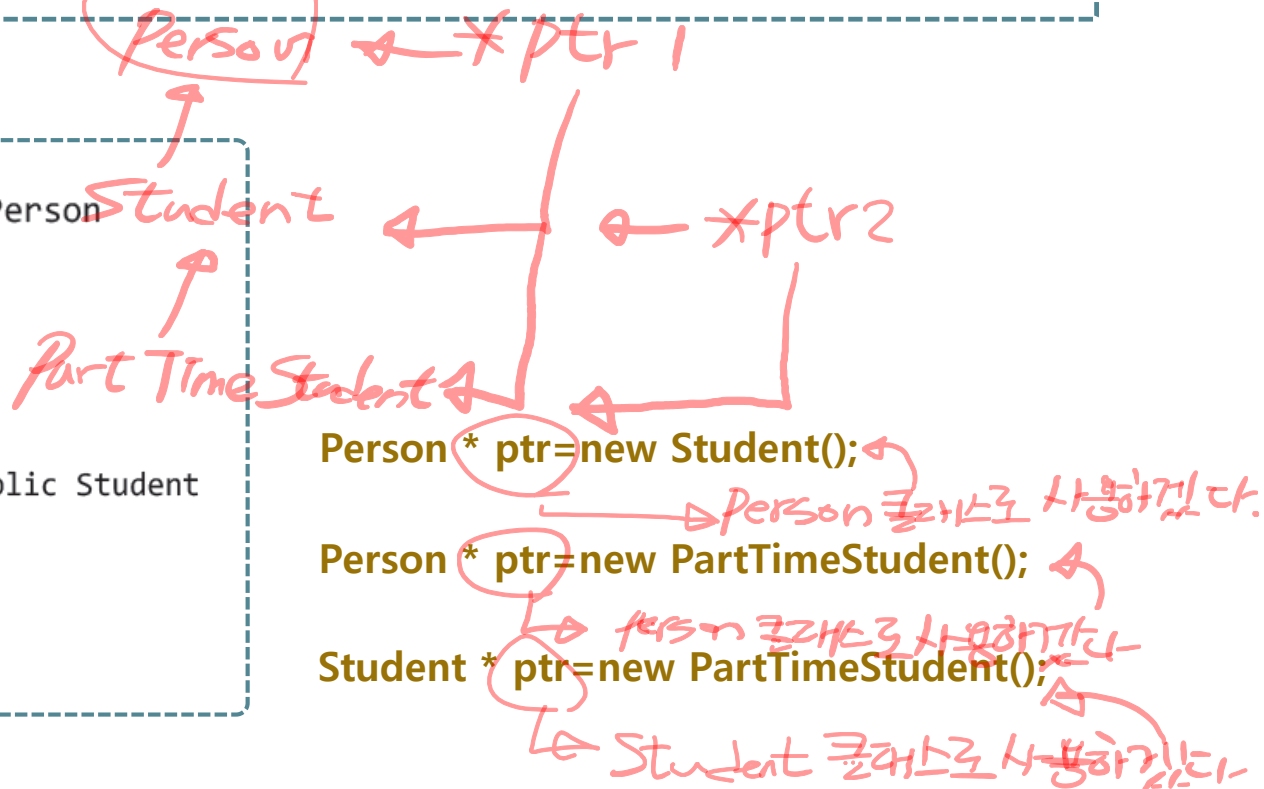
윤성우 저 열혈강의 C++ 프로그래밍 개정판

# 객체의 주소 값을 저장하는 객체 포인터 변수

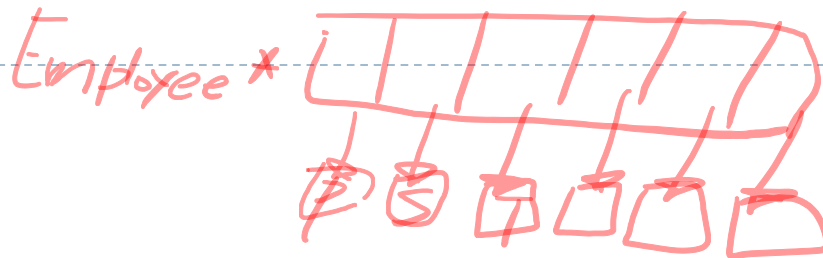
"C++에서, AAA형 포인터 변수는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 가리킬 수 있다(객체의 주소 값을 저장할 수 있다)."

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

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



# 오렌지미디어 급여관리 확장성 문제 1차 해결

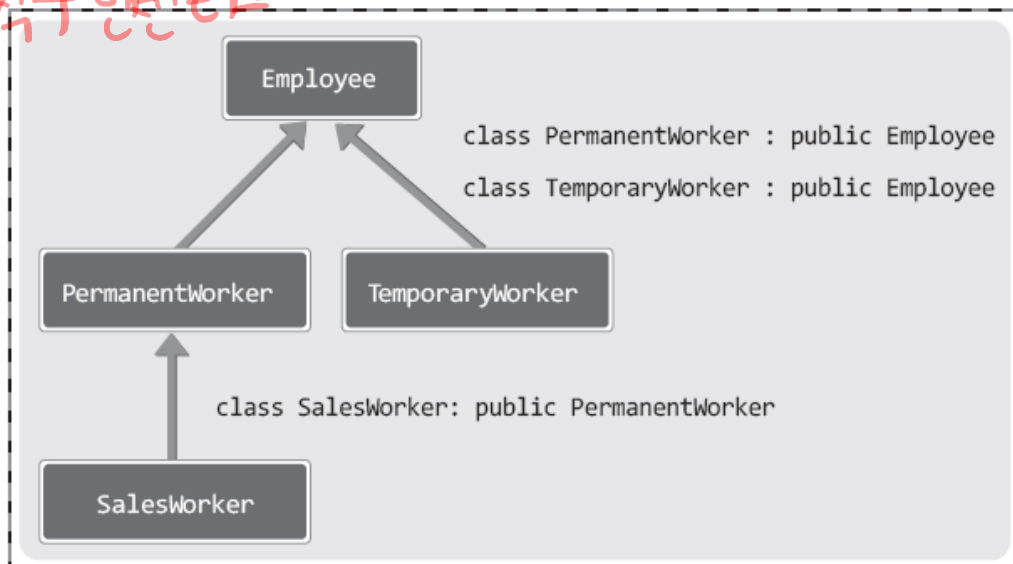


- 고용인 Employee
- 정규직 PermanentWorker
- 영업직 SalesWorker
- 임시직 TemporaryWorker

“정규직, 영업직, 임시직 모두 고용의 한 형태이다(고용인이다).”

“영업직은 정규직의 일종이다.”

정규직 + 임시직



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;
    }
};
```

1. AddEmployee(new S...())

왼쪽의 EmployeeHandler 클래스는 Employee 객체를 처리하는 컨트롤 클래스로 변경되었다.

# 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!

정의를 안하면 부모의 GetPay()만 호출됨

# 윤성우의 열혈 C++ 프로그래밍



## Chapter 08-2. 가상함수(Virtual Function)

윤성우 저 열혈강의 C++ 프로그래밍 개정판




# 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!
    . . . .
}
```

복모는 자식포인터를  
받을 수 있음

# 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();    (×)
    . . .
}
```

자신 클래스 접근하기

예제 EmployeeManager2.cpp와 EmployeeManager3.cpp의 주식처리 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;
}
```

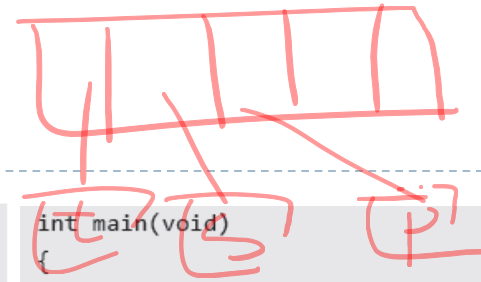
```
FirstFunc
SecondFunc
ThirdFunc
```

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; }
};
```

오버라이딩 된 함수가 virtual이면  
오버라이딩 한 함수도 자동 virtual

```
class Third: public Second
{
public:
    virtual 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;
}
```

ThirdFunc  
ThirdFunc  
ThirdFunc

실행결과

포인터의 형에 상관 없이 포인  
터가 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 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 가상함수**라 하며, 하나 이상의 순수 가상함수를 멤버로 두어서 객체 생성이 불가능한 클래스를 가리켜 **추상 클래스**라 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!**

# 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!

# 윤성우의 열혈 C++ 프로그래밍



## Chapter 08-3. 가상 소멸자와 참조자의 참조 가능성

윤성우 저 열혈강의 C++ 프로그래밍 개정판



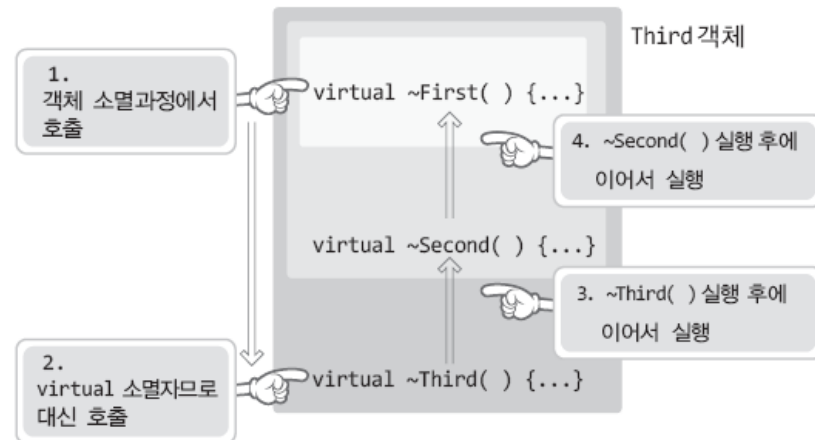
# 가상 소멸자(Virtual Destructor)

```
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).”

→ 자식 포인터는 부모를 가리킬 수 있음

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

→ 참조자도 마찬가지

```
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();

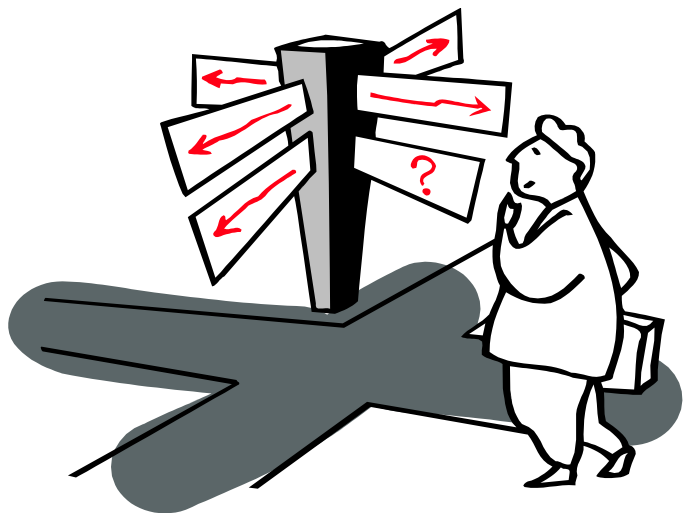
    Second & sref=obj;
    sref.FirstFunc();
    sref.SecondFunc();
    sref.SimpleFunc();

    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()
```



Chapter 08이 끝났습니다. 질문 있으신지요?