OOP ( object oriented programming )

#### Encapsulation (캡슐화)

객체가 구현된 방식의 세부 정보를 사용자로부터 숨기면서 유지하는 방식

(public 과 private 접근 지정자 사용)

캡슐화를 사용하는 이유

1. 사용하기 쉽고, 복잡성을 줄여준다.

현실에서 예를 들자면, 리모컨을 사용할 때 사용자는 리모컨이 어떻게 동작하는지 원리를 이해하지 않아도 사용할 수 도록 구현되어 있어 매우 효과적이다. 만일 리모컨이 어떻게 구현되었는지 이해해야지만 사용할 수 있다면 사용자는 리모컨을 사용하기 어려웠을 것입니다. 이러한 점을 토대로 할 때, 캡슐화는 프로그램의 복잡성을 줄여주고, 실수도 줄여줍니다.

1. 데이터 보호 , 오용 방지

전역 변수나 public 멤버 변수를 지정하게 되면 아무나 해당 변수에 접근할 수 있으므로 엄격하게 제어할 수 없어서 위험합니다. 또한 이 데이터를 조작하여 악용하는 사례가 발생할 수 있기에 캡슐화를 사용합니다.

오용 사례

class IntArray {

public:

int arr[5] ;

}

int main(void){

IntArray array;

array.arr[10] = 2 ; // error (index out of bounds)

return 0;

}

만약 배열을 잘못된 인덱스에 접근하면 에러가 발생하게 된다.

캡슐화를 사용할 경우

class IntArray {

private:

int arr[5];

public:

void setValue(int x, int value ) {

if ( x >= 0 && x < 5 ) {

arr[x] = value;

}

}

int main (void) {

IntArray array;

array.setValue(2, 5) ;

return 0 ;

}

이와 같이 array에 접근하기 위해서는 setValue라는 함수를 통해서만 접근할 수 있도록 하여 에러가 발생하지 않고 배열에 접근하기 위해서는 setValue를 강제로 사용하게 만들 수 있습니다.

C++에서 캡슐화 하는 방법

접근 지정자 (private 과 public)을 사용하여 캡슐화 해준다.

class animal {

private:

int color ;

int weight ;

int height ;

public:

void setColor (string color) ;

void setWeight ( int w ) ;

void setHeight ( int h ) ;

string getColor() ;

int getWeight() ;

int getHeight() ;

}

int main(void) {

animal cat ;

cat.setColor(“black”);

cout << “Cat Color: “ << cat.getColor() << endl;

return 0;

}

이와 같이 멤버 변수를 private 으로 지정하게 되면 반드시 public 멤버 함수를 통해서만 멤버 변수에 접근할 수 있었습니다. 그렇다보니 경우에 따라서 객체의 멤버 함수가 아닌 함수도 private 멤버 함수에 접근해야만 할 경우가 발생하게 됩니다.

이러한 경우 번거로움을 줄이기 위해 C++에서는 friend 라는 새로운 접근 제어 키워드를 제공해주었습니다.

##### < friend >

지정한 대상에 한해 해당 객체의 모든 멤버에 접근할 수 있는 권한을 부여해줍니다.

friend 키워드는 전역 함수, 클래스, 멤버 함수의 세 가지 형태로 사용할 수 있습니다.

-----------------------------------------------------------------------------------------

###### friend 전역 함수

프렌드 함수는 해당 클래스의 멤버 함수는 아니지만, 멤버 함수와 같은 접근 권한을 가지게 된다.

friend 전역 함수 선언

friend [리턴 타입] 함수 이름

#include <iostream>

**using** **namespace** std;

**class** A {

**private**:

**int** x = 10;

**public**:

**friend** **int** getX(**const** A& t);

};

**int** getX(**const** A& t){

**return** t.x;

// friend 전역함수가 아니라면 private 멤버 변수인 x에 접근 불가

}

**int** main(**void**){

A classA;

**int** x = getX(classA);

cout << "x of class named A : " << x << endl; // 10

**return** 0;

}

⇒ 이와 같이 A라는 클래스의 멤버 변수인 x 에 friend를 사용하여 접근할 수 있었습니다.

-----------------------------------------------------------------------------------------

###### friend class

해당 클래스의 모든 멤버 함수가 특정 클래스의 프렌드인 클래스를 의미함.

<사용하는 경우>

두 클래스가 기능상으로 밀접한 관계가 있고, 상대방 클래스의 private 멤버에 접근할 필요가 있는 경우 class 자체를 friend로 선언하여 사용한다.

#include <iostream>

**using** **namespace** std ;

**class** buyer{

**private**:

**int** haveMoney;

**public**:

buyer(**int** money){

haveMoney = money;

}

**void** showMoney() {

cout << "Buyer Money: " << haveMoney << endl;

}

**friend** **class** seller ; // friend class

};

**class** seller {

**private**:

**int** totalMoney;

**public**:

seller(**int** money){

totalMoney = money;

}

**void** showMoney() {

cout << "Seller Money: " << totalMoney << endl;

}

**void** sellProduct(buyer& b , **int** price){ //

**this**->totalMoney += price;

b.haveMoney -= price;

}

};

**int** main(**void**){

seller \*SELLER = **new** seller(100000) ;

buyer \*BUYER = **new** buyer(30000);

SELLER->showMoney();

BUYER->showMoney();

cout << "buyer bought the book of seller" << endl;

SELLER->sellProduct(\*BUYER, 20000);

SELLER->showMoney();

BUYER->showMoney();

**return** 0;

}

-----------------------------------------------------------------------------------------

##### static member variable

객체 별로 할당되지 않고 클래스의 모든 객체가 공유하는 멤버 변수를 의미

즉, 멤버 변수가 정적(static)으로 선언되면, 해당 클래스의 모든 객체에 대하 여 하나의 데이터만이 유지 관리가 됩니다.

정적 멤버 변수도 접근 지정자의 영향을 받아 private 으로 지정해놓을 경우 멤버 함수 또는 friend를 이용하여 접근해야한다.

##### static member function

객체를 생성하지 않고, 클래스 이름으로 호출하여 사용한다.

<static member function 특징>

1. 객체를 생성하지 않고, 클래스 이름으로 호출 가능
2. 객체 생성을 안하여 this 포인터를 사용할 수 없음
3. 특정 객체와 결합하지 않으므로 정적 멤버 변수만 사용 가능

상수 멤버 변수 (constant member variable)

한 번 초기화 하면, 값을 변경할 수 없음.

선언: const [타입] 멤버 변수 이름 ;

상수 멤버 함수 (constant member function)

호출한 객체의 데이터를 수정할 수 없는 멤버 함수   
선언: 상수 멤버 함수 원형 const ;

-----------------------------------------------------------------------------------------

#### 상속(Inheritance)

상속은 높은 코드 재활용성을 제공하며, 클래스 간의 계층적 관계를 구성하게 함으로써 다형성의 문법적 토대를 마련해줍니다. 이러한 이유로 인해 객체 지향 프로그래밍에서 상속은 중요한 특징 중에 하나입니다.

##### 클래스 상속(class inheritance)

C++에서 클래스 상속이란 기존에 정의되어 있는 클래스의 모든 멤버 변수와 멤버 함수를 물려받아, 새로운 클래스를 작성하는 것을 의미

* 기존에 정의되어 있는 클래스

기초 클래스(basic class), 부모 클래스 (parent class) , 상위 클래스 (super class)

* 상속된 새로운 클래스

파생 클래스(derived class), 자식 클래스 (child class) , 하위 클래스 (sub class)

상속의 장점

1. 기존에 작성된 클래스를 재활용 가능.

2. 공통적인 부분은 기초 클래스에 미리 작성하여, 파생 클래스에서 중복되는 부분을 따로 만들지 않아도 된다.

##### 파생 클래스 (derived class)

파생 클래스란

기초 클래스 (basic class) 의 멤버 변수와 멤버 함수들을 모두 물려받아 새롭게 작성된 클래스를 의미

다른 이름으로는 자식 클래스 (child class) , 하위 클래스 (sub class) 라고 불린다.

###### 파생 클래스 선언

기존 클래스가 하나일 경우 ( 단일 상속 )

class [클래스 이름] : [접근 제어 지시자] [기존 클래스] { … }

ex) class Student : public Person { ... }

기존 클래스가 두 개 이상일 경우 ( 다중 상속 )

class [클래스 이름] : [[접근 제어 지시자] [기존 클래스1], [접근 제어 지시자] [기존 클래스2], … ] { … }

ex) class Stduent : public Person, public soccerPlayer { … }

만일 접근 제어 지시자를 입력하지 않을 경우 , 접근 제어 지시자에 자동으로 private 이 들어가게 된다.

###### 파생 클래스 특징

1. 반드시 자신만의 생성자를 작성해야함.

2. 기초 클래스의 접근할 수 있는 모든 멤버 변수들이 저장됨.

3. 기초 클래스의 접근할 수 있는 모든 멤버 함수를 사용할 수 있음.

4. 필요한 만큼 멤버 변수 또는 함수를 추가 할 수 있음.

<상속 예제>

#include <iostream>

#include <string>

**using** **namespace** std;

**class** Person{

**private**:

string name;

**int** age;

**public**:

Person(string name, **int** age){

**this**->name = name;

**this**->age = age;

}

**void** displayAge(){

cout << "My age is "<< age << endl;

}

**void** overRidingFunc();

};

**void** Person::overRidingFunc(){

cout << "This function exists in Parent class(super class)" << endl;

}

**class** Student : **public** Person{

**private**:

**int** student\_id;

**public**:

Student(**int** id, string name, **int** age);

**void** overRidingFunc();

};

**void** Student::overRidingFunc(){ // member function overriding

cout << "This is Overriding." << endl;

cout << "This function exists in child class(subClass)" << endl;

}

Student::Student(**int** id, string name, **int** age):Person(name,age){

student\_id = id;

}

**int** main(**void**){

Student student(12345678,"Pepe", 30);

student.displayAge(); // method of Person class

student.Person::overRidingFunc();

student.overRidingFunc(); // overriding

}

##### 멤버 함수 오버라이딩 (Overriding)

자식 클래스가 부모 클래스로부터 상속 받은 멤버 함수를 무시하고, 같은 이름의 함수를 새롭게 정의하는 것을 의미

자식 클래스는 상속받은 멤버 함수를 그대로 사용하여도 되지만, 자신의 클래스에 필요한 동작을 위해 해당 멤버 함수를 재정의하여 사용할 수 있도록 한 것

(단, 오버라이딩은 멤버 함수의 동작만을 재정의 하는 것이므로, 함수의 원형은 기존 멤버 함수와 같아야 한다.)

#### Polymorphism

서로 다른 객체가 동일한 메시지에 대하여 서로 다른 응답을 할 수 있도록 하는 기능

##### 가상 함수 (virtual function)

가상 함수는 파생 클래스에서 재정의하기를 기대하는 멤버 함수입니다.

선언

virtual 멤버 함수 원형;

순수 가상 함수 (pure virtual function)

순수 가상 함수는 파생 클래스에서 반드시 재정의를 한 후 사용해야하는 멤버 함수입니다.

선언

virtual 멤버 함수 원형=0;

위 같이 함수만 존재하고 함수의 본체가 없다는 의미로 선언부 끝에 ”=0” 을 추가해줍니다.

##### 추상 클래스 (abstract class)

한 개 이상의 순수 가상 함수 (pure virtual function)을 포함하는 클래스를 의미

추상 클래스의 특징

1. 추상 클래스는 OOP의 중요한 특징인 다형성을 가진 함수의 집합을 정의할 수 있게 해줍니다.
2. 즉, 추상 클래스에서 멤버 함수를 순수 가상 함수로 선언해놓으면, 이 클래스로 부터 파생된 모든 클래스는 반드시 이 가상 함수를 재정의하여야 사용할 수 있습니다.
3. 추상 클래스는 순수 가상 함수를 포함하고 있기 때문에, 자신의 인스턴스를 생성할 수 없습니다.

추상 클래스는 동작이 정의되지 않은 순수 가상 함수를 포함하고 있으므로, 인스턴스를 생성할 수 없습니다.

따라서 추상 클래스는 먼저 상속을 통해 파생 클래스를 만들고, 만든 파생 클래스에서 순수 가상 함수를 모두 오버라이딩하고 나서야 비로소 파생 클래스의 인스턴스를 생성할 수 있게 됩니다.

하지만 추상 클래스 타입의 포인터와 참조는 바로 사용할 수 있습니다.

#include <iostream>

**using** **namespace** std;

**class** Animal {

**public**:

**virtual** ~Animal() {}; // 가상 소멸자 선언

**virtual** **void** Cry()=0;

};

**class** Dog : **public** Animal {

**public**:

**virtual** **void** Cry() {

cout << "Bark Bark !!" << endl;

}

};

**class** Cat : **public** Animal {

**public**:

**virtual** **void** Cry() {

cout << "Miaow ~!!" << endl;

}

};

**int** main(**void**){

// Polymorphism (다형성)

Animal \*dog = **new** Dog();

Animal \*cat = **new** Cat();

dog->Cry();

cat->Cry();

**delete** dog;

**delete** cat;

**return** 0;

}