

객체지향언어



10장 가상함수와 다형성

학습목표

- 함수를 호출할 때 호출할 멤버함수를 결정하기 위한 동적 바인딩과 가상함수를 설명할 수 있고, 사용할 수 있다.
- C++ 스타일의 형 변환을 설명할 수 있고, 사용할 수 있다.
- 추상 클래스의 개념을 설명할 수 있고, 추상 클래스를 사용한 함수의 다형성을 이용하여 프로그램 할 수 있다.

■ 형 변환

- 포인터를 활용하는 경우 포인터와 저장되는 메모리공간(주소)의 자료형이 같아야 함
- 자료형이 다른 경우에는 묵시적이던지 명시적이던지 형 변환 발생
- 객체 포인터에 객체의 주소를 저장하는 경우에도 동일한 클래스를 활용해야 함

[예시]

```
int a = 10, *pa;
```

```
double d = 6.5, *pd;
```

```
pa = &a;           // 자료형이 같다.
```

```
pd = &d;           // 자료형이 같다.
```

```
pd = pa;          // 자료형이 다르므로 컴파일 에러가 발생한다.
```

```
pa = &d;          // 자료형이 다르므로 컴파일 에러가 발생한다.
```

```
pd = (double *)pa; // 자료형이 다르므로 명시적인 형 변환을 하였다.
```

```
pa = (int *)&d;    // 자료형이 다르므로 명시적인 형 변환을 하였다.
```

■ 상속 관계에 있는 클래스들 사이에 형 변환

■ 업 캐스팅(up casting)

- 기반 클래스의 포인터 변수에 파생 클래스의 주소가 저장되는 경우로 파생 클래스 형을 기반 클래스 형으로 형 변환
- 업 캐스팅인 경우는 컴파일러가 자동으로 형 변환

■ 다운 캐스팅(down casting)

- 파생 클래스의 포인터 변수에 기반 클래스의 주소가 저장되는 경우로 기반 클래스 형을 파생 클래스의 형으로 변환
- 다운 캐스팅의 경우는 컴파일러는 자동으로 형 변환을 하지 않고, 컴파일 오류가 발생
- 필요하다면 기반 클래스 형을 파생 클래스 형으로 명시적인 형 변환

■ ex10_1.cpp (1) (상속 관계에 있는 클래스들 사이에 형 변환)

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

```
class Vehicle                                // 기반 클래스
{
protected:
    int number ;
public:
    Vehicle(int n) : number(n) { };
    void show() const ;
};
```

```
void Vehicle::show() const
{
    cout << "Vehicle::show() ==> " ;
    cout << "번호: " << number << endl ;
}
```

■ ex10_1.cpp (2) (상속 관계에 있는 클래스들 사이에 형 변환)

```
class Bus : public Vehicle
```

```
// 파생 클래스
```

```
{  
private:  
    int person;  
public:  
    Bus(int n, int p) : Vehicle(n)  
    {  
        person = p;  
    };  
    void show() const ;  
};  
  
void Bus::show() const  
{  
    cout << "Bus::show() ==> " ;  
    cout << "번호: " << number << ", 승객수: " << person << endl ;  
}
```

■ ex10_1.cpp (3) (상속 관계에 있는 클래스들 사이에 형 변환)

```
void main()
```

```
{  
    Vehicle *pv1, *pv2, v(1234);  
    Bus *pb1, *pb2, b(2345, 10);  
    pv1 = &v;  
    pv1->show();  
    pb1 = &b;  
    pb1->show();  
    cout << endl;
```

```
Vehicle::show() ==> 번호: 1234  
Bus::show() ==> 번호: 2345, 승객수: 10  
  
업캐스팅 : 자식 클래스형에서 부모 클래스형으로 변환.  
Vehicle::show() ==> 번호: 2345  
Vehicle::show() ==> 번호: 2345  
  
다운캐스팅 : 부모클래스형에서 자식클래스형으로 변환.  
Bus::show() ==> 번호: 1234, 승객수: -858993460  
계속하려면 아무 키나 누르십시오 . . .
```

```
cout << "업캐스팅 : 자식 클래스형에서 부모 클래스형으로 변환." << endl;
```

```
pv2 = pb1;           // 업 캐스팅 : 자동으로 형 변환
```

```
pv2->show();
```

```
pv2 = &b;           // 업 캐스팅
```

```
pv2->show();
```

```
cout << endl;
```

```
cout << "다운캐스팅 : 부모 클래스형에서 자식 클래스형으로 변환." << endl;
```

```
//pb2 = pv2;           // 다운 캐스팅으로 오류 발생
```

```
//pb2= &v;           // 다운 캐스팅으로 오류 발생
```

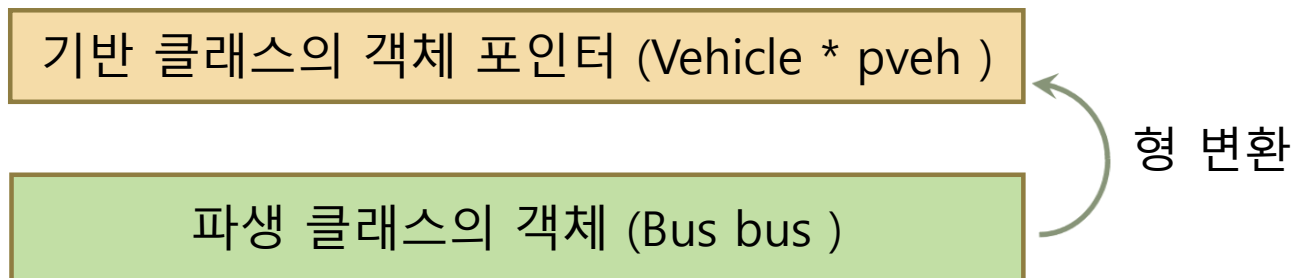
```
pb2 = (Bus *)&v;           // 명시적(강제) 형 변환을 하여야 한다.
```

```
pb2->show();
```

```
}
```

■ 업 캐스팅

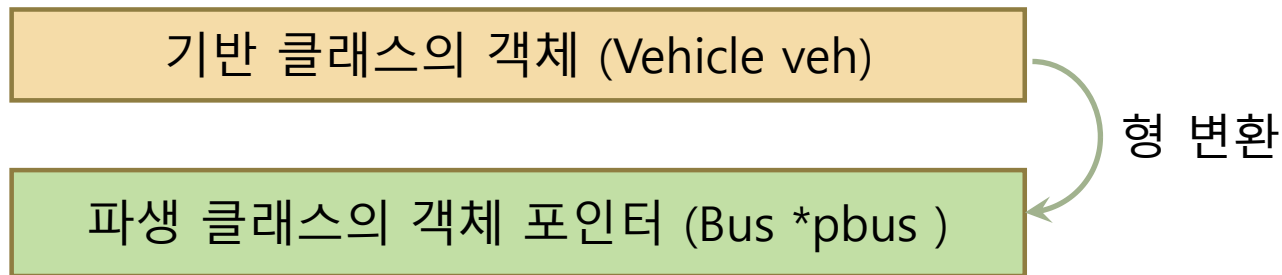
- 파생 클래스 객체의 주소 값이 기반 클래스의 객체 포인터 변수에 저장되는 경우



```
pveh = &bus; // 업 캐스팅  
            // (파생 클래스의 형이 기반 클래스의 형으로 변환)  
            // 자동 형 변환이 된다.
```


■ 다운 캐스팅

- 기반 클래스 객체의 주소 값이 파생 클래스의 객체 포인터 변수에 저장되는 경우



```
pbus = &veh; // 다운 캐스팅  
            // (기반 클래스의 형이 파생 클래스의 형으로 변환)  
            // 자동 형 변환이 안 된다. --> 에러 발생
```

형 변환

■ [예시]

```
class Vehicle {           // 부모 클래스
public:
    void show();           // 부모 클래스의 show() 함수
};
class Bus : public Vehicle { // 자식 클래스
public:
    void show();           // 자식 클래스의 show() 함수를 재정의
};
```

[업 캐스팅의 예]

```
void main() {
    Bus bus;                // 자식 클래스의 객체
    Vehicle *pveh = &bus;   // 업 캐스팅 (자동 형 변환
    pveh->show();            // 데이터는 bus이지만, Vehicle 클래스의 show() 함수를 호출
}
```

[다운 캐스팅의 예]

```
void main() {
    Vehicle veh;            // 부모 클래스의 객체
    // Bus *pbus = &veh;    // 다운 캐스팅 (오류)
    Bus *pbus = (Bus *)&veh; // 명시적 형 변환이 필요. pbus->show();
    // 데이터는 veh 이지만, Bus 클래스의 show() 함수를 호출
}
```

■ ex10_2.cpp (1) (업 캐스팅)

```
#include <iostream>
#include <cstring>
using namespace std;
class Vehicle                                // 기반 클래스
{
protected:
    int number ;
public:
    Vehicle(int n):number(n) { };
    void show() const ;
};
void Vehicle::show() const
{
    cout << "Vehicle::show() ==> 번호: " << number << endl ;
}
class Bus : public Vehicle                  // 파생 클래스
{
private:
    int person;
public:
    Bus(int n, int p):Vehicle(n) { person = p; };
    void show() const ;
};
```

■ ex10_2.cpp (2) (업 캐스팅)

```
void Bus::show() const
{
    cout << "Bus::show() ==> 번호: " << number << ", 승객수: " << person << endl ;
}
void main()
{
    Vehicle *pveh;           // 부모 클래스의 객체 포인터
    Vehicle veh(1234);
    pveh = &veh;             // 객체포인터의 클래스와 객체의 클래스가 같은 경우
    cout << "(pveh = &veh) ==> " ;
    pveh->show() ;

    Bus bus(3456, 40);       //자식 클래스의 객체
    pveh = &bus;
                                // 부모 클래스의 객체 포인터가 자식 클래스의 객체를 가리킨다.
    cout << "(pveh = &bus) ==> " ;
    pveh->show();             // 함수는 자동으로 부모의 형으로 변환 --> 업 캐스팅
    cout << endl;
}
```

```
<pveh = &veh> ==> Vehicle::show() ==> 번호: 1234
<pveh = &bus> ==> Vehicle::show() ==> 번호: 3456

계속하려면 아무 키나 누르십시오 . . .
```

■ 바인딩(binding)

- 함수 호출이 해당 함수의 정의와 결합되는 것을 의미함
- 함수의 바인딩 방법
 - 정적 바인딩(static binding)
 - 동적 바인딩(dynamic binding)

■ 바인딩 방법의 비교

바인딩 방법	바인딩 시점	특징	대상
정적 바인딩	컴파일 시간에 호출함수가 결정	속도: 빠르다 융통성: 없다	일반 함수
동적 바인딩	실행 시간에 호출함수가 결정	속도: 느리다 융통성: 있다	가상 함수

■ 정적 바인딩(static binding)

- 컴파일 시점에 호출할 함수가 결정
- 컴파일 할 때 호출될 함수가 미리 정해지므로 이른 바인딩(early binding)이라고도 함
- 컴파일 시점에는 선언된 객체 포인터의 자료형에 대한 정보만 있고 그 포인터가 어느 객체를 가리키는지는 알 수 없음
- 변수의 값이 정해지는 시점은 실행할 때 결정 됨

ex10_2.cpp 예제는 정적 바인딩을 한 프로그램으로

```
pveh = &bus;  
pveh->show();
```

출력결과는 데이터 값이 객체의 값이지만 함수 호출은 객체 포인터에 의해 정해 짐

```
Vehicle::show() ==> 번호: 3456
```

■ 동적 바인딩(dynamic binding)

- 실행할 때 호출할 멤버함수를 결정하여 멤버함수를 호출하는 방법
- 호출할 함수가 나중에 정해지므로 늦은 바인딩(late binding)이라고도 함
- 동적 바인딩을 하려면 기반 클래스에서 재정의할 함수 앞에 **virtual** 이라는 키워드를 붙여 **가상함수로 선언**하여야 함
- 가상(virtual)함수로 선언하면 **실행 시점에 호출할 함수가 결정됨**

[예시]

```
class Vehicle
```

```
{
```

```
...
```

```
    virtual void show( )
```

```
{
```

```
        cout << "Vehicle::show() ==> 번호: " << number << endl;
```

```
}
```

```
...
```

```
}
```

■ 가상함수(virtual)

- 호출할 함수가 실행 시점에 결정되어야 하므로, 가상함수가 선언된 클래스의 객체가 생성되면 가상함수 테이블을 만들어 가상함수 주소를 따로 저장하여 관리
- 가상함수가 호출되면 해당 가상함수 테이블로 가서 주소를 찾아 그 주소에 해당하는 함수를 호출
- 실행시점에 객체 포인터가 어느 객체를 가리키고 있느냐에 따라 바인딩할 함수를 결정하여 함수를 호출. 즉, **동적 바인딩을 제공**

■ 가상함수의 장/단점

- 동적 바인딩을 하면 융통성 있게 함수를 호출할 수 있음
- 메모리에 부담을 주고 실행시점에 바인딩할 함수를 결정하여야 하므로 처리 속도가 지연됨

■ ex10_3.cpp (1) (가상 함수를 사용한 동적 바인딩)

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

class Vehicle {
protected:
    int number;
public:
    Vehicle(int n) : number(n) { };
    virtual void show() const ;           // 기반 클래스에서 가상함수로 선언
};

void Vehicle::show() const {
    cout << "Vehicle::show() ==> 번호: " << number << endl;
}

class Bus : public Vehicle {
private:
    int person;
public:
    Bus(int n, int p) : Vehicle(n) {    person = p;    };
    void show() const ;
};
```

■ ex10_3.cpp (2) (가상 함수를 사용한 동적 바인딩)

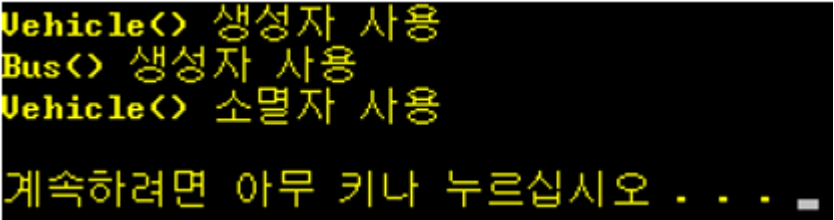
```
void Bus::show() const {  
    cout << "Bus::show() ==> 번호: " << number << ", 승객수: " << person << endl;  
}
```

```
void main()  
{  
    Vehicle *pveh;  
    Vehicle veh(1234);  
    pveh = &veh;  
    cout << "(pveh = &veh) ==> " ;  
    pveh->show(); // Vehicle 클래스의 show() 호출  
  
    Bus bus(3456, 40);  
    pveh = &bus;  
    cout << "(pveh = &bus) ==> " ;  
    pveh->show(); // Bus 클래스의 show() 호출  
    cout << endl;  
}
```

```
<pveh = &veh> ==> Vehicle::show() ==> 번호: 1234  
<pveh = &bus> ==> Bus::show() ==> 번호: 3456, 승객수: 40  
계속하려면 아무 키나 누르십시오 . . .
```

■ ex10_4.cpp (소멸자 사용 예)

```
#include <iostream>
using namespace std;
class Vehicle
{
protected:
    int number;
public:
    Vehicle() { cout << "Vehicle() 생성자 사용" << endl; }
    ~Vehicle() { cout << "Vehicle() 소멸자 사용" << endl; }
};
class Bus : public Vehicle
{
public:
    Bus() { cout << "Bus() 생성자 사용" << endl; };
    ~Bus() { cout << "Bus() 소멸자 사용" << endl; };
};
void main()
{
    Vehicle *pveh = new Bus;
    delete pveh;
    cout << endl;
}
```

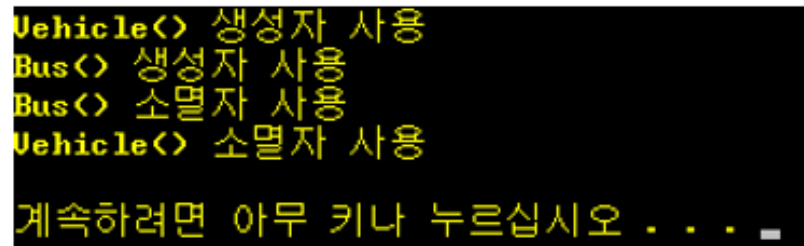


```
Vehicle<> 생성자 사용
Bus<> 생성자 사용
Vehicle<> 소멸자 사용

계속하려면 아무 키나 누르십시오 . . .
```

■ ex10_5.cpp (가상 소멸자 사용 예)

```
#include <iostream>
using namespace std;
class Vehicle {
protected:
    int number;
public:
    Vehicle() {cout << "Vehicle() 생성자 사용" << endl; }
    virtual ~Vehicle() {cout << "Vehicle() 소멸자 사용" << endl; }
};
class Bus : public Vehicle {
public:
    Bus() {    cout << "Bus() 생성자 사용" << endl; };
    ~Bus() {   cout << "Bus() 소멸자 사용" << endl; };
};
void main(){
    Vehicle *pveh = new Bus;
    delete pveh;
    cout << endl;
}
```



```
Vehicle<> 생성자 사용
Bus<> 생성자 사용
Bus<> 소멸자 사용
Vehicle<> 소멸자 사용
```

```
계속하려면 아무 키나 누르십시오 . . .
```

■ 순수 가상함수(pure virtual function)

- 함수의 정의 없이 함수의 원형만을 기반 클래스에 선언해 둔 가상함수
- 순수 가상함수는 멤버함수를 선언할 때 virtual 키워드를 선언문의 앞에 붙이고 마지막 부분에 '= 0'이라고 표기
- 순수 가상함수가 만들어지면 **자식 클래스에서는 반드시 해당 함수를 재정의 (override)**하여야 함

```
virtual 반환_자료형 함수명(인수, ... ) = 0;
```

[예시]

```
class Vehicle {  
private:  
    int number;  
public:  
    virtual void show() = 0;  
    ...  
};
```

■ 추상 클래스(abstract class)

- 순수 가상함수를 하나 이상 가지는 클래스를 추상 클래스
- 추상 클래스는 함수의 정의가 없으므로 객체를 생성할 수 없음
- 추상 클래스는 객체 포인터는 생성할 수 있음
- 추상 클래스는 자식 클래스에서 재정의해야 하는 순수 가상함수를 하나 이상 포함하고 있으므로 반드시 자식 클래스를 가지고 있어야 함
- 추상 클래스는 가상함수를 가지고 있는 클래스이므로 해당 함수는 **동적 바인딩**

■ ex10_6.cpp (1) (추상 클래스와 함수의 동적 바인딩)

```
#include <iostream>
using namespace std;
class Vehicle                // 추상 클래스
{
protected:
    int number;
public:
    virtual void show() = 0;    // 기반 클래스에서 순수 가상함수로 선언
};

class Bus : public Vehicle
{
private:
    int person;
public:
    Bus(int n, int p)
    {
        number = n;
        person = p;
    };
    void show();
};
```

■ ex10_6.cpp (3) (추상 클래스와 함수의 동적 바인딩)

```
void Bus::show()
{
    cout << "Bus::show() ==> 번호: " << number << ", 승객수: " << person << endl;
}
```

class Truck : public Vehicle

```
{
private:
    double cargo;
public:
    Truck(int n, double c)
    {
        number = n;
        cargo = c;
    }
    void show();
};

void Truck::show()
{
    cout << "Truck::show() ==> 번호: " << number << ", 적재정량: " << cargo << endl;
}
```


■ ex10_6.cpp (4) (추상 클래스와 함수의 동적 바인딩)

```
void main()
{
    Vehicle *pveh ;
    Bus bus(2345, 40);

    cout << "[bus] " ;
    bus.show();

    Truck truck(4567, 2.5);
    cout << "[truck] " ;
    truck.show();

    pveh = &bus ;
    cout << "[pveh = &bus] " ;
    pveh->show();

    pveh = &truck ;
    cout << "[pveh = &truck] " ;
    pveh->show();
    cout << endl;
}
```

```
[bus] Bus::show() ==> 번호: 2345, 승객수: 40
[truck] Truck::show() ==> 번호: 4567, 적재정량: 2.5
[pveh = &bus] Bus::show() ==> 번호: 2345, 승객수: 40
[pveh = &truck] Truck::show() ==> 번호: 4567, 적재정량: 2.5
계속하려면 아무 키나 누르십시오 . . .
```

C++ 스타일의 형 변환

■ C 스타일의 형 변환 연산자

- 무조건 형 변환하는 것으로 생각지 않은 결과가 발생할 수도 있음
- 문법적으로 형 변환이 가능한 경우에 형 변환하는 것인지 혹은 형 변환할 수 없는 경우에 형 변환하는 것인지 알기가 어려움
- C 스타일의 형 변환은 사용하지 않는 것이 좋음

■ C++ 스타일의 형 변환

- `static_cast< >()` // 안전한 형 변환
- `const_cast< >()` // `const`나 `volatile` 속성을 제거하는 형 변환
- `reinterpret_cast< >()` // 위험한 형 변환
- `dynamic_cast< >()` // 클래스 사이의 형 변환

static_cast **<int>** **(32.0f)** // float 형의 32.0을 int 형으로 변환

→ 형 변환하려는 값 혹은 변수 : 32.0를 형 변환하려고 한다.

→ 변환되는 자료형 : int 형으로 형 변환한다.

→ 형 변환 연산자

■ static_cast 연산자

- C 스타일 형 변환과 같은 가장 기본적인 형 변환 연산자
- A 타입에서 B 타입으로 묵시적인 형 변환이 가능한 경우에 명시적으로 형 변환할 때 static_cast를 사용
- static_cast 연산자는 컴파일 시 자동 형 변환이 가능한지 확인해서 형 변환할 수 있으면 형 변환을 수행하고 형 변환을 할 수 없으면 컴파일 오류를 발생
- static_cast 연산자는 서로 상속 관계에서 부모 클래스에서 자식 클래스로 논리적으로 가능하면 형 변환할 수 있으나 컴파일 오류를 발생하지 않으므로 발생할 수 있는 문제는 프로그래머가 책임져야 함

[예시]

```
float f = 32.0f;  
int n;  
n = static_cast<int>(f);      // f를 int 형으로 형 변환하여 n에 넣어라
```

■ ex10_7.cpp (1) (static_cast 연산자 사용 예)

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

```
class Parent                                // 기반 클래스
{
protected:
    int num;
public:
    Parent() { };
    void show() { cout << "class Parent " << endl; };
};
```

```
class Child : public Parent                // 파생 클래스
{
public:
    Child(int n) { num = n; }
    void show() { cout << "class Child : " << num << endl; }
};
```

C++ 스타일의 형 변환

■ ex10_7.cpp (2) (static_cast 연산자 사용 예)

```
void main()
{
    Parent *p_P1 = new Parent;           // 문제없다.
                                         // Parent형 객체를 생성해서 Parent형 포인터에 보관

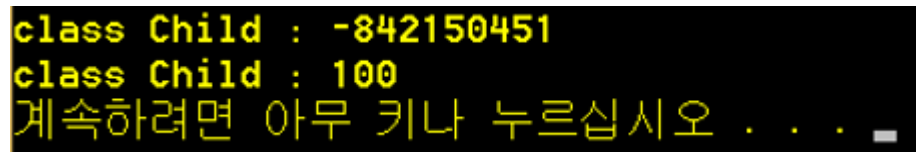
    Parent *p_P2 = new Child(100);       // 업 캐스팅 --> 문제없다
                                         // Child형 객체를 생성해서 Parent형 포인터에 보관

    Child *p_C1 = static_cast<Child*>(p_P1); // 잘못된 형 변환. 컴파일 오류는 발생하지
                                         // 않는다. 실행 오류는 프로그래머 책임

    Child *p_C2 = static_cast<Child*>(p_P2); // 문제없다.

    p_C1->show();                          // 실행 시 오류가 발생한다.
    p_C2->show();

    delete p_P1;
    delete p_P2;
}
```



```
class Child : -842150451
class Child : 100
계속하려면 아무 키나 누르십시오 . . .
```

■ `const_cast` 연산자

- `const` 속성이나 `volatile` 속성을 제거할 때 사용
- `const_cast` 연산자는 포인터, 참조 혹은 멤버 포인터로부터 `const` 속성을 제거
- 참조에 의한 전달 방식으로 함수에게 `const` 상수를 인수로 넘겨줄 수 없을 때 `const_cast` 연산자를 사용하면 유용
- `volatile` 제한자
 - `volatile` 제한자는 프로그램 코드가 변경되지 않더라도 어떤 메모리 위치에 있는 값이 변경될 수 있다는 것을 나타냄
 - 예를 들면 직렬 포트로부터 들어오는 정보가 저장되는 하드웨어 위치를 지시하는 포인터인 경우, 프로그램이 아니라 하드웨어가 그 메모리의 내용을 변경
 - `volatile` 제한자 목적은 컴파일러의 최적화 능력을 개선하는 것임
 - 어떤 변수를 `volatile` 이라고 선언하지 않으면 컴파일러가 최적화를 자유로이 수행할 수 있음
 - 변수를 `volatile`이라고 선언하면 이러한 최적화를 하지 못하게 함

C++ 스타일의 형 변환

■ ex10_8.cpp (const_cast 연산자 사용 예)

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

```
int add(int &x, int &y) {
    int z;
    z = x + y;
    return z;
}
```

```
int main(void) {
    int a = 10;
    const int b = 20;
    int sum;
```

```
//sum = add(a, b);    // b는 상수이므로 포인터 및 참조자로 넘겨지지 못한다.
                     // const 속성을 제거하면 넘길 수 있다.
```

```
int c = const_cast<int&>(b);
```

```
sum = add(a, c);
cout << a << " + " << b << " = " << sum << endl;
return 0;
}
```

10 + 20 = 30

계속하려면 아무 키나 누르십시오 . . .

■ reinterpret_cast 연산자

- reinterpret_cast는 일반적으로 허용하지 않는 위험한 형 변환을 하고자 할 때 사용
- 이 때 발생하는 문제는 프로그래머가 책임져야 하므로 주의해야 함

[예시]

```
int a, b;  
a = reinterpret_cast<int>(&b);
```


■ dynamic_cast 연산자

- 상속 관계에 있는 클래스의 포인터 혹은 참조에서 형 변환할 때 사용
- **dynamic_cast는 실행 중에 안전성을 검사하여 안전한 경우에만 허가**
 - 업 캐스팅은 문제가 없지만, 다운 캐스팅은 포인터가 가리키고 있는 객체의 실제 형이 무엇이냐에 따라 안전할 수도 있고 위험할 수도 있음
- dynamic_cast는 해당 클래스가 가상함수를 하나 이상 가지고 있어야 사용할 수 있음
- 형 변환에 문제가 있는 경우라면 포인터인 경우에는 NULL 값을 반환

■ ex10_9.cpp (1) (dynamic_cast 연산자 사용 예)

```
#include <iostream>
using namespace std;
class Parent                                // 기반 클래스
{
protected:
    int num;
public:
    Parent() { };
    virtual void show(){ };
};
class Child : public Parent                // 파생 클래스
{
public:
    Child(int n) {
        num = n;
    }
    void show() {      cout << "class Child : " << num << endl;      }
};
```

C++ 스타일의 형 변환

■ ex10_9.cpp (2) (dynamic_cast 연산자 사용 예)

```
void main() {  
    Parent *p_P1 = new Parent;           // 문제없다.  
                                           // Parent형 객체를 생성해서 Parent형 포인터에 보관  
  
    Parent *p_P2 = new Child(100);       // 업 캐스팅 --> 문제없다  
                                           // Child형 객체를 생성해서 Parent형 포인터에 보관  
  
    Child *p_C1 = dynamic_cast< Child* >( p_P1 );  
                                           // p_P1을 Child형 포인터로 형 변환  
                                           // p_P1이 가리키는 객체는 원래 Parent형이므로 형 변환은 잘못된  
                                           // 형 변환을 수행하지 않고 NULL 값을 반환  
  
    Child *p_C2 = dynamic_cast< Child* >( p_P2 );  
                                           // p_P2를 Child형 포인터로 형 변환 (다운 캐스팅)  
                                           // p_P2가 가리키는 객체는 원래 Child형이므로  
                                           // 이 형 변환은 문제가 없다 --> 성공  
                                           // p_C1은 NULL 값으로 실행 불가  
  
    //p_C1->show();  
    if (p_C1 == NULL)  
        cout << "NULL" << endl;  
    p_C2->show();  
    delete p_P1;  
    delete p_P2;  
}
```

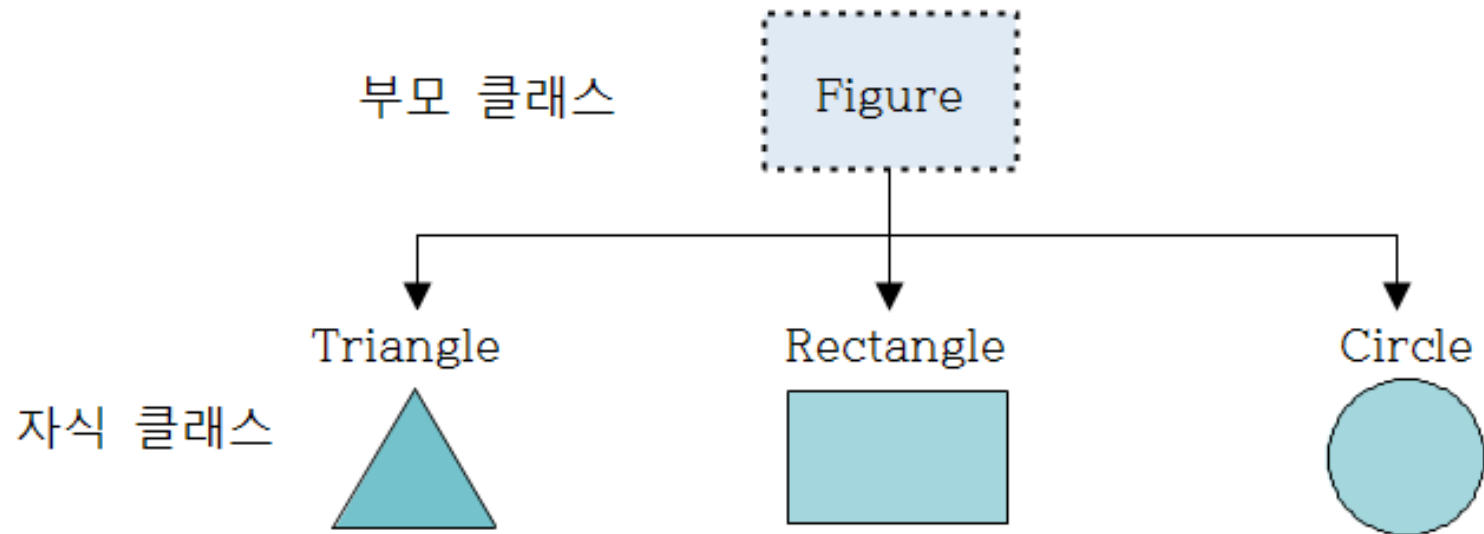
```
NULL  
class Child : 100  
계속하려면 아무 키나 누르십시오 . . .
```

■ 함수의 다형성과 추상 클래스

- 추상 클래스는 상속을 위한 기반 클래스로 사용
 - 여러 종류의 관련 클래스를 동일한 함수(인터페이스)로 접근할 수 있도록 추상 클래스를 설계하고, 그 다음에 추상 클래스를 부모로 갖는 자식 클래스를 설계
 - 여러 관련 클래스의 객체가 추상 클래스의 순수 가상함수를 통하여 접근 가능
 - 모든 파생 클래스들이 기반 클래스에서 순수 가상함수로 정의한 함수로 호출할 수 있음
- **함수의 이름은 하나인데 여러 가지 용도로 사용할 수 있으므로 함수의 다형성 (polymorphism)을 제공**
 - 함수 다중 정의(function overriding)도 다형성을 나타내지만 인수(매개변수)의 자료형이 나 반환 자료형이 다를 수 있음
 - 순수 가상함수는 함수 이름, 인수(매개변수)의 자료형이나 인수 개수, 반환형까지 같은 함수의 재정의(function overriding)를 이용한 다형성을 제공

■ 함수의 다형성의 예(1) - 클래스 상속 관계

- 삼각형, 직사각형, 원과 같은 도형에서 면적을 구하는 예



■ 함수의 다형성의 예(2) – 클래스 상속 관계

```
class Figure
{
protected:
    double area ;
public:
    virtual void area() = 0;           // 순수 가상함수
};

class Triangle : public Figure
{
private:
    int width ;
    int height ;
public:
    void area() { cout << "Triangle Area = " << area << endl ; } //area() 함수 재정의
};
```

■ 함수의 다형성의 예(3) – 클래스 상속 관계

```
class Rectangle : public Figure
```

```
{
```

```
private:
```

```
    int width ;
```

```
    int height ;
```

```
public:
```

```
    void area() { cout << "Rectangle Area = " << area << endl ; } //area() 함수 재정의  
};
```

```
class Circle : public Figure
```

```
{
```

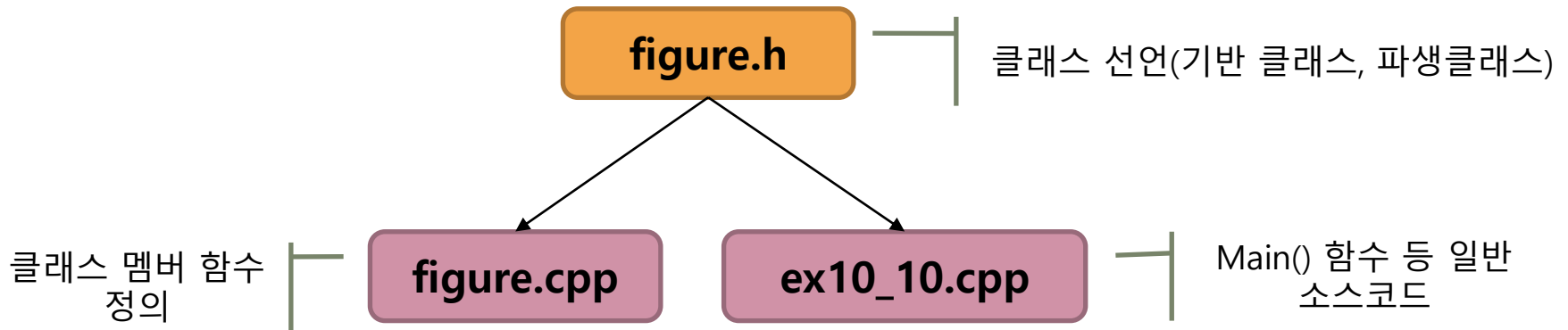
```
private:
```

```
    int radius ;
```

```
public:
```

```
    void area() { cout << "Circle Area = " << area << endl ; } //area() 함수 재정의  
};
```

■ 여러 파일 분할 구현 예시



■ figure.h (1) (파일 분할 구현 - 클래스의 선언 부분)

```
class Figure                                //기본 클래스(추상 클래스)
{
protected:
    double area ;                          // 공통 멤버 변수 선언
public:
    virtual void showArea() = 0;           //순수 가상함수
};
```


■ figure.h (2) (파일 분할 구현 - 클래스의 선언 부분)

```
class Triangle : public Figure {
private:
    int width , height ;
public:
    Triangle(int w, int h) ;           // 일반 생성자
    void showArea();                  // showArea() 함수 재정의 위한 선언
};

class Rectangle : public Figure {
private:
    int width , height ;
public:
    Rectangle(int w, int h) ;         // 일반 생성자
    void showArea();                  // showArea() 함수 재정의 위한 선언
};

class Circle : public Figure {
private:
    int radius ;
public:
    Circle(int r) ;                   // 일반 생성자
    void showArea() ;                 // showArea() 함수 재정의 위한 선언
};
```

■ figure.cpp(1) (파일 분할 구현-클래스의 멤버함수 정의 부분)

```
#include <iostream>
#include "figure.h" //사용자 정의(figure.h) 헤더 파일 포함
using namespace std;
const double PI = 3.14159;
Triangle::Triangle(int w, int h) : width(w), height(h) {
    area = w * h / 2;
}
void Triangle :: showArea() { // showArea() 함수 재정의
    cout << "Triangle Area = " << area << endl ;
}
Rectangle::Rectangle(int w, int h) : width(w), height(h) {
    area = w * h;
}
void Rectangle :: showArea() { // showArea() 함수 재정의
    cout << "Rectangle Area = " << area << endl ;
}
Circle::Circle(int r) : radius(r) {
    area = PI * r * r;
}
void Circle :: showArea() { // showArea() 함수 재정의
    cout << "Circle Area = " << area << endl ;
}
```

■ ex10_10.cpp (파일 분할 구현- 함수의 다형성)

```
#include <iostream>
#include "figure.h"
using namespace std;
void main()
{
```

```
    Figure *ptr ;
    Triangle tri(20, 10) ;
    Rectangle rec(20, 10) ;
    Circle cir(10) ;
```

ptr = &tri;

```
cout << "[ptr = &tri] : " ;
ptr->showArea();
```

ptr = &rec;

```
cout << "[ptr = &rec] : " ;
ptr->showArea();
```

ptr = ○

```
cout << "[ptr = &cir] : " ;
ptr->showArea();
```

```
}
```

//사용자 정의(figure.h) 헤더 파일 포함

```
[ptr = &tri] : Triangle Area = 100
[ptr = &rec] : Rectangle Area = 200
[ptr = &cir] : Circle Area = 314.159
계속하려면 아무 키나 누르십시오 . . .
```

// Triangle 클래스의 showArea() 호출

// Rectangle 클래스의 showArea() 호출

// Circle 클래스의 showArea() 호출



Thank You
