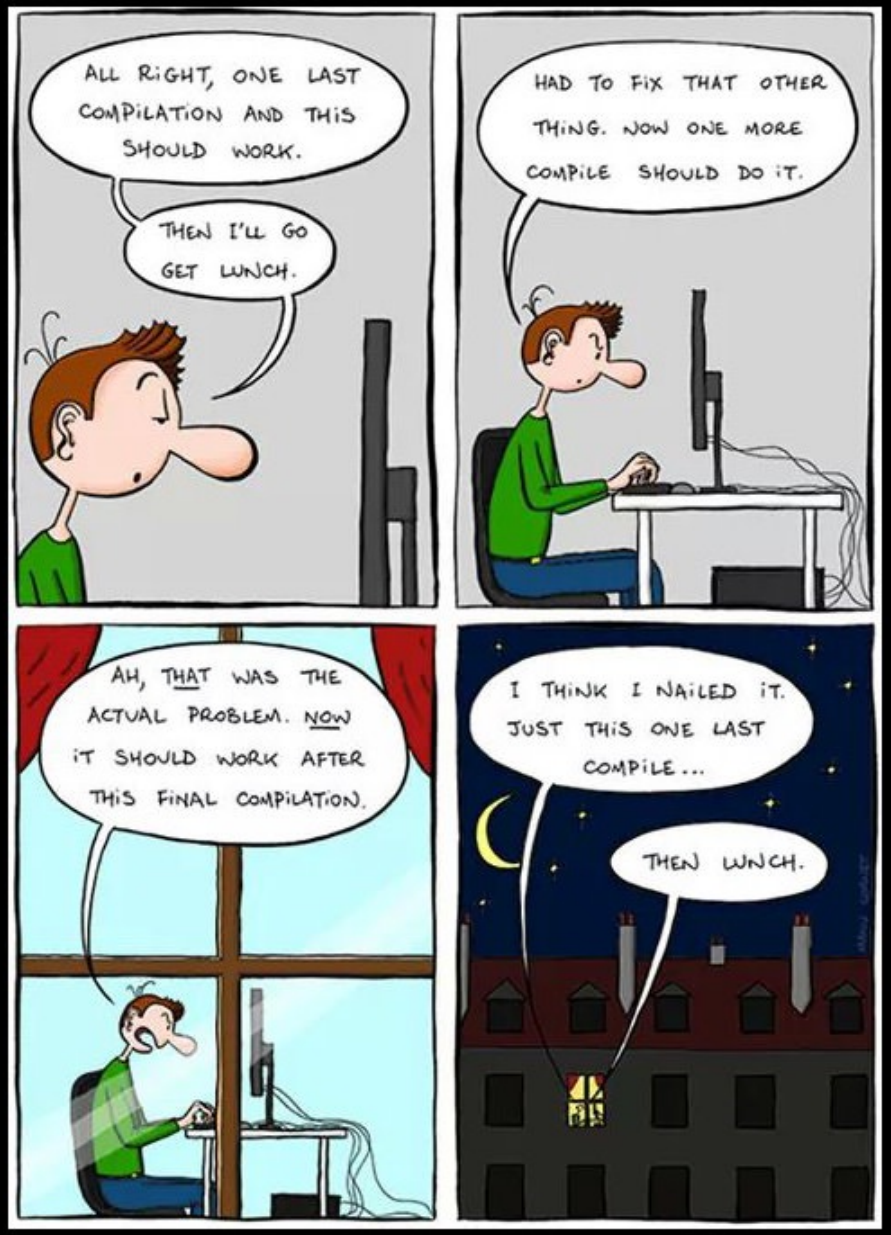


Assignment 1
is over!

Lecture 8

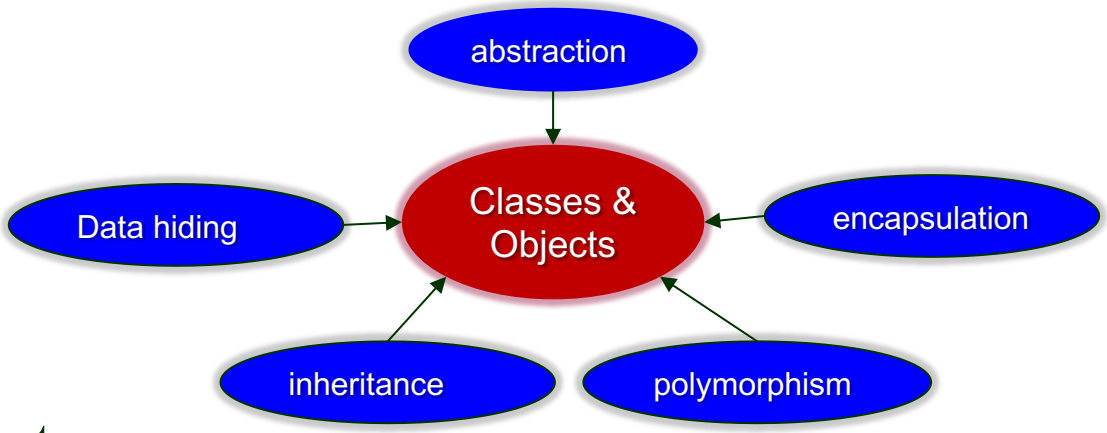
Inheritance

Topics covered by last lecture

- String as an array of characters
- String as an object of string class
- Stream and file I/O
- Static variables
- Constant variables
- Copy constructor

Topics covered by this lecture

- Composition
- Inheritance
  - Class declaration
  - Inheritance type
  - Inherited access
  - Constructor and destructor
- Type conversion
- Class hierarchies



Composition

There are two ways to define a new data type from existing data type:

- Composition: declare a new data type by using existing data types
- Inheritance: derive a new data type from existing data types.
- Use which for what?
  - Composition: *A is part of B*. e.g. engine is part of car.
  - Inheritance: *A is a B*. e.g. a SmartPlayer is a Player. A "ComputerScienceBook" is a Book.

Composition vs Inheritance

- Objects in a composited class can communicate each other via their interface (public functions).

```
class Board {
   TicTacToe grid[boardSize][boardSize];
   BoardCoordinate focus;
public:
   bool addMove(int,int,int,int);
   char checkWin();
   void printBoard();
   . . .
};
```

```
class Player {
protected:
   char playerSymbol;
public:
   virtual void getMove(Board*,int&,int&)=0;
   char getSymbol();
   . . .
};
```

```
class Game {
  Board* board;
  Player *players[2];
public:
  Game(Board*,Player*,Player*);
  void play();
};
```

A game consists of a game board and two players - Composition

```
void Game::play() {
   while(!board.checkWin()) {
         int x1, y1, x2, y2;
         player[0]->getMove(board,x1,y1);
         player[1]->getMove(board,x2,y2);
         board.addMove(x1,y1,x2,y2);
          board.printBoard();
      }
   }
}
```

Composition vs Inheritance

```
class Player {
protected:
  char playerSymbol;
public:
  virtual void
       getMove(Board*,int&,int&)=0;
  char getSymbol();
  . . .
};
```

All these classes are almost the same except the implementation of *getMove* function

```
class HumanPlayer : public Player {
public:
 void getMove(Board*,int& x,int& y) {
    cin >> x >> y;
    x--;y--;
};
```

```
class RandomPlayer : public Player {
public:
 void getMove(Board* bPtr,int& x,int& y) {
   do {
      x = index / bPtr->getSize();
      y = index % bPtr->getSize();
   } while(!bPtr->isValid(x,y))
   return;
};
```

```
class MonteCarloPlayer : public Player {
    double simulation(Board b);
    double expansion(Board b);
public:
 void getMove(Board*,int&,int&);
};
```

Inheritance

A Manager is an Employee

```
class Employee {
      public:
          string      firstName, lastName;
          int         employeeId;
          Date        hiringDate;
};
class Manager: public Employee {
      public:
          int         level ;
          string      officeNumber;
};
```

- Manager is derived from Employee. (Derivation), or Employee is a base class for Manager.
- Manager has all the members of Employee in addition to its own members.

```
Manager m1;
m1. firstName = “John”; m1.lastName = “Smith”;
```

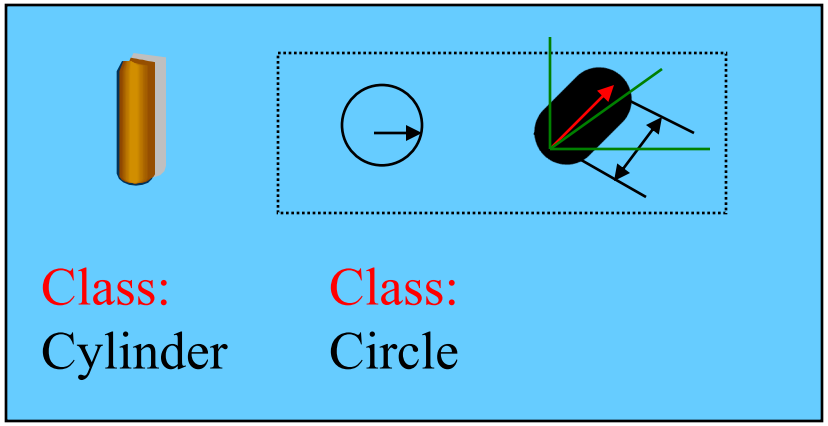
Inheritance

- Inheritance allows one class to be derived from existing classes by acquiring, or inheriting, their *data items* and *member functions*. It can then be altered by adding new data members or member functions, or by modifying (overriding) existing member functions and their access privileges.

Inheritance Declaration

```
class Circle {
    private:
        double radius;
    public:
        Circle(double r=1.0) { radius = r;}
        double calVal();
};

double Circle::calVal(void)   // this calculate the area of the circle
{
    return (PI * radius * radius);
}
```



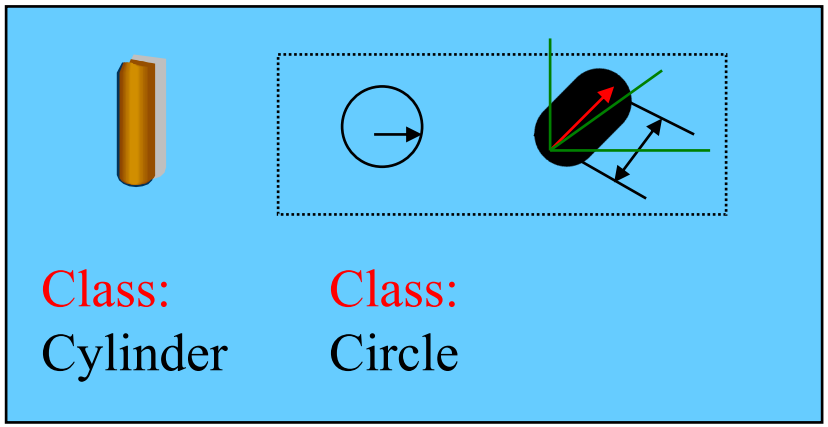
It is important to have a constructor to initialise data members!

Inheritance Declaration

Initialise the data members of the base class.

```
class Cylinder : public Circle {
    private:
        double length;
    public:
        Cylinder(double r, double l): Circle(r),length(l) {}
        double calVal();
};

double Cylinder::calVal(void)
{
    return (length*Circle::calVal());
}
```



Inheritance Declaration

```
int main() {
  Circle circle1, circle2(2);  // create two Circle obj
  Cylinder cylinder1(3,4);      // create one Cylinder obj

  cout << "The area of circle_1 is " << circle1.calVal() << endl;
  cout << "The area of circle_2 is " << circle2.calVal() << endl;
  cout << "The volume of cylinder1 is " << cylinder1.calVal()
       << endl;

  circle2 = cylinder1;  // assign a cyl to a Circle
  cout << "\nThe area of circle_1 is now "  << circle2.calVal() << endl;
}
```

Circle.cpp

Member ownership and access

- A member of a derived class can use any public or protected members of its base class.

```
class Employee {
private:
    string firstName, lastName;
    char middleInitial;
protected:
    string fullName() {return firstName+' '+middleInitial+' '+lastName; }
public:
     // constructor is needed here.
    void print() {cout << "Employee " << fullName() << endl;}
};
class Manager: public Employee {
    int level;
public:
    // constructor is needed here.
    void printManager() { cout << ”Manager " << fullName() << “ at  level =“
                                  level << endl; }
};
```

Function overriding

- A member of a derived class can override a public or protected members of its base class.

```
class Employee {
private:
    string firstName, lastName;
    char middleInitial;
protected:
    string fullName() {return firstName+' '+middleInitial+' '+lastName; }
public:
     // constructor is needed here.
    void print() {cout << "Employee " << fullName() << endl;}
};
class Manager: public Employee {
    int level;
public:
    // constructor is needed here.
    void print() { cout << ”Manager " << fullName() << “ at  level =“
                                    level << endl; }
};
```

Function overriding

Use member functions of base class

- You can also call directly the *print()* member function of the base class in the derived class by using scope operator :

```
void Manager::print() const {
    Employee::print() ;          // print Employee information
    // Print extra information for managers
    cout << “at level:” << level << endl ;
}
```

manager.cpp

Override existing code

Suppose you received a task to work on a project with existing code. You can keep the existing code untouched by overriding the functions you do not like (you might need to delete useless variables and change the accessibility of some data members or member functions from private to protected).

Overriding.cpp

Accessibility

Base case: `private, protected, public`

```
class B {
private:
   int x;
protected:
   int y;
public:
   int z;
};
```

```
int main() {
    B b;
    cout<<b.x; //wrong
    cout<<b.y; //wrong
    cout<<b.z; //correct
};
```

Accessibility

Accessibility.cpp

Base case: `private, protected, public`
Derived: `private, protected, public`
Outside: accessible or not accessible

```
class B {
private:
   int x;
protected:
   int y;
public:
   int z;
};
```

```
class D: public B  {
private:
    int r;
public:
    void sum() {
        r = x+y+z; //wrong
        r = y+z; //correct
    }
};
```

```
int main() {
    B b;
    cout<<b.x; //wrong
    cout<<b.y; //wrong
    cout<<b.z; //correct
    D d;
    cout<<d.x; //wrong
    cout<<d.y; //wrong
    cout<<d.z; //correct
    cout<<d.r; //wrong
    d.sum(); //correct
};
```

Inheritance type

How to make data access correct?

```
class B {
private:
   int x;
protected:
   int y;
   int getX() {
     return x;
   }
public:
   int z;
};
```

```
class D: public B  {
protected:
    int r;
public:
    void sum() {
       r = getX()+y+z;
    }
};
//all correct
```

```
class E: public D  {
public:
    void print() {
        cout<<getX()+y+z+r;
    }
};

//all correct
```

```
int main() {
    E e;
    e.z = 0;
    e.sum()
    e.print();
};
//all correct
```

Accessibility2.cpp

Inheritance type

How do they affect data access?

Three ways to extend a class:

```
class DerivedClassName: public BaseClassName
class DerivedClassName: private BaseClassName
class DerivedClassName: protected BaseClassName
```

```
class B {
private:
  int x;
protected:
  int y;
  int getX() {
     return x;
   }
public:
  int z;
};
```

```
class D: private B  {
protected:
   int r;
public:
   void sum() {
      r = getX()+y+z;
   }
};
```

```
class E: protected D  {
public:
   void print() {
     cout<<getX()+y+z+r;
  }
};
```

```
int main(  {
    E e;
    e.z = 0;
    e.sum()
    e.print();
};
```

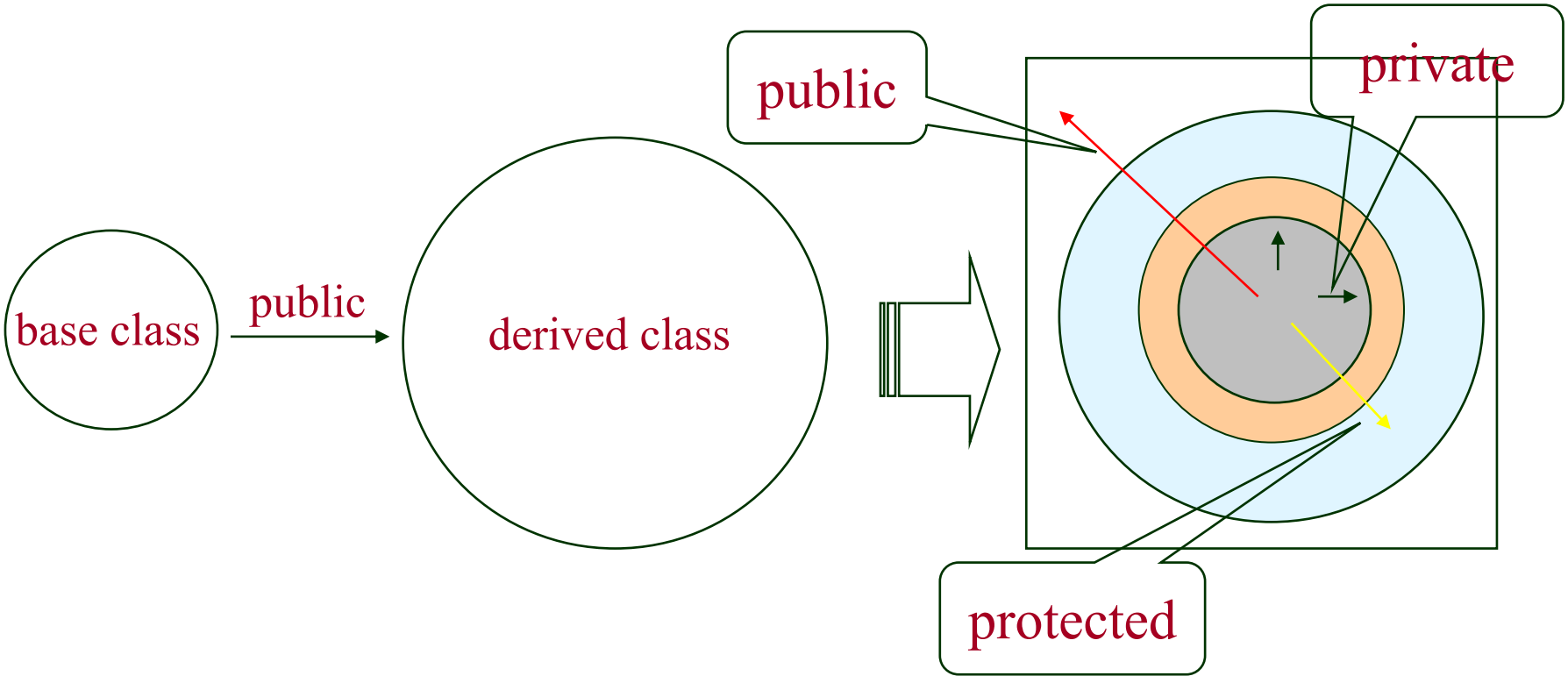
Incorrect

Incorrect

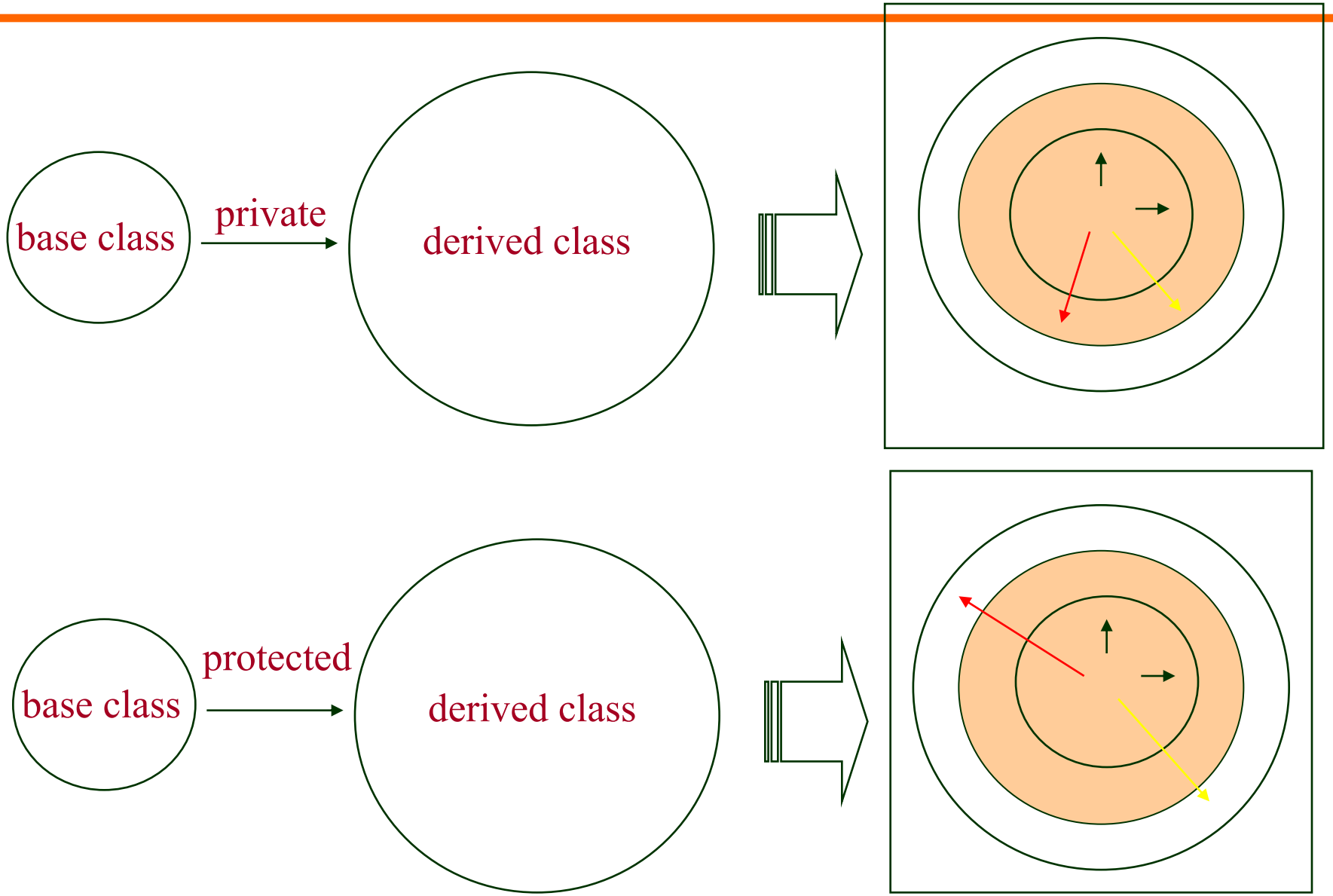
Accessibility3.cpp

Class Access

The manner in which a derived class is declared provides a mean of further limiting member access to inherited classes. In the example of Cylinder class, the class access was declared as public. This provides any derived class of Cylinder class the ability to access the base class (Circle)'s members with the same privileges as the base class.



Class Access



Class Access

class Cylinder : public Circle

Data member in base class / Type of inheritance	: public	: private	: protected
private	Inaccessible from derived class inaccessible outside	Inaccessible from derived class Inaccessible outside	Inaccessible from derived class inaccessible outside
protected	protected in derived class Inaccessible outside	private in derived class Inaccessible outside	protected in derived class Inaccessible outside
public	public in derived class Accessible outside	private in derived class Inaccessible outside	protected in derived class Inaccessible outside

```
class Circle {
private: double radius;
public:  double calVal();
};
```

The constructor initializer list

- It is important that every member item of an object is initialized.
- When a member item of a base class is private, the derived class can only initialize the variables through initializer list via constructors.

```
class Circle {
    private:
        double radius;
    public:
        Circle(double r ) {
            radius = r;
         }
        double calVal();
};
```

```
class Cylinder : public Circle {
    private:
        double length;
    public:
        Cylinder(double r, double l):
            Circle(r), length(l) {}
        double calVal();
};
```

Constructors in inheritance

There must be suitable constructors in base class and derived classes.

```
class A {
    private:
      int val;
    public:
      A(int x) { val=x; }
};
class B: public A {
    public:
      B(int x): A(x) { }
      B(int x) { }          // INCORRECT
      B() : A(5) { }
      B() { }               // INCORRECT
};

class C: public A {   // INCORRECT no constructor for class C
};
```

ab.cpp

Constructor and destructor execution

- First execute the constructor of base class and then of derived class when an object of derived class is created.
- First execute the destructor of derived class and then of base class when an object of derived class is destroyed.

```
class A {
   public: A() { cout << “ctor:A()” << endl ; }
           ~A() { cout << “dtor:~A()” << endl ; }
};
class B : public A {
   public: B() { cout << “ctor:B()” << endl ; }
           ~B() { cout << “dtor:~B()” << endl ; }
};
int main() {
   B b ;
}
```

call.cpp

managerOrder.cpp

Type conversion and casting

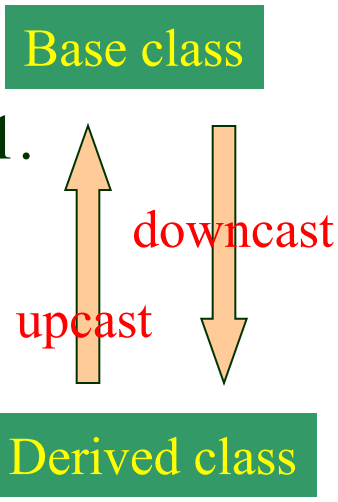
- **Upcasting:** If a class "Derived" has a public base class "Base" then a "Derived" object can be assigned to a variable of type "Base" without explicit casting.

```
Employee staff1("John", "Smith", 'D');
Manager staff2("Peter", "Wang",' ');
staff2.setLevel(2);
staff1 = staff2; //object slicing
staff1.print();
```

managerCopy.cpp

Note:

- Only the members of Employee class are copied to staff1.
- Inheritance type should be public.
- A better way to upcast an object is to use pointers.



ObjectSlicing.cpp

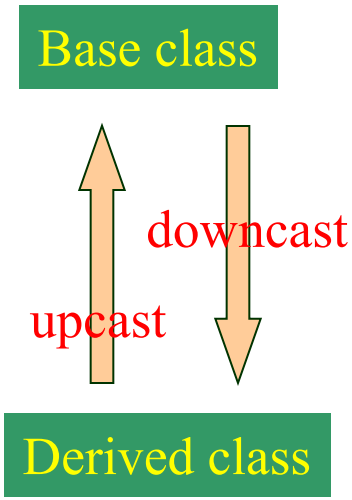
Type conversion and casting

- **Downcasting**: *A base class object cannot be automatically converted to the derived class.*

```
staff2 = staff1; //invalid
```

- You may use static_cast<> to force a downcasting in your risk

```
void g ( Manager mm, Employee ee) {
    Employee* pe = &mm;                    // ok:
    Manager* pm = &ee;                     // INCORRECT
    pm->level = 2;                         // INCORRECT
    pm = static_cast<Manager*>(pe) ;  // works.
    pm->level = 2;                 // fine.
    // …
}
```



managerCopy.cpp

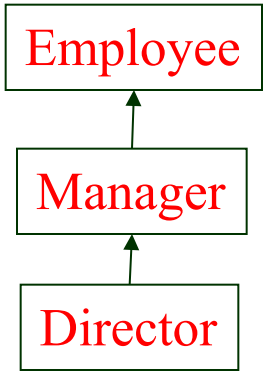
Automatic type conversion and casting

- Safety Issues of type conversions:
  - Upcasting is always possible and safe especially use pointers.
  - Down-casting needs to be performed with great care.
  - To allow safe down-casting, C++ introduced the concept of dynamic casting. The technique is available for polymorphic classes (will be introduced in Lecture 10).

Class Hierarchies

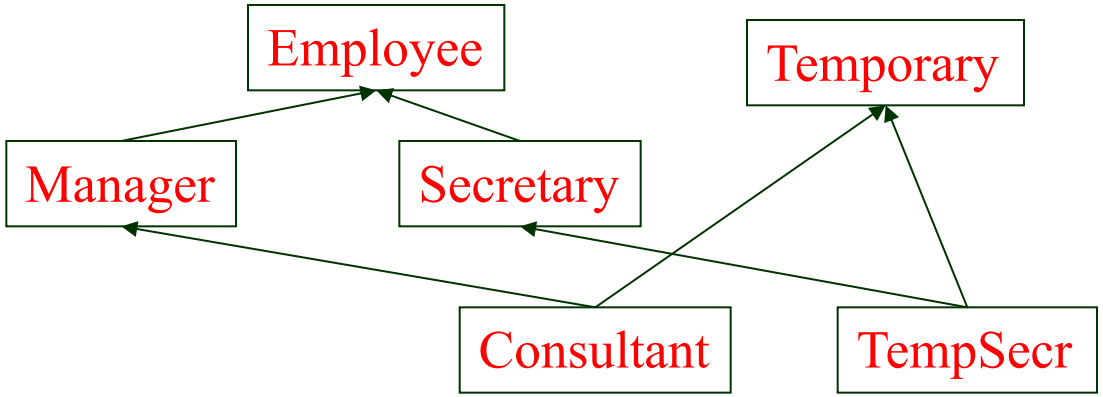
- A derived class can itself be a base class.

```
class Employee { /*… */ };
class Manager : public Employee { /* … */ };
class Director : public Manager { /* … */ };
```



- Multiple Inheritance

```
class Temporary { /* … */ };
class Secretary : public Employee { /* … */ };
class TempSecr : public Temporary, public Secretary { /* … */ };
class Consultant : public Temporary, public Manager {/*…*/};
```



Homework

- Read textbook: Chapter 14
- Assignment 1 demonstration is in this week. Each student will take 10 to 15 minutes. You would experience a significant delay for the demonstration depending on the size of your class.
- Assignment 2 is available now. Please check it out and let me know if you have any questions.