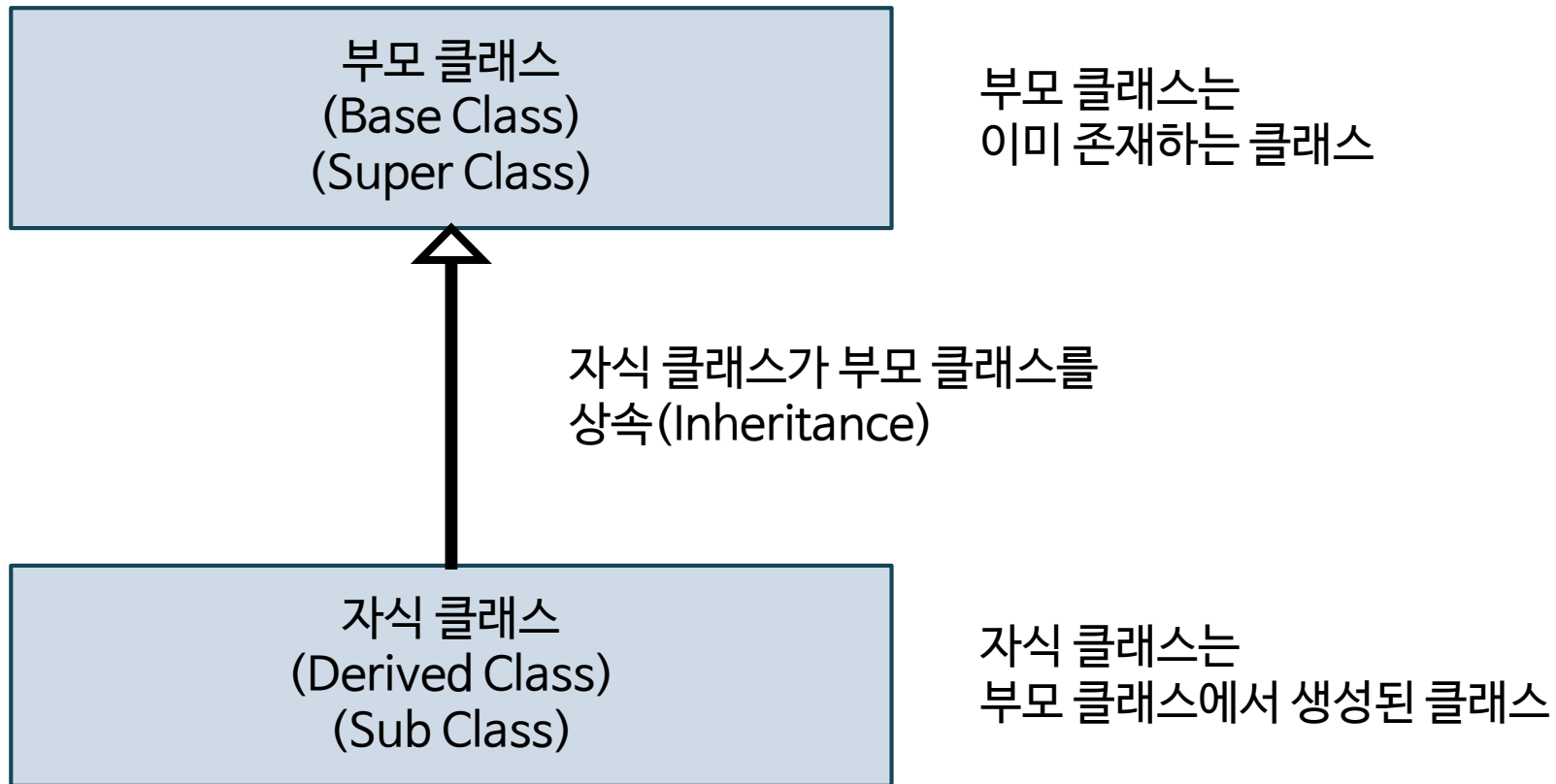




C++ 스터디

8. 상속(Inheritance)

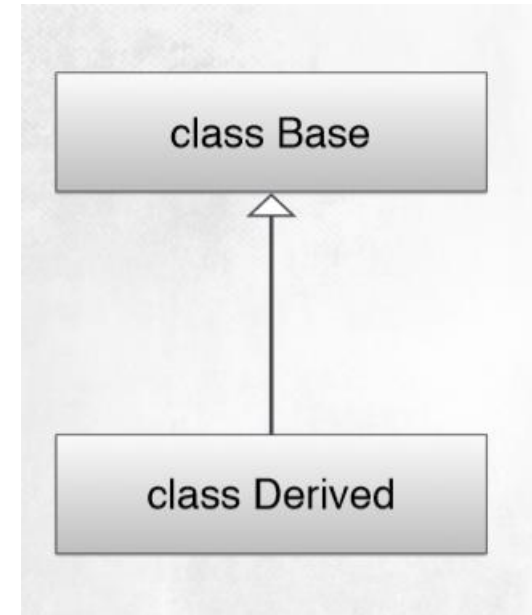
개념 정리



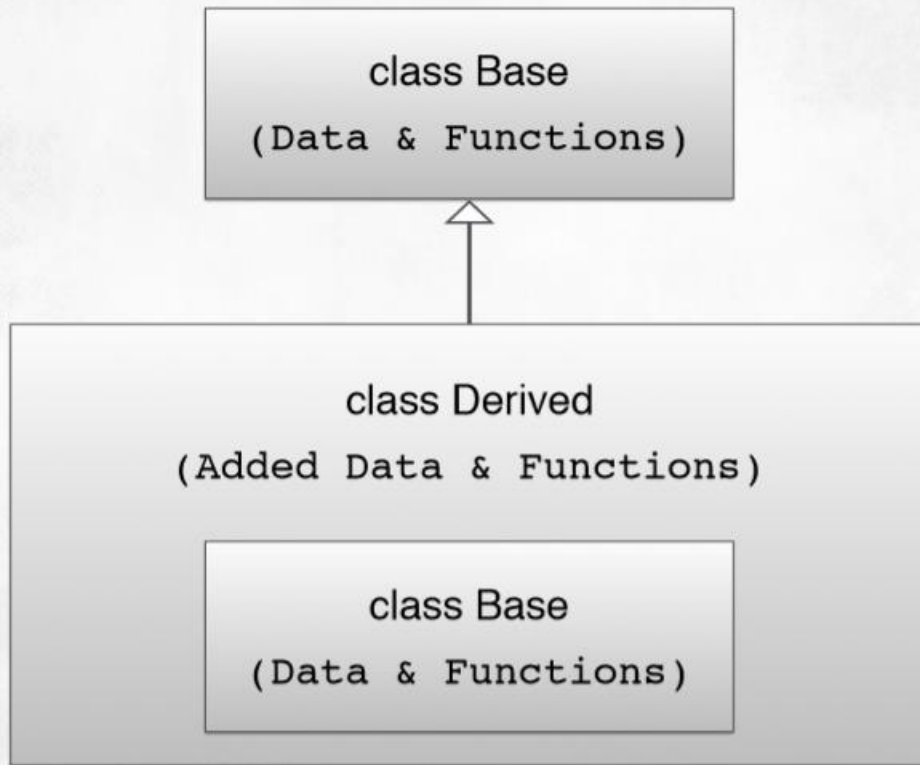
상속하는 법!

```
1
2 class Base {
3     |
4     |};
5
6 class Derived : public Base {
7     |
8     |};
9
```

위와 같이
: public (상속받을 클래스 이름)
을 중간에 끼워주면 된다!



부모 클래스, 자식 클래스



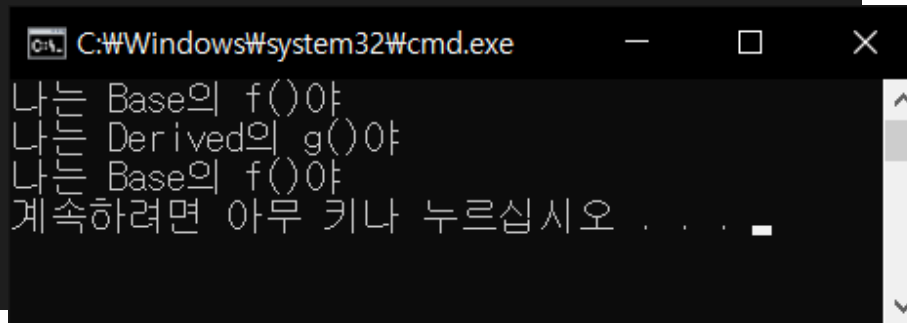
부모 클래스의 모든 멤버변수와
멤버함수(메소드)를 가지고 있다.
즉, 부모+ α 의 클래스

부모 클래스의 public

```
1
2  #include <iostream>
3  using namespace std;
4
5  class Base {
6  public:
7      void f() { cout << "나는 Base의 f()야" << endl; }
8  };
9
10 class Derived : public Base {
11 public:
12     void g() { cout << "나는 Derived의 g()야" << endl; }
13 };
14
15 int main() {
16     Base myB;
17     Derived myD;
18     myB.f();
19     myD.g();
20     myD.f();
21 }
22
```

Derived 클래스의 객체가
Base 클래스의 메소드를
사용하는 모습

Base 클래스의 모든 것을
Derived 클래스도
가지고 있기 때문



```
C:\Windows\system32\cmd.exe
나는 Base의 f()야
나는 Derived의 g()야
나는 Base의 f()야
계속하려면 아무 키나 누르십시오...
```

부모 클래스의 private

```
1
2  #include <iostream>
3  using namespace std;
4
5  class Base {
6  private:
7      int n = 1;
8  };
9
10 class Derived : public Base {
11 public:
12     int get_n() { return n; }
13 };
14
15 int main() {
16     Base myB;
17     Derived myD;
18     cout << myD.get_n() << endl;
19 }
20
```

int Base::n

멤버 "Base::n" (선언된 줄 7)에 액세스할 수 없습니다.

Derived 클래스의 **내부**에서
Base 클래스의 private값을
가져올 수가 없다...?!

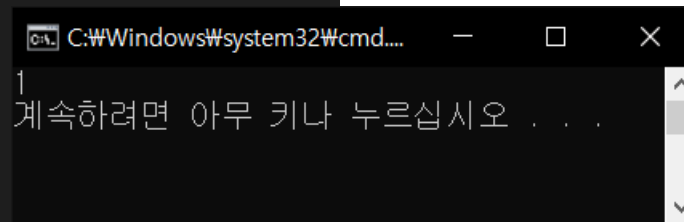
아무리 자식클래스여도 부모의
사생활을 침범할 수는 없는
법이기 때문!

새로운 접근제어지시자 protected

```
1
2     #include <iostream>
3     using namespace std;
4
5     class Base {
6     protected:
7         int n = 1;
8     };
9
10    class Derived : public Base {
11    public:
12        int get_n() { return n; }
13    };
14
15    int main() {
16        Base myB;
17        Derived myD;
18        cout << myD.get_n() << endl;
19    }
20
```

이럴때 필요한 것이
protected라는
접근제어지시자이다!

protected를 사용한 부분은
상속으로 넘어갈 경우 private로
취급되기 때문에 자식클래스의
내부에서도 사용이 가능하다!



새로운 접근제어지시자 protected

```
1
2  #include <iostream>
3  using namespace std;
4
5  class A
6  {
7  private:
8      int a = 1;
9  };
10
11  class B
12  {
13  protected:
14      int b = 2;
15  };
16
17  int main()
18  {
19      A a;
20      B b;
21      cout << a.a << b.b << endl;
22  }
```

int B::b

멤버 "B::b" (선언됨 줄 14)에 액세스할 수 없습니다.

클래스 외부에서 접근할 수 없다는 점은 private와 같다.

역할을 같지만 상속되는 방식의 차이인 것!

새로운 접근제어지시자 protected

표로 정리하면 다음과 같다!

	상속되기 전	상속된 후
public	어디서든 사용 가능!	어디서든 사용 가능!
protected	클래스 내부에서 접근 가능	클래스 내부에서 접근 가능
private	클래스 내부에서 접근 가능	접근하기 힘든 깊은 곳으로...

다음 슬라이드 내용



새로운 접근제어지시자 protect

```
1
2  #include <iostream>
3  using namespace std;
4
5  class Base {
6  public:
7      int get_public() { return n; }
8  protected:
9      int get_protected() { return n; }
10 private:
11     int n = 1;
12 };
13
14 class Derived : public Base {
15 public:
16     int get_n() { return get_protected(); }
17 };
18
19 int main() {
20     Base myB;
21     Derived myD;
22     cout << myD.get_n() << endl;
23     cout << myD.get_public() << endl;
24 }
25
```

상속된 private는 사실 아예 접근이 불가능한 것은 아니다...!

접근이 가능한 메소드를 통해서 깊은 곳에 빠진 데이터를 가져올 수 있다.

상속 타입

```
1
2 class Base {
3     |
4     };
5
6 class Derived : public Base {
7     |
8     };
9
```

사실 위의 빨간 상자가 표시한 곳은 ‘접근제어지시자’의 자리이다!

따라서 public뿐만 아니라 private와 protected도 사용될 수 있으며,
지금까지의 내용은 public타입의 상속에 관한 내용이었다!

상속과 접근제어지시자

잘 정리되어 있는 표를 가져왔습니다...!

```
class Base {  
};  
class Derived : {TYPE} Base {  
};
```

다음 페이지에 간단하게 한글로도
정리해 드릴테니
그거 보고 다시 보러 오세요

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	public in derived class. Can be accessed directly by any non- static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Protected	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Private	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.

상속과 접근제어지시자

위 슬라이드의 표는 좀 복잡하게 서술되어 있어서 한글로 간단히 표를 만들어 드리죠

		상속타입		
		public	protected	private
부모 클래스의 접근제어지시자	public	상속 타입을 따라감!		
	protected	protected		private
	private	깊은 곳으로...		

상속한 클래스를
상속할 경우 차이가 있음

만약 한번만 상속한다고 하면...?

부모 클래스의 접근제어지시자	public	상속 타입을 따라감!
	protected	어쨌든 클래스 내부에서만 접근 가능!
	private	깊은 곳으로...

훨씬 간단하게
정리할 수 있다!

상속과 접근제어지시자

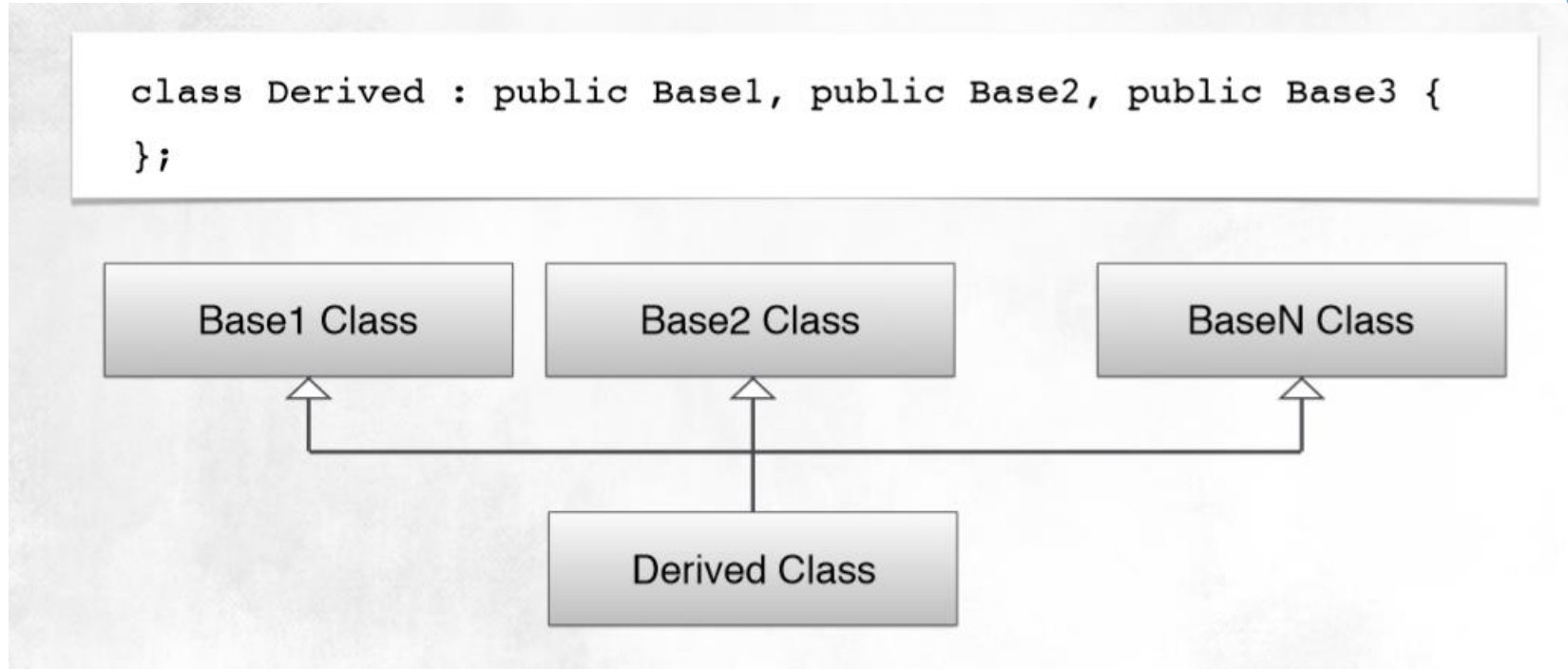
```
1
2  #include <iostream>
3  using namespace std;
4
5  class Base {
6  public:
7      void f() { cout << "나는 Base의 f()야" << endl; }
8  };
9
10 class Derived : private Base {
11 public:
12     void g() { cout << "나는 Derived의 g()야" << endl; }
13 };
14
15 int main() {
16     Base myB;
17     Derived myD;
18     myB.f();
19     myD.g();
20     myD.f();
21 }
22
23
```

void Base::f()
함수 "Base::f" (선언됨 줄 7)에 액세스할 수 없습니다.

6번 슬라이드의 예시에서
public을 private로 바꿔었더니
접근할 수 없는 모습

f() 메소드는 부모 클래스에서
public이었기 때문에
자식 클래스에선
상속타입을 따라서
private가 되었기 때문...!

다중상속



가능하긴 한데 흔히 쓰진 않는다.

오버라이딩 (Overriding)

- ▶ 부모 클래스에 존재하던 함수를 자식 클래스에서 재정의하여 사용하는 것을 overriding이라고 한다.
- ▶ 부모 클래스에서 쓰이던 기능에 다른 기능을 추가하고 싶을 때, 기존과 다른 새로운 기능을 구현하고 싶을 때 주로 사용한다.

오버로딩 (Overloading)

- ▶ 기존에 정의된 함수 또는 연산자와 같은 이름을 가진 함수 또는 연산자를 정의할 수 있다.
(연산자 오버로딩이 여기에 포함)
- ▶ 이 때 기존에 정의된 것과 새로 만들 함수의 매개변수 형태는 **달라야 한다!!** (개수는 상관 없음)
- ▶ 여러 개의 오버로딩된 함수 중 사용자가 실행한 것과 적합한 것을 컴파일러가 알아서 매칭시켜줌 (overload resolution)

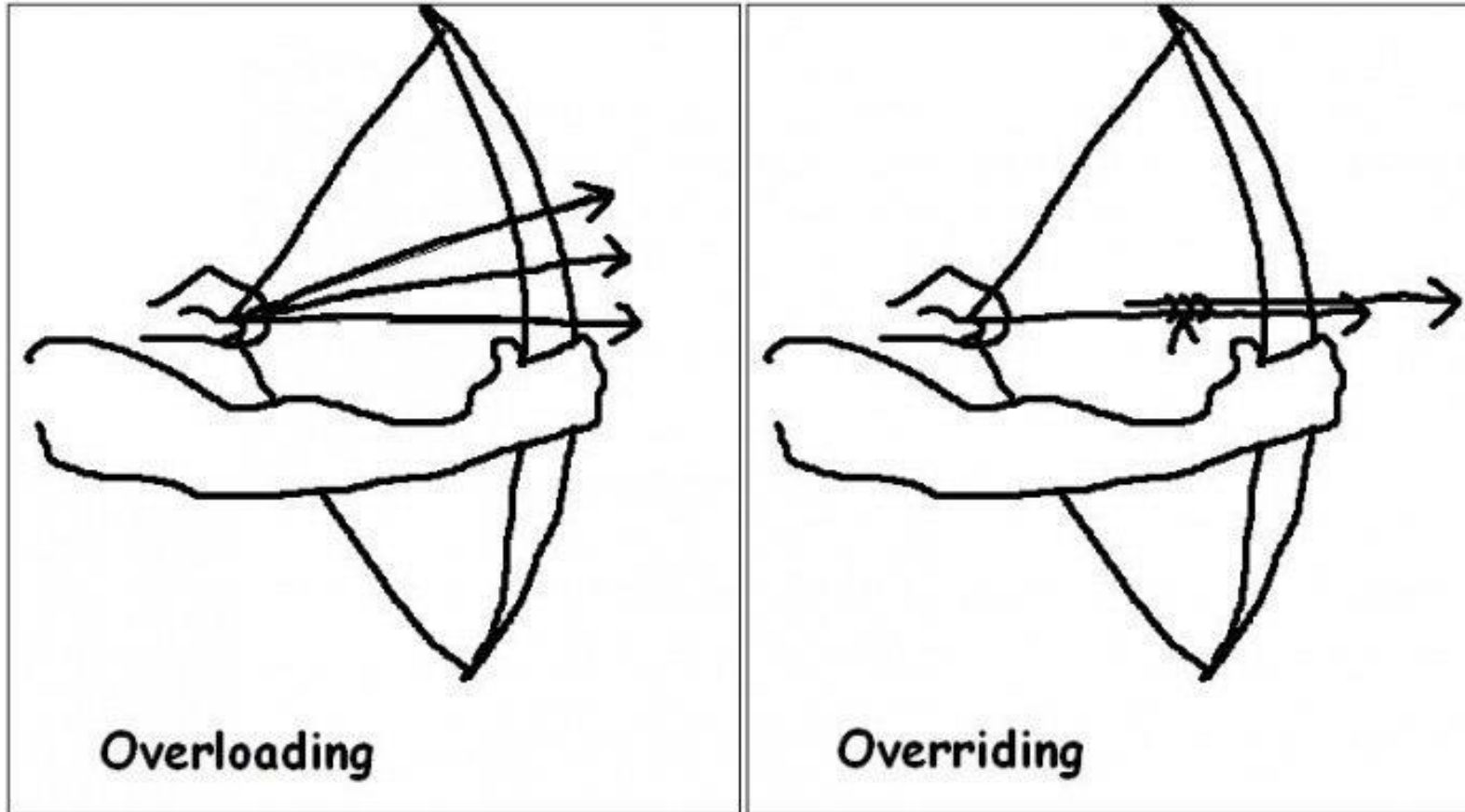
오버로딩과 오버라이딩의 차이는??

- ▶ 이름이 비슷하지만 그 개념은 다르다 (이름이 같다는 것은 동일)

	오버로딩	오버라이딩
리턴 타입	상관 없음	
조건	일반 함수나 같은 클래스 내에서 같은 이름의 함수를 여러 개 만들 때	자식클래스가 부모클래스의 함수를 재정의할 때
매개변수	타입이 달라야 함(개수는 상관x)	상관없음

- ▶ 더 자세한 내용은 <https://thrillfighter.tistory.com/164>

오버로딩과 오버라이딩의 차이는??



overriding과 overloading의 차이점을 잘 보여주는 예시

virtual & override

- ▶ 상속 관련 키워드 - virtual, override (필수는 아님)
- ▶ virtual (부모클래스에서 사용) : 부모클래스에서 virtual 키워드가 있는 함수는 오버라이딩을 해서 쓸 수 있다고 “명시”해놓은 것
- ▶ override (자식클래스에서 사용) : 자식클래스에서 override 키워드가 있는 함수는 부모클래스에 있는 함수와 다른 기능을 수행할 수 있다고 “명시”해놓은 것
- ▶ 자식클래스에서 부모클래스에 있는 함수를 오버라이딩하여 사용할 경우, 해당 함수를 virtual function(가상함수)라고도 한다.

virtual & override - virtual

```
6
7 // 이걸 부모 클래스
8 class Text {
9     private:
10         string text;
11     public:
12         // 생성자에서 text에 값을 넣어줌
13         Text(const string& t): text(t) {}
14
15         // get함수
16         virtual string get() const { return text; }
17
18         // text 뒤에 extra를 붙여주는 함수
19         virtual void append(const string& extra) { text += extra; }
20     };
21
```

virtual의 위치는 사진과 같이
함수의 반환형 앞에 쓴다

virtual & override - override

```
23
24
25 // 부모 클래스에서 데코레이션이 더해진 자식 클래스
26 class FancyText: public Text {
27     private:
28         string left_bracket;
29         string right_bracket;
30         string connector;
31     public:
32         // 어떤 것으로 데코레이션을 할지 입력받는 생성자
33         FancyText(const string& t, const string& left, const string& right, const string& conn):
34             Text(t), left_bracket(left), right_bracket(right), connector(conn) {}
35
36         // 데코레이션을 가미한 get함수
37         string get() const override { return left_bracket + Text::get() + right_bracket; }
38
39         // 데코레이션을 가미한 append함수
40         void append(const string& extra) override { Text::append(connector + extra); }
41     };
42
```

override의 위치는 사진과 같이 ()와 {}사이에 쓴다

-> 부모클래스에서 **virtual**을 써준 함수에만 override를 쓸 수 있다!!

virtual & override - override

```
23
24
25 // 부모 클래스에서 데코레이션이 더해진 자식 클래스
26 class FancyText: public Text {
27     private:
28         string left_bracket;
29         string right_bracket;
30         string connector;
31     public:
32         // 어떤 것으로 데코레이션을 할지 입력받는 생성자
33         FancyText(const string& t, const string& left, const string& right, const string& conn):
34             Text(t), left_bracket(left), right_bracket(right), connector(conn) {}
35
36         // 데코레이션을 가미한 get 함수
37         string get() const override { return left_bracket + Text::get() + right_bracket; }
38
39         // 데코레이션을 가미한 append 함수
40         void append(const string& extra) override { Text::append(connector + extra); }
41     };
42
```

앞에 Text::가 뭐지??
부모클래스의 함수를 쓰기 위해 함수
앞에 붙여주는 namespace!

여기서 Text(t)는 부모클래스의
생성자를 이용!
(부모클래스의 변수)text(t)와 동일!

함수 오버라이딩을 사용하여 부모클래스에 있던 함수에 무언가를
덧붙여 이름만 같은 새로운 함수를 만들었다!

virtual & override - override

```
// 부모 클래스와 겹은 같지만 안의 내용들을 비워버린 자식 클래스
class FixedText: public Text {
public:
    // text를 항상 "FIXED"로 고정시킴
    FixedText(): Text("FIXED") {}

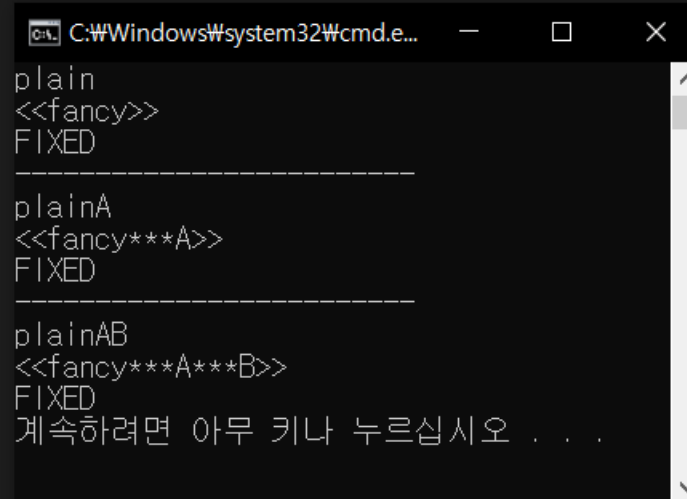
    // get함수는 굳이 바꿔줄 필요가 없겠다

    // 뭘 입력하든 아무것도 안하게 만든 함수
    void append(const string& override { /*아무것도 안함*/ }
};
```

앞에서와 반대로 오히려 기능을 축소시키거나
아무것도 안 하는 함수로 바꿀 수도 있다!

virtual & override - override

```
int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "***");
    FixedText t3;
    cout << t1.get() << '\n';
    cout << t2.get() << '\n';
    cout << t3.get() << '\n';
    cout << "-----\n";
    t1.append("A");
    t2.append("A");
    t3.append("A");
    cout << t1.get() << '\n';
    cout << t2.get() << '\n';
    cout << t3.get() << '\n';
    cout << "-----\n";
    t1.append("B");
    t2.append("B");
    t3.append("B");
    cout << t1.get() << '\n';
    cout << t2.get() << '\n';
    cout << t3.get() << '\n';
}
```

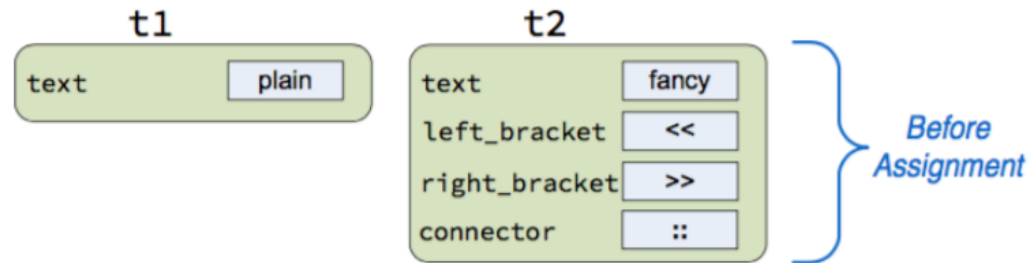


```
C:\Windows\system32\cmd.e...
plain
<<fancy>>
FIXED
-----
plainA
<<fancy***A>>
FIXED
-----
plainAB
<<fancy***A***B>>
FIXED
계속하려면 아무 키나 누르십시오 . . .
```

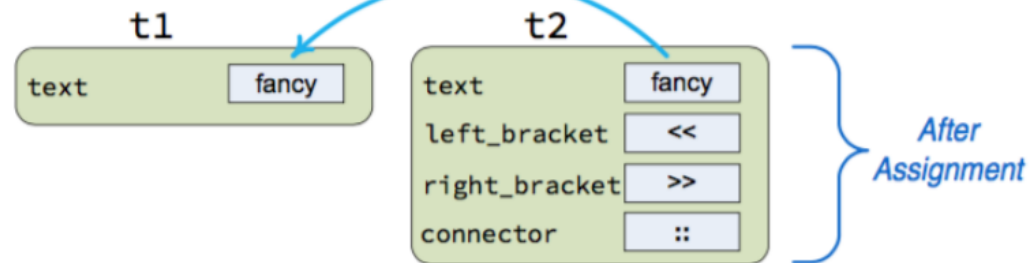
같은 이름의 함수들이 조금씩 다른 작업을 수행하고 있는 모습

부모 클래스, 자식 클래스 정리

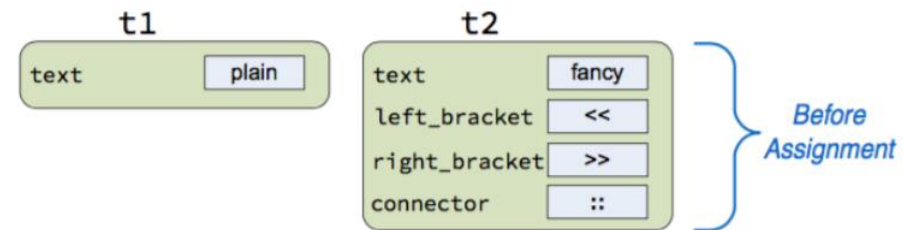
```
Text t1("plain");  
FancyText t2("Fancy", "<<", ">>", "::");
```



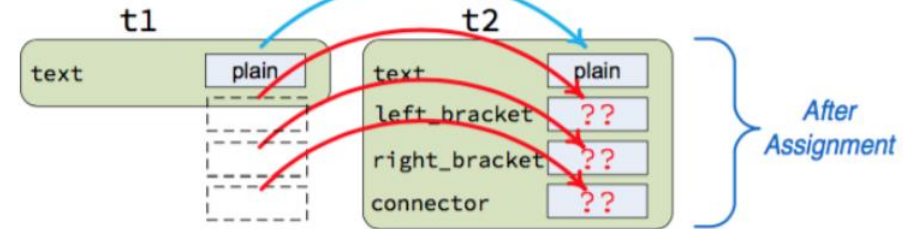
```
t1 = t2;
```



```
Text t1("plain");  
FancyText t2("Fancy", "<<", ">>", "::");
```



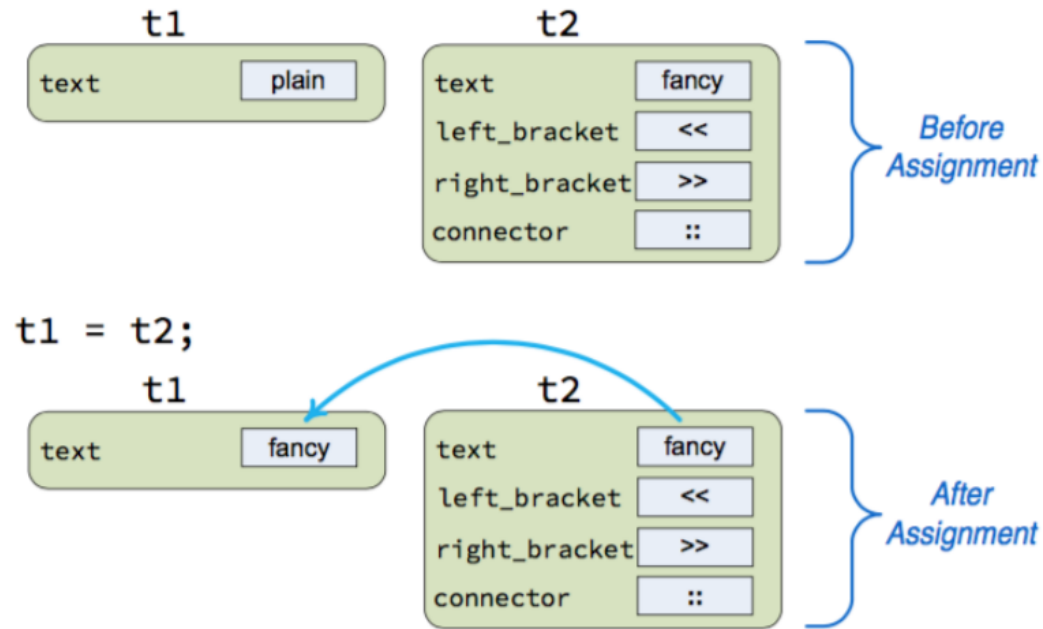
```
t2 = t1;
```



Text = 부모, FancyText = 자식
부모에는 자식을 대입할 수 있지만 반대는 안 된다
그 이유는? 부모 클래스와 자식 클래스가 is-a 관계이기 때문!!

부모 클래스, 자식 클래스 정리

```
Text t1("plain");  
FancyText t2("Fancy", "<<", ">>", "::");
```



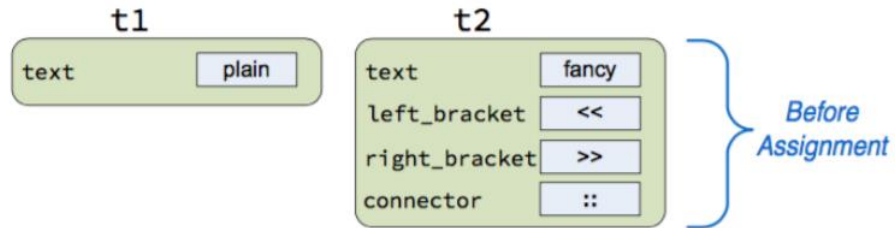
```
int main() {  
    Text t1("plain");  
    FancyText t2("fancy", "<<", ">>", "::");  
    cout << t1.get() << " " << t2.get() << endl;  
    t1 = t2; // t2를 t1에 복사!  
    cout << t1.get() << " " << t2.get() << endl;  
}
```

C:\Windows\system32\cmd.e...
plain <<fancy>>
fancy <<fancy>>
계속하려면 아무 키나 누르십시오 . . .

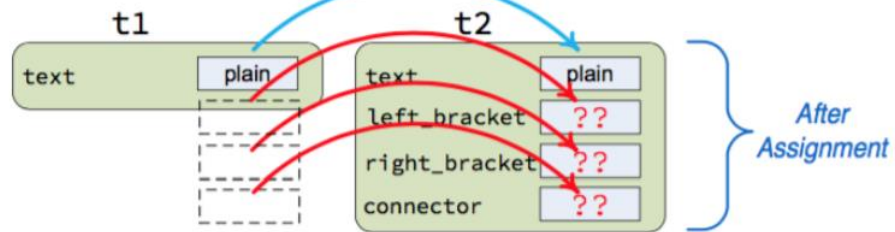
실제 코드에서도 정상 작동!

부모 클래스, 자식 클래스 정리

```
Text t1("plain");  
FancyText t2("Fancy", "<<", ">>", "::");
```



```
t2 = t1;
```



```
int main() {  
    Text t1("plain");  
    FancyText t2("fancy", "<<", ">>", "::");  
    cout << t1.get() << " " << t2.get() << endl;  
    t2 = t1; // t1을 t2에 복사?!  
    cout << endl;  
}
```

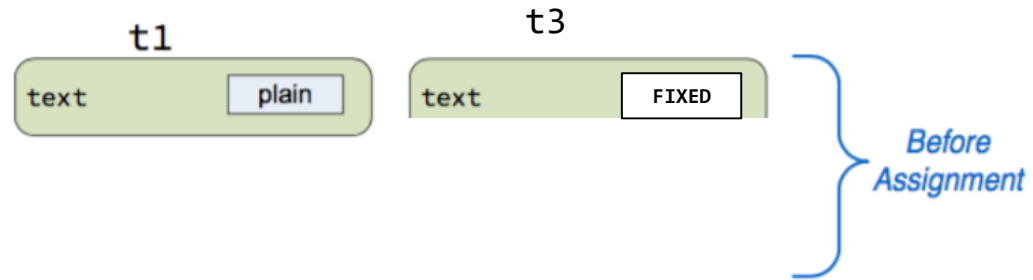
Text t1

이러한 피연산자와 일치하는 "=" 연산자가 없습니다.
피연산자 형식이 FancyText = Text입니다.

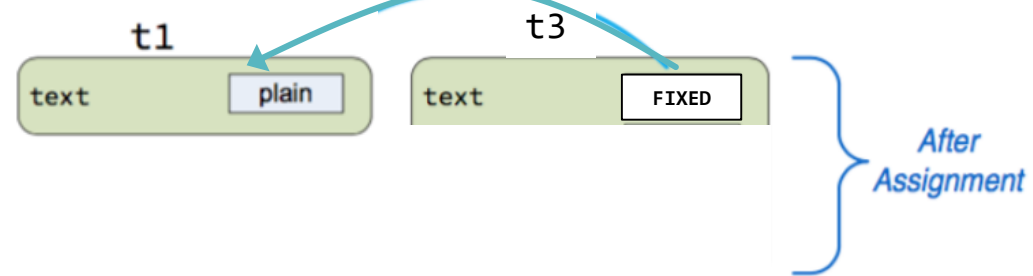
빨간 줄? 왼쪽 그림이 설명해주고 있다

부모 클래스, 자식 클래스 정리

```
Text t1("plain");  
FixedText t3;
```



```
t1 = t3;
```



```
int main() {  
    Text t1("plain");  
    FixedText t3;  
    cout << t1.get() << " " << t3.get() << endl;  
    t1 = t3; // t3를 t1에 복사!  
    cout << t1.get() << " " << t3.get() << endl;  
}
```

```
C:\Windows\system32\cmd.exe  
plain FIXED  
FIXED FIXED  
계속하려면 아무 키나 누르십시오 . . .
```

t3도 아주 잘 된다!

is-a 관계

- 자식 클래스 is-a 부모 클래스 (is-a 관계)
ex) Every employee is-a person
- 부모 클래스 is not a 자식 클래스 (is-a 관계가 아님)
ex) Not every person is an employee.

is-a 관계 (상속관계)

- ▶ is-a 관계는 자식에서 부모로의 단방향 관계이다.
- ▶ 이를 **상속의 조건**이라고 하는데 여기에는 is-a 이외에 has-a 관계 (포함관계) 도 있다.
- ▶ has-a 관계 (궁금한 사람은 읽어 보세요) : <https://cpp11.tistory.com/40>

과제

▶ 랩 8 ㄱ