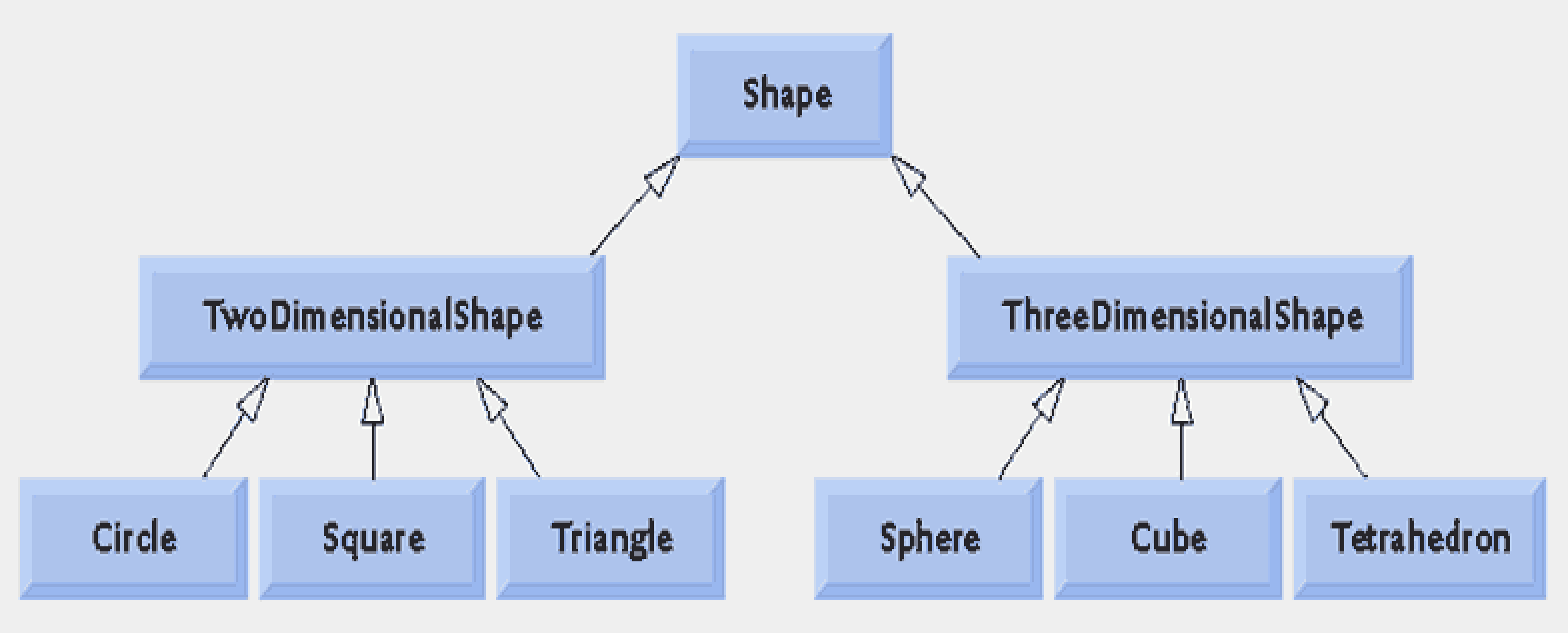
Inheritance

Jiun-Long Huang
National Chiao Tung University

Introduction

- Code reusability is an important programming concept.
- C++ facilitates this concept more efficiently than C by inheritance
- When using inheritance, a new class can be created by establishing parent-child relationships with existing classes.
    - is-a relationship

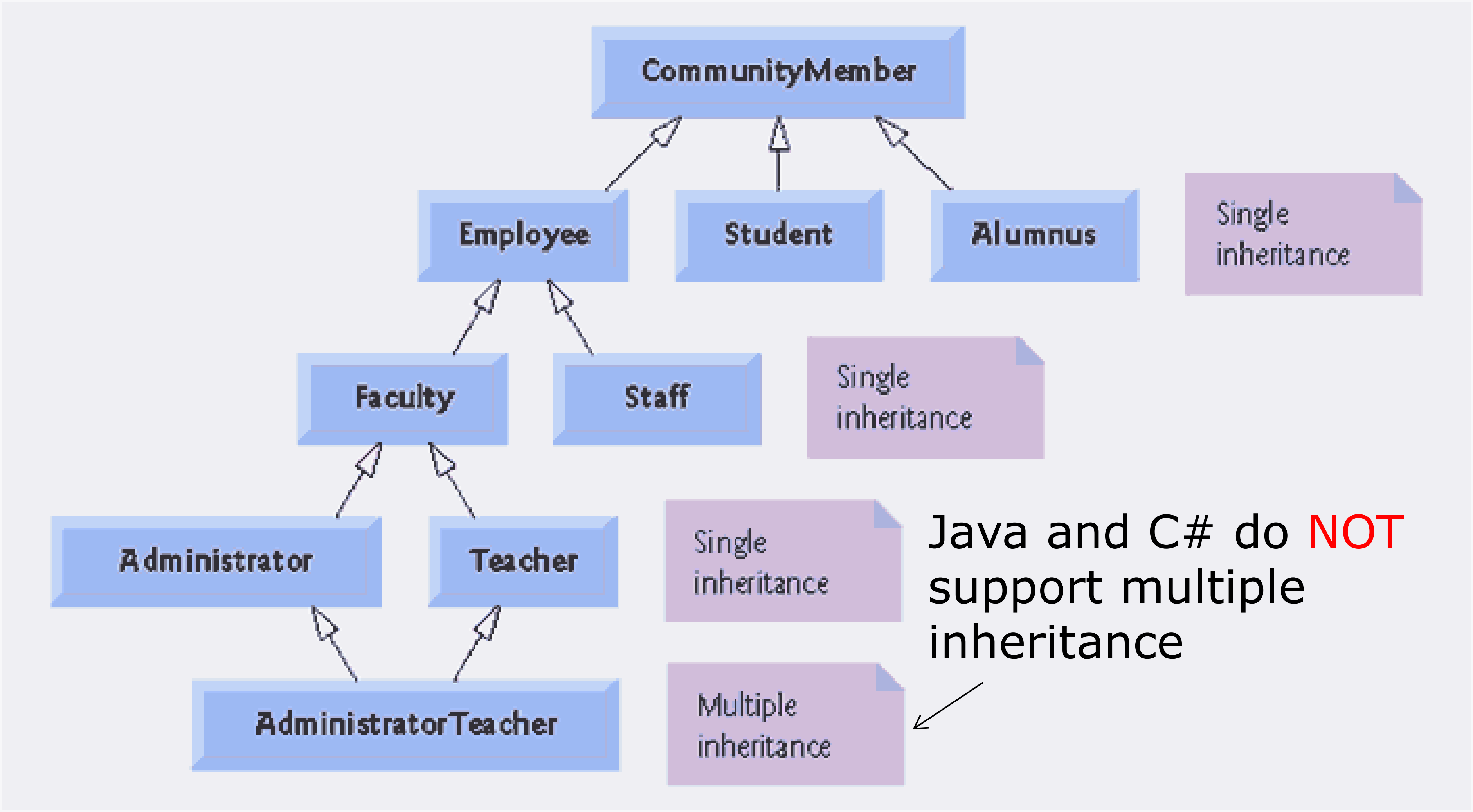
- Inheritance enables a hierarchy of classes to be designed.



- Base/parent/super classes and derived/child/sub classes
  - Object of one class is an object of another class
  - Base class typically represents larger set of objects than derived classes
- Example:
  - Base class: Vehicle
    - Includes cars, trucks, boats, bicycles, etc.
  - Derived class: Car
    - Smaller, more-specific subset of vehicles

- Derived class automatically has base class’s:
  - Member variables
  - Member functions
- Derived class can add additional member functions and variables
- Derived class can modify member functions and variables inherited from base class
  - Override

Multiple Inheritance



Java and C# do NOT support multiple inheritance

Inheritance

```
class derived_classname : access_specifier base_classname
{
  //specific properties of the derived-class
}
```

- access_specifier can be
  - private (default),
  - protected or
  - public

Private/Protected/Public Data Members

- A member of a class can be private, protected, or public :
  - If it is private, its name can be used only by member functions and friends of the class in which it is declared.
  - If it is protected, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class
  - If it is public, its name can be used by any function.

From *The C++ Programming Language*

Private Inheritance

- Consider a class D derived from a base class B :
- If B is a private base, (class D : private B)
  - B’s public and protected members can be used only by member functions and friends of D, and
  - only friends and members of D can convert a D to a B.

From *The C++ Programming Language*

Protected Inheritance

- If B is a protected base, (class D : protected B)
  - B’s public and protected members can be used only by member functions and friends of D and by member functions and friends of classes derived from D, and
  - only friends and members of D and friends and members of classes derived from D can convert a D to a B.

From *The C++ Programming Language*

Public Inheritance

- If B is a public base, (class D : public B)
  - B’s public members can be used by any function,
  - B’s protected members can be used by members and friends of D and members and friends of classes derived from D, and
  - any function can convert a D to a B.

From *The C++ Programming Language*

Private Inheritance

	Private member	Protected member	Public member
Base class B and its friends	O	O	O
Derived class D and its friends	X	O	O
Classes derived from D and their friends	X	X	O
Other	X	X	O

Protected Inheritance

	Private member	Protected member	Public member
Base class B and its friends	O	O	O
Derived class D and its friends	X	O	O
Classes derived from D and their friends	X	O	O
Other	X	X	O

Public Inheritance

	Private member	Protected member	Public member
Base class B and its friends	O	O	O
Derived class D and its friends	X	O	O
Classes derived from D and their friends	X	O	O
Other	X	X	O

- Who knows the "is-a" relationship (inheritance)?

	Private Inheritance	Protected Inheritance	Public Inheritance
Derived class D and its friends	O	O	O
Classes derived from D and their friends	X	O	O
Other	X	X	O

```
class X
{
};

class Y : private X // ERROR
//class Y : protected X // ERROR
//class Y : public X // OK
{
};

void func(X a)
{
}

int main()
{
  class Y y;
  func(y);
}
```

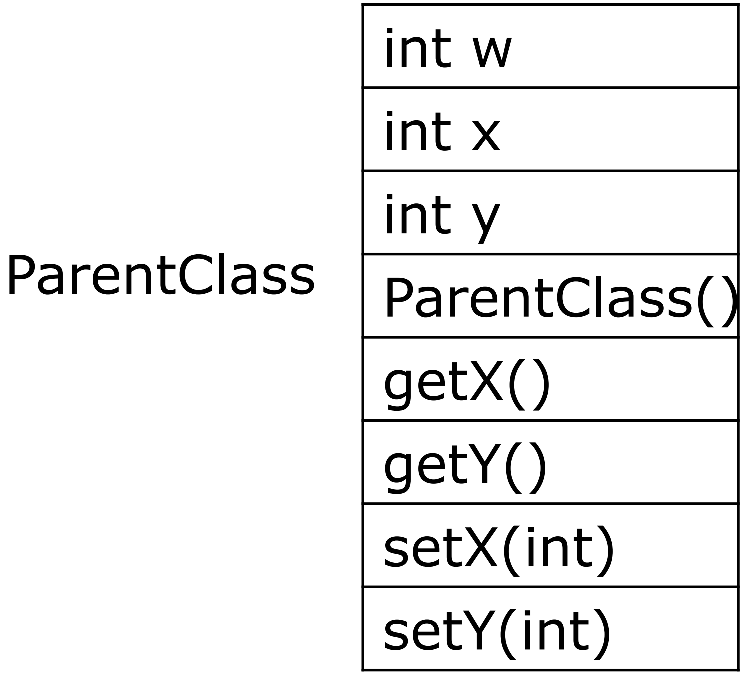
```
a.cc: In function `int main()':
a.cc:16: error: `X' is an
inaccessible base of `Y'
a.cc:16: error:   initializing
argument 1 of `void func(X)'
```

An object of class Y (the derived class) is an object of class X (the base class

Remarks

- For private inheritance:
  - Private inheritance is included in the language for completeness
- For protected inheritance
  - It’s something you don’t use very often, but it’s in the language for completeness.
- Public inheritance is used in most cases

```
#include <iostream>
using namespace std;
class ParentClass
{
private:
  int w;
protected:
  int x, y;
public:
  ParentClass() {
    cout<<"Parent's constructor"<<endl;
    x=1;
    y=1;
  }
  int getX() const { return x; }
  int getY() const { return y; }
  void setX(int n) { x=n; }
  void setY(int n) { y=n; }
};
```

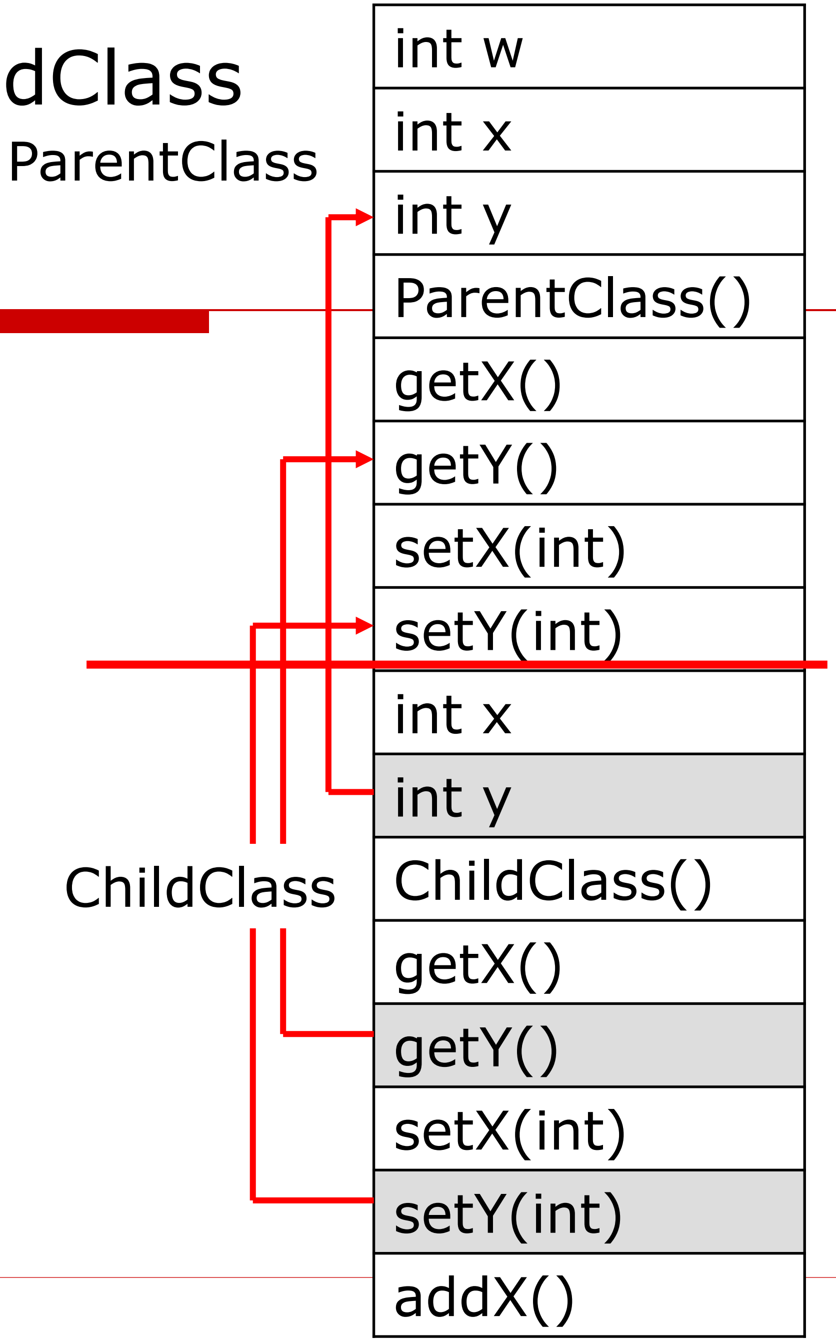


ParentClass

int w
int x
int y
ParentClass()
getX()
getY()
setX(int)
setY(int)

```
class ChildClass : public ParentClass
{
protected:
  int x;  //Override
public:
  ChildClass() {
    cout<<"Child's constructor"<<endl;
    x=0;
  }
  int getX() const { //Override
     return x;
  }
  void setX(int n) { //Override
     x=n*3;
  }
  void addX() { //Add member function
     x++;
  }
};
```

An object of class ChildClass



ParentClass

int w

int x

int y

ParentClass()

getX()

getY()

setX(int)

setY(int)

ChildClass

int x

int y

ChildClass()

getX()

getY()

setX(int)

setY(int)

addX()

```
int main() {
  cout << "sizeof(ParentClass) = " << sizeof(ParentClass)
       << endl;
  cout << "sizeof(ChildClass) = " << sizeof(ChildClass)
       << endl;
  ParentClass * p;
  ChildClass c;
  p=&c;
  cout<<p->getX()<<" "<<p->getY()<<endl;
  cout<<c.getX()<<" "<<c.getY()<<endl;
  c.setX(2);
  p->setX(4);
  c.setY(3);
  p->setY(3);
  cout<<p->getX()<<" "<<p->getY()<<endl;
  cout<<c.getX()<<" "<<c.getY()<<endl;
}
```

```
sizeof(ParentClass) = 12
sizeof(ChildClass) = 16
Parent's constructor
Child's constructor
1 1
0 1
4 3
6 3
```

- The following data members and functions cannot be inherited from a base class:
  - private members (private data members and private member functions)
    - inaccessible
  - constructor and destructor functions
  - friend functions
  - static functions
  - operator functions that overload the assignment operator

Upcasting and Downcasting

- Up-casting
  - One can cast an object of a derived class to an object of the base class of the derived class

```
class Vehicle {...};
class Car: public Vehicle {...};
void func(Vehicle & v) {...}

Car c();
func((Vehicle)c);
func(static_cast<Vehicle>(c));
func(c); // automatic up-casting
```

Upcasting and Downcasting (contd.)

- Down-casting
  - Try to cast an object of a derived class to an object of the base class of the derived class

```
class Vehicle {...};
class Car: public Vehicle {...};
void func2(Car & c) {...}

Vehicle v();
Func2(v); // X
func2((Car)v); // X
func2(static_cast<Car>(v)); // X
```

Upcasting and Downcasting (contd.)

```
class Vehicle {...};
class Car: public Vehicle {...};
void func(Vehicle & v)
{
  // do something for Vehicle
  Car & c=v; // Error: compilation error
  //Car * C=(Car *)v; // X
  //Car * C=static_cast<Car *>(v); // X
  // do something for Car
}
```

Successful compilation. However, a vehicle is not always a car

http://www.cppreference.com/wiki/keywords/dynamic_cast

Run-Time Type Information

- RTTI (Run-Time Type Information, or Run-Time Type Identification) refers to a C++ system that makes information about an object's data type available at runtime.
  - The dynamic_cast<> operation and typeid operator in C++ are part of RTTI.
  - With C++ run-time type information, you can perform safe typecasts and manipulate type information at run time.

http://en.wikipedia.org/wiki/Run-time_type_information

Run-Time Type Information (contd.)

- RTTI is available only for classes which are polymorphic, which means they have at least one virtual method.
  - In most cases, one added a virtual destructor into a class to make it polymorphic
- RTTI is optional with some compilers—you choose at compile time whether to include the function
- RTTI is the C++ implementation of a more generic concept called reflection or, more specifically, type introspection.

```
class Vehicle {virtual ~Vehicle(){}};
class Car: public Vehicle {};
void func(Vehicle * v)
{
  // do something for Vehicle
  Car * c=dynamic_cast<Car *> v;
  if (c==NULL) {
    // show error message
  }
 else {
  // do something for Car
  }
}
```

v is not pointing to an object of class Car of the derived class of Car

Operator typeid

- Two usages
  - typeid(type)
  - typeid(expression)
- Header <typeinfo> must be included before using typeid operator.
- Return value
  - The reference to type_info object corresponding to the given type or type of the given expression.

type_info

- Members
  - operator==
  - operator!=
    - Comparison operators. They return whether the two types describe the same type.
    - A derived type is not considered the same type as any of its base classes.
  - Name
    - Returns a null-terminated character sequence with a human-readable name for the type.

http://www.cplusplus.com/reference/std/typeinfo/type_info/

```
// type_info example
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {};
class Derived : public Base {};
class Poly_Base {public: virtual void Member(){}};
class Poly_Derived: public Poly_Base {};

int main() {
  // built-in types:
  int i;
  int * pi;
  cout << "int is: " << typeid(int).name() << endl;
  cout << "  i is: " << typeid(i).name() << endl;
  cout << " pi is: " << typeid(pi).name() << endl;
  cout << "*pi is: " << typeid(*pi).name() << endl << endl;
  // non-polymorphic types:
  Derived derived;
  Base* pbase = &derived;
  cout << "derived is: " << typeid(derived).name() << endl;
  cout << " *pbase is: " << typeid(*pbase).name() << endl;
  cout << boolalpha << "same type? ";
  cout << ( typeid(derived)==typeid(*pbase) ) << endl <<
          endl;
  // polymorphic types:
  Poly_Derived polyderived;
  Poly_Base* ppolybase = &polyderived;
  cout << "polyderived is: " << typeid(polyderived).name()
       << endl;
  cout << " *ppolybase is: " << typeid(*ppolybase).name()
       << endl;
  cout << boolalpha << "same type? ";
  cout << ( typeid(polyderived)==typeid(*ppolybase) ) <<
       endl << endl;
}
```

Possible Output

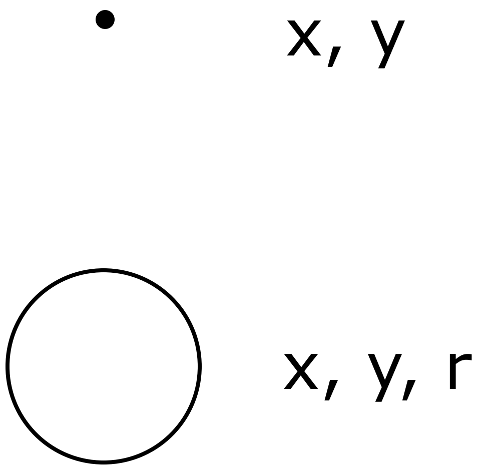
- The result of type_info.name() is compiler-dependent

```
int is: i
  i is: i
 pi is: Pi
*pi is: i

derived is: 7Derived
 *pbase is: 4Base
same type? false

polyderived is:
12Poly_Derived
 *ppolybase is:
12Poly_Derived
same type? true
```

A Bad Example



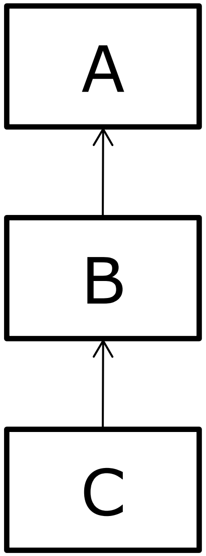
```
class Point
{
protected:
  float x,y;
public:
  Point(float x, float y) {
    this->x=x; this->y=y;
  }
};
class Circle : public Point
{
protected:
  float r;
public:
  Circle(float x, float y, float r) {
    this->x=x; this->y=y; this->r=r;
  }
};
```

Why the example is not good?

Constructing and Destroying Derived Classes

- Every base and derived class have its own constructor and destructor functions.
  - Constructors and destructors cannot be inherited.
- When instantiating an object of a derived class, constructors of all of its parent classes are executed prior to the derived class constructor.

- When creating a child, all of its parents have to be created before the child can be created



Constructor calls:
A→B→C

Destructor calls:
C→B→A

Parameter Lists

- If a base class constructor has arguments, these arguments also have to be added to the argument list of any class derived from this base class.

```
derived_class(arg_list1) : base_class(arg_list2)
{
  //body of derived class constructor
}
```

Example

```
class Vehicle {
protected:
  int maxSpeed;
  int capacity;//maximal number of people in the vehicle
public:
  Vehicle(int s, int c) { maxSpeed=s; capacity=c; }
};
class Car : public Vehicle {
Protected:
  int doorNumber;
public:
  Car(int s, int c, int n) : Vehicle(s, c)
  {
    doorNumber=n;
  }
};
```

Multiple Inheritance

- When implementing direct multiple inheritance, a derived class can directly inherit more than one base class.
- To define a class that directly inherits multiple base classes, the general format is

```
class Derived_class:access specifier Base1_class,
  access specifier Base2_class, ...,
  access specifier BaseN class
{
  //body of derived class
};
```

From *The C++ Programming Language*

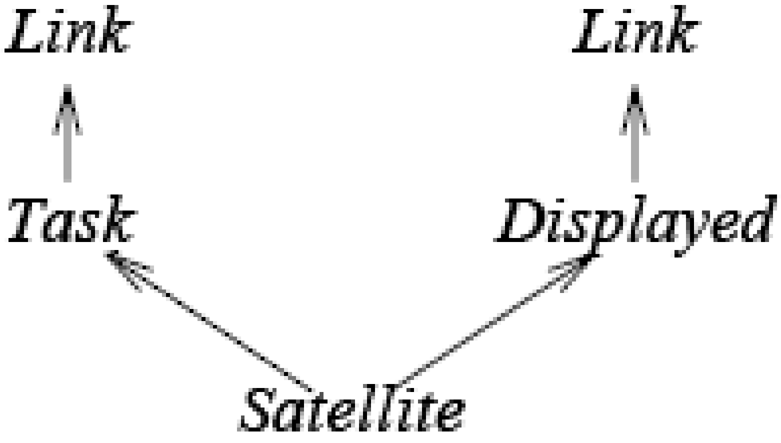
Example

```
class Task {
  ...
  void delay(int);
};
class Displayed {
  ...
  void draw(void);
};
class Satellite : public Task, public Displayed {
  ...
}
void f(Satellite s)
{
  s.draw(); // Displayed::draw();
  s.delay(10); // Task::delay(10);
}
```

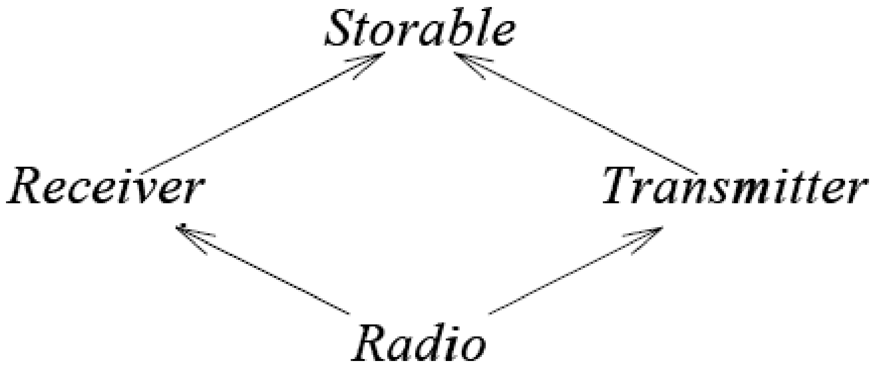
Ambiguity

```
class Task {
  ...
  int getValue(void);
};
class Displayed {
  ...
  int getValue(void);
};
class Satellite : public Task, public Displayed {
  ...
}
void f(Satellite s) {
  int a;
  a=s.getValue(); // ERROR: ambiguious
  a=s.Task::getValue(); // OK
  a=s.Displayed::GetValue(); // OK
}
```

Ambiguity



```
class Link {
  Link * next;
}
class Task : public Link {
  ...
};
class Displayed : public Link {
  ...
};
class Satellite : public Task, public Displayed {
  ...
}
void f(Satellite s) {
  s.next=NULL; // ERROR: ambiguous
  s.Link::next=NULL; // ERROR: ambiguous
  s.Task::Link::next=NULL; // OK
}
```



Ambiguity

- Every virtual base of a derived class is represented by the same (shared) object.

```
class Transmitter : public virtual Storable {
public:
  void write();
};
class Receiver : public virtual Storable {
public :
  void write();
};
class Radio : public Transmitter, public Receiver {
public :
  void write();
};
```

Remarks

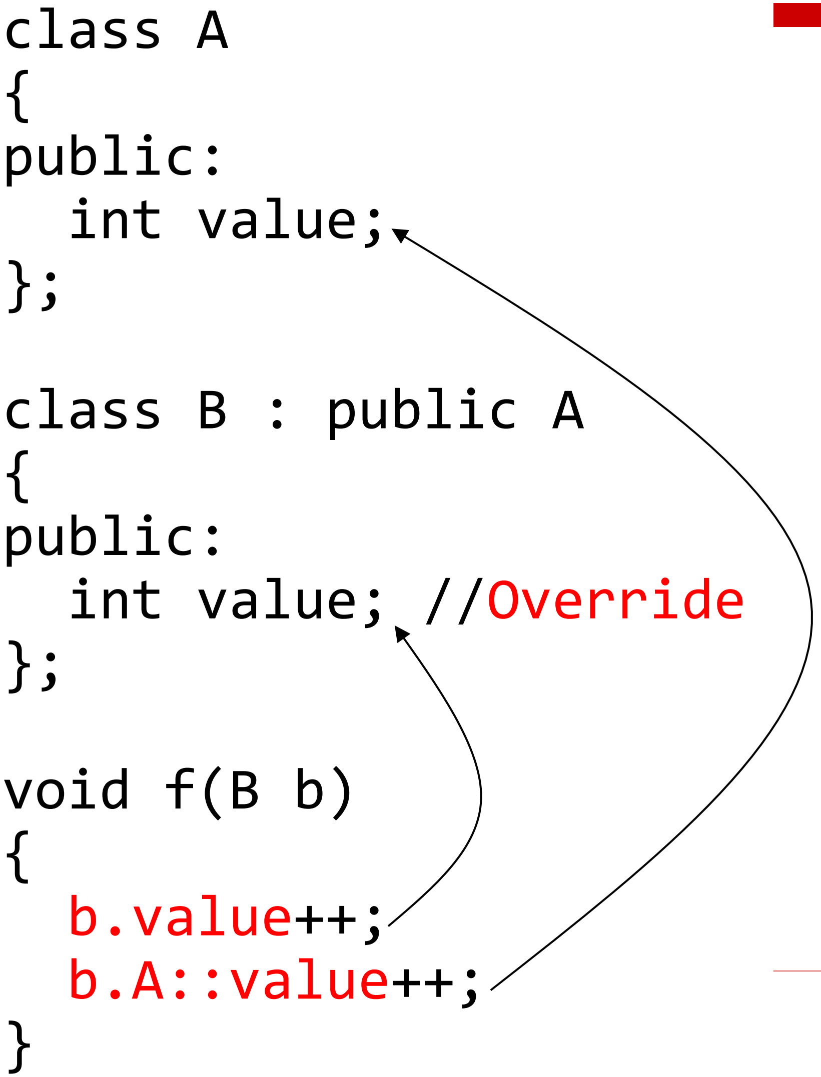
- Some believe that multiple inheritance should never be used
  - Java and C# do not support multiple inheritance
- Multiple inheritance should only be used be experienced programmers!

Dominating and Overriding Base Class Members

- □ A data member of a derived class dominates a data member inherited from a base class that uses the same identifier.
- □ The base class name followed by the scope resolution operator (::) is used only in cases in which it is necessary to distinguish inherited members from the members with the same name declared within the derived class.

```
derived_object.base_class_name::inherited_member
```

Example



```
class A
{
public:
  int value;
};

class B : public A
{
public:
  int value; //Override
};

void f(B b)
{
  b.value++;
  b.A::value++;
}
```