C++ Programming

12th Study: Object-Oriented Programming (8/8)

- Multiple inheritance
- vtable(Virtual function table)
- RTTI(Runtime Type Information)

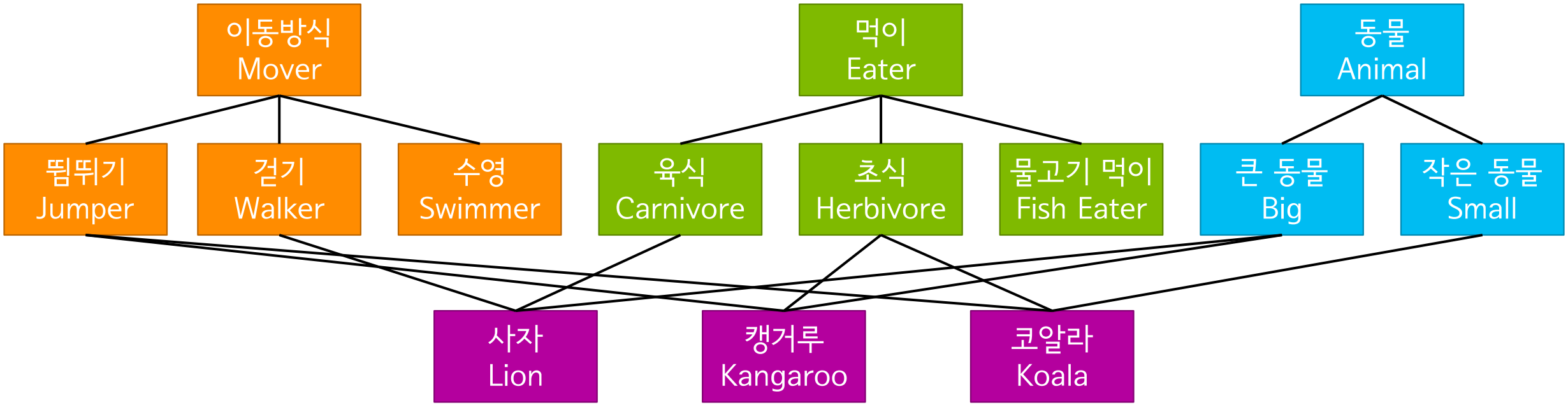
C++ Korea 옥찬호 (utilForever@gmail.com)

Multiple inheritance

A feature in which an object or class can inherit characteristics and features from more than one parent object or parent class

Multiple inheritance

- 지금까지 봤던 예제들은 모두 단일 상속!
  즉, 자식 클래스는 하나의 부모 클래스만 가짐
- 다중 상속을 통해 하나 이상의 부모 클래스를 가질 수도 있음



Multiple inheritance

- 다중 상속은 특정 상황에서는 아주 유용함
  - 근데 언제 유용할까…?
    - 첨가 클래스를 이용할 때
    - 컴포넌트 기반 클래스를 모델링 할 때
- 하지만 항상 염두에 둬야 하는 단점들도 있음
  - 많은 프로그래머들이 다중 상속을 좋아하지 않음
  - C++은 명시적으로 다중 상속 관계를 지원하지만 Java와 같은 언어는 다중 상속을 배제함

Multiple inheritance

- 다중 상속이 꺼려지는 이유
  - 다중 상속 관계는 그림을 나타내기에 복잡
    - 클래스 계층은 프로그래머로 하여금 코드 간 관계를 이해하기 쉽도록 도움
    - 그런데 다중 상속에서는 서로 전혀 관계없는 클래스들을 부모로 가질 수 있어 어느 클래스들이 지금 작업 중인 객체의 코드에 영향을 미치는지 파악하기 쉽지 않음
  - 다중 상속은 간명할 수도 있는 클래스 구조를 망가뜨릴 수 있음
    - 이전 예제에서 뜀뛰기를 하는 동물이 모두 같은 먹이를 먹는다면?
    - 계층이 분리되어 있기 때문에 이동 방식과 먹이가 같아도 추가적인 자식 클래스 없이 부모 클래스에 합칠 방법이 없음

Multiple inheritance

- 다중 상속이 꺼려지는 이유
  - 다중 상속 계층은 구현하기가 까다로움
    - 만약 두 부모 클래스가 같은 행동을 다른 방식으로 구현했다면?
    - 이 경우 같은 부모 클래스를 가지는 두 자식 클래스를 부모 클래스로 가질 수 있을까? 해당 행동은 어느 쪽 부모 클래스를 따라야 하는가?
    - 이런 상황이 발생할 가능성 때문에 구현이 까다로움
- 이런 이유 때문에 다른 언어에서는 다중 상속을 제거했고, 보통은 다중 상속을 사용하지 않아도 충분함
  - 클래스 계층을 다시 생각하거나, 디자인 패턴을 사용

Multiple inheritance

- 그래도 C++에서는 되니까, 메커니즘을 설명해 보자면…

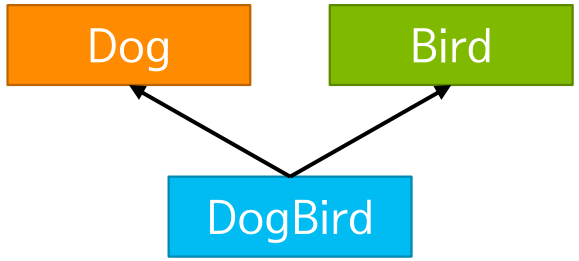
```
class Baz : public Foo, public Bar
{
    // Something
};
```

  - Baz 객체는 Foo와 Bar의 모든 public 멤버 변수 / 함수를 지원
  - Baz 객체의 멤버 함수에서 Foo와 Bar의 protected 멤버 변수 / 함수에 접근할 수 있음
  - Baz 객체는 Foo 또는 Bar로 업 캐스팅될 수 있음
  - 새로운 Baz 객체를 생성할 때 자동으로 Foo와 Bar의 디폴트 생성자가 호출됨
    호출 순서는 상속 목록에 나열된 순서를 따름
  - Baz 객체를 삭제하면 Foo와 Bar의 소멸자가 생성자 호출의 역순으로,
    즉 상속 목록에 나열된 순서의 반대 순서로 호출됨

Multiple inheritance

- DogBird(개새) = Dog(개) + Bird(새)

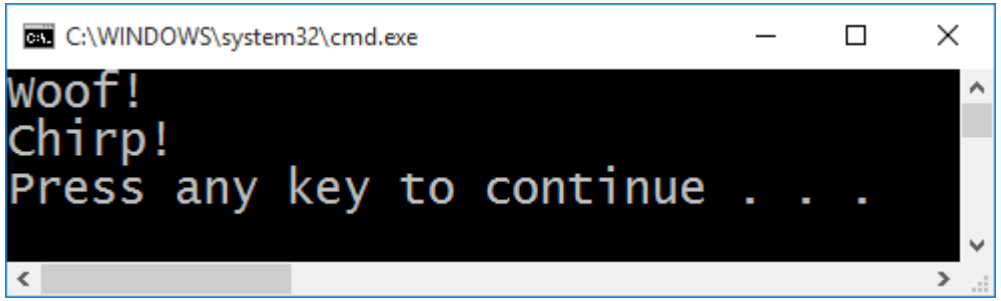
```
class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
};

class DogBird : public Dog, public Bird
{ };
```



Multiple inheritance

- 복수의 부모를 가진 클래스 객체라고 해도 이용방법은 같음
  - 사용자는 클래스가 두 개의 부모를 가졌다는 사실 자체를 몰라도 됨
  - 알아야 할 사항은 두 부모 클래스가 지원하는 멤버 변수 / 함수!
    - 이 경우 DogBird 객체는 Dog와 Bird 클래스의 모든 public 멤버 함수를 지원

```
DogBird confusedAnimal;
confusedAnimal.bark();
confusedAnimal.chirp();
```



Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호한 이름
    - Dog 클래스와 Bird 클래스가 같은 멤버 함수 eat()을 가졌다면 어떻게 될까?
    - Dog와 Bird는 아무런 상관이 없지만, DogBird 클래스에서 동시에 존재함

```
class Dog {
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() { cout << "The dog has eaten." << endl; }
};

class Bird {
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() { cout << "The bird has eaten." << endl; }
};
```

Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호한 이름

```
DogBird confusedAnimal;
confusedAnimal.eat();
```

```
error C2385: ambiguous access of 'eat'
note: could be the 'eat' in base 'Dog'
note: or could be the 'eat' in base 'Bird'
```

  - 해결 방법 : 객체를 업 캐스트하거나 모호성이 해소되도록 스코프 지정 연산자 사용

```
static_cast<Dog>(confusedAnimal).eat(); // 슬라이싱, Dog::eat() 호출
confusedAnimal.Dog::eat();              // 명시적으로 Dog::eat() 호출
```

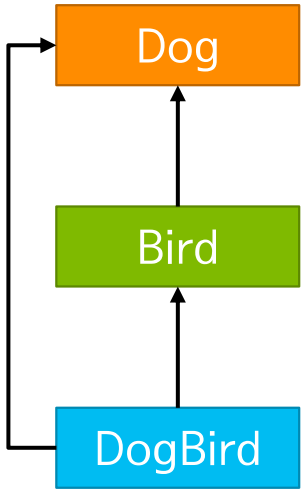
  - 다른 해결 방법 : 자식 클래스의 멤버 함수 구현부에서도 부모 클래스의 이름을 스코프 지정 연산자로 지정해 멤버 함수의 모호성을 해소할 수 있음

```
void DogBird::eat()
{
    Dog::eat(); // 명시적으로 Dog의 eat() 호출
}
```

Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호성을 발생시키는 다른 방법 : 같은 클래스를 두 번 상속받는 경우

```
class Dog
{
public:
    virtual void eat()
    {
        cout << "The bird has eaten." << endl;
    }
};
class Bird : public Dog {};
class DogBird : public Bird, public Dog {};
```



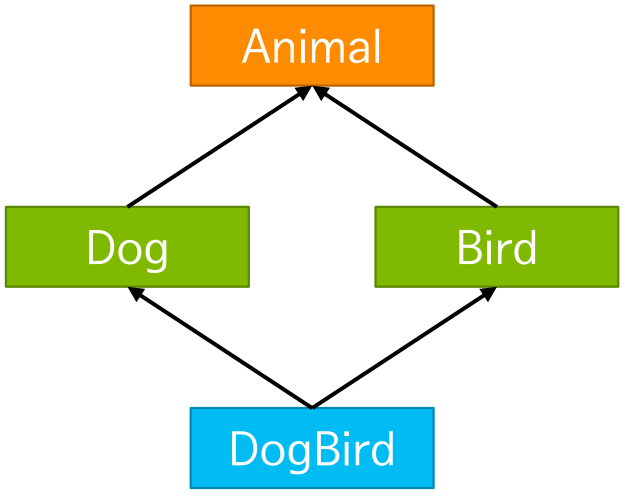

```
DogBird confusedAnimal;
confusedAnimal.eat();
```

```
error C2385: ambiguous access of 'eat'
note: could be the 'eat' in base 'Dog'
note: or could be the 'eat' in base 'Dog'
```

Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호한 베이스 클래스
    - 모호한 베이스 클래스가 발생하는 가장 흔한 시나리오는 부모 클래스들이 같은 부모를 공통적으로 가지는 경우

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

class Dog : public Animal { … };
class Bird : public Animal { … };
```




```
confusedAnimal.sleep();
```

```
error C2385: ambiguous access of 'sleep'
note: could be the 'sleep' in base 'Animal'
note: or could be the 'sleep' in base 'Animal
```

Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호한 베이스 클래스
    - 다이아몬드 형태의 클래스 계층을 가지는 상황에서는 최상위 클래스를 모든 멤버 함수가 순수 가상 함수인 추상 클래스로 만드는 것이 가장 좋은 방법!
      (하지만 DogBird 클래스는 어느 부모의 sleep() 함수를 호출해야 할 지 명시해야 함)

```
class Animal {
public:
    virtual void sleep() = 0;
};

class Dog : public Animal {
public:
    virtual void sleep() { … }
};
```

```
class Bird : public Animal {
public:
    virtual void sleep() { … }
};

class DogBird : public Dog, public Bird  {
public:
    virtual void sleep() { Dog::sleep(); }
};
```

Multiple inheritance

- 다중 상속이 문제를 일으키는 상황
  - 모호한 베이스 클래스
    - 하지만 더 좋은 방법이 있었으니, 바로 virtual 베이스 클래스(virtual base class)! (Animal을 virtual로 상속받으면 공통 부모에 대해서 하나의 객체만 생성되기 때문에 sleep()이 실행될 때 어느 쪽 객체를 이용해야 할 지 선택할 필요가 없어짐)

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

class Dog : public virtual Animal { … };
class Bird : public virtual Animal { … };

confusedAnimal.sleep(); // Animal이 virtual이기 때문에 모호하지 않음
```

vtable(Virtual function table)

A mechanism to support run-time method binding

vtable

- virtual이 아니어도 오버라이딩을 할 수 있음 → 멤버 함수 재정의를 통해

```
class Super
{
public:
    void go() { cout << "Super::go()" << endl; }
};

class Sub : public Super
{
public:
    void go() { cout << "Sub::go()" << endl; }
};
```

```
Sub mySub;
mySub.go();
```

→ Sub::go()

```
Sub mySub;
Super& ref;
ref = mySub;
ref.go();
```

→ Super::go()

  - 하지만 메서드가 virtual이 아니므로 실제로 오버라이딩된 것이 아님
    - Sub 클래스에서 Super의 go()와는 별개인 새로운 멤버 함수 go()를 정의했을 뿐
    - go()는 virtual이 아니기 때문에 서브클래스에서의 오버라이딩 여부를 따지지 않음

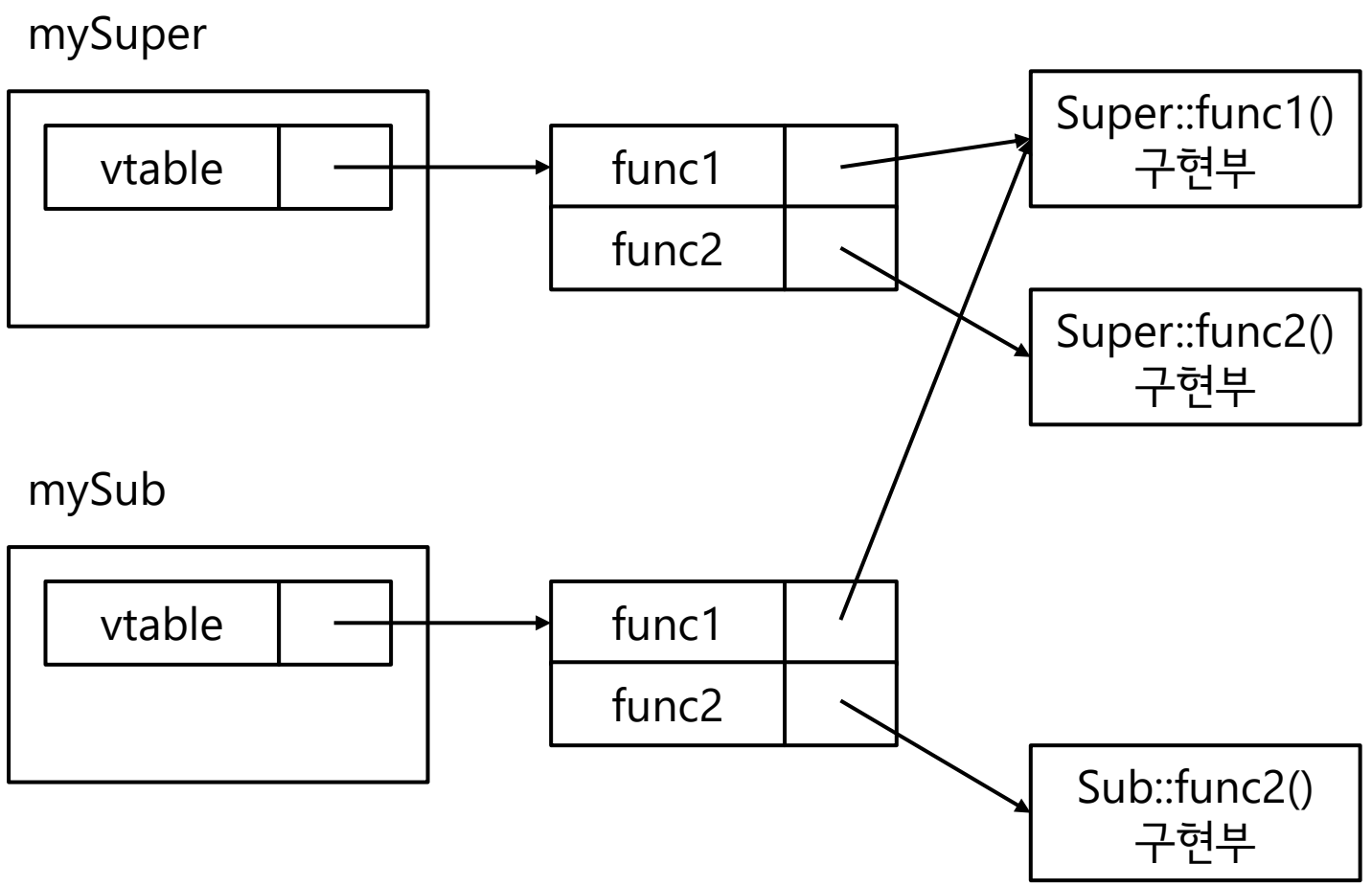
vtable

- virtual의 내부 구현
  - 멤버 함수 숨김이 일어나는 이유를 알려면 virtual 키워드가 어떻게 구현되는지 알아야 함
    - 컴파일러가 클래스 정의 코드를 컴파일하면 그 클래스의 모든 데이터 멤버 변수 및 멤버 함수가 들어 있는 바이너리 객체가 만들어짐
    - virtual이 아닌 멤버 함수의 경우 멤버 함수 호출 시 제어권의 전달이 컴파일 타임 타입에 맞춰서 직접 하드코딩됨
    - 하지만 virtual로 선언된 멤버 함수의 경우 vtable(Virtual Table, 가상 테이블)이라 부르는 특별한 메모리 영역을 찾아보게 됨
      - 하나 이상의 virtual 메서드를 가진 클래스는 각각 자신만의 vtable을 통해 virtual 멤버 함수들의 구현부를 가리키는 포인터들을 관리함. 어떤 객체의 멤버 함수가 호출되면 그 객체에 딸린 vtable에서 해당 멤버 함수의 오버라이딩된 포인터를 찾아서 객체의 실제 타입에 맞춰 올바른 버전의 멤버 함수가 실행되도록 함

vtable

- virtual의 내부 구현

```
class Super
{
public:
    virtual void func1() { }
    virtual void func2() { }
};

class Sub : public Super
{
public:
    virtual void func2() { }
};

Super mySuper;
Sub mySub;
```



vtable

- virtual 키워드에 대한 논쟁 (쓸 것인가, 말 것인가?)
  - 이전에 모든 멤버 함수를 virtual로 선언하길 권장한 적이 있음
  - 컴파일러가 모든 멤버 함수를 virtual로 취급하게 만들 수도 있음
    (Java 언어는 모든 멤버 함수는 무조건 virtual로 취급)
  - 그러나 C++에서 virtual 키워드가 존재하는 이유는 오버헤드 때문
    - virtual 멤버 함수를 호출하기 위해서는 포인터를 역참조해
      실행할 멤버 함수의 위치를 찾아오는 부가적인 작업이 필요
    - 지금은 오버헤드를 무시할 수 있을 만큼 작지만, C++ 언어를 만들 당시에는
      선택권을 주는 것이 더 낫다고 판단함 → 만약 오버라이딩 자체가 필요하지 않다면
      virtual을 사용할 필요가 없고 그에 따라 오버헤드도 없어지게 됨
    - 코드 크기에도 영향을 미침 (각 객체에 멤버 함수 포인터를 저장할 공간 추가로 필요)

vtable

- virtual 소멸자의 필요성
  - 하지만 오버헤드에 상관 없이 소멸자는 반드시 virtual로 선언해야 함
    - 소멸자가 virtual이 아닌 경우 객체가 소멸하더라도 해제되지 않은 메모리 영역이 남을 수 있기 때문

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

class Sub : public Super {
private:
    char* mString;
public:
    Sub() { mString = new char[30]; }
    ~Sub() { delete[] mString; }
};
```

```
Super* ptr = new Sub(); // mString은 이 시점에 할당됨
delete ptr;             // ~Super가 호출되었지만
                        // virtual이 아니기 때문에
                        // ~Sub는 호출되지 않음
```

RTTI(Runtime Type Information)

A C++ mechanism that exposes information about an object's data type at runtime

RTTI

- C++은 런타임이 아닌 컴파일 타임에 많은 것들이 결정됨
  - 오버라이딩의 경우, 멤버 함수를 간접적으로 호출하기 때문에 동작하는 것이지 객체 스스로 자신이 속한 클래스가 무엇인지 알고서 대응하는 것이 아님
- C++에도 객체의 런타임 정보를 얻을 수 있는 기능이 있음 바로 런타임 타입 정보, RTTI(Run-Time Type Information)!
  - RTTI는 객체가 속한 클래스와 연동되는 여러 가지 기능을 제공
    - dynamic_cast : RTTI를 이용해 객체 계층 간에 타입 변환을 안전하게 할 수 있도록 해줌
    - typeid : 런타임에 객체의 타입이 무엇인지 알 수 있게 해줌 (대부분은 virtual 멤버 함수를 이용해 객체의 타입에 따른 작업 처리가 가능하기 때문에 typeid에 의존해야 할 상황은 많지 않음)

RTTI

- dynamic_cast 예제 (하나 이상의 가상 함수를 가지고 있어야 함)

```
#include <iostream>

using namespace std;

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

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

참고 : http://prostars.net/55

RTTI

- dynamic_cast 예제 (하나 이상의 가상 함수를 가지고 있어야 함)

```
Base* base1 = new Base;
Base* base2 = new Derived;
Derived* derived1 = new Derived;
Derived* derived2 = nullptr;

// 컴파일 오류 : 타입 변환을 할 수 없다.
// derived2 = base1;

// 컴파일 성공 : 런타임에 타입 변환에 실패하며
// nullptr을 반환한다.
derived2 = dynamic_cast<Derived*>(base1);
if (derived2 == nullptr)
    cout << "Runtime Error" << endl;
```

참고 : http://prostars.net/55

RTTI

- dynamic_cast 예제 (하나 이상의 가상 함수를 가지고 있어야 함)

```
// 컴파일 오류 : 타입 변환을 할 수 없다.
// derived2 = base2;

// 컴파일 성공 : 런타임에 타입 변환에 성공하며
// Derived 타입의 포인터를 반환한다.
derived2 = dynamic_cast<Derived*>(base2);
if (derived2)
    derived2->print();

// 컴파일 성공 : 이런 경우에는 캐스팅이 필요 없다.
derived2 = derived1;
```

참고 : http://prostars.net/55

RTTI

- typeid 예제 (하나 이상의 가상 함수를 가지고 있어야 함)

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Animal
{
    virtual void something() { }
};

class Dog : public Animal { };
class Bird : public Animal { };
```

```
void speak(const Animal& animal) {
    if (typeid(animal) == typeid(Dog&))
        cout << "Woof!" << endl;
    else if (typeid(animal) == typeid(Bird&))
        cout << "Chirp!" << endl;
}


Dog dog;
Bird bird;

Animal& animal = dog;
speak(animal);
animal = bird;
speak(bird);
```

```
Woof!
Chirp!
```