C++ 프로그래밍

□ 상속

○ 상속의 개념

* 클래스 사이의 상속 : 객체 생성 시, 자신의 멤버 뿐 아니라 부모 클래스의 멤버를 포함한다.
* 기본 클래스 : 상속해주는 클래스, 부모 클래스
* 파생 클래스 : 상속받는 클래스, 자식 클래스
  - 기본 클래스의 속성과 기능을 물려받고 자신만의 속성과 기능을 추가
* Java, C#
* 간결한 클래스 작성
  - 코드 중복 제거와 수정 용이
* 클래스 간의 계층적 분류 및 관리의 용이함
* 클래스 재사용과 확장을 통한 소프트웨어 생산성 향상
* C++은 다중 상속을 허용

○ 상속 선언

* ' : ' 으로 상속을 표시

기본 형식

```
class Student : public Person {
        // Person을상속받는Student 선언
};
class StudentWorker : public Student  {
        // Student를상속받는StudentWorker선언
};
```

상속 - Point / ColorPoint

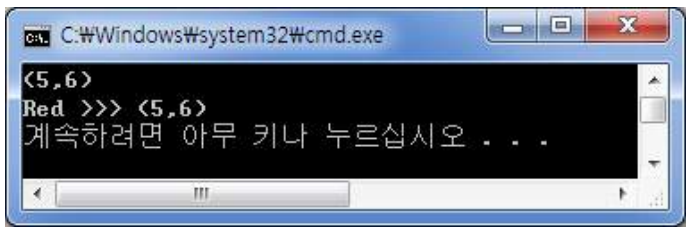
```
#include <iostream>
#include <string>

using namespace std;

#ifndef POINT_H
#define POINT_H

class Point{
private:
        int x, y;

public:
        void setPoint(int x, int y){
                this->x = x;
                this->y = y;
        }
        void showPoint(){
                cout << "(" << this->x << "," << this->y << ")" << endl;
        }
};
#endif
```




```
#include "Point.h"

#ifndef COLORPOINT_H
#define COLORPOINT_H

class ColorPoint : public Point{
protected:
        string color;

public:
        void setColor(string color){
                this->color = color;
        }
        void showColor(){
                cout << this->color << " >>> ";
                showPoint();
        }
};
#endif
```

```
#include "ColorPoint.h"
#include "Point.h"

int main(){
        ColorPoint cp;

        cp.setPoint(5, 6);
        cp.showPoint();

        cp.setColor("Red");
        cp.showColor();

        return 0;
}
```

○ 상속과 객체 포인터

* 업 캐스팅(up-casting) // casting : 배역을 정함
  - 위쪽 배역으로 결정한다. // 자식 -> 부모
  - 파생 클래스 포인터가 기본 클래스 포인터에 치환되는 것
  - 업 캐스팅 시 명시적 형 변환은 불필요
    // 부모 클래스는 하나이므로 형 변환이 필요하지 않다.

```
ani_a = kim;  //ani_a = (Animal*)kim;
```

* 다운 캐스팅
  - 아래쪽 배역으로 결정 // 부모 -> 자식
  - 기본 클래스의 포인터가 파생 클래스의 포인터에 치환되는 것
  - 다운 캐스팅 시 명시적 형 변환이 반드시 필요
    // 다운 캐스팅 시 자식 클래스는 부모에서 어느 클래스로 가야할지 지정해야하기 때문

```
kim = (Human*)ani_a;
```


```
int main() {
  ColorPoint cp;
  ColorPoint *pDer;
  Point* pBase = &cp; // 업캐스팅

  pBase->set(3,4);
  pBase->showPoint();

  pDer = (ColorPoint *)pBase; // 다운캐스팅
  pDer->setColor("Red"); // 정상 컴파일
  pDer->showColorPoint(); // 정상 컴파일
}
```

(강제 타입 변환 반드시 필요)

○ 접근 지정자

* private 멤버
  - 선언된 클래스 내에서만 접근 가능
  - 파생 클래스에서도 기본 클래스의 private 멤버 직접 접근 불가

* public 멤버
  - 선언된 클래스나 외부 어떤 클래스, 모든 외부 함수에 접근 허용
  - 파생 클래스에서 기본 클래스의 public 멤버 접근 가능

* protected 멤버
  - 선언된 클래스에서 접근 가능 (자식 클래스에서 사용 가능)
  - 파생 클래스에서만 접근 허용, 다른 클래스나 외부 함수에서는 protected 멤버 접근 불가

○ 상속 - 생성자 및 소멸자

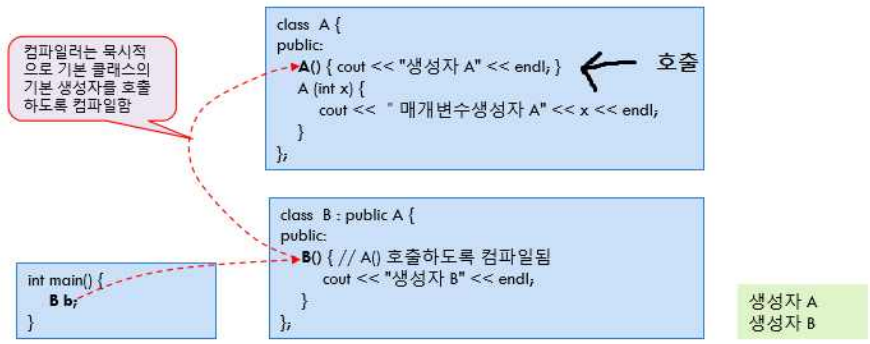
* 생성자 실행 순서 (자식 클래스의 객체 생성 시)
  - 부모 클래스의 생성자 실행 -> 자식 클래스의 생성자 실행
  - 부모가 먼저 생성 되어야 함

* 소멸자 실행 순서
  - 파생 클래스의 객체가 소멸될 때 : 파생 클래스의 소멸자 실행 -> 기본 클래스의 소멸자 실행

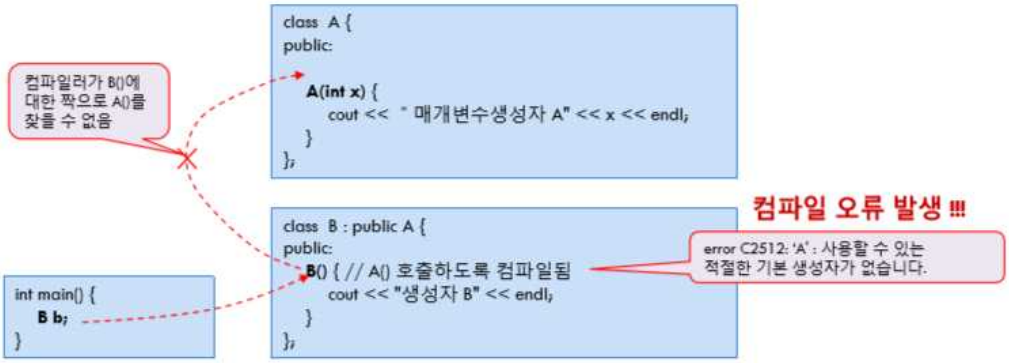
* 자식 클래스의 생성자 구현 시 함께 실행할 기본 생성자를 지정할 수 있다.

* 생성자를 지정하지 않았을 경우 컴파일러가 기본 생성자를 호출한다.

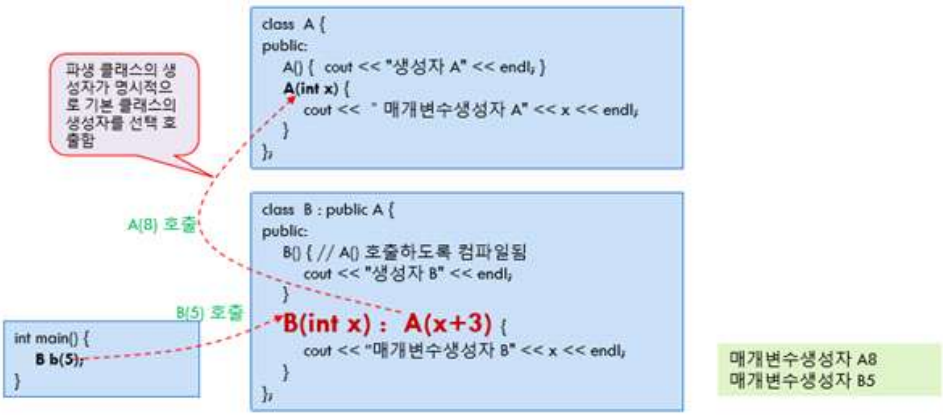
파생 클래스의 생성자에서 기본 클래스의 기본 생성자 호출



* 기본 생성자를 지정하지 않으면 컴파일 오류가 난다.



* 자식 클래스에서 기본 클래스의 생성자를 선택



○ 가상 상속

* 다중 상속으로 인한 기본 클래스 멤버의 중복 상속 해결
* 가상 상속
  - 자식 클래스의 선언문에 부모 클래스 앞에 virtual 선언
  - 자식 클래스의 객체가 생성될 때, 기본 클래스의 멤버는 오직 한 번만 생성

// 기본 클래스의 멤버가 중복하여 생성되는 것을 방지

```
class In : virtual public BaseIO { // In 클래스는 BaseIO 클래스를 가상 상속함
...
};

class Out : virtual public BaseIO { // Out 클래스는 BaseIO 클래스를 가상 상속함
...
};
```

// 과제 연습문제 4

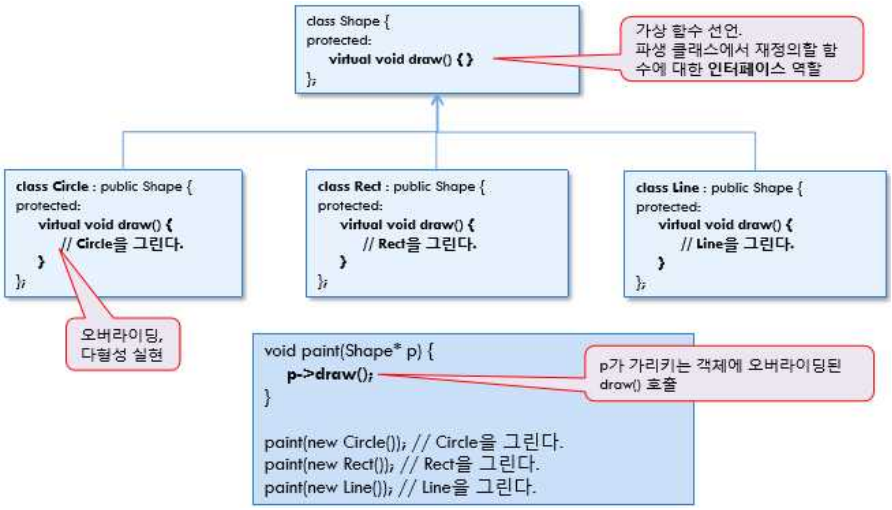
○ 오버라이딩(가상함수)

* 가상 함수(virtual function)
  - virtual : 동적 바인딩을 하도록 하는 명령어

* 동적 바인딩
  - 어떤 함수를 호출할 것인가를 결정하는 시기를 실행시간으로 미루는 것
  - 컴파일 시 호출할 함수를 결정하는 정적 바인딩과 대조

* 함수 오버라이딩
  - 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언
  - 기본 클래스의 가상 함수의 존재감을 상실시킴
  - 함수 재정의 / 다형성의 한 종류

* 오버라이딩 예
  - Circle을 Shape에 넣으면 업 캐스팅이 일어남
  - p->draw(); // 본래의 태생을 보므로 Circle의 p->draw()를 부른다.



상속 - Overriding

```
#include "Shape.h"

void main(){
        Shape *sp1, *sp2, *sp3;
        Circle C;
        Line L;
        Rectangle R;

        sp1 = &C;
        sp2 = &L;
        sp3 = &R;

        cout << "[동적바인딩]" << endl;
        sp1->paint();
        sp2->paint();
        sp3->paint();
}
```

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

class Shape{
public:
        void paint(){
                draw();
        }
protected:
        virtual void draw(){
                cout << "Shape Draw" << endl;
        }
};

class Circle : public Shape{
public:
        virtual void draw(){
                cout << "Circle Draw" << endl;
        }
};

class Line : public Shape{
public:
        virtual void draw(){
                cout << "Line Draw" << endl;
        }
};

class Rectangle : public Shape{
public:
        virtual void draw(){
                cout << "Rectangle Draw" << endl;
        }
};
```

* 오버라이딩 (가상함수)
  - 다른 도형들을 효율적으로 관리하기 위해 필요함

* 업캐스팅
  - 업캐스팅을 사용하여 Shape로 올라가 맞는 형태로 바꾼다.

* 오버라이딩 성공 조건
  - 가상 함수 이름, 매개 변수 타입과 개수, 리턴 타입이 모두 일치해야 함
  - 가상 함수의 virtual 지시어는 상속되므로 파생 클래스에서는 virtual 생략 가능

```
class Base {
public:
    virtual void fail();
    virtual void success();
    virtual void g(int);
};

class Derived : public Base {
public:
    virtual int fail(); // 오버라이딩 실패. 리턴 타입이 다름
    virtual void success(); // 오버라이딩 성공
    virtual void g(int, double); // 오버로딩 사례. 정상 컴파일
};
```

```
class Base {
public:
    virtual void f();
};

class Derived : public Base {
public:
    virtual void f(); // virtual void f()와 동일한 선언
};
```

생략 가능

* 상속이 반복되는 경우
  - 부모 클래스 뿐만 아니라 할아버지 클래스까지 있는 경우도 동적바인딩이 된다.
  - 상속 반복에 상관없이 처음 생성된 클래스를 따라서 오버라이딩이 된다.

* 범위지정 연산자(::)
  - 기본클래스::가상함수() 형태로 기본 클래스의 가상 함수를 정적 바인딩으로 호출

상속 - Overriding -> 범위 지정 연산자(::)

```
#include "Shape.h"

void main(){
        Shape *sp1, *sp2, *sp3;
        Circle C;
        Line L;
        Rectangle R;

        sp1 = &C;
        sp2 = &L;
        sp3 = &R;

        cout << "[동적바인딩]" << endl;
        sp1->paint();
        sp2->paint();
        sp3->paint();
}
```

```
선택 C:₩WINDOWS₩system32₩...
[동적바인딩]
Circle Draw
Line Draw
Rectangle Draw
계속하려면 아무 키나 누르십
```

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

class Shape{
public:
        void paint(){
                draw();
        }
protected:
        virtual void draw(){
                cout << "Shape Draw" << endl;
        }
};

class Circle : public Shape{
public:
        virtual void draw(){
                cout << "Circle Draw" << endl;
        }
};

class Line : public Shape{
public:
        virtual void draw(){
                cout << "Line Draw" << endl;
        }
};

class Rectangle : public Shape{
public:
        virtual void draw(){
                cout << "Rectangle Draw" << endl;
        }
};
```

○ 순수 가상함수

* 순수 가상함수

- 함수의 코드가 없고 선언만 있는 가상 멤버 함수
- 순수 가상함수를 하나라도 가지고 있으면 추상 클래스가 된다.
- 선언 방법 : 멤버 함수의 원형=0; 으로 선언

* 추상클래스

- 추상 클래스는 객체를 생성할 수 없다.
- 추상 클래스의 포인터는 선언할 수 있다.
- 상속에서 기본 클래스의 역할을 하기 위함
  - 순수 가상 함수를 통해 파생 클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할
  - 추상 클래스의 모든 멤버 함수를 순수 가상 함수로 선언할 필요 없음
  - 파생클래스에서 구현해야 할 함수만 순수 가상함수로 구현

* 추상 클래스의 상속

- 추상 클래스를 단순 상속하면 자동 추상 클래스가 됨

* 추상 클래스의 구현

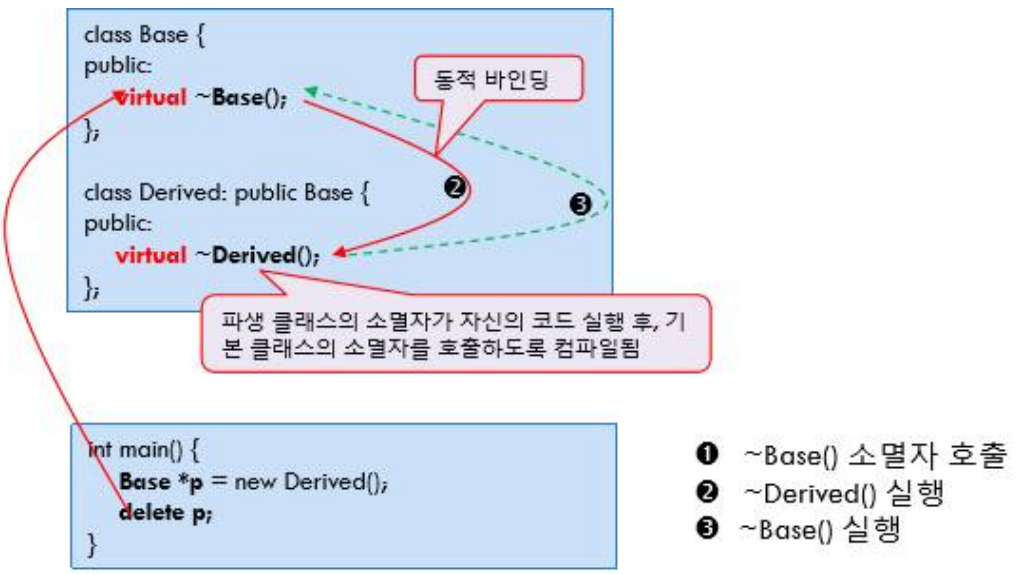
- 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩하면 파생 클래스는 일반 클래스가 됨

```
class Shape { // Shape은 추상 클래스
  Shape *next;
   public:
  void paint() {
    draw();
  }
  virtual void draw() = 0; // 순수 가상 함수
};
void Shape::paint() {
  draw(); // 순수 가상 함수라도 호출은 할 수 있다.
}
```

○ 가상 소멸자

* 가상 소멸자

- 상속관계에서 파생 클래스의 소멸자도 수행하기 위해 소멸자를 가상 소멸자로 선언한다.
- 소멸자를 virtual 키워드로 선언해서 소멸자 호출 시 동적 바인딩이 되도록 한다.



□ 표준 템플릿 라이브러리(STL)

○ 제네릭 함수

* 제네릭(일반화) 함수
  - 함수나 클래스를 일반화시키고 매개 변수 타입을 지정하여 틀에서 찍어내듯 코드를 생산
  - 매개변수 타입마다 다른 함수를 만들 필요가 없다.

* 템플릿(Template) : 틀
  - 함수나 클래스를 일반화시키는 키워드 // 오버로딩을 일일이 입력할 필요가 없다.

* 제네릭 타입 선언
  - 일반화를 위해 필요한 데이터 타입
  - 구체화 : 들어오는 매개변수 타입에 따라 타입이 바뀐다.

기본 형식

```
templat <class 이름>       // 1개의 제네릭 타입을 가진 템플릿 선언
templat <typename 이름>  // 1개의 제네릭 타입을 가진 템플릿 선언
templat <class a1, class a2, class a3> // 3개의 제네릭 타입을 가진 템플릿 선언

void myswap (a1 &a, a1 &b){  // 제네릭 함수 a1이라는 타입을 정한다.
    a1 tmp;   // a1이 들어오는 매개변수 타입으로 바뀐다. (객체도 가능)
    tmp = a;
    a = b;
    b = tmp;
}
```

○ 제네릭 클래스

* 제네릭(일반화) 클래스
  - 제네릭 클래스의 선언과 구현은 모두 헤더 파일에서 함

제네릭 클래스 예제 - MyStack

```
/* 제네릭 클래스 선언 */
template <class T>
class MyStack {
    int top;
    T data[100]; // T타입의 배열
public:
    MyStack(){ tos = -1; };
    void push(T element);
    T pop();
};

/* 제네릭 클래스 구현 */
template <class T>
void MyStack<T>::push(T value){
    data[++top] = value;
}

template <class T>
T MyStack<T>::pop(){
    return data[top--;]
}
```

```
/* 클래스 구체화 및 객체 활용 */
#include <iostream>
using namespace std;

#include "MyStack.h"

void main(){
        MyStack<int> iStack; // <데이터타입>
        MyStack<double> dStack;
        iStack.push(3);
        int n = iStack.pop();

        dStack.push(3, 5);
        double d = dStack.pop();
}
```

○ STL : 표준 템플릿 라이브러리

* 컨테이너 - 템플릿 클래스

- 데이터를 담아두는 자료 구조를 표현한 클래스
- 리스트, 큐, 스택, 맵, 셋, 벡터

컨테이너 클래스	설명	헤더 파일
vector	동적 크기의 배열을 일반화한 클래스	<vector>
deque	앞 뒤 모두 입력 가능한 큐 클래스	<deque>
list	빠른 삽입/삭제 가능한 리스트 클래스	<list>
set	정렬된 순서로 값을 저장하는 집합 클래스, 값은 유일	<set>
map	(key, value) 쌍으로 값을 저장하는 맵 클래스	<map>
stack	스택을 일반화한 클래스	<stack>
queue	큐를 일반화한 클래스	<queue>

* iterator - 컨테이너 원소에 대한 포인터

- 컨테이너의 원소들을 순회하면서 접근하기 위해 만들어진 컨테이너 원소에 대한 포인터

iterator의 종류	iterator에 ++ 연산 후 방향	read/write
iterator	다음 원소로 전진	read/write
const_iterator	다음 원소로 전진	read
reverse_iterator	지난 원소로 후진	read/write
const_reverse_iterator	지난 원소로 후진	read

* 알고리즘 - 템플릿 함수

- 컨테이너 원소에 대한 복사, 검색, 삭제, 정렬 등의 기능을 구현한 템플릿 함수
- 컨테이너의 멤버 함수가 아닌 전역 함수

STL 알고리즘 함수			
copy	merge	random	rotate
equal	min	remove	search
find	move	replace	sort
max	partition	reverse	swap

* STL 관련 헤더 파일과 이름 공간

* 헤더파일

- 컨테이너 클래스를 사용하기 위한 헤더 파일
  - vector 클래스 : #include <vector>
  - list 클래스 : #include <list>

- 알고리즘 함수를 사용하기 위한 헤더 파일
  - #include <algorithm> // 알고리즘 함수에 상관 없음

* 이름 공간

- std : STL이 선언된 이름 공간

○ STL 컨테이너 - vector

* vector 특징
  - 가변 길이 배열을 구현한 제네릭 클래스 // 벡터의 길이에 제한이 없음
  - 원소의 저장, 삭제, 검색 등 다양한 멤버 함수 지원
  - 벡터에 저장된 원소는 인덱스로 접근 가능 // 0부터 시작

* vector 클래스의 주요 멤버와 연산자

멤버와 연산자 함수	설명
push_back(element)	벡터의 마지막 element 추가
at(int index)	index 위치의 원소에 대한 참조 리턴
begin()	벡터의 첫 번째 원소에 대한 참조 리턴
capacity()	벡터의 용량 리턴
end()	벡터의 끝(마지막 원소 다음)을 가리키는 참조 리턴
empty()	벡터가 비어있으면 true 리턴
erase(iterator it)	벡터에서 it이 가리키는 원소 삭제, 삭제 후 자동으로 벡터 조정
size()	벡터에 들어있는 원소의 개수 리턴
operater[]()	지정된 원소에 대한 참조 리턴
operator=()	이 벡터를 다른 벡터에 치환(복사)