디지털 영상처리 연구실 연구보고서

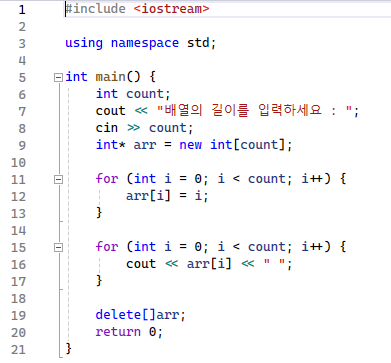
정지우

**동적 할당**

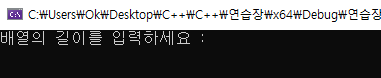
|  |  |  |
| --- | --- | --- |
|  | 정적 할당 | 동적 할당 |
| 특징 | 컴파일 단계에서 메모리가  미리 할당되고 프로그램이  끝나면 소멸한다 | 사용자가 원할 때  메모리영역을 할당하고  수동으로 반납하여  소멸 시킨다. |
| 메모리를 쓰지않거나  많이 할당 시켜놓으면,  낭비가 생길 수 있음 | 필요한 만큼 메모리를  할당 가능하지만,  '직접' 반납하여야 한다. |
| 메모리 영역 크기 변경 불가 | 메모리 영역 크기 변경 가능 |

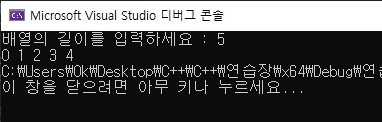
[ new 연산자 ]

자료형\*포인터 변수 = new 자료형[개수]



이와 같이 메모리 크기를 변수(count)로 입력받아 지정할 수 있다.





**상속**

**클래스 상속)**

이미 정의되어 있는 기반 클래스로부터 파생 클래스가 정의된다.

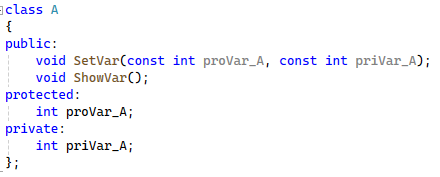
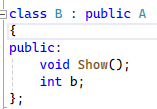
class 파생 클래스 : 접근지정자 기반클래스

{

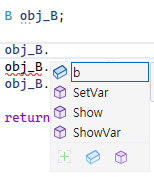
맴버 함수와 맴버 변수 선언;

};

이를 통해 파생 클래스의 객체는 자신의 맴버뿐만 아니라, 기반클래스에서 허락된 맴버도 참조 가능하다.



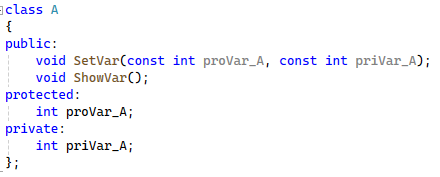
A 를 기반으로 하는 B 파생클래스를 선언하였고,

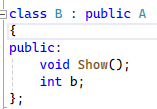


B의 객체 obj\_B를 생성하고 객체가 사용할 수 있는 맴버 함수,변수를 보면, B의 맴버 함수,변수 뿐만 아니라, A의 맴버 함수인 SetVar와 ShowVar수 있다.

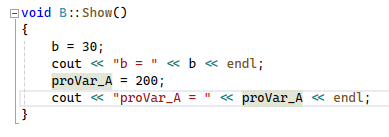
**접근 지정자)**

|  |  |  |  |
| --- | --- | --- | --- |
| 접근 지정자 | 기반 클래스에서 | 파생 클래스에서 | 외부(main)에서 |
| public | 참조 가능 | 참조 가능 | 참조 가능 |
| **protected** | **참조 가능** | **참조 가능** | **참조 불가능** |
| private | 참조 가능 | 참조 불가능 | 참조 불가능 |



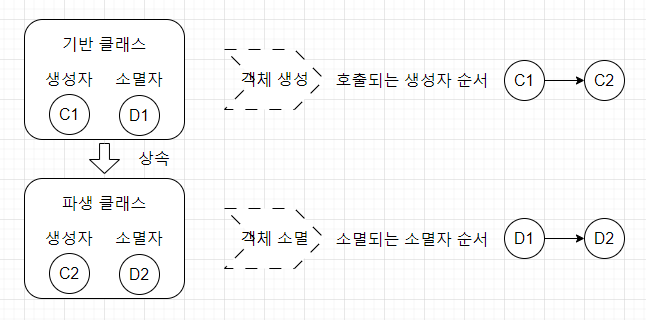


위와 같이, A 를 기반으로 하는 B 파생클래스를 선언하였고



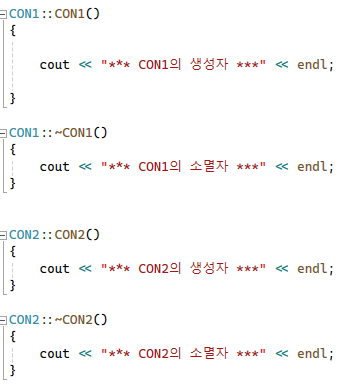
B 파생 클래스에서 A 클래스의 Protected 지정자로 해둔 proVar\_A 맴버변수를 참조가능한것을 볼 수 있다.

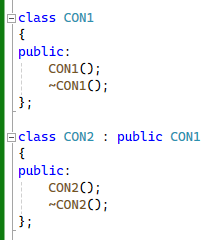
**생성자와 소멸자)**

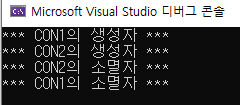
 앞서 배운 클래스의 생성자와 소멸자 형식은 동일하지만,  
클래스 상속에서 생성자와 소멸자 호출 순서는 다음과 같다.

D1

D2

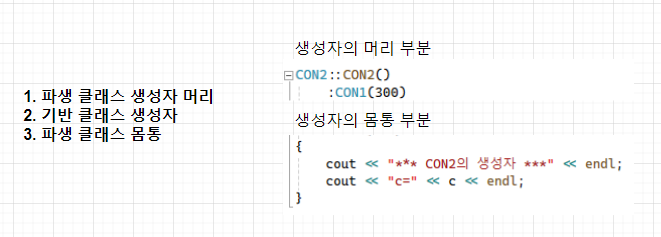
예시)



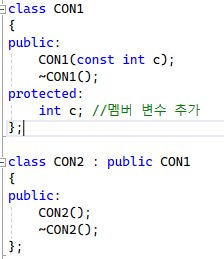
실행하면 아래 그림과 같이 기반 생성자 - 파생 생성자 - 파생 소멸자 - 기반 소멸자 이다.

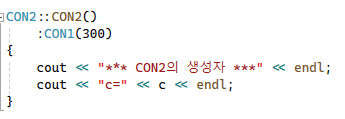
**유의 사항 1)**

더욱 정확한 순서는,



그렇기 때문에, 만일 기반 클래스의 생성자가 매개변수를 필요로 하는데   
파생 클래스의 생성자 머리에서 초기화 시켜주지 않으면 **에러**가 발생한다.

예시)



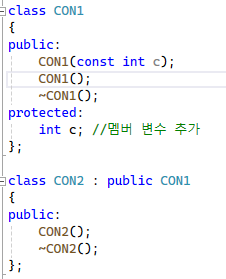
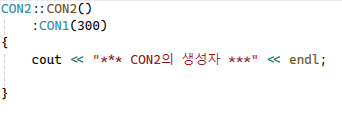
왼쪽 사진처럼 CON1의 생성자가 매개변수를 필요하면, 파생클래스의 생성자 머리부분에서 그 값을 **전달**시켜 줘야 한다.

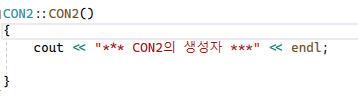
**유의 사항 2)**

생성자는 오버로딩 될 수 있으며, 기반 클래스의 생성자가 오버로딩 된 경우에는   
필요한 매개변수에 따라 알아서 실행된다.

예시)

1번





2번

1번 처럼 기반클래스의 생성자에 매개변수를 넘겨주면 매개변수가 필요한 생성자가 실행되고,

2번 처럼 기반클래스의 생성자에 아무런 매개변수도 전달하지 않으면, 필요없는 생성자가   
실행된다.

**상속: 고급**

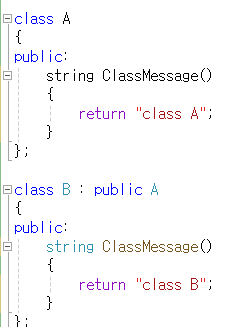
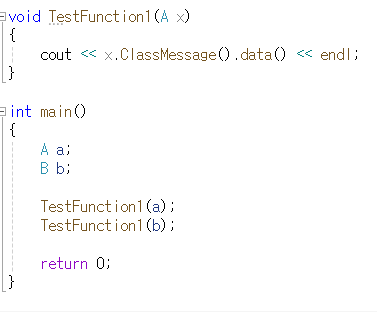
**상속에서 맴버 함수 재정의)**

기존의 클래스 내의 함수 오버로딩말고, 상속을 통한 기반클래스의 맴버함수를   
 파생클래스에서 정의 하는 방법이다.

함수 오버로딩: 함수의 매개변수를 다르게 정의하는 것을 의미한다.

- 함수의 동작이 유사하면서 다루는 대상 데이터형이 다를 경우 함수의 이름을 매번 변경하지 않아도 되는 편리함이 있다.

함수 오버라이딩: 기반클래스의 맴버 함수를 파생 클래스에서 다시 정의하는것을 의미한다.





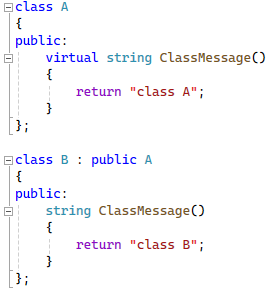
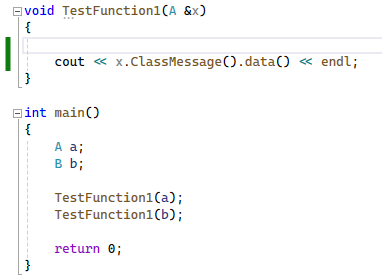
위의 예시를 보면 출력 예시가 이렇게나올거 같지만, 실제로는   
  
이렇게 나온다.

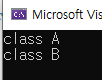
① 기반 클래스의 맴버 함수가 **가상 함수**로 선언되어야 한다.

② 함수에서 매개변수로 전달된 변수는 객체의 **주소**를 포함하여야 한다.

가상 함수는 선언할 때 virtual 을 추가된 형태이다. 그리고 함수에서 특정 클래스의 객체를  
참조하려면 단순한 재정의가 아닌 적절한 객체의 맴버를 참조하는 **동적결합**이 되어야 하고  
이렇게 하나의 함수가 여러형태로 정의 되는것을 **다형성**이라고 하는데

다형성으로 인해 각 클래스의 객체의 주소를 전달받아야 다형성이 활용된다.  
 (만일 주소를 전달안받으면 가상 함수를 설정하여도 A객체만 활용된다.)

****

****

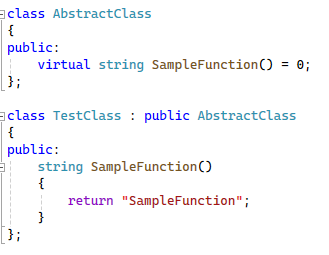
**추상 클래스)**

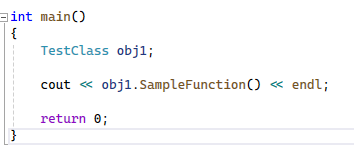
추상 클래스는 객체 지향 프로그래밍에서 중요한 특징인 다형성을 가진 함수의 집합을 정의할 수 있게 해줍니다.

**즉**, 반드시 사용해야 하는 함수를 지정함으로써, 반드시 파생 클래스에서 재정의 해야합니다.

virtual 반환형 함수이름() = 0;

추가로, 추상 클래스는 순수 가상함수를 포함하고 있어서 객체 생성이 불가하다.

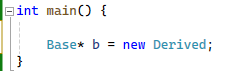
예시)



출력하면 다음과 같은 출력이 나온다.

**동적 형 변환)**

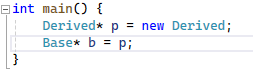
int, double 과 같은 자료 형 변환이 아닌, 기반과 파생 클래스형으로 변환이 동적 형 변환의 대상이다.



기반클래스를 Base 파생클래스를 Derived라고 했을때,

메모리 어딘가에 Derived타입의 객체가 생성이되고, 그 객체에대한 포인터를 기반클래스의 포인터가 받는것을 **업 캐스팅**이라고 한다.

이를 풀어쓰면,



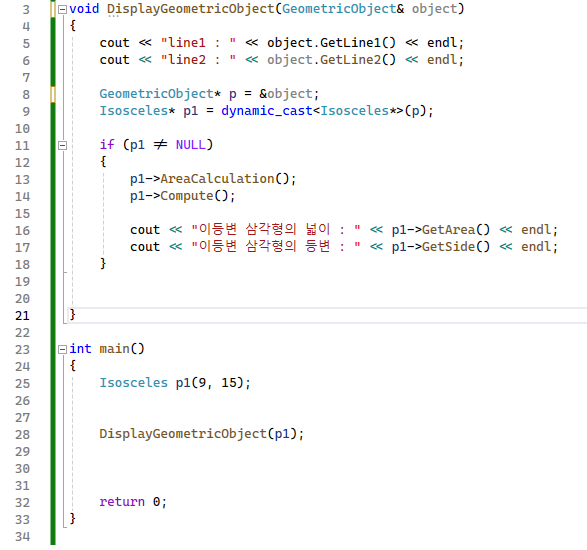
이렇게 형변환이 가능하다는 것을 알 수 있다.

**즉**, 자식 클래스 포인터 -> 부모 클래스 포인터 (묵시적)

double a=1; 이라고 했을때 묵시적으로 형변환 되는거라고 이해하면 쉽다.

**다운 캐스팅**

여기서 기반 클래스는 GeometricObject 이고, 파생 클래스는 Isosceles 이다.

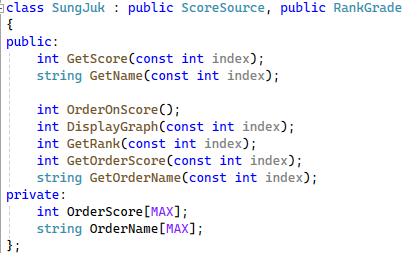


객체 포인터 1 = dynamic\_cast<클래스 이름\*>(객체 포인터 2)

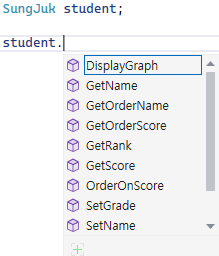
여기서 DisplayGeometricObject() 함수를 통해 파생 클래스의 객체를 넘기고,  
이를 GeometricObject\* p = &object; 업캐스팅을 하여 받고 다시 dynamic\_cast를 활용하여  
다운캐스팅 하는것을 볼 수 있다.

여기서 NULL이 출력되는 이유는 dynamic\_cast는 시도를 하는것으로,  
p가 가르키는 객체가 실제로 Isosceles 이면 성공하고 아니면 NULL을 반환하게 되는것이다.

**다중 상속)**



위의 그림과 같이 상속 받는 기반클래스를 , (콤마)로 구분해서 명시하면된다.

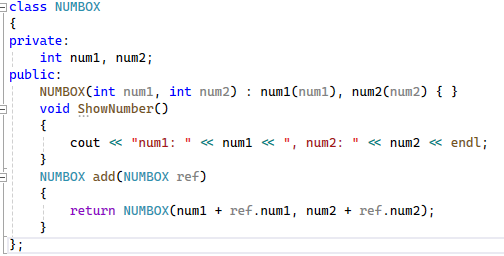


이렇게 기반클래스 2개의 맴버를 참조할 수 있다.

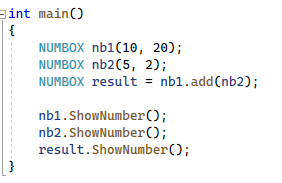
**연산자 오버로딩**

**객체 연산)**

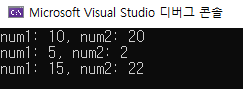
우선적으로 일반 클래스의 객체끼리 어떻게 연산이 가능한지 알아볼것이다.



위의 그림과 같이 클래스를 반환하는 add라는 맴버함수를 지정하고 매개변수로 다른 객체를 받아 연산을 하고 새로운 객체에 넣어주는 방식이 있다.

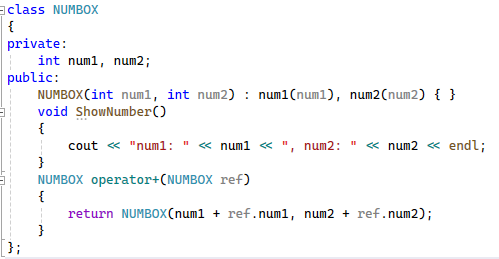


**result라는 객체**에 **nb1객체**와 **nb2객체**의 값을 더한값이 들어가게 되며 출력하게 되면 다음과 같다.

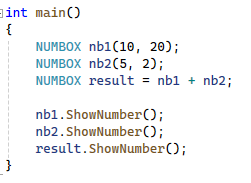


**연산자 오버로딩)**

이전의 객체연산처럼 함수로 만들어서 연산하는 방식보다   
연산기호 자체를 **재정의** 해서 더욱 간편하게 나타낼수 있는 방식이다.



함수반환형 opreator 연산자(연산 대상);



이렇게 '+' 기호에 새로운 의미를 넣어 재정의 하여 연산자 오버로딩 하는것이다.