게임 자료구조와 알고리즘

-CHAPTER14-

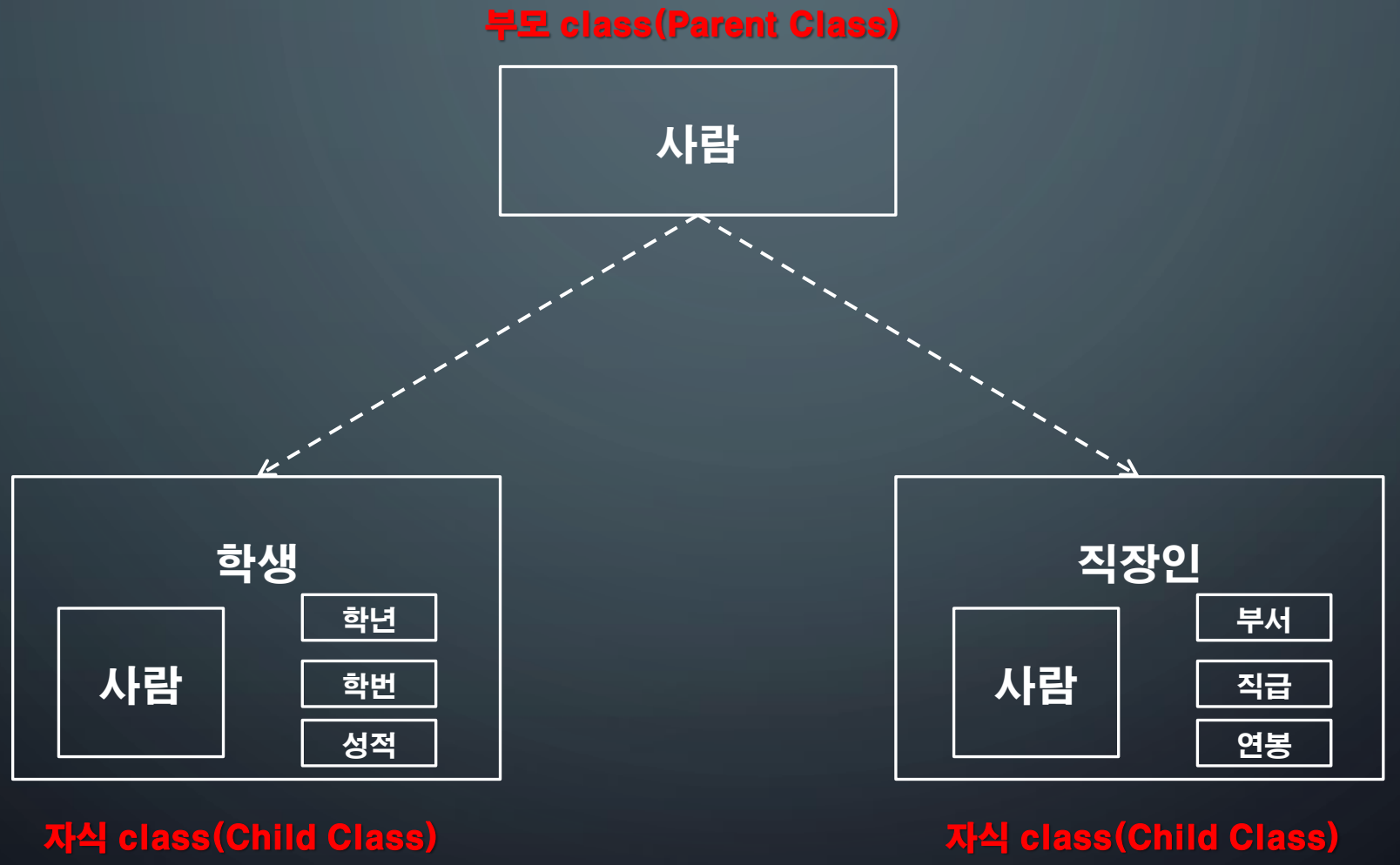
SOULSEEK

목차

상속

1. 상속

- 다른 class의 멤버 변수, 멤버 함수를 자신의 멤버인 것 처럼 사용이 가능하다.
- Class의 **재사용**과 관련이 있다.

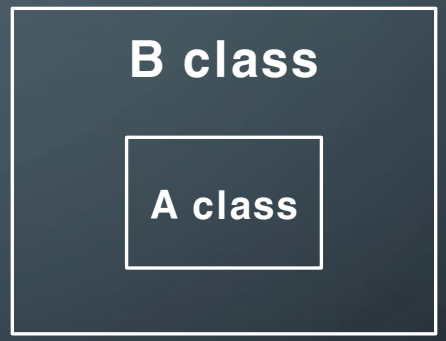


1. 상속

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

class A
{
public:
	A()
	{
		cout << "A Class 생성" << endl;
	}
};

class B : public A
{
	public:
	B()
	{
		cout << "B Class 생성" << endl;
	}
};

void main()
{
	B b;
}
```

1. 상속

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

class A
{
public:
	A()
	{
		cout << "A의 생성자 호출" << endl;
	}
	~A()
	{
		cout << "A의 소멸자 호출" << endl;
	}
};
class B : public A
{
public:
	B()
	{
		cout << "B의 생성자 호출" << endl;
	}
	~B()
	{
		cout << "B의 소멸자 호출" << endl;
	}
};
void main()
{
	B b;
}
```



1. 상속

상속의 특징

1. **has ~ a(~ 은 ~ 을 가지고 있다.) 의 관계 - 상속관계**
   Person Class / Phone Class

   Ex.
   사람 , Phone Class
   핸드폰은 사람을 가지고있다 (X)
   사람은 핸드폰을 가지고있다 (O)

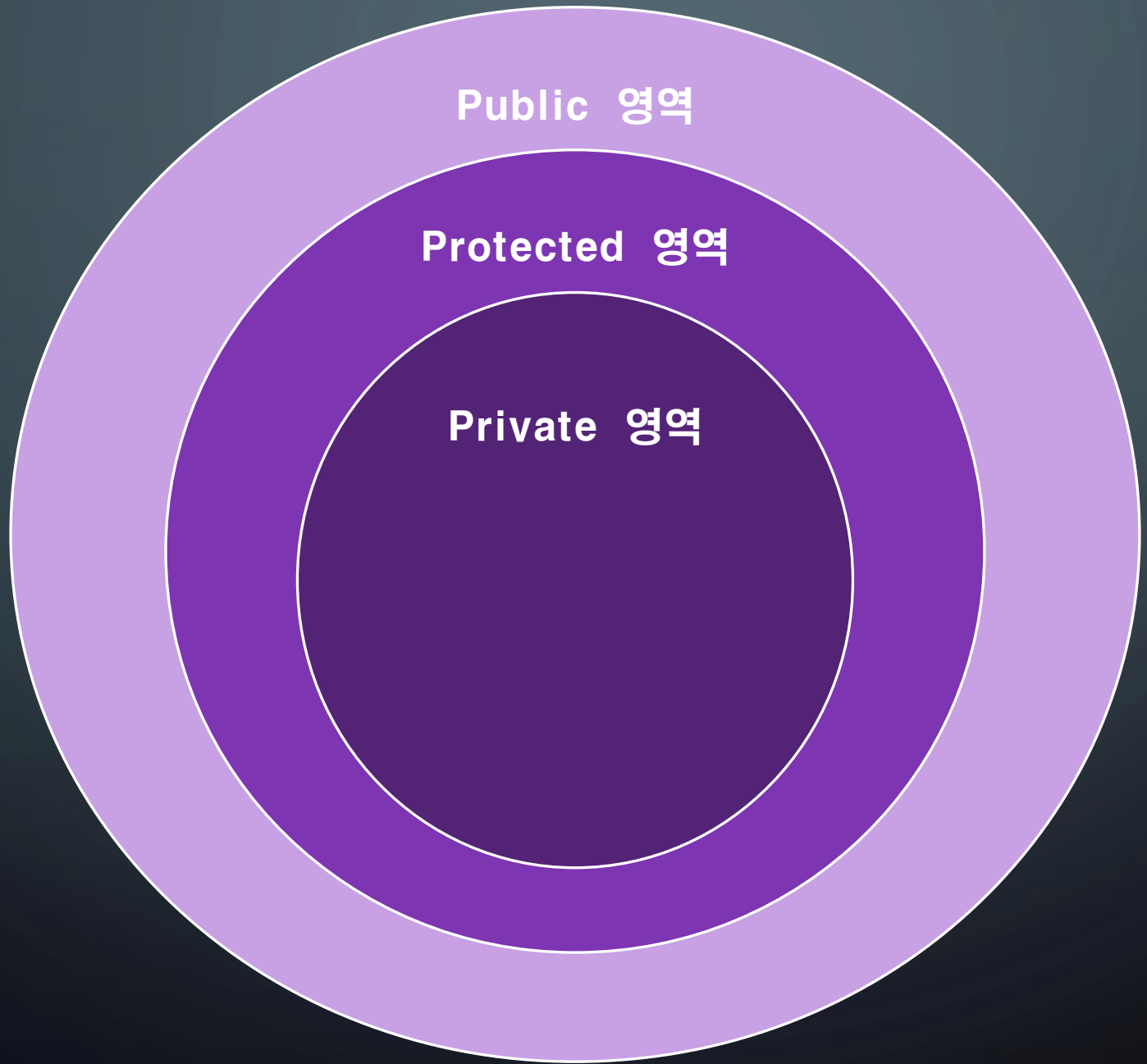
2. **is ~ a(~ 은 ~이다.) 의 관계 - 포함관계**
   Person Class / Student Class

   Ex.
   사람 , 학생 Class
   사람은 학생이다 (X)
   학생은 사람이다 (O)

1. 상속

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

class A
{
private:
	int m_ia;
protected:
	int m_ib;
public:
	int m_ic;
	void Test()
	{
		m_ia = 10;
		m_ib = 10;
		m_ic = 10;
	}
};
class B : public A
{
	void Test()
	{
		m_ia = 10;
		m_ib = 10;
		m_ic = 10;
	}
};
class C : public B
{
	void Test()
	{
		m_ia = 10;
		m_ib = 10;
		m_ic = 10;
	}
};

void main()
{
	B b;
	b.m_ia = 10;
	b.m_ib = 10;
	b.m_ic = 10;
}
```

1. 상속



1. 상속

오버라이딩(Overriding)

- 부모**class**의 함수의 처리동작이 마음에 들지 않을 경우 **동일한 이름**으로 자식**class**가 만들어 **재정의** 한다.

```
class Mammal
{
public:
	void speak(int cnt)
	{
		cout << cnt << "번 짖다" << endl;
	}
	void speak()
	{
		cout << "짖다" << endl;
	}
};
class Dog : public Mammal
{
public:
	void speak()
	{
		cout << "멍멍" << endl;
	}
};
```

```
void main()
{
	Mammal dongmul;
	Dog jindo;
	dongmul.speak();
	dongmul.speak(3);
	jindo.speak();
	//jindo.speak(5);
}
```

1. 상속

UpCasting

- 자식**Class**객체의 **주소값**을 **부모Class 포인터 변수**에 담아 사용한다.
- 여러 자식**Class**의 **부모가 동일** 할 경우 해당 부모 **Class**에 여러 자식**Class**를 **모아** 일괄 처리한다.**(ex. 부모 : 동물 자식 : 고양이, 강아지,원숭이)**

```
class Mammal
{
public:
	void speak(int cnt)
	{
		cout << cnt << "번 짖다" << endl;
	}
	void speak()
	{
		cout << " 짖다" << endl;
	}
};
class Dog : public Mammal
{
public:
	void speak()
	{
		cout << "멍멍" << endl;
	}
};
```

```
void main()
{
	Mammal* ptr;
	Dog jindo;
	ptr = &jindo;
	ptr->speak();
	ptr->speak(5);
}
```

1. 상속

학습과제

Login Class를 만들어 Computer Class 와 함께 상속관계를 만들어 보자.

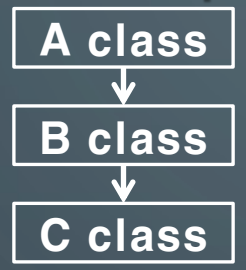
다중상속

2. 다중상속

- 부모 **class**가 **둘 이상**으로 상속 받은 경우
- 기본적으로 **C++에서만 지원**하며 다른 언어에서는 지양하는 편

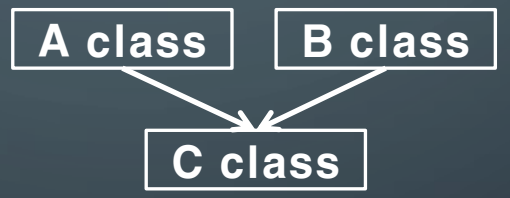
해당 자식 Class가 다른 Class의 부모가 되는 경우

```
class A : {};
class B : public A{};
class C : public B{};
```



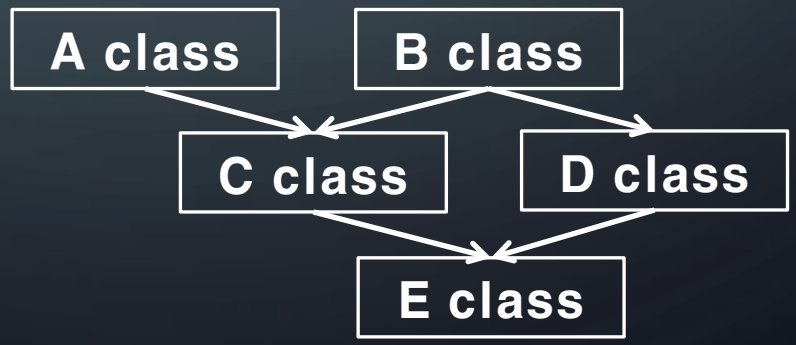
해당 class 의 부모가 2개 이상 인 경우

```
class A : {};
class B : {};
class C : public A,public B {};
```



두 가지가 혼합된 형태

```
class A : {};
class B : {};
class C : public A,public B {};
class D : public B{};
class E : public C,public D {};
```



2. 다중상속

```
class A
{
public:
	void func1()
	{
		cout << "A함수 입니다." << endl;
	}
};
class B
{
public:
	void func2()
	{
		cout << "B함수 입니다." << endl;
	}
};

class C : public A, public B
{
public:
	void func3()
	{
		func1();
		func2();
	}
};
```

```
void main()
{
	C c;
	c.func3();
}
```

2. 다중상속

```
class A
{
public:
	void func()
	{
		cout << "A함수 입니다." << endl;
	}
};
class B
{
public:
	void func()
	{
		cout << "B함수 입니다." << endl;
	}
};

class C : public A, public B
{
public:
	void func3()
	{
		func();
		func();
	}//에러
};
```

```
void main()
{
	C c;
	c.func3();
}
```

2. 다중상속

```
class A
{
public:
	void func()
	{
		cout << "A함수 입니다." << endl;
	}
};
class B
{
public:
	void func()
	{
		cout << "B함수 입니다." << endl;
	}
};

class C : public A, public B
{
public:
	void func3()
	{
		A::func();
		B::func();
	}//에러
};
```

```
void main()
{
	C c;
	c.func3();
}
```

2. 다중상속

```
class A
{
public:
	A()
	{
		cout << "A함수 생성자." << endl;
	}
};
class B : public A
{
public:
	B()
	{
		cout << "B함수 생성자." << endl;
	}
};
class C : public A
{
public:
	C()
	{
		cout << "C함수 생성자." << endl;
	}
};
class D : public B, public C
{
public:
	D()
	{
		cout << "D함수 생성자." << endl;
	}
};
```

```
void main()
{
	D d;
}
```

2. 다중상속

학습과제

- 성별,나이,이름을 저장하고 출력하는 **Person Class** 와 학년,반,번호 를 저장하고 출력하는 **School Class**를 만들고 위의 두Class를 상속받은 Student **Class**를 만들어 출력하시오. **(단 Person Class와 School Class의 맴버함수는 Student Class에서만 호출가능)**

가상함수

3. 가상함수

- 자식의 주소를 부모 포인터에 **업캐스팅** 했을 시 **오버라이딩**된 함수를 사용하게 해주는 방법
- 부모class 에서 작성한다.
- 소멸자 함수도 virtual 을 써주어야 한다.
- 형식 : virtual 반환자료형 함수이름(매개변수)

```
class Bumo
{
public:
	void func()
	{
		cout << "Bumo함수 입니다." << endl;
	}
};
class Jasic : public Bumo
{
public:
	void func()
	{
		cout << "Jasic함수 입니다." << endl;
	}
};
```

```
void main()
{
	Jasic ob;
	Bumo* mom = &ob;
	mom->func();
}
```

3. 가상함수

```
class Bumo
{
public:
	virtual void func()
	{
		cout << "Bumo함수 입니다." << endl;
	}
};
class Jasic : public Bumo
{
public:
	void func()
	{
		cout << "Jasic함수 입니다." << endl;
	}
};
void main()
{
	Jasic ob;
	Bumo* mom = &ob;
	mom->func();
}
```

3. 가상함수

순수가상함수

- 부모 **class**에서는 **사용하지 않는 함수** 이지만 자식**class**는 무조건 동일한 이름으로 함수를 **오버라이딩**을 적용하게 끔 **강제**하는 방법
- 형식 **: virtual** 반환자료형 함수이름**(**매개변수**) = 0;**

가상함수 **Code,** 순수가상함수 **Code**를 확인해보자**.**

3. 가상함수

다중 상속으로 인한 부모**class**를 **여러 번 생성** 못하게 끔 제어하는 방법

```
class A
{
public:
	A()
	{
		cout << "A함수 생성자." << endl;
	}
};
class B : public A
{
public:
	B()
	{
		cout << "B함수 생성자." << endl;
	}
};
class C : public A
{
public:
	C()
	{
		cout << "C함수 생성자." << endl;
	}
};
```

```
class D : public B, public C
{
public:
	D()
	{
		cout << "D함수 생성자." << endl;
	}
};

void main()
{
	D d;
}
```

3. 가상함수

```
class A
{
public:
	A()
	{
		cout << "A함수 생성자." << endl;
	}
};
class B : virtual public A
{
public:
	B()
	{
		cout << "B함수 생성자." << endl;
	}
};
class C : virtual public A
{
public:
	C()
	{
		cout << "C함수 생성자." << endl;
	}
};
```

```
class D : public B, public C
{
public:
	D()
	{
		cout << "D함수 생성자." << endl;
	}
};

void main()
{
	D d;
}
```

3. 가상함수

학습과제

- **RPG**게임을 객체지향으로 설계 한 후에 무기와 상점기능 추가.

FRIEND

4. FRIEND

- **Friend 멤버 함수 :** 전역 함수를 **Friend** 선언화 함으로써 멤버 함수처럼 **private 멤버 변수**에 접근이 가능하다.
- **Friend Class :** 다른 **Class**를 마치 **자신의 Class 처럼 private**에 접근할 수 있다.
- **Friend** 전역 함수 **:** 자신의 멤버 함수를 **전역 함수로 변환시킬** 수 있다.

-장점

1. 연산자 오버로딩등 예외적인 경우에 필요하다.
2. **Class**사용 하는 측면에서 추가적으로 **private** 영역에 접근하는 함수를 만들지 않아도 되어서 편리하다.

-단점

1. 객체 지향개념이 모호해진다.
2. **Class**간의 의존도를 높혀 유연성 있는 코드가 아니게 된다.

4. FRIEND

Friend 멤버 함수

```
class A
{
private:
	int x, y;
public:
	friend void Setxy(A& a);
	A() { x = 0; y = 0; }
	void Showxy()
	{
		cout << "x = " << x << endl;
		cout << "y = " << y << endl;
	}
};

void Setxy(A& a)
{
	cout << "x, y좌표 입력 : " << endl;
	cin >> a.x >> a.y;
}
void main()
{
	A a;
	a.Showxy();
	Setxy(a);
	a.Showxy();
}
```

4. FRIEND

Friend Class

```
class A
{
private:
	int x, y;
public:
	friend class B;
	A(int a, int b) : x(a), y(b) { }

	void Showxy()
	{
		cout << "x = " << x << endl;
		cout << "y = " << y << endl;
	}
};
class B
{
private:
	int x, y;
public:
	B() : x(0), y(0) {}

	void GetA(A &a)
	{
		x = a.x; y = a.y;
	}

	void Showxy()
	{
		cout << "x = " << x << endl;
		cout << "y = " << y << endl;
	}
};
```

```
void main()
{
	A a(10, 15);
	B b;
	a.Showxy();
	b.Showxy();
	b.GetA(a);
	a.Showxy();
	b.Showxy();
}
```

4. FRIEND

Friend 전역 함수

```
class A
{
private:
	int x, y;
public:
	A(int a, int b) : x(a), y(b) { }
	friend void Showxy(A& a)
	{
		cout << "x = " << a.x << endl;
		cout << "y = " << a.y << endl;
	}
};

void main()
{
	A a(10, 15);
	Showxy(a);
}
```

예외처리

4. 예외처리

- 해당 코드를 실행중에 발생 할 수 있는 오류를 처리할 수 있게 미리 예방하는 방법.

```
ex.
try
{
    오류가 발생할 수 있는 부분
    throw 발생한 오류정보
}
catch(매개변수)
{
    예외처리
}

void main()
{
    int i = 5, j = 0;
    if (j == 0)
        cout << j << " :으로 나눌 수 없습니다." << endl;
    else
        cout << i / j << endl;
}
```

4. 예외처리

```
void main()
{
	int i = 5, j = 0;
	try
	{
		if (j == 0)
		throw j;
		cout << i / j << endl;
	}
	catch (int k)
	{
		cout << "0으로 나눌 수 없습니다." << endl;
	}
}
```

4. 예외처리

```
void main()
{
	int i = 5, j = 0;
	try
	{
		if (j == 0)
		throw "j가 0\n";
		cout << i / j << endl;
	}
	catch (int k)
	{
		cout << " 0으로 나눌 수 없습니다." << endl;
	}
	catch (char* k)
	{
		cout << k << endl;
	}
}
```