```
class Base {
    int b;
public:
    void f1() {..}
};
```

```
int main() {
    Base obj B;
```

obj B

Base :: b

```
obj B. f1();
```

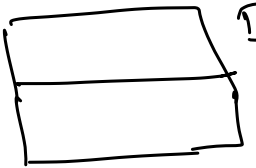
b is private. → `obj B. b = 10;`

```
class Derived : Base {
    int d;
public:
    void f2() {..}
};
```

private inheritance

```
Derived obj D;
```

obj D



Base :: b

Derived :: d

d is private → `obj D. d = 10;`

```
obj D. f2();
```

b is private → obj D. b = 100; ✗

→ obj D. f1(); ✗

f1 is private in derived class due to private inheritance.

```
class Derived : public Base {
    int d;
public:
    void f2() {...}
};
```

public inheritance

```
int main() {
    Derived obj D;
    obj D. f1();
```

✓

Derived is inherited publicly so Base class members will retain their access specifiers in Derived class

```
class Base {
    int b1;
protected:
    int b2;
public:
    void f1() {
        b1 = 10;
        b2 = 20;
    }
};
```

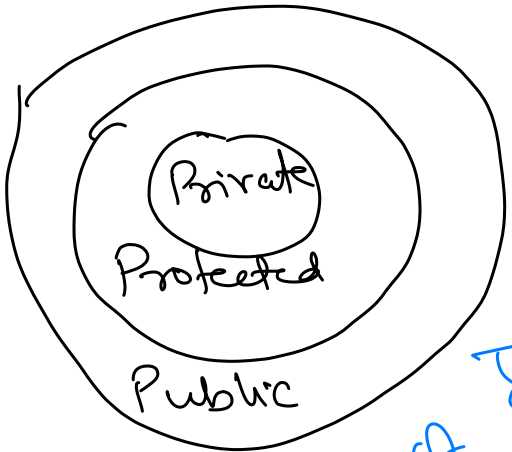
```
class Derived : public Base {
    int d1;
protected:
    int d2;
public:
    void f2() {
        d1 = 10;
        d2 = 20;
        b1 = 100;
        b2 = 200;
        f1();
    }
};
```

b1 is private in Base and hence not accesible in Derived class

f1 is public in Base

this → f1()

```
int main() {
    Base Obj B;

    Obj B. b1 = 10;  ✗ → b1 is private
    Obj B. b2 = 20;  ✗ → b2 is protected.
```

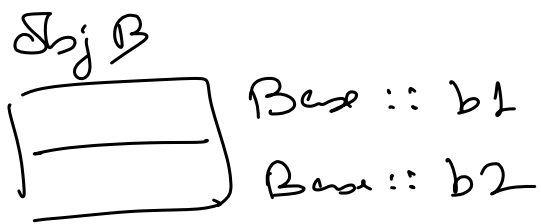


Base class member

Type of inheritance	public	protected	private
public	public	protected	Not accessible
protected	protected	protected	Not accessible
private	private	private	Not accessible

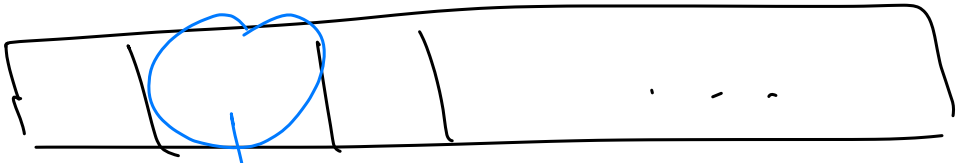
access specifier of Base class member in Derived class

```
int main() {
    Derived objD;

    objD.f2();

    Base objB;
```



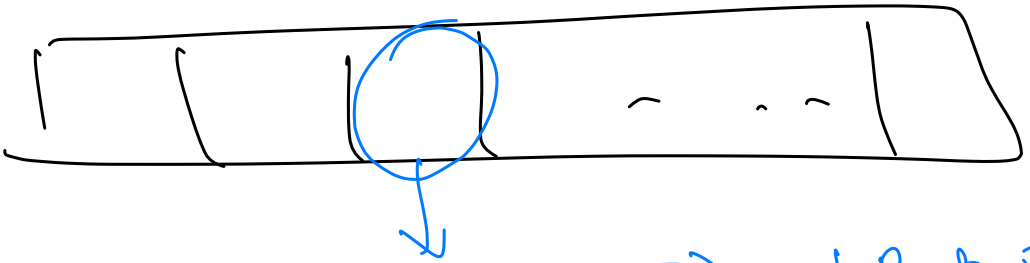
```
class Base {
    int b;
public:
    Base() : b(0) {
        std::cout << "Base default constructor\n";
    }
    ~Base() {
        std::cout << "Base destructor\n";
    }
    void setB(int b) {
        this->b = b;
    }
    int getB() {
        return b;
    }
};

class Derived : public Base {
    int d;
public:
    Derived() : d(0) {
        std::cout << "Derived default constructor\n";
    }
    ~Derived() {
        std::cout << "Derived destructor\n";
    }
```

```
    void setD(int d) {
        this->d = d;
    }
    int getD() {
        return d;
    }
};

int main() {
    Derived objD;

    objD.setB(10);
    objD.setD(5);

    std::cout << "Size of Base = " << sizeof(Base) << "\n";
    std::cout << "Size of Derived = " << sizeof(Derived) << "\n";
    std::cout << "getB for Derived = " << objD.getB() << "\n";

    return 0;
}
```



Line * ⟹ Line* pLines[10]

← array where each element will be a address of line object



Rect * ⟹ Rect* pRects[10]

⇐ array where each element will be a address of rect object

Physical size of array → How much memory is allocated?

Logical size of array → How many elements are stored?

```
struct Point {
    int x;
    int y;
};

class Line {
    Point pts[2];
public:
    Line(int x1, int y1, int x2, int y2);
    ~Line();
    void Move(int cx, int cy);
    void Draw();
};

class Rect {
    Point pts[2];
public:
    Rect(int x1, int y1, int x2, int y2);
    ~Rect();
    void Move(int cx, int cy);
    void Draw();
};
```

```
Line::Line(int x1, int y1, int x2, int y2) {
    std::cout << "Line::Line()\n";
}
Line::~Line() {
    std::cout << "Line::~Line()\n";
}
void Line::Move(int cx, int cy) {
    std::cout << "Line::Move()\n";
}
void Line::Draw() {
    std::cout << "Line::Draw()\n";
}

Rect::Rect(int x1, int y1, int x2, int y2) {
    std::cout << "Rect::Rect()\n";
}
Rect::~Rect() {
    std::cout << "Rect::~Rect()\n";
}
void Rect::Move(int cx, int cy) {
    std::cout << "Rect::Move()\n";
}
void Rect::Draw() {
    std::cout << "Rect::Draw()\n";
}
```

```
int main() {
    Line* pLines[10];
    int lineCount = 0;

    Rect* pRects[10];
    int rectCount = 0;

    std::cout << "\nCreating objects\n";
    // Create some objects.
    pLines[lineCount] = new Line(0, 0, 1, 1);
    ++lineCount;
    pLines[lineCount] = new Line(2, 2, 3, 3);
    ++lineCount;

    pRects[rectCount] = new Rect(10, 10, 20, 20);
    ++rectCount;

    std::cout << "\nDrawing objects\n";
    // Draw first line and rect
    pLines[0]->Draw();
    pRects[0]->Draw();

    std::cout << "\nDeleting objects\n";
    // Delete the memory for objects created.
    for (int i = 0; i < lineCount; ++i)
        delete pLines[i];
```

```
    for (int i = 0; i < rectCount; ++i)
        delete pRects[i];

    return 0;
}
```

```
struct Point {
    int x;
    int y;
};

class Object {
protected:
    Point pts[2];
public:
    Object(int x1, int y1, int x2, int y2);
    ~Object();
    void Move(int cx, int cy);
    void Draw();
};

class Line : public Object {
public:
    Line(int x1, int y1, int x2, int y2);
    ~Line();
    void Draw();
};

class Rect : public Object {
public:
    Rect(int x1, int y1, int x2, int y2);
    ~Rect();
    void Draw();
```

```
Object::Object(int x1, int y1, int x2, int y2) {
    std::cout << "Object::Object()\n";
}
Object::~Object() {
    std::cout << "Object::~Object()\n";
}
void Object::Move(int cx, int cy) {
    std::cout << "Object::Move()\n";
}
void Object::Draw() {
    std::cout << "Object::Draw()\n";
}

Line::Line(int x1, int y1, int x2, int y2) : Object(x1, y1, x2, y2) {
    std::cout << "Line::Line()\n";
}
Line::~Line() {
    std::cout << "Line::~Line()\n";
}
void Line::Draw() {
    std::cout << "Line::Draw()\n";
}

Rect::Rect(int x1, int y1, int x2, int y2) : Object(x1, y1, x2, y2) {
    std::cout << "Rect::Rect()\n";
}
```

```
Rect::~Rect() {
    std::cout << "Rect::~Rect()\n";
}
void Rect::Draw() {
    std::cout << "Rect::Draw()\n";
}


int main() {
    Object* pObjects[20];
    int objectCount = 0;

    // Create some lines and rectangles in some order
    std::cout << "\nCreating Line\n";
    pObjects[objectCount++] = new Line(0, 0, 1, 1);
    std::cout << "\nCreating Rect\n";
    pObjects[objectCount++] = new Rect(1, 1, 2, 2);
    std::cout << "\nCreating Rect\n";
    pObjects[objectCount++] = new Rect(10, 10, 20, 20);
    std::cout << "\nCreating Line\n";
    pObjects[objectCount++] = new Line(5, 5, 10, 10);

    std::cout << "\nMove first line\n";
    // Move first line
    pObjects[0]->Move(10, 10);
```

Due to inheritance each Derived class object has a base class object in it ⇒ address of Derived class object can be stored in Base class pointer/reference

```
    std::cout << "\nDraw all\n";
    // Draw all objects in the order of creation.
    for (int i = 0; i < objectCount; ++i) {
        pObjects[i++]->Draw();
    }

    std::cout << "\nDelete all\n";
    // Delete the memory for objects created.
    for (int i = 0; i < objectCount; ++i) {
        delete pObjects[i];
    }

    return 0;
}
```

Object *

```
class Base {
    int b;
public:
    void f1() {..}
    void f2() {..}
};

class Derived : public Base {
    int d;
public:
    void g1() {..}
    void f2() {..}
};
```

function overriding

```
int main() {
    Base obj B;
    Derived obj D;

    obj B . f2();
    obj D . f1();
    obj D . f2()
```

obj B → Base

f2() ⇓ Base :: f2

obj D → Derived

f1() ⇓ Base :: f1

obj D → Derived

f2() ⇓ Derived :: f2

Base *pb = &objD;

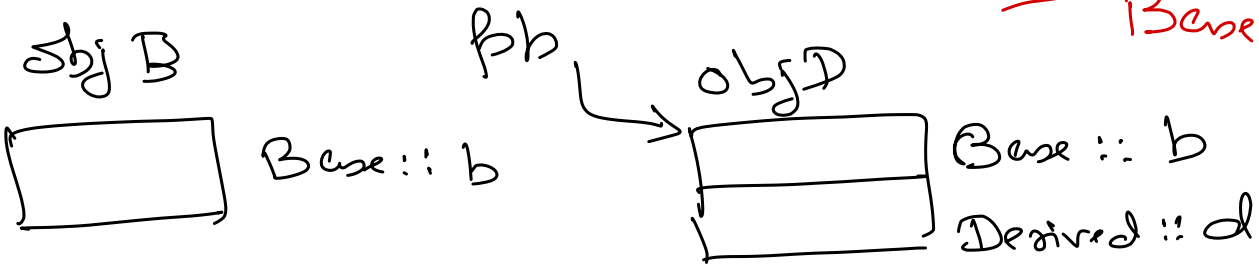
Object slicing

objD.Base::f2();

Derived

Base::f2

Base class pointer/reference pointing or referencing to Derived class object



pb -> d = 10;

Base *

Is d a member of Base? No

$\underbrace{pb \rightarrow}_{Base *}$ f2(); ✓ ⟹ Base :: f2

Is f2 a member of Base? YES

Is it accessible? YES

Base * pb1 = new Derived();

⟹ Base :: Base()

Derived :: Derived()

pb1 →

Base :: b

Derived :: d

⋮

delete $\underbrace{pb1}_{Base *}$; ⟹ Base :: ~Base()

```
void swap(int& a, int &b) {
    int t;

    t = a; a = b; b = t;
}
void swap(float& a, float &b) {
    float t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "Before swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "Before swap f1 = " << f1 << ", f2 = " << f2 << "\n";
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```

```
template<class T>
void swap(T& a, T& b) {
    std::cout << "Calling template function\n";

    T t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "\nBefore swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "\nBefore swap f1 = " << f1 << ", f2 = " << f2 << "\n";
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```

```
template<class T>
void swap(T& a, T& b) {
    std::cout << "Calling template function\n";

    T t;

    t = a; a = b; b = t;
}

template<>
void swap(float& a, float& b) {
    std::cout << "Calling template specialization for float\n";

    float t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "\nBefore swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "\nBefore swap f1 = " << f1 << ", f2 = " << f2 << "\n";
```

```
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```