

Intro to Inheritance in C++

CSE 205 - Week 8

Lec Raiyan Rahman

Dept of CSE, MIST

raian@cse.mist.ac.bd



Two Ways of Building buildings



Completely finish each piece or floor before building the next one. Less organized. Suitable for small buildings.



Make the skeleton of the floor first, then finish the floor. Then move to next floor. Suitable for multi storied buildings!

Two Ways of Coding



Structured Programming Languages

Write code to implement a process. usually more suitable for less data-centric, smaller projects.



Object Oriented Programming Languages

Define the “objects” and what they do first, then write processes accordingly. Usually suitable more data-centric, bigger projects.

Let's see an example code (CT + Mid Mark Calculation)

The code is more process centric. The structure for the necessary data (student ID, marks etc) are not very structured and intuitive.

Check out “CT marks - using C.c”

The code is more data centric. The structure for storing the data for each student is clearly defined. Then we add necessary processes for that student (like finding min CT, total marks etc).

Check out “CT marks - using C++.cpp”

Codes: [click here](#)

But how is the C++ (OOP) way of coding helpful?

- It more closely resembles the real life problems scenarios involving information/databases. For example, our MCAM system. It'd be much more intuitive to write in an OOP language than a structured one.
- The data and the necessary processes are coupled together in a class (***encapsulation***, as you've already seen in sec A). Easier for developers to develop, maintain, debug in large projects as well!
- Reuse of code through ***inheritance***. Much less code to write, also helpful in resembling how a system is in real life. (We'll see more on this in today's class)
- Flexibility through ***polymorphism***. (As we'll soon see in later classes).

Reusing or Reworking - A Metaphor



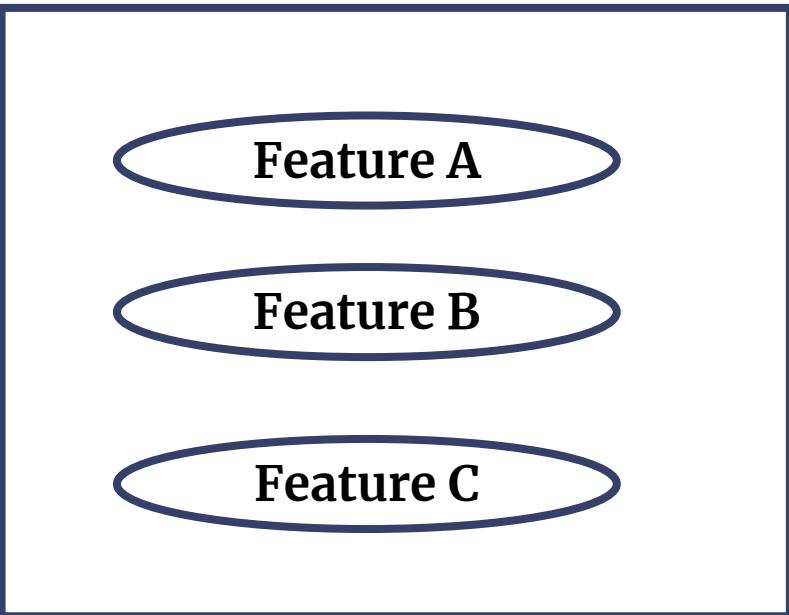
Day 1 Progress



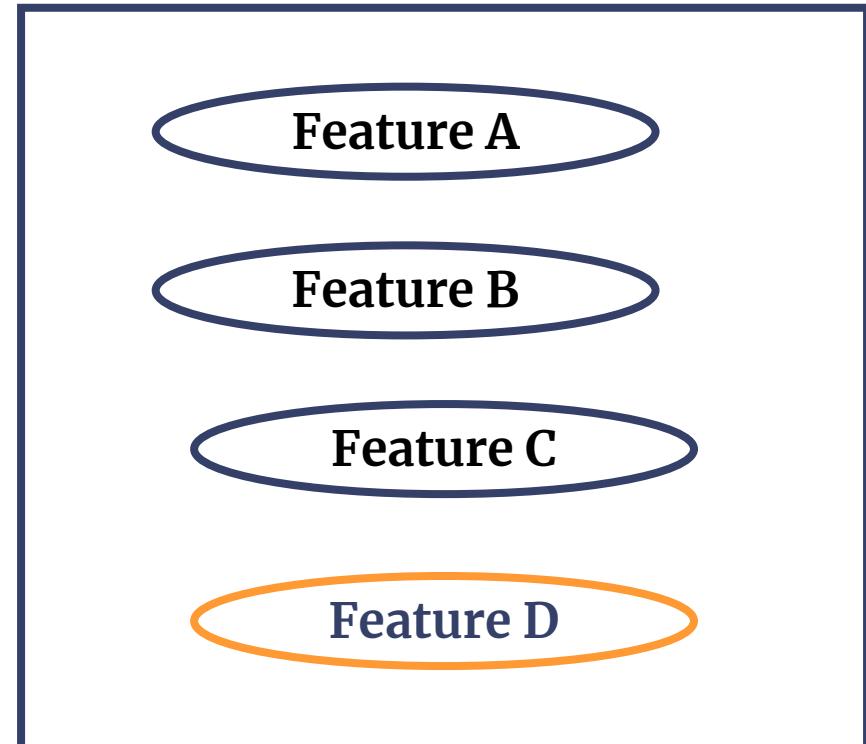
On day 2, do you start all over again or just put in the missing piece?

No, you always **build on top of what you've already done**. The same is for coding.
OOP allows code (and **structural**) reuse as you see logically fit!

An example case of code reuse

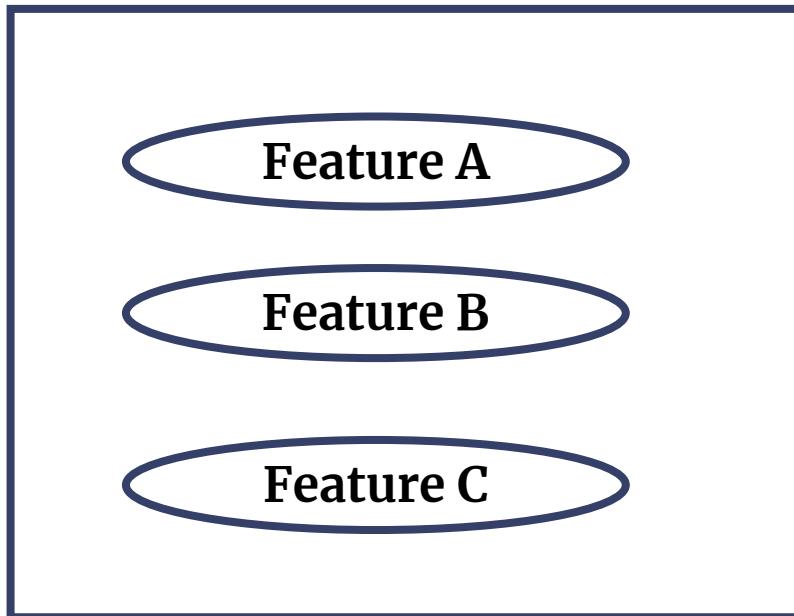


You have a class like this.



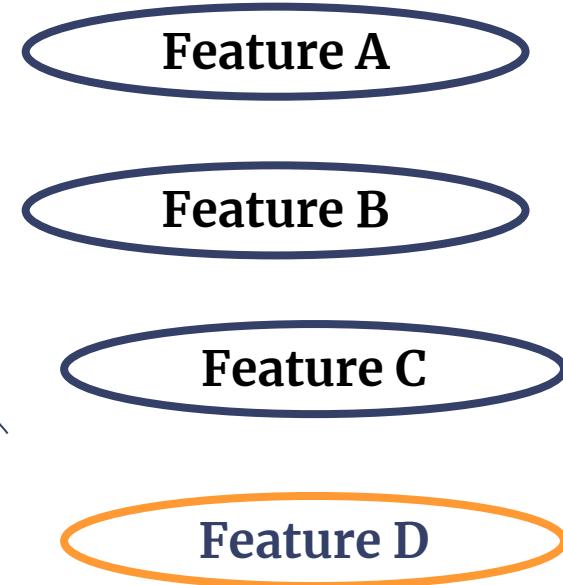
You need another class like this

What if you could ask the **compiler** to



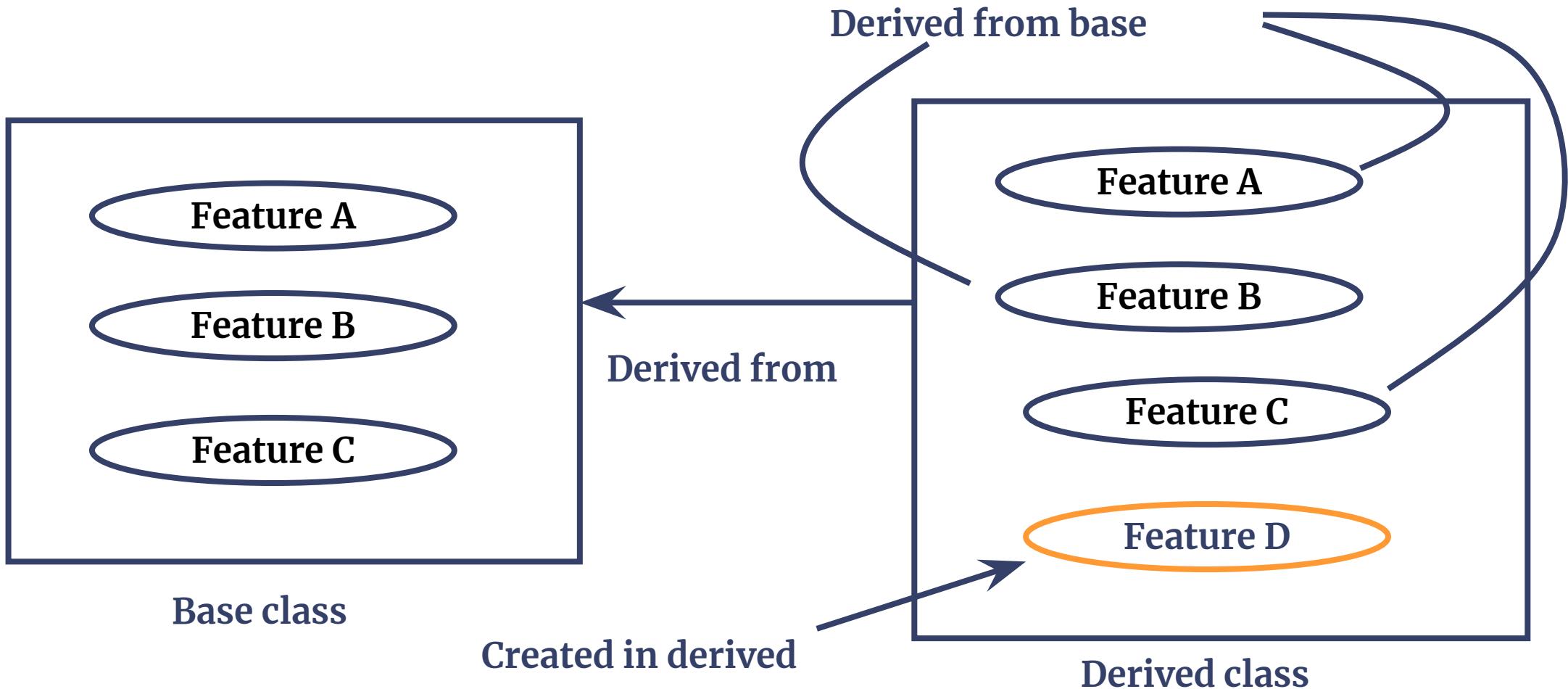
You have a class like this.

Step 1
Copy the class
structure for this
portion from here



Step 2
Then, I'll just tell you what feature "D" is

Well, that's exactly what you can do!



But before we see how we can ask the compiler to “copy” the structure of a class to a new one, let’s have some refreshers on some key concepts!

Two Types of Roles



Computer Maker



Computer User

Two Types of Roles

Class Designer (Class Maker)

Those who design classes

Developer (Class User)

Those who use the classes (by making objects) and calling the functions of a class.

Two Types of Access Modifiers - A Metaphor

(Think like a Class Designer)

private

public

Two Types of Access Modifiers - A Metaphor



private

Your toothbrush
(Only you can use it)

public

you = A C++ class!

Two Types of Access Modifiers - A Metaphor



private

Your toothbrush
(Only you can use it)



public

Your toothpaste (Anyone
can use it)

Inheritance

Say the first class “X” is, $X = 1 + 2 + 3 + 4$

Inheritance

You need a class named Y that's like:

X = 1 + 2 + 3 + 4

Y = 1 + 2 + 3 + 4 + 5

Inheritance

Creating a class from another class(s)

X = 1 + 2 + 3 + 4

Y = X + 5

So, you're inheriting the class definition of class 'X' to make a new class named 'Y'. Then, only adding the special variables and/or functions class Y may have on your own.

Less Code. Closer resemblance to real systems as well!

Inheritance

Creating a class from another class(s)

```
class Point
{
public:
    int x;
    int y;
};
```

Inheritance

Creating a class from another class(s)

```
class Point
{
public:
    int x;
    int y;
};
```

```
class Point3D
{
public:
    int x;
    int y;
    int z;
};
```

Inheritance

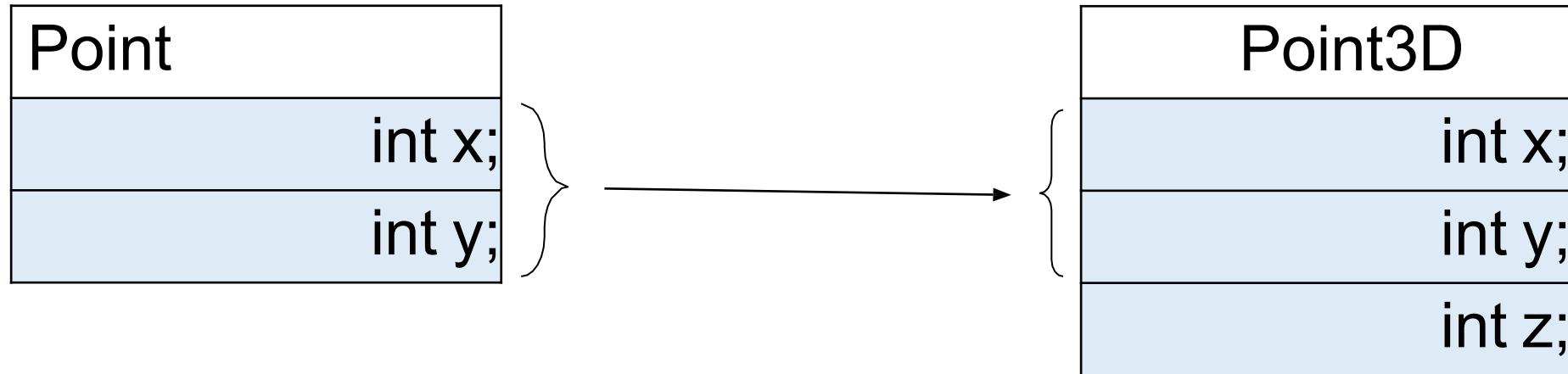
Creating a class from another class(s)

```
class Point
{
public:
    int x;
    int y;
};
```

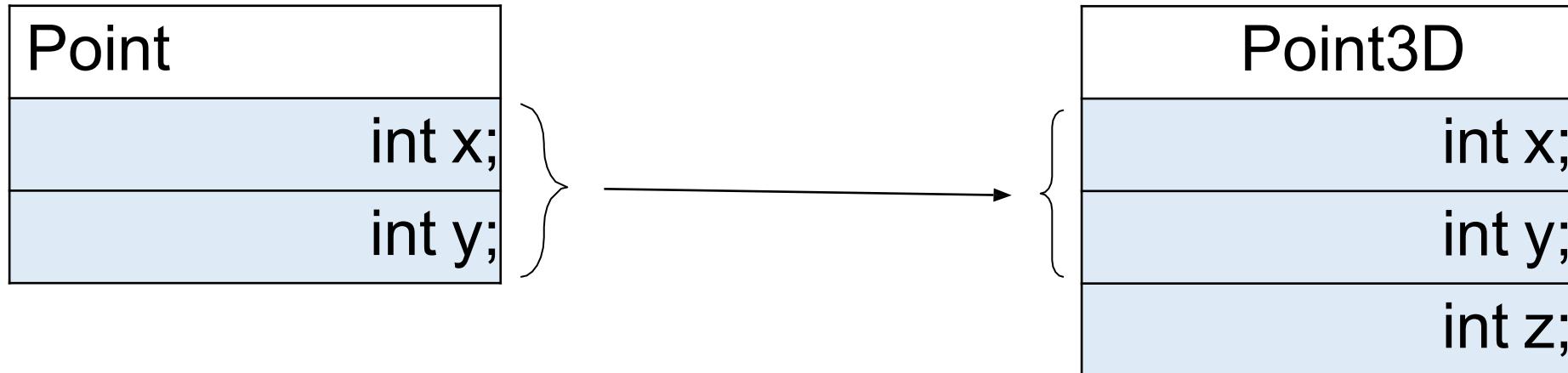
```
class Point3D
{
public:
    int x;
    int y;
    int z;
};
```



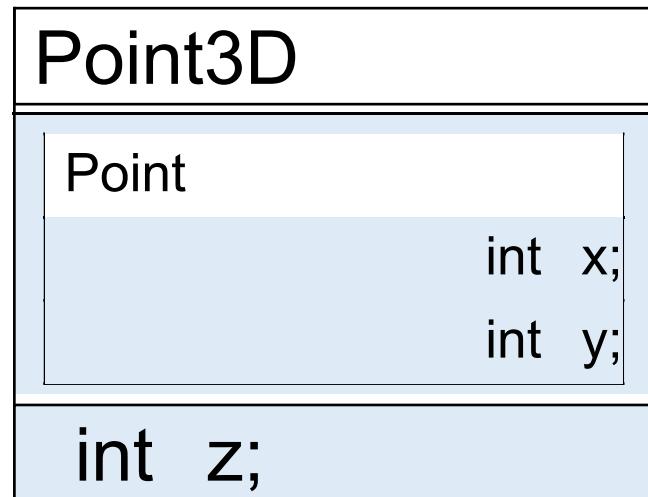
Inheritance



Inheritance



We need something like this:



Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```

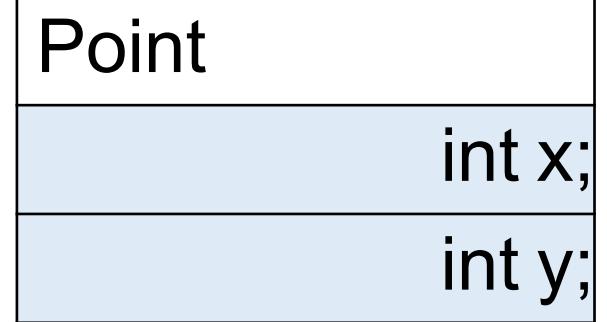
Point
int x;
int y;

Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```

Notice the colon (:)

```
class Point3D :
{
```



Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```

Not 'class Point'

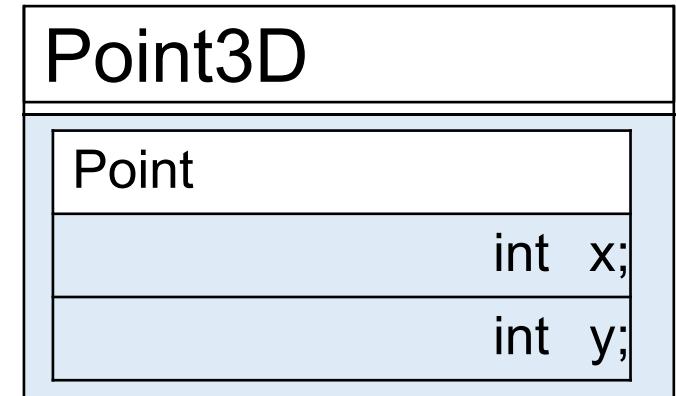
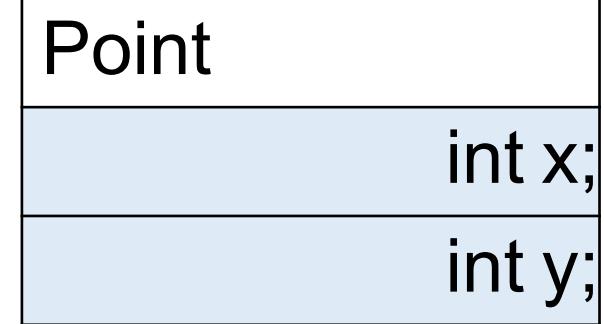
```
class Point3D : public Point
{
```

Access Specifier for inheritance.

Public + Public = Public

Public + Private = private

Private + Public = private



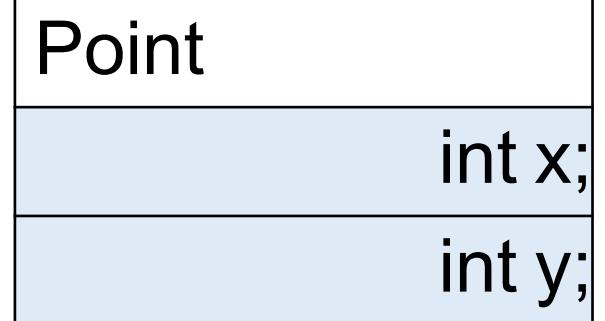
Visibility of inherited members

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Public	Public	Private	Protected
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected

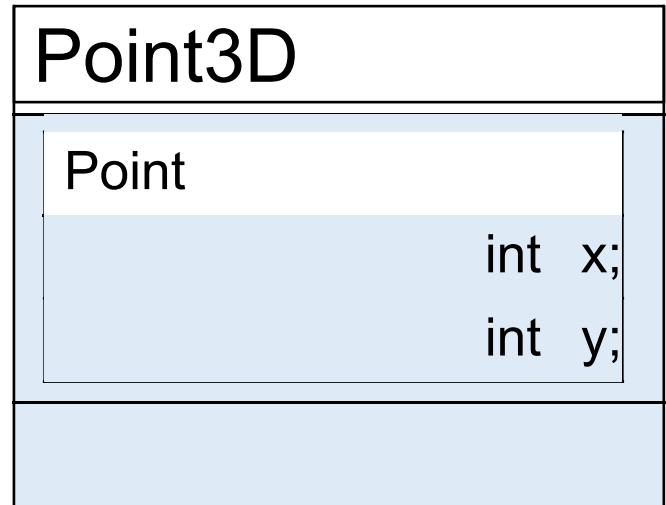
Not Inherited = Can't be accessed directly from the derived class.

Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```

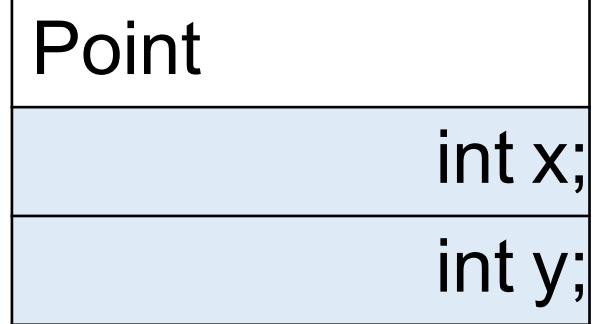


```
class Point3D : public Point
{
public:
};
```



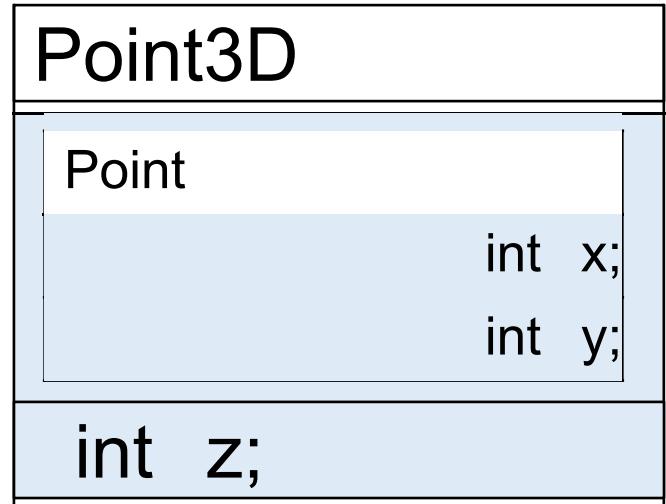
Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```



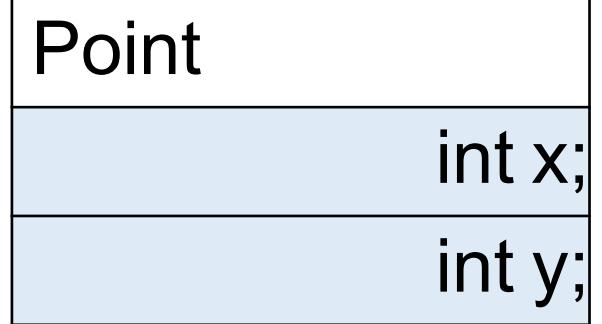
```
class Point3D : public Point
{
public:
    int z;
};
```

Class designer's view

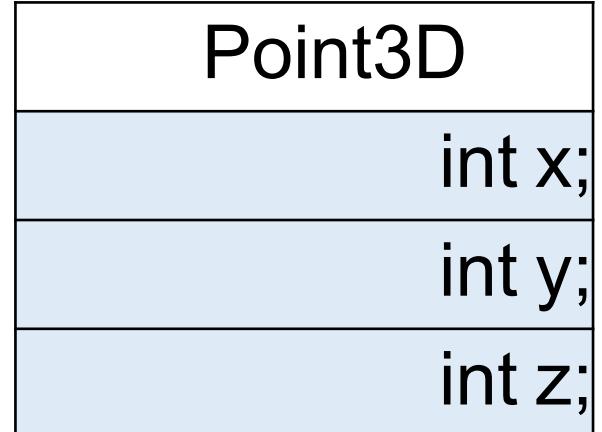


Inheritance Syntax

```
class Point
{
public:
    int x;
    int y;
};
```



```
class Point3D : public Point
{
public:
    int z;
};
```



Developer's view

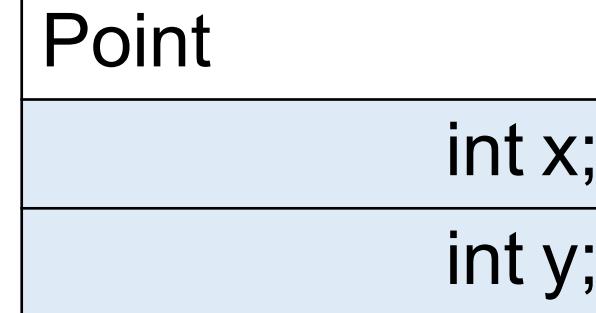
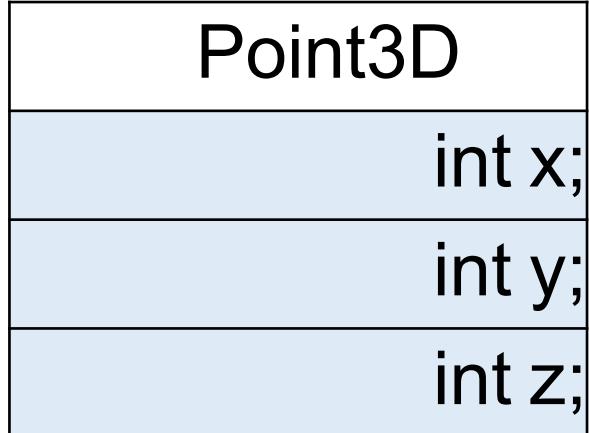


Terminology

Derived class

Base class

```
class Point3D : public Point
```



A More Useful Example

An undergraduate student in MIST receives marks in Theory and Sessional individually.

Design a Student class that will hold the respective marks. Write appropriate getter and setter functions (except constructor).

Student Class

Student

```
int theory;  
int sessional;
```

```
void setMarks(...);  
int getTheory();  
int getSessional();
```

Extending Student Class

Student

```
int theory;  
int sessional;
```

```
void setMarks(...);  
int getTheory();  
int getSessional();
```

L4Student

```
...  
int thesis;
```

```
...  
void setThesis(...);  
int getThesis();
```

Extending Student Class

```
class L4Student
```

Student

```
int theory;  
int sessional;
```

```
void setMarks();  
int getTheory();  
int getSessional();
```

L4Student

Extending Student Class

```
class L4Student : public Student
```

Student

```
int theory;  
int sessional;
```

```
void setMarks();  
int getTheory();  
int getSessional();
```

L4Student

Student

```
int theory; int sessional;
```

```
void setMarks();  
int getTheory();  
int getSessional();
```

Extending Student Class

```
class L4Student : public Student
```

Student

```
int theory;  
int sessional;
```

```
void setMarks();  
int getTheory();  
int getSessional();
```

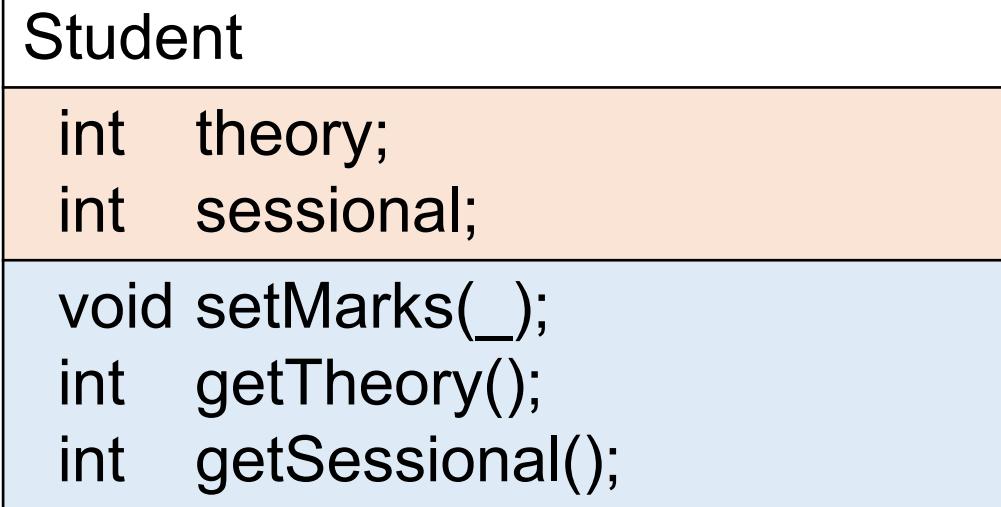
L4Student

```
Student
```

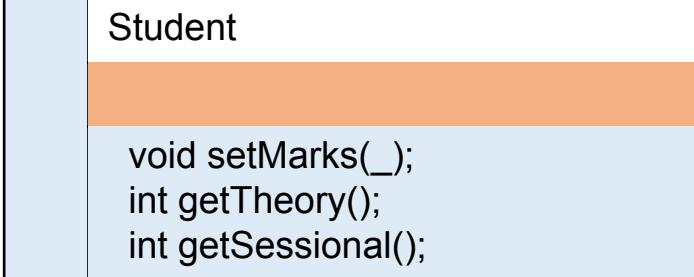
```
void setMarks();  
int getTheory();  
int getSessional();
```

Extending Student Class

```
class L4Student : public Student
{
private:
```



L4Student



Extending Student Class

```
class L4Student : public Student
{
private:
    int thesis;
```

Student

```
int theory;
int sessional;
```

```
void setMarks();
int getTheory();
int getSessional();
```

L4Student

```
int thesis;
```

Student

```
void setMarks();
int getTheory();
int getSessional();
```

Extending Student Class

```
class L4Student : public Student
{
private:
    int thesis;
public:
    void setThesis(int t)
    {
        thesis = t;
    }
}
```

Student

```
int theory;
int sessional;
```

```
void setMarks();
int getTheory();
int getSessional();
```

L4Student

```
int thesis;
```

Student

```
void setMarks();
int getTheory();
int getSessional();
```

```
void setThesis()
```

Extending Student Class

```
class L4Student : public Student
{
private:
    int thesis;
public:
    void setThesis(int t)
    {
        thesis = t;
    }
    int getThesis()
    {
        return thesis;
    }
};
```

Student

```
int theory;
int sessional;
```

```
void setMarks();
int getTheory();
int getSessional();
```

L4Student

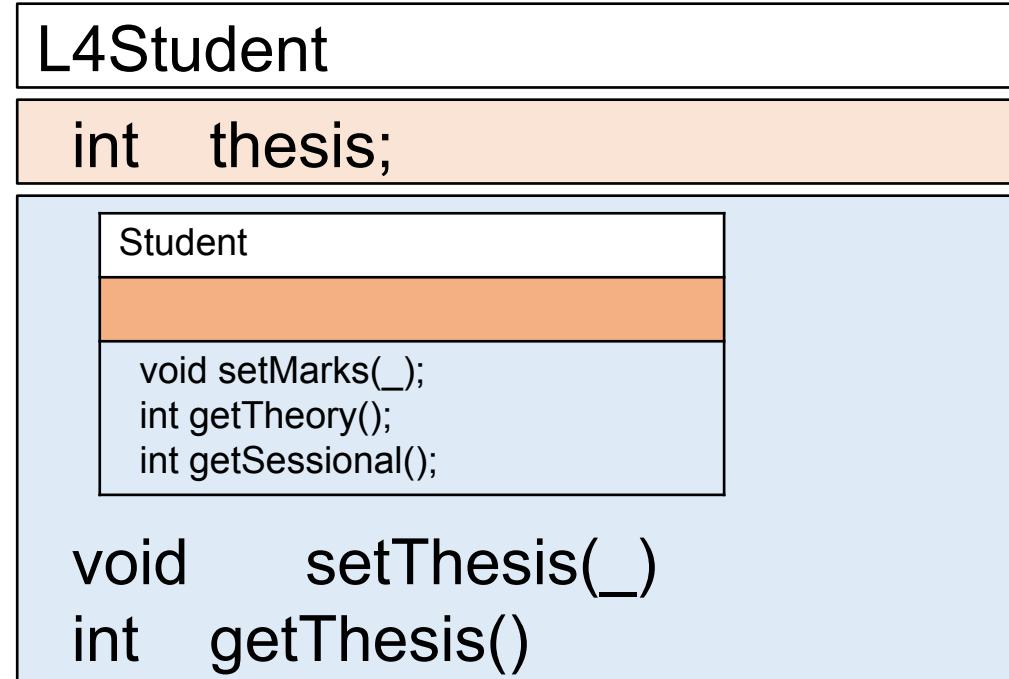
```
int thesis;
```

Student

```
void setMarks();
int getTheory();
int getSessional();
```

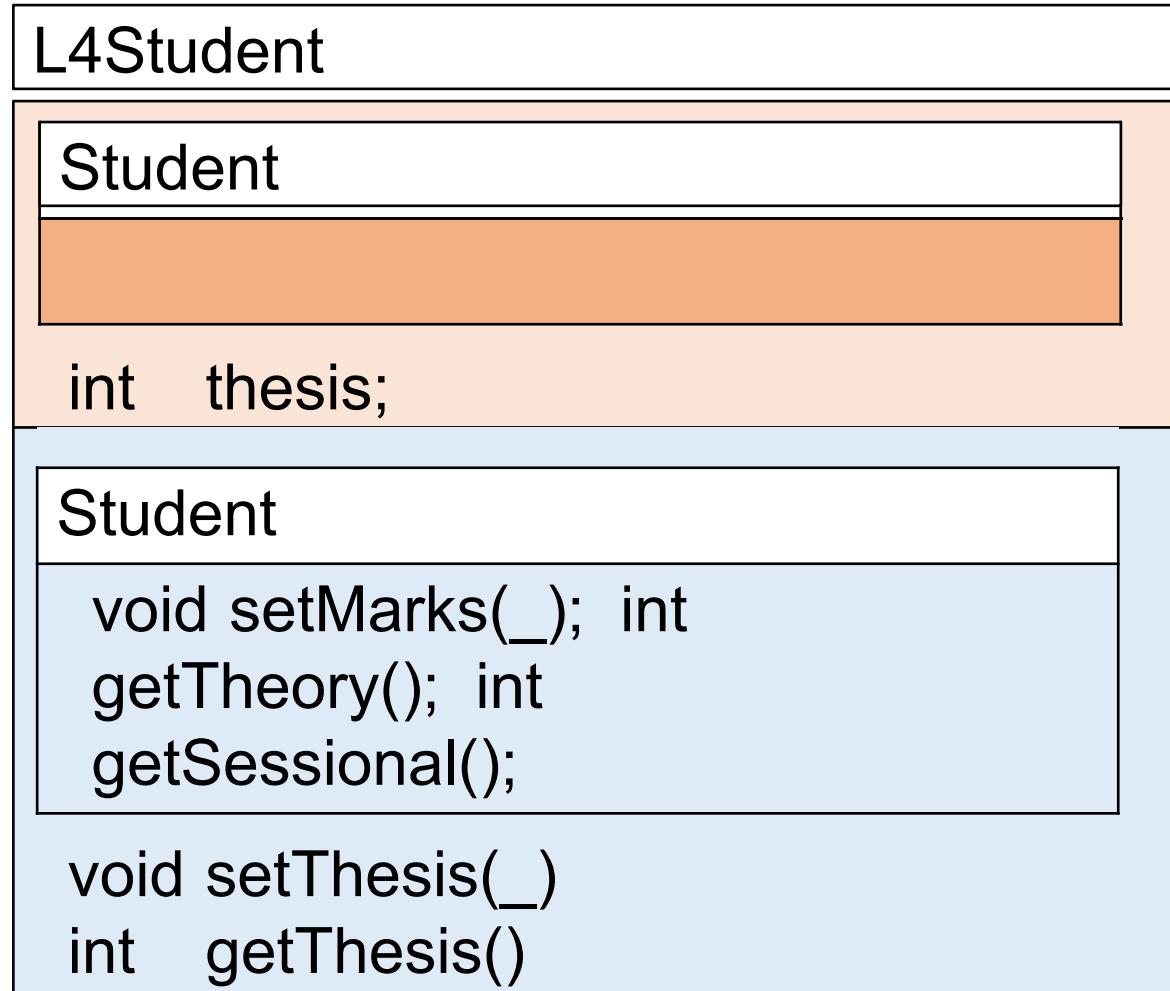
```
void setThesis_()
int getThesis()
```

L4Student Class



Class designer's view

L4Student Class



*Class designer's view
(Expanded)*

L4Student Class

L4Student

```
void      setMarks(_);
int       getTheory();
int       getSessional();
void      setThesis(_);
int       getThesis();
```

Developer's view

Notice that

L4Student is a Student

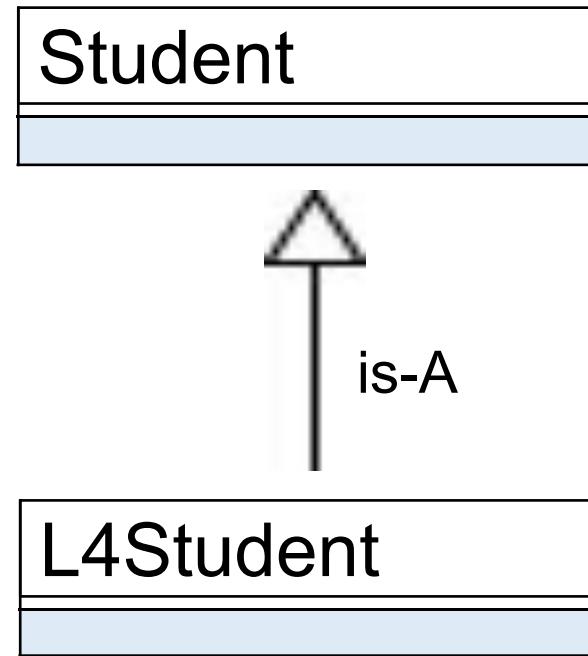
Point3D is a Point

Triangle is a Shape

SalaryAccount is an Account

Car is a Vehicle

is-A Notation (Applicable for public inheritance)

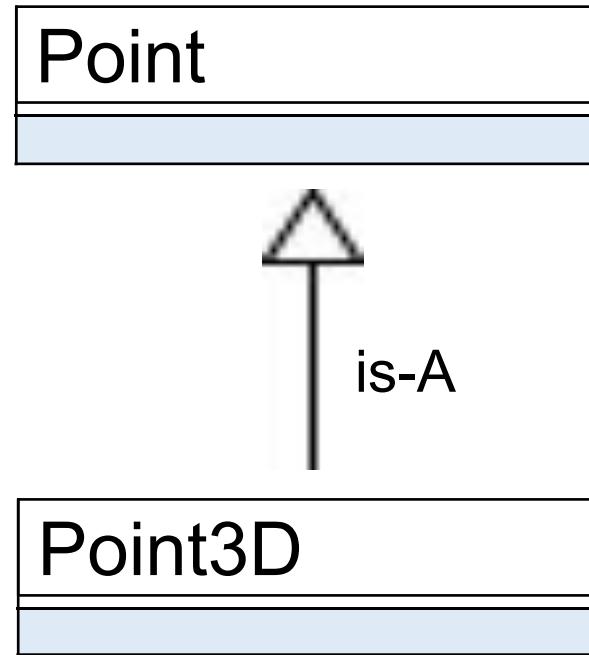


L4Student is-A **Student**

Derived-Class is-A **Base-Class**

The direction of the arrow may look counter-intuitive, but it's accurate.

is-A Notation (Applicable for public inheritance)



Point3D is-A **Point**

Derived-Class is-A **Base-Class**

The direction of the arrow may look counter-intuitive, but it's accurate.

Let's Check out some inheritance examples in c++ codes.

Codes: [click here](#)

Private Inheritance

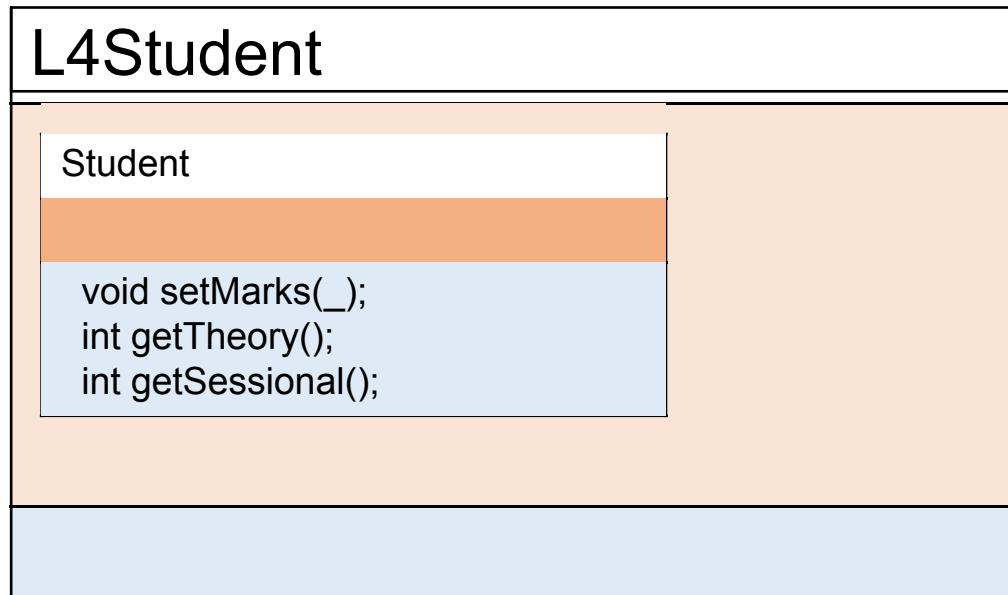
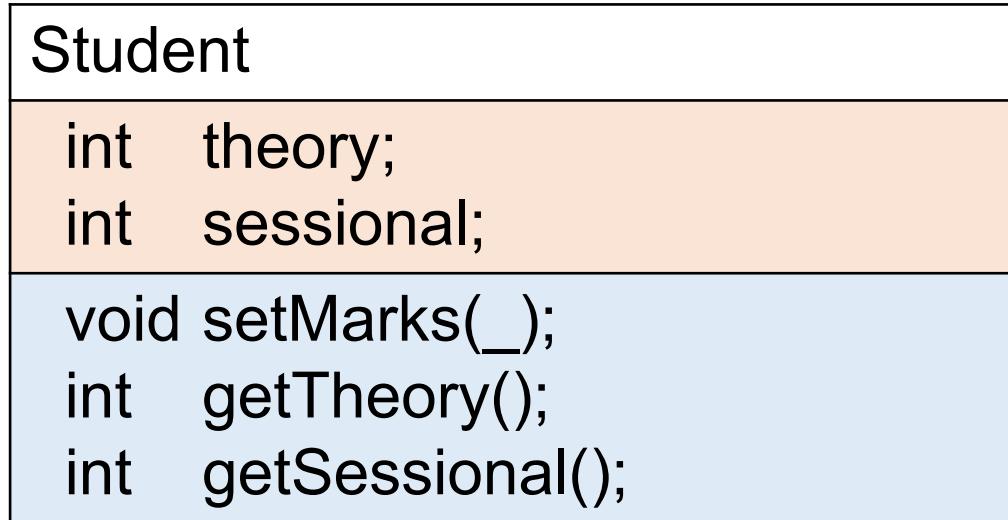
```
class L4Student : private Student
```

Student
int theory;
int sessional;
void setMarks();
int getTheory();
int getSessional();

L4Student

Private Inheritance

```
class L4Student : private Student
```



Private Inheritance

```
class L4Student : private Student
```

Student
int theory; int sessional;
void setMarks(); int getTheory(); int getSessional();

L4Student
Student
Student
void setMarks(); int getTheory(); int getSessional();

Access Specifier Summary

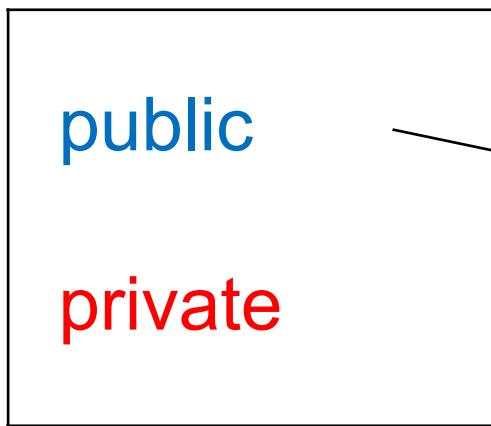
public Inheritance



Access Specifier Summary

private Inheritance

Base Class



Derived Class



A better usage of Private Inheritance

```
class result : private Student
```

Student

```
int theory;  
int sessional;
```

```
void setMarks();  
int getTheory();  
int getSessional();
```

Result

Student

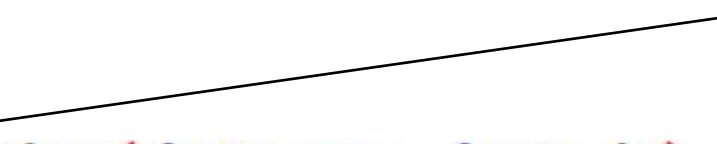
Student

```
void setMarks();  
int getTheory();  
int getSessional();
```

```
int entryMarks()  
int getResult();
```

A better usage of Private Inheritance

```
class Result : private Student
{
public:
    int entryMarks(int a, int b)
    {
        setMarks(a, b);
    }
    int publishResult()
    {
        int total = getTheory() + getSessional();
        return total;
    }
};
```



This is called Delegate Function, *Because it ‘delegates’ Its duty to another Function.*

Note

Result is not a Student! So it's not a is-a relation.
It's a has-a relation. Result has-a Student.

More Examples of has-a

Car has-an Engine

Body has-an Organ

School has-a Classroom

Course has-a ClassTest

A Third Type of Access Modifier



private

Your toothbrush
(Only you can use it)

protected



public

Your toothpaste
(Anyone can use it)

A Third Type of Access Modifier



private

Your toothbrush
(Only you can use it)



protected

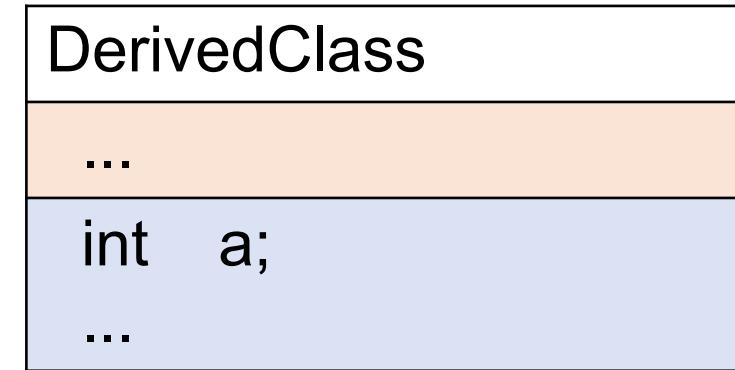
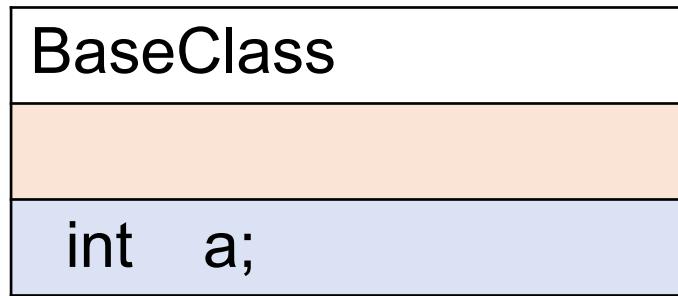
Your home (your relatives can use it too)



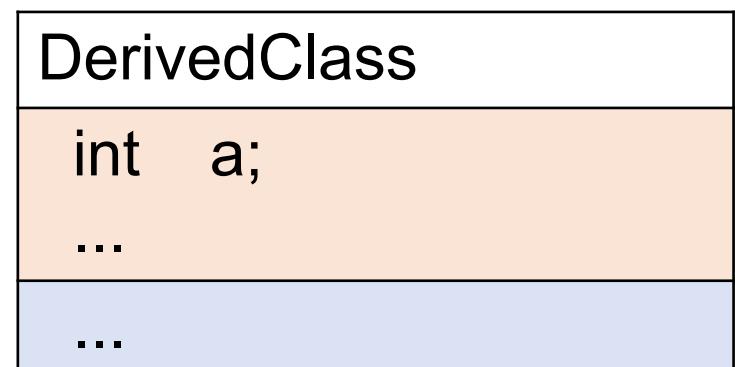
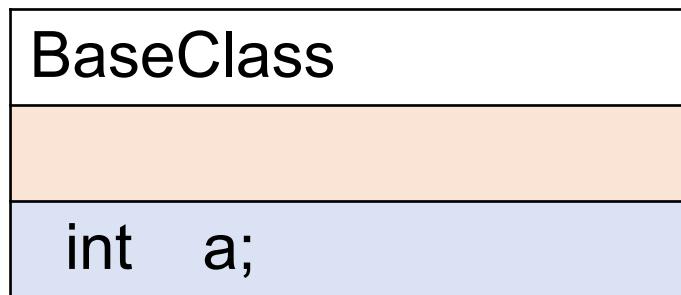
public

Your toothpaste
(Anyone can use it)

Public Inheritance

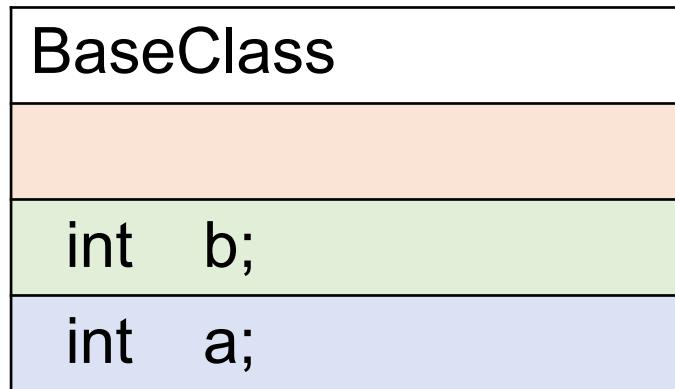


Private Inheritance



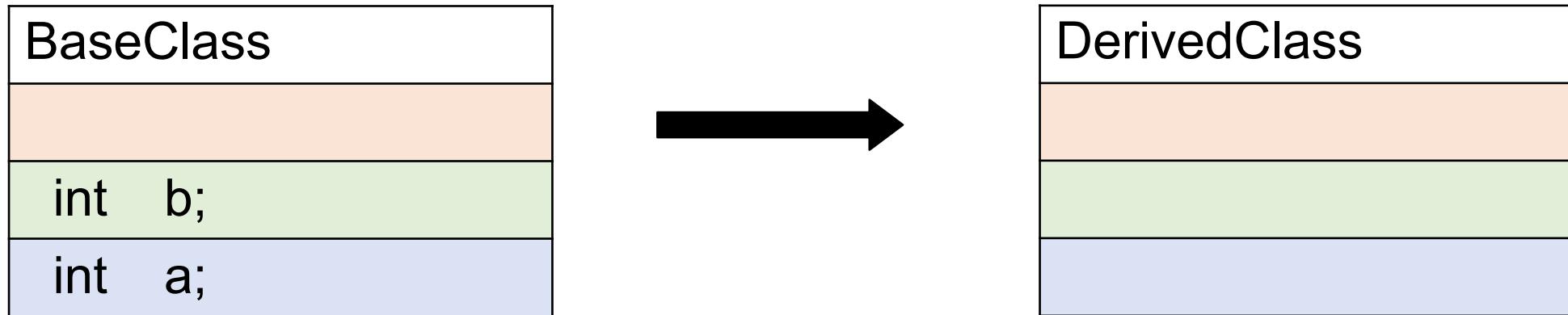
We need something that is private in both Base Class and Derived Class

Protected Member



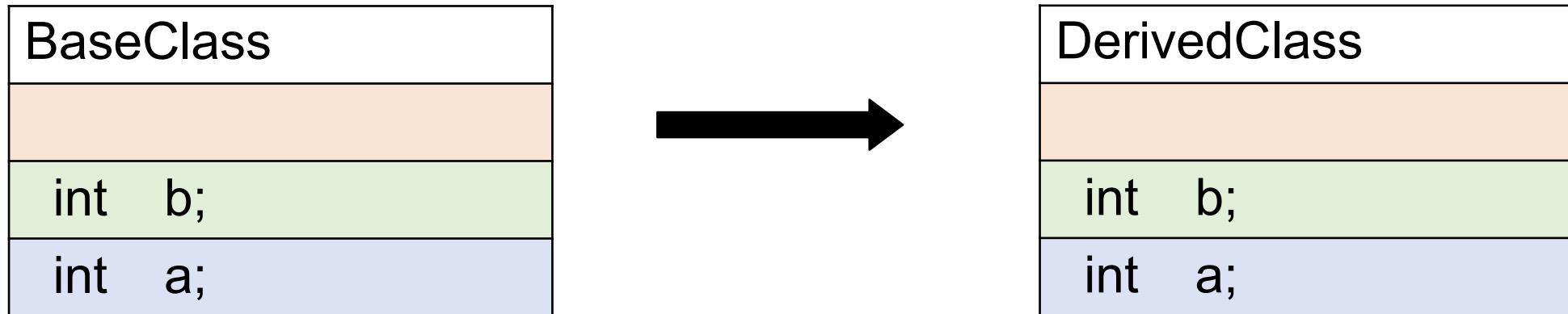
‘b’ is inaccessible from outside of **BaseClass** (just like a private member)
But it’s accessible in its derived classes

Public Inheritance



```
class DerivedClass : public BaseClass
{
}
```

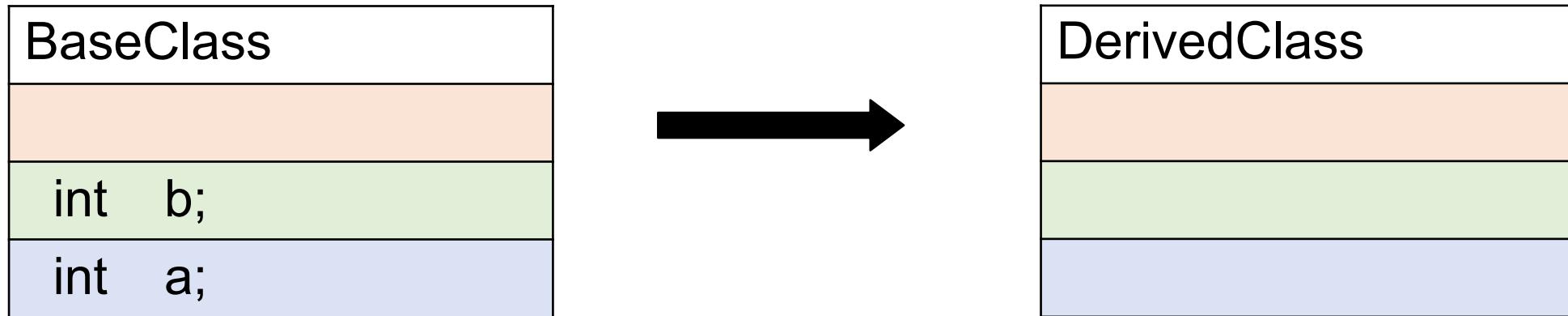
Public Inheritance



```
class    DerivedClass : public BaseClass
{
}
```

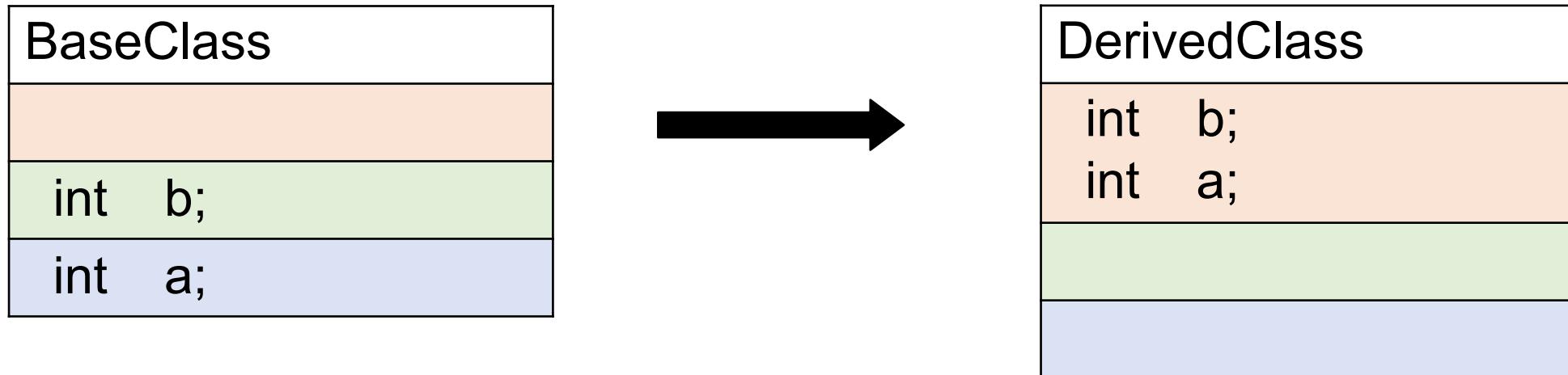
'b' is inaccessible from outside of **DerivedClass** (just like a private member)
But it's accessible in its derived classes!

Private Inheritance



```
class DerivedClass : private BaseClass
{  
}
```

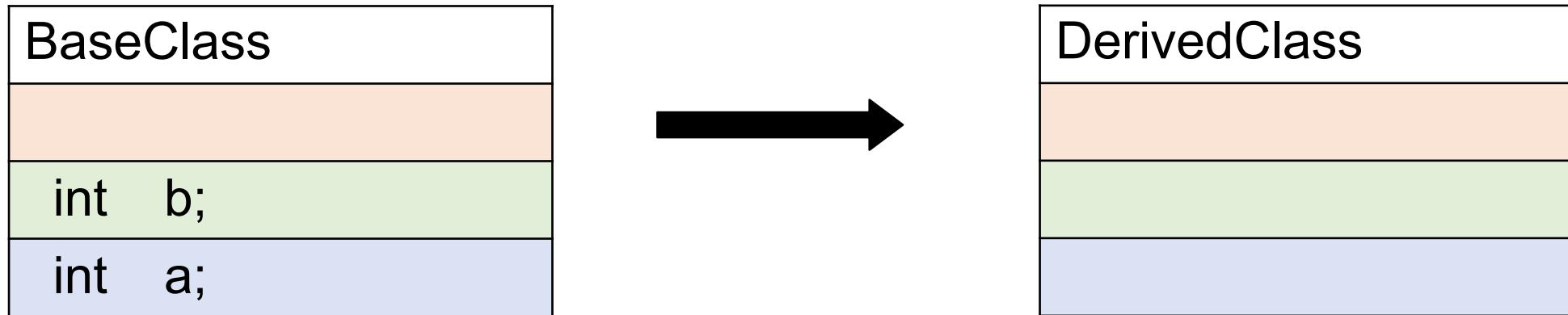
Private Inheritance



```
class     DerivedClass :  private BaseClass
{
}
```

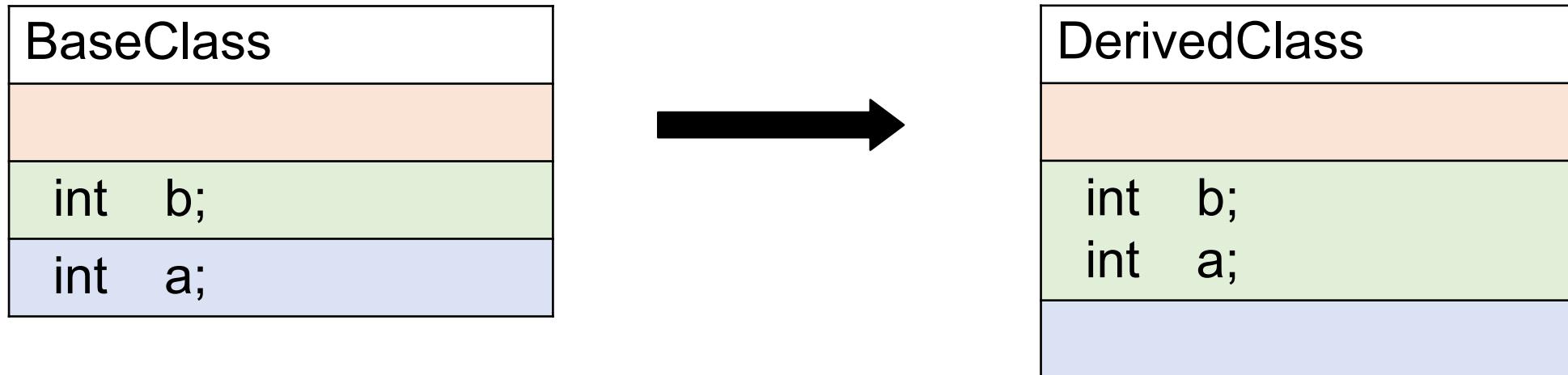
Both 'b' and 'a' becomes private. **Neither** will be accessible from its derived class or outside class anymore.

Protected Inheritance



```
class DerivedClass : protected BaseClass
{  
}
```

Protected Inheritance



```
class    DerivedClass :  protected  BaseClass  
{  
}
```

Both 'b' and 'a' becomes protected. **Neither** will be accessible from outside this `DerivedClass`, **except** its derived classes.

Access Specifier Summary (Final)

Inheritance	Base Class	Derived Class
<i>public</i>	private: protected: public:	private: protected: public:
<i>protected</i>	private: protected: public:	private: protected: public:
<i>private</i>	private: protected: public:	private: protected: public:

The diagram illustrates the inheritance of access specifiers from a base class to a derived class. It uses three columns: Inheritance, Base Class, and Derived Class. The Inheritance column lists three levels: public, protected, and private. The Base Class column lists the same three levels with specific access specifiers: private, protected, and public. The Derived Class column also lists these three levels. Solid arrows point from the Inheritance level to the corresponding access specifier in the Base Class column. Dashed arrows point from the Inheritance level to the corresponding access specifier in the Derived Class column. Blue boxes highlight the final access specifiers in the Derived Class column for each row.

Source: <https://flylib.com/books/en/2.937.1.175/1/>

Modifying Behavior of Derived class

- Modification of functions
- Modification of variables
- Modification of visibility

Modifying Function Behavior (Overriding)

```
class Point
{
protected:
    int x;
    int y;
public:
    void setPoint(int _x, int _y)
    {
        x = _x;
        y = _y;
    }

    void show()
    {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

Point p1;
p1.setPoint(3, 4);
p1.show();           //(3, 4)
```

Modifying Function Behavior (Overriding)

```
class Point3D : public Point
{
    int z;
public:
    void setZ(int a, int b, int c)
    {
        setPoint(a, b);
        z = c;
    }
};

Point3D p2;
p2.setZ(5, 6, 7);
p2.show();           //(5, 6)
```

Modifying Function Behavior (Overriding)

```
class Point3D : public Point
{
    int z;
public:
    void setZ(int a, int b, int c)
    {
        setPoint(a, b);
        z = c;
    }
    void show()
    {
        cout << "(" << x << ", " << y << ", " << z << ")" << endl;
    }
};

Point3D p2;
p2.setZ(5, 6, 7);
p2.show();           //(5, 6, 7)
```

Modifying Function Behavior (Overriding)

When a function is called, it tries to invoke the nearest definition.

Rule: The prototype of the overriding function must be exactly the same as the original function in Base class. i.e.:

- Same function name
- Same parameter list
- Same return type

Only the function body can be different.

Modifying Function Behavior (Overriding)

Let's Override the `getTotalMarks()` in Derived class `ArchStudent` in our example.

Modifying Variable Behavior

This is not encouraged, but can be done.

```
class Point
{
protected:
    int x;
    int y;
    int resourceId = 50;
public:
    void setPoint(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void show()
    {
        cout << resourceId << " (" << x << ", " << y << ")" << endl;
    }
};
```

Modifying Variable Behavior

```
class Point3D : public Point
{
    int z;
protected:
public:
    void setZ(int a, int b, int c)
    {
        setPoint(a, b);
        z = c;
    }
    void show()
    {
        cout << resourceID << " (" << x << ", " << y << ", " << \
z << ")" << endl;
    }
};

Point p1;
p1.setPoint(3, 4);
p1.show();           //50 (3, 4)
Point3D p2;
p2.setZ(5, 6, 7);
p2.show();           //50 (5, 6, 7)
```

Modifying Variable Behavior

```
class Point3D : public Point
{
    int z;
protected: int resourceId = 60;
public:
void setZ(int a, int b, int c)
{
    setPoint(a, b);
    z = c;
}
void show()
{
    cout << resourceId << " (" << x << ", " << y << ", " << \
z << ")" << endl;
}
};

Point p1;
p1.setPoint(3, 4);
p1.show();           //50 (3, 4)
Point3D p2;
p2.setZ(5, 6, 7);
p2.show();           //60 (5, 6, 7)
```

Accessing Members of Base Class

```
class Point3D : public Point
{
    int z;
protected: int resourceId = 60;
public:
    void setZ(int a, int b, int c)
    {
        setPoint(a, b);
        z = c;
    }
    void show()
    {
        cout << Point::resourceID << " (" << x << ", " << y << ", " << \
        z << ")" << endl;
    }
};

Point p1;
p1.setPoint(3, 4);
p1.show();           //50 (3, 4)
Point3D p2;
p2.setZ(5, 6, 7);
p2.show();           //50 (5, 6, 7)
```

Modifying Visibility

```
class Base
{
protected:
    void hiddenMethod()
    {
        cout << "This should not be publicly exposed!";
    }
public:
    void publicMethod()
    {
        cout << "This should be called from outside!";
    }
};

class Derived : public Base
{
protected:
    Base::publicMethod;
public:
    Base::hiddenMethod;
};

Base b;                                //Error!
b.hiddenMethod();                         //OK
b.publicMethod();                         //OK

Derived d;                               //OK
d.hiddenMethod();                         //Error!
d.publicMethod();                        //Error!
```

For Further Reading:

1. Intro to Inheritance: https://www.w3schools.com/cpp/cpp_inheritance.asp
2. A Deeper look into Inheritance: <https://www.learnCPP.com/cpp-tutorial/introduction-to-inheritance/>
3. Delegate Functions in C++: <http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2002/n1363.htm>

References/Courtesy:

Lec Saidul Hoque Anik

Lec Anika Binte Islam

Thank you

More on Inheritance in C++

CSE 205 - Week 9

[Lec Raiyan Rahman](#)

Dept of CSE, MIST

raian@cse.mist.ac.bd

CC: Lec Saidul Hoque Anik

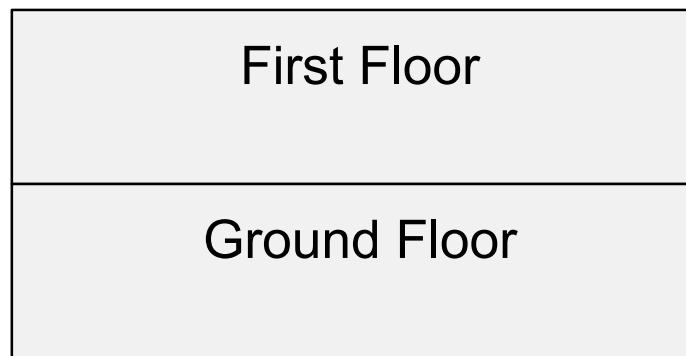


How derived class is constructed

How derived class is constructed

Ground Floor

How derived class is constructed



How derived class is constructed



How derived class is constructed



How derived class is constructed

```
class GroundFloor
{
public:
    GroundFloor()
    {
        cout << "Ground Floor is created" << endl;
    }
};

class FirstFloor: public GroundFloor
{
public:
    FirstFloor()
    {
        cout << "First Floor is created" << endl;
    }
};

int main()
{
    FirstFloor f;
```

What will happen in this case?

```
class GroundFloor
{
public:
};

class FirstFloor: public GroundFloor
{
public:
    FirstFloor()
    {
        cout << "First Floor is created" << endl;
    }
};

int main()
{
    FirstFloor f;
}
```

What will happen in this case?

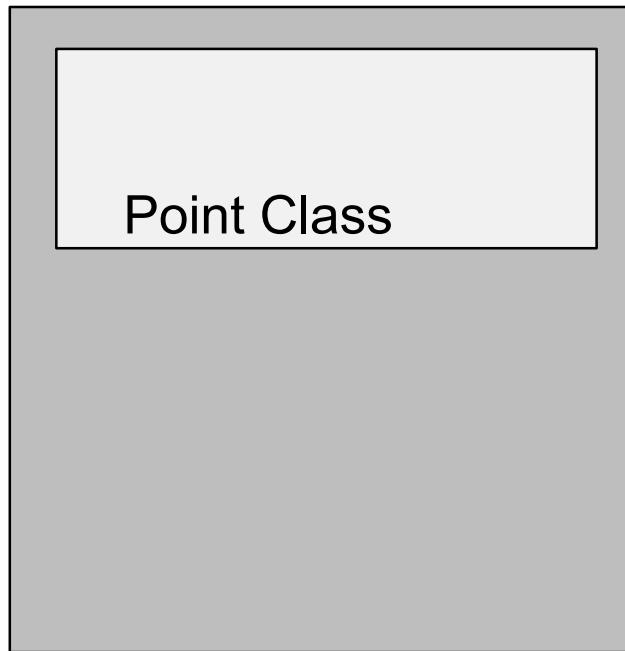
```
class GroundFloor
{
public:
    GroundFloor()
    {
        cout << "Ground Floor is created" << endl;
    }
};

class FirstFloor: public GroundFloor
{
public:
    ...
};

int main()
{
    FirstFloor f;
}
```

How derived class is constructed

Which constructor will be called first?

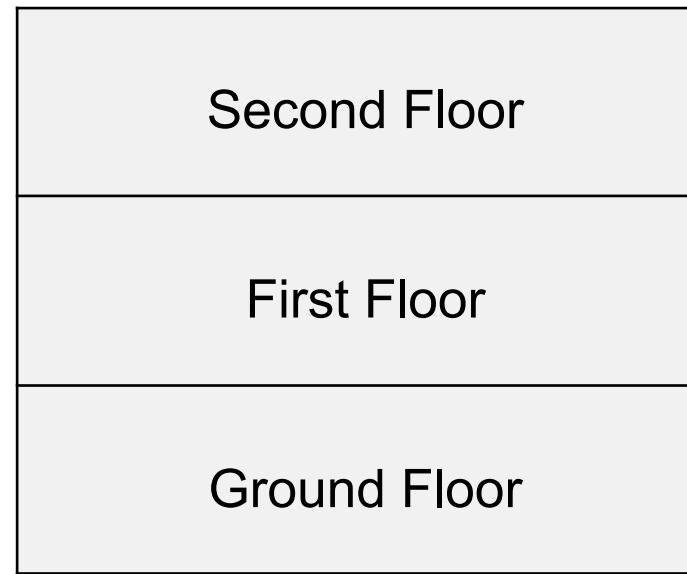


Point3D Class

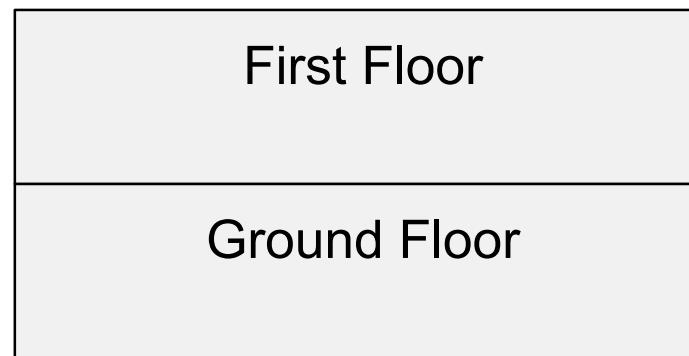
Destructor Call Sequence



Destructor Call Sequence



Destructor Call Sequence



Destructor Call Sequence

Ground Floor

Destructor Call sequence

```
class GroundFloor
{
public:
    GroundFloor()
    {
        cout << "Ground Floor is created" << endl;
    }
    ~GroundFloor()
    {
        cout << "Ground Floor is destroyed" << endl;
    }
};

class FirstFloor: public GroundFloor
{
public:
    FirstFloor()
    {
        cout << "First Floor is created" << endl;
    }
    ~FirstFloor()
    {
        cout << "First Floor is destroyed" << endl;
    }
};
```

```
int main()
{
    FirstFloor f;
```

Parameterized Constructor

```
class FirstFloor: public GroundFloor
{
public:
FirstFloor()
{
    cout << "First Floor is created" << endl;
}
FirstFloor(int r)
{
    cout << "First Floor is created with " << r << " rooms" << endl;
}
~FirstFloor()
{
    cout << "First Floor is destroyed" << endl;
}
};
```

How do we pass the parameter from FirstFloor?

```
class GroundFloor
{
public:
    GroundFloor()
    {
        cout << "Ground Floor is created" << endl;
    }
    GroundFloor(int r)
    {
        cout << "Ground Floor is created with " << r << " rooms" << endl;
    }
    ~GroundFloor()
    {
        cout << "Ground Floor is destroyed" << endl;
    }
};
```

How do we pass the parameter from FirstFloor?

```
class FirstFloor: public GroundFloor
{
public:
    FirstFloor()
    {
        cout << "First Floor is created" << endl;
    }
    FirstFloor(int r, int g) : GroundFloor(g)
    {
        cout << "First Floor is created with " << r << " rooms" << endl;
    }
    ~FirstFloor()
    {
        cout << "First Floor is destroyed" << endl;
    }
};

int main()
{
    FirstFloor f(3, 1);
}
```

Notice the syntax



Task

Write the Point and Point3D class with constructor

Task

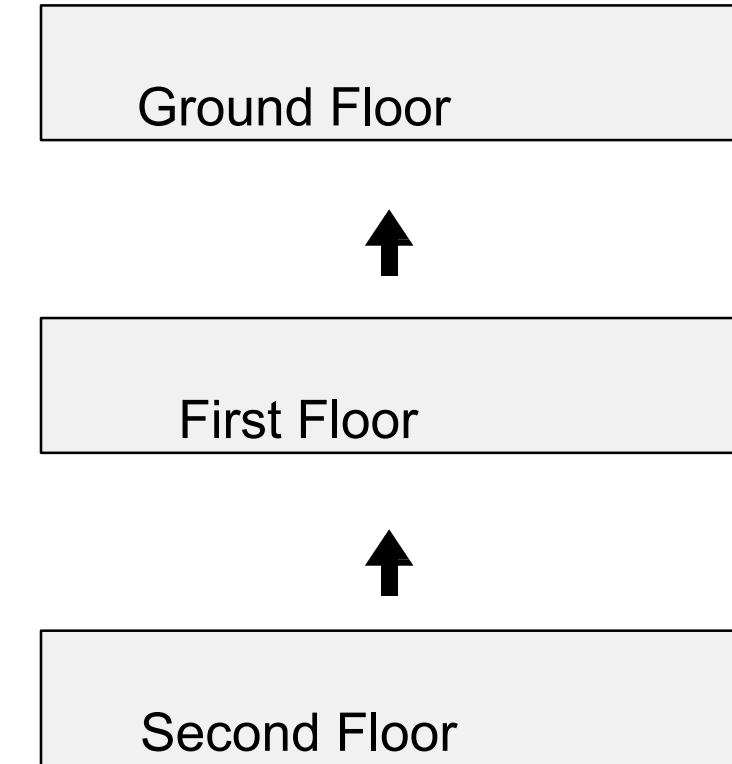
Write the Point and Point3D class with constructor

```
class Point
{
protected:
    int x;
    int y;
public:
    Point(int a, int b)
    {
        x = a;
        y = b;
    }
    void display()
    {
        cout << x << " " << y << endl;
    }
};

class Point3D : public Point
{
    int z;
public:
    Point3D(int a, int b, int c) : Point(a, b)
    {
        z = c;
    }
    void display()
    {
        cout << x << " " << y << " " << z << endl;
    }
};
```

Multilevel Inheritance

What will be the constructor of SecondFloor?



Multilevel Inheritance

What will be the constructor of SecondFloor?

```
class SecondFloor : public FirstFloor
{
public:
    SecondFloor(int a, int b, int c) : FirstFloor(b, c)
    {
        cout << "Second Floor is created with " << a << " rooms" << endl;
    }
};
```

Task

Write down a Point4D class with constructor

Task

Write down a Point4D class with constructor

```
class Point4D : public Point3D
{
    int w;
public:
    Point4D(int a, int b, int c, int d) : Point3D(a, b, c)
    {
        w = d;
    }
    void display()
    {
        cout << x << " " << y << " " << z << " " << w << endl;
    }
};
```

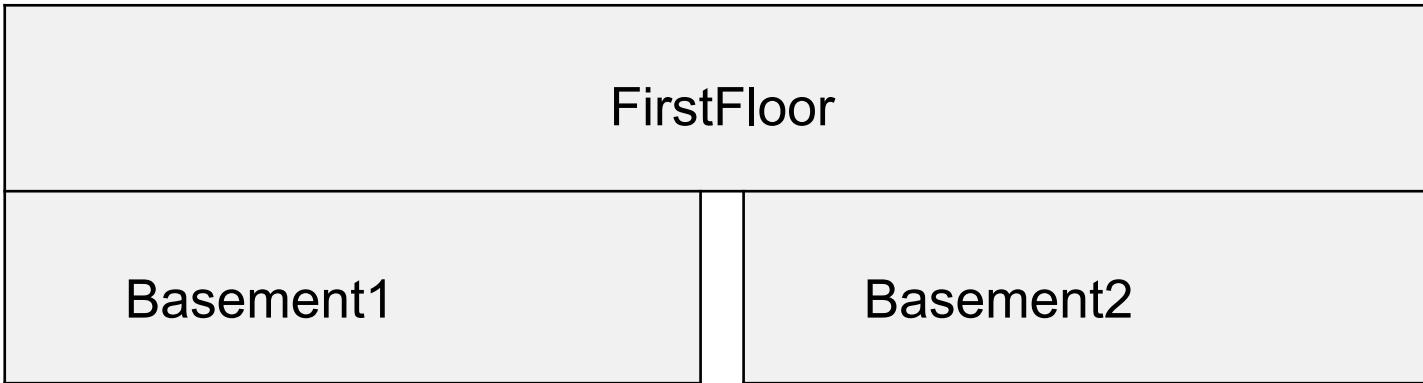
Hierarchical Inheritance/Multiple Inheritance

Derived Class with multiple base classes



Hierarchical Inheritance

What will be the order of constructor?



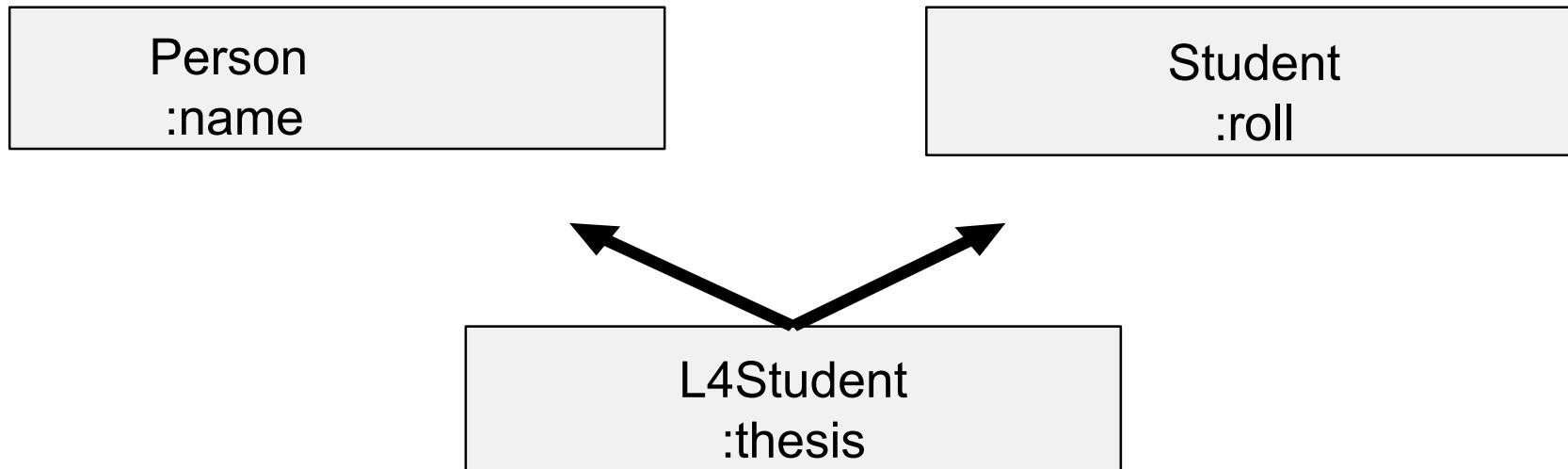
Hierarchical Inheritance

What will be the order of destructor?



Hierarchical Inheritance

Derived Class with multiple base classes



Hierarchical Inheritance

Derived Class with multiple base classes

```
class Person
{
    char name[100];
public:
    Person(char * n)
    {
        strcpy(name, n);
    }
    void printName()
    {
        cout << name << endl;
    }
};
```

```
class Student
{
    int roll;
public:
    Student(int r)
    {
        roll = r;
    }
    void printRoll()
    {
        cout << roll << endl;
    }
};
```

Hierarchical Inheritance

Derived Class with multiple base classes

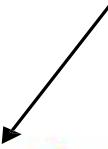
```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first



Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first

*This will be called
after first*

Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first

This will be called after first

This will be called at last

Hierarchical Inheritance

What will be the order of destructor?

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

Thank you

Runtime Polymorphism, Early and Late Binding

CSE 205 - Week 10 Class 1

[Lec Raiyan Rahman](#)

Dept of CSE, MIST

raian@cse.mist.ac.bd

CC: Lec Saidul Hoque Anik



Polymorphism

Same name, different functionality

Polymorphism

Same name, different functionality

Function Overloading

- i) Multiple function with same name
- ii) Different number of parameters, or
- iii) Different argument data-type

Polymorphism

Same name, different functionality

Function Overloading

```
void findMax(int a, int b)
{
    if (a >= b)
        cout << a << " is the max value" << endl;
    else
        cout << b << " is the max value" << endl;
}

void findMax(double a, double b)
{
    if (a >= b)
        cout << a << " is the max value" << endl;
    else
        cout << b << " is the max value" << endl;
}

findMax(10, 20);           //20
findMax(22.5, 10.3);      //22.5
```

Function Overriding

- i) Function with same name in derived class
- ii) Exactly same number of parameters, and
- iii) Same argument data-type

Function Overriding

```
class Point
{
public:
    int x;
    int y;
    void display()
    {
        cout << x << ", " << y << endl;
    }
};

class Point3D : public Point
{
public:
    int z;
    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }
};
```

```
int main()
{
    Point p1;
    p1.x = 10;
    p1.y = 20;
    p1.display();

    Point3D p2;
    p2.x = 10;
    p2.y = 20;
    p2.z = 30;
    p2.display();
}
```

Object Slicing

In C++, when a derived object is copied into an object of base type, the copy will not have the additional methods of derived class.

The modifications are ‘sliced-off’.

```
void printPoint(Point p)
{
    p.display();
}

int main()
{
    Point p1;
    p1.x = 10;
    p1.y = 20;
    printPoint(p1);

    Point3D p2;
    p2.x = 100;
    p2.y = 200;
    p2.z = 300;
    printPoint(p2);
}
```

Object Slicing

p1

```
int x;  
int y;  
void display();
```

p2

```
int x;  
int y;  
void display();  
int z;  
void display();
```

p



Object Slicing

p1

```
int x;  
int y;  
void display();
```



p2

```
int x;  
int y;  
void display();  
int z;  
void display();
```

p

```
int x;  
int y;  
void display();
```

p1 can be copied into p, no problem

Object Slicing

p1

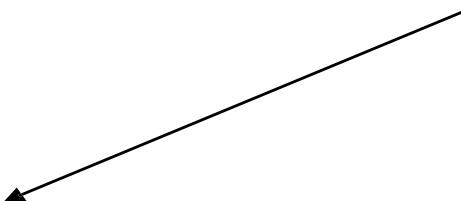
```
int x;  
int y;  
void display();
```

p2

```
int x;  
int y;  
void display();  
int z;  
void display();
```

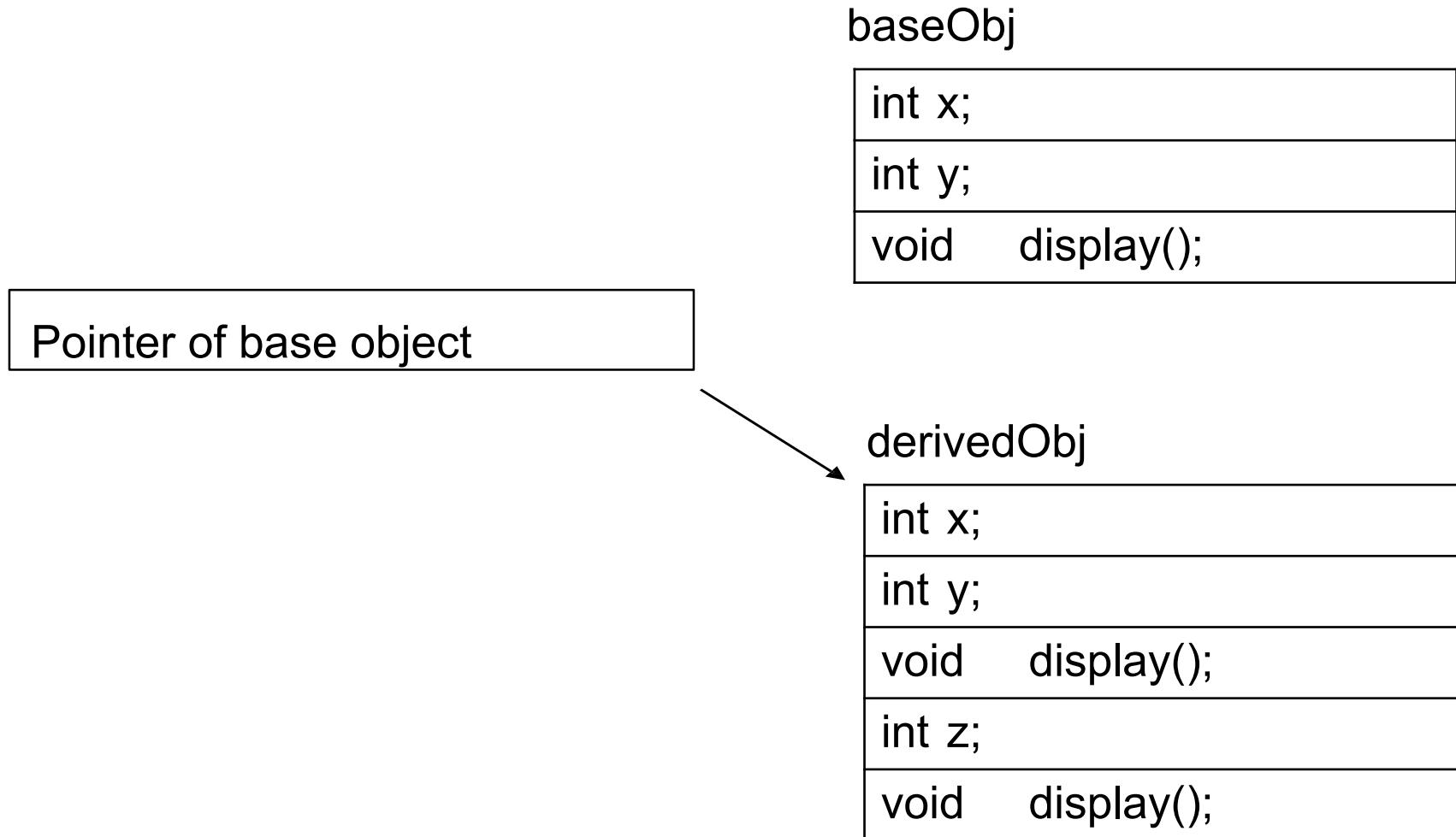
p

```
int x;  
int y;  
void display();  
int z;  
void display();
```



When p2 is copied into p, extra members
are sliced off

What if we use a pointer?



What if we use a pointer?

```
class Base
{
public:
    void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    void display()
    {
        cout << "Hello from Derived class" << endl;
    }
};

void callDisplay(Base * obj)
{
    obj->display();
}

int main()
{
    Base b1;
    Derived d1;
    callDisplay(&b1);
    callDisplay(&d1);
}
```

Early Binding

- Compiler matches the function call with the correct function definition at compile time.
- **All function calls are by default early-bound**
- Also known as compile time binding or static binding

```
findMax(10, 20);          //20
findMax(22.5, 10.3);      //22.5
```

```
void printPoint(Point p)
{
    p.display();
}
```

- Early Bound functions

Early Binding

```
void callDisplay(Base * obj)
{
    obj->display();
}
```

During compile time, the compiler sees that obj is of Base type. So it assigns/binds the base version of display() function in this line.

Early Binding

```
class Base
{
public:
    void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    void display()
    {
        cout << "Hello from Derived class" << endl;
    }
};

void callDisplay(Base * obj)
{
    obj->display();
}

We want the compiler to  
stop early binding
```

```
int main()
{
    Base b1;
    Derived d1;
    callDisplay(&b1);
    callDisplay(&d1);
}
```

Late Binding

We mention 'virtual' before the function that needs to be binded during run-time

```
class Base
{
public:
    virtual void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    void display()
    {
        cout << "Hello from Derived class" << endl;
    }
};

void callDisplay(Base * obj)
{
    obj->display();
}

int main()
{
    Base b1;
    Derived d1;
    callDisplay(&b1);
    callDisplay(&d1);
}
```

Late Binding

```
void callDisplay(Base * obj)
{
    obj->display();
}
```

When the compiler sees that the **display()** function is virtual, it does not bind the line with any function address.

The call is resolved during run-time. At run-time, the type of object is determined (whether it is base class or derived class), and display() function of the corresponding class is called.

This is known as late binding, or **dynamic method dispatch**.

*All virtual functions are late-binded.

Early binding vs. Late Binding

Which function will be early binded?

```
class Base
{
public:
    void f1()
    {
        cout << "f1 is called" << endl;
    }
    virtual void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    void f1()
    {
        cout << "f1 is overrided" << endl;
    }
    void display()
    {
        cout << "Hello from Derived class" << endl;
    }
};
```

```
int main()
{
    Base b1;
    Derived d1;

    Base * ptr;

    ptr = &b1;
    ptr->f1();
    ptr->display();

    ptr = &d1;
    ptr->f1();
    ptr->display();
}
```

Early binding vs. Late Binding

Which function will be early binded? Answer: f1(), because it is not virtual.

```
class Base
{
public:
    void f1()
    {
        cout << "f1 is called" << endl;
    }
    virtual void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    void f1()
    {
        cout << "f1 is overrided" << endl;
    }
    void display()
    {
        cout << "Hello from Derived class" << endl;
    }
};
```

```
int main()
{
    Base b1;
    Derived d1;

    Base * ptr;

    ptr = &b1;
    ptr->f1();
    ptr->display();

    ptr = &d1;
    ptr->f1();
    ptr->display();
}
```

Early binding vs. Late Binding

Which one makes the overall program faster?

Early binding vs. Late Binding

Which one makes the overall program faster?

Early binding. Because the program has already decided which version of function is to be called before run-time.

How Late Binding is implemented

vptr and vtable

vtable is an array that holds the address of all the virtual functions in a class. vptr holds the address of the array inside the class.

Each time a class is derived, vptr is inherited, and new vtable is generated for the derived class.

The process is done during the compilation.

How Late Binding is implemented

vptr and vtable

Base Class

```
f1();  
void display();
```

Derived Class

```
f1();  
void display();
```

How Late Binding is implemented

vptr and vtable

Base Class

```
f1();  
virtual void display();
```

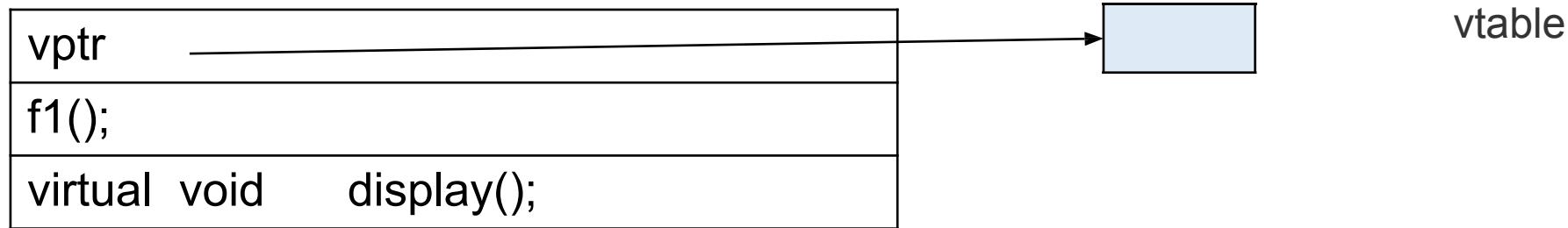
Derived Class

```
f1();  
void display();
```

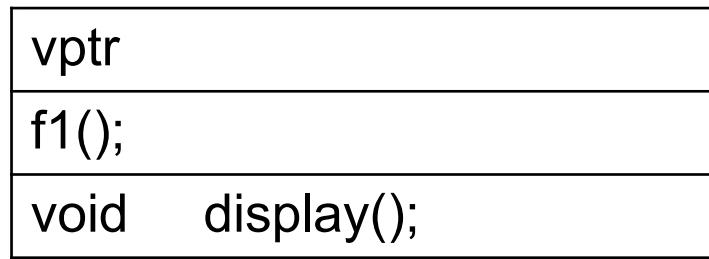
How Late Binding is implemented

vptr and vtable

Base Class



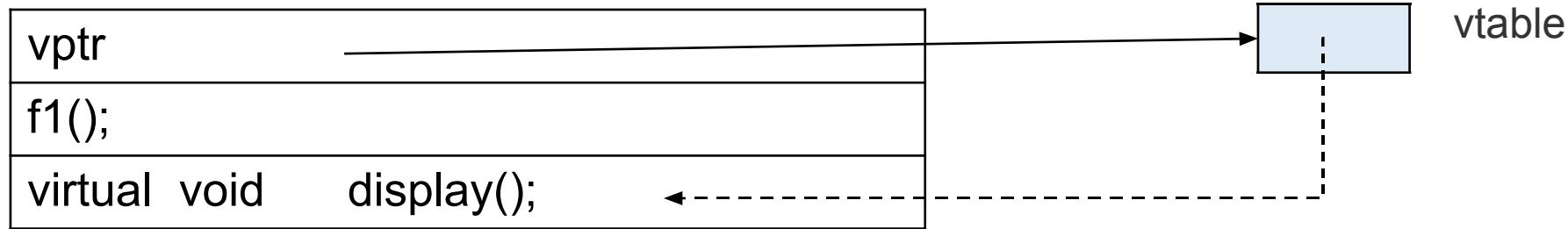
Derived Class



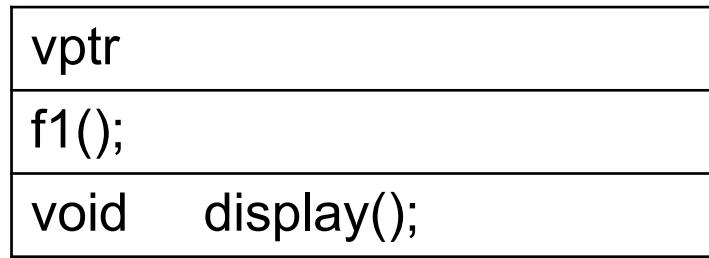
How Late Binding is implemented

vptr and vtable

Base Class



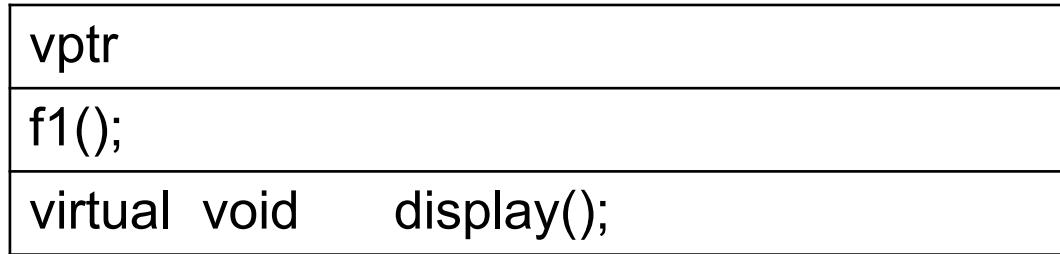
Derived Class



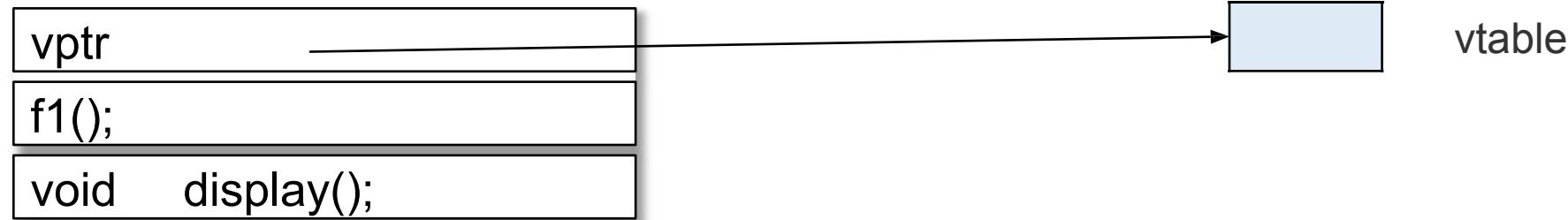
How Late Binding is implemented

vptr and vtable

Base Class



Derived Class



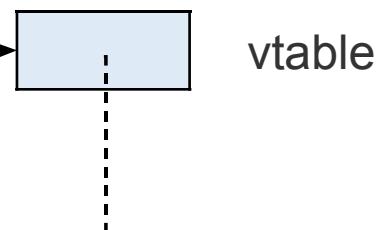
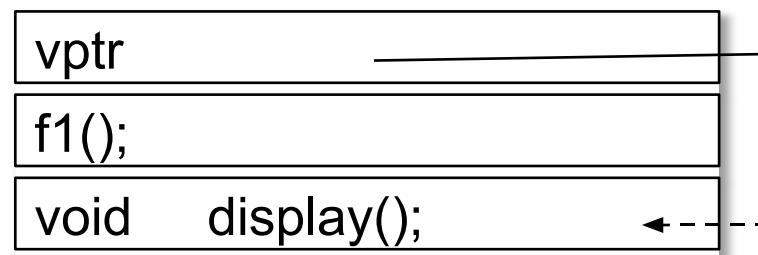
How Late Binding is implemented

vptr and vtable

Base Class



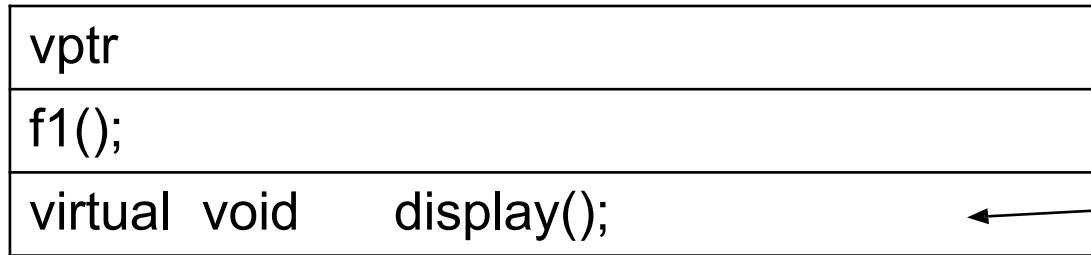
Derived Class



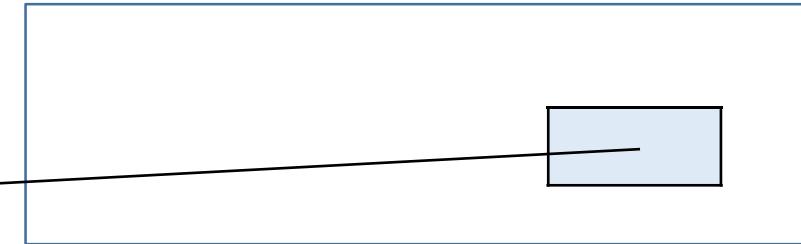
How Late Binding is implemented

vptr and vtable

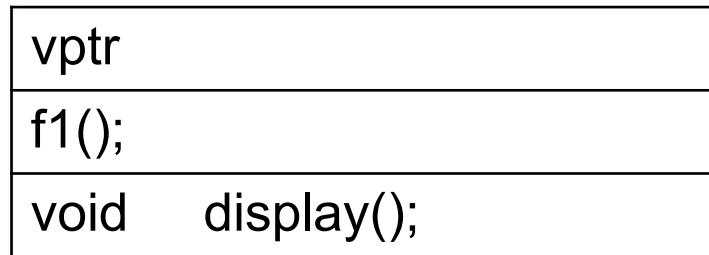
Base Class



baseObj



Derived Class



```
Base b1;  
Derived d1;  
  
Base * ptr;  
  
ptr = &b1;  
ptr->f1();  
ptr->display();
```

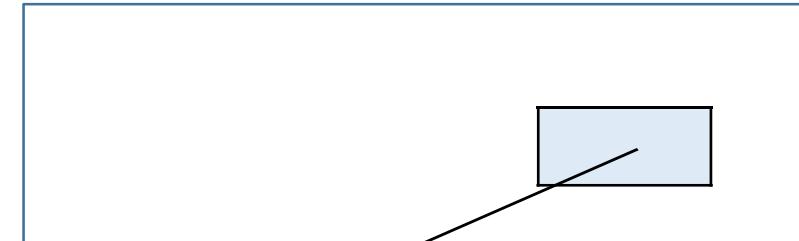
How Late Binding is implemented

vptr and vtable

Base Class

vptr
f1();
virtual void display();

derivedObj



Derived Class

vptr
f1();
void display();

```
ptr = &d1;  
ptr->f1();  
ptr->display();
```

One cool thing about Late Binding

Using pointer, we can also call private virtual functions! Why?

```
class Base
{
public:
    void f1()
    {
        cout << "f1 is called" << endl;
    }
    virtual void display()
    {
        cout << "Hello from Base class" << endl;
    }
};

class Derived : public Base
{
    void display()
    {
        cout << "Hello from Derived class, but it is private." << endl;
    }
public:
    void f1()
    {
        cout << "f1 is overridden" << endl;
    }
};

int main()
{
    Base b1;
    Derived d1;

    Base * ptr;
    ptr = &b1;
    ptr->f1();
    ptr->display();

    ptr = &d1;
    ptr->f1();
    ptr->display(); //The private version
}
```

One cool thing about Late Binding

Using pointer, we can also call private virtual functions! Why?

Because access specifier is checked **only** during compile time.

Virtual Destructor

Virtual Destructor

When we dynamically allocate a derived object using a base pointer, only the base destructor is called during the deletion process because of early binding.

In order to prevent this early binding, we mention the keyword `virtual` before the destructor of the base class.

Virtual Destructor

```
class Base
{
public:
    Base()
    {
        cout << "Hello from base." << endl;
    }
    ~Base()
    {
        cout << "Goodbye from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Hello from Derived." << endl;
    }
    ~Derived()
    {
        cout << "Goodbye from Derived class" << endl;
    }
};
```

Virtual Destructor

```
int main()
{
    Base * ptr;
    ptr = new Derived();
    delete ptr;
}
```

Q: Which constructor will be called?



Virtual Destructor

```
int main()
{
    Base * ptr;

    ptr = new Derived();

    delete ptr;
}
```

*Q: Which constructor will be called?
A: base and then derived.*



Virtual Destructor

```
int main()
{
    Base * ptr;

    ptr = new Derived();

    delete ptr;
}
```

Q: *Which destructor will be called?*

Virtual Destructor

```
int main()
{
    Base * ptr;

    ptr = new Derived();

    delete ptr;
}
```

Q: *Which destructor will be called?*

A: Only Base, because ptr is of type 'Base'. So compiler will early bind only the destructor of Base.

Virtual Destructor

```
class Base
{
public:
    Base()
    {
        cout << "Hello from base." << endl;
    }
    virtual ~Base()
    {
        cout << "Goodbye from Base class" << endl;
    }
};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Hello from Derived." << endl;
    }
    ~Derived()
    {
        cout << "Goodbye from Derived class" << endl;
    }
};
```

We mention *virtual* so that the compiler does not perform early binding.

Abstract Class and Interfaces

CSE 205 - Week 10 Class 2 & 3

[Lec Raiyan Rahman](#)

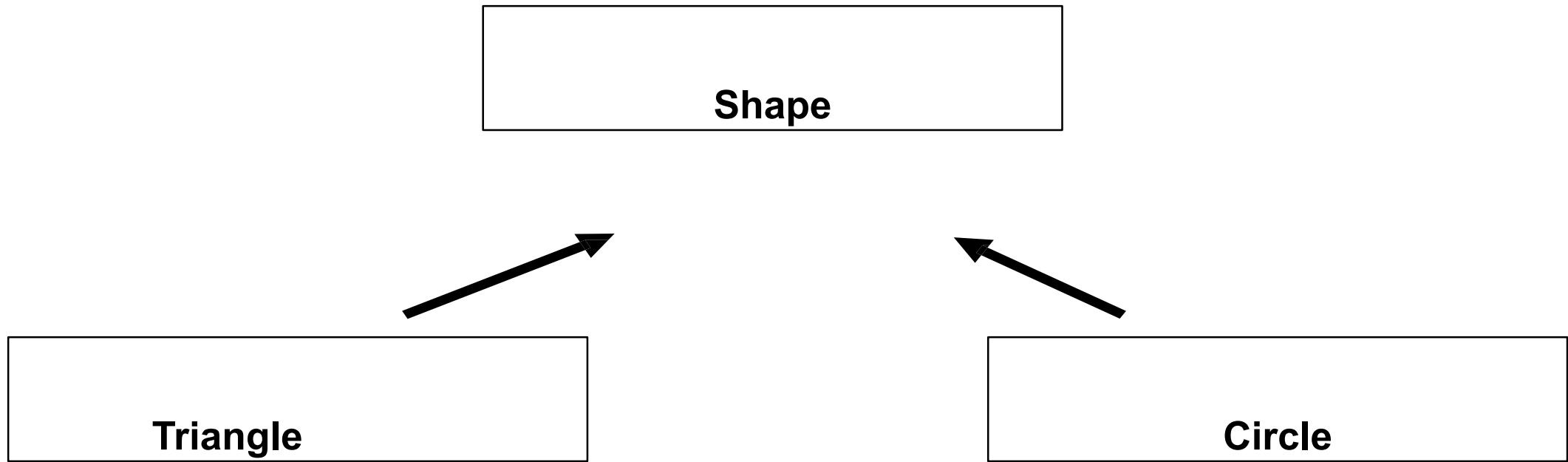
Dept of CSE, MIST

raian@cse.mist.ac.bd

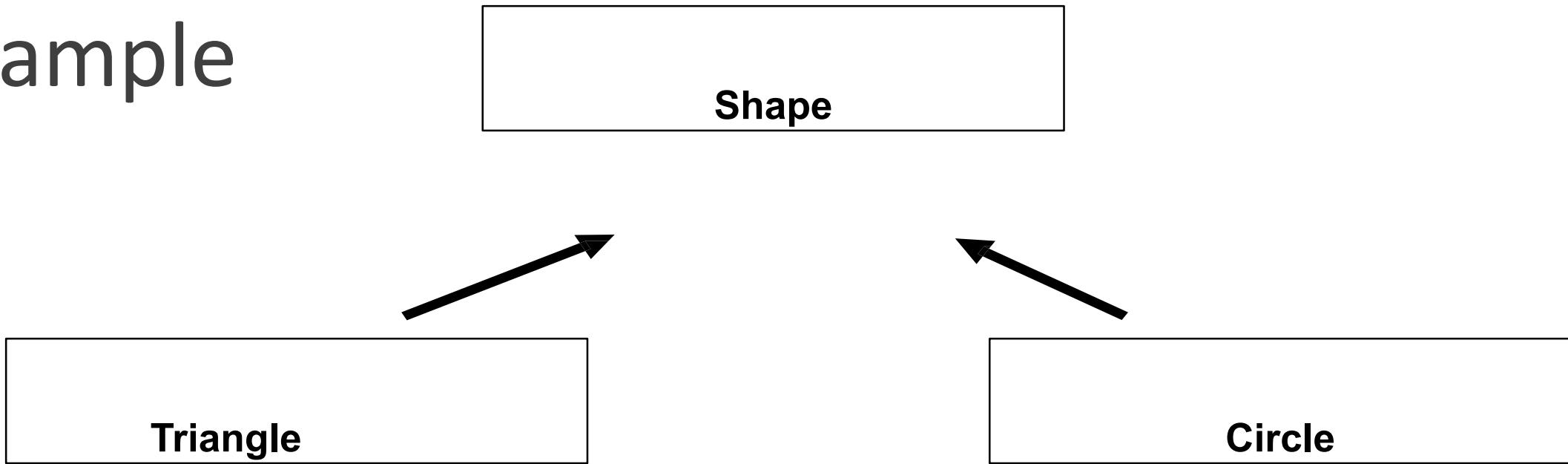
CC: Lec Saidul Hoque Anik



Consider the following
example



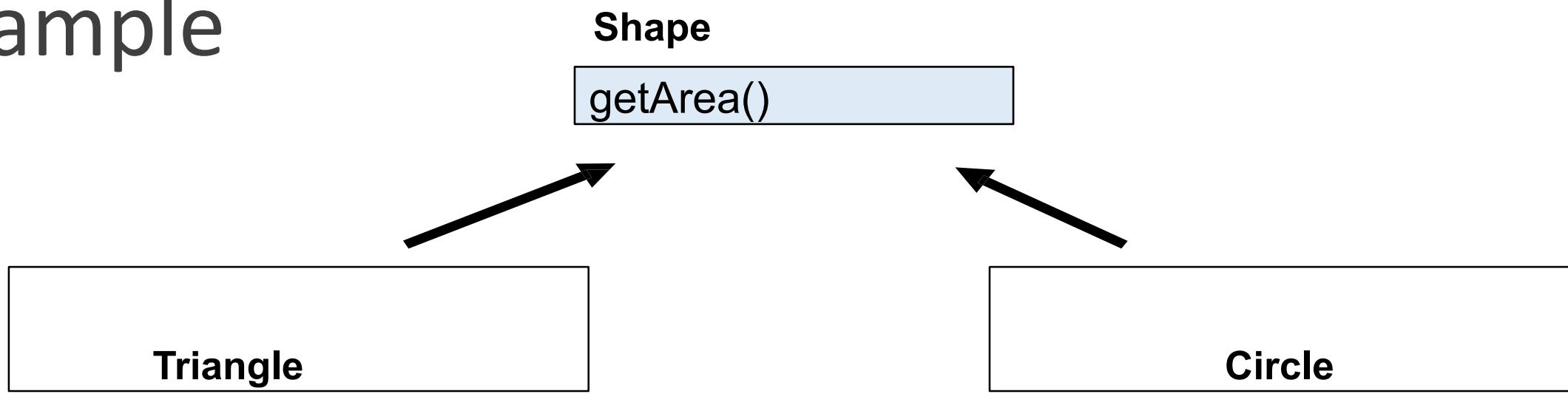
Consider the following example



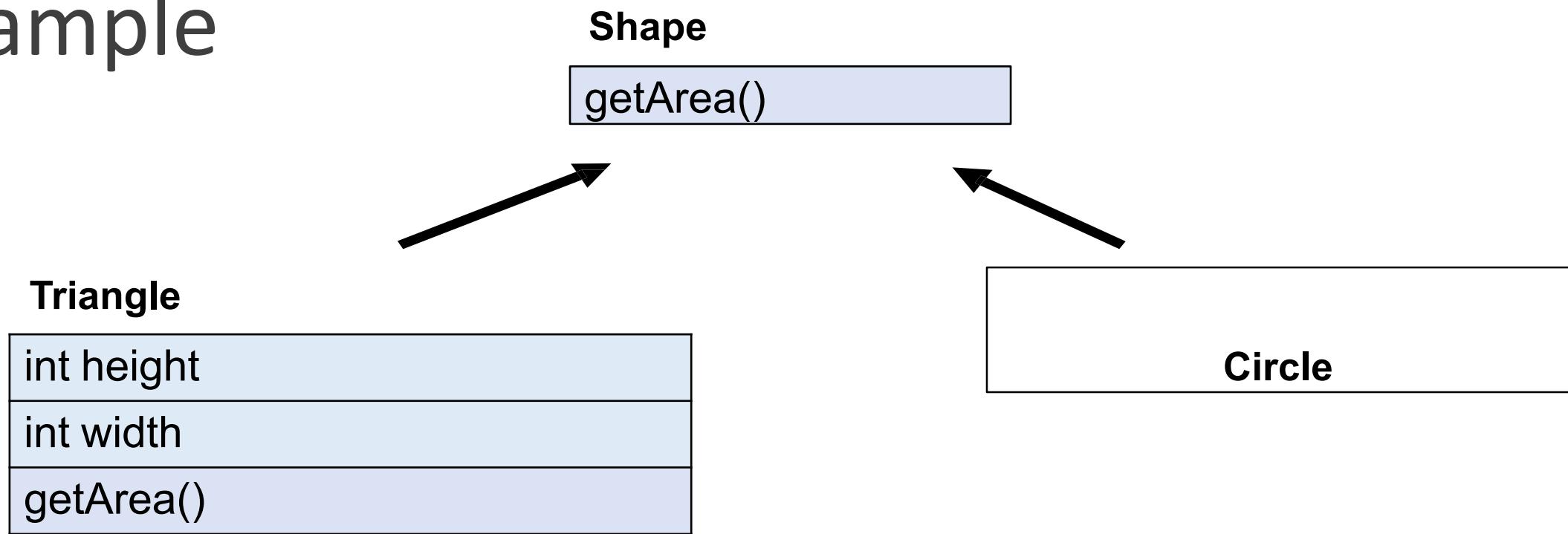
We want to call the following function from the main function:

```
void printArea(Shape * obj)
{
    cout << "Area is " << obj->getArea() << endl;
}
```

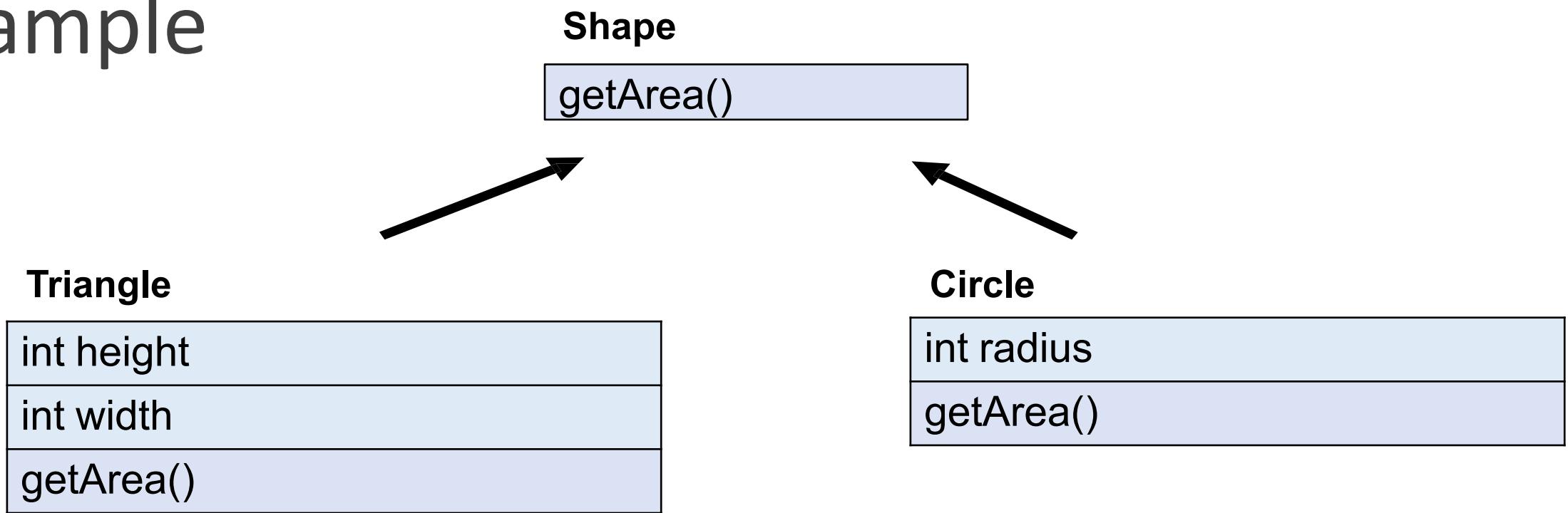
Consider the following
example



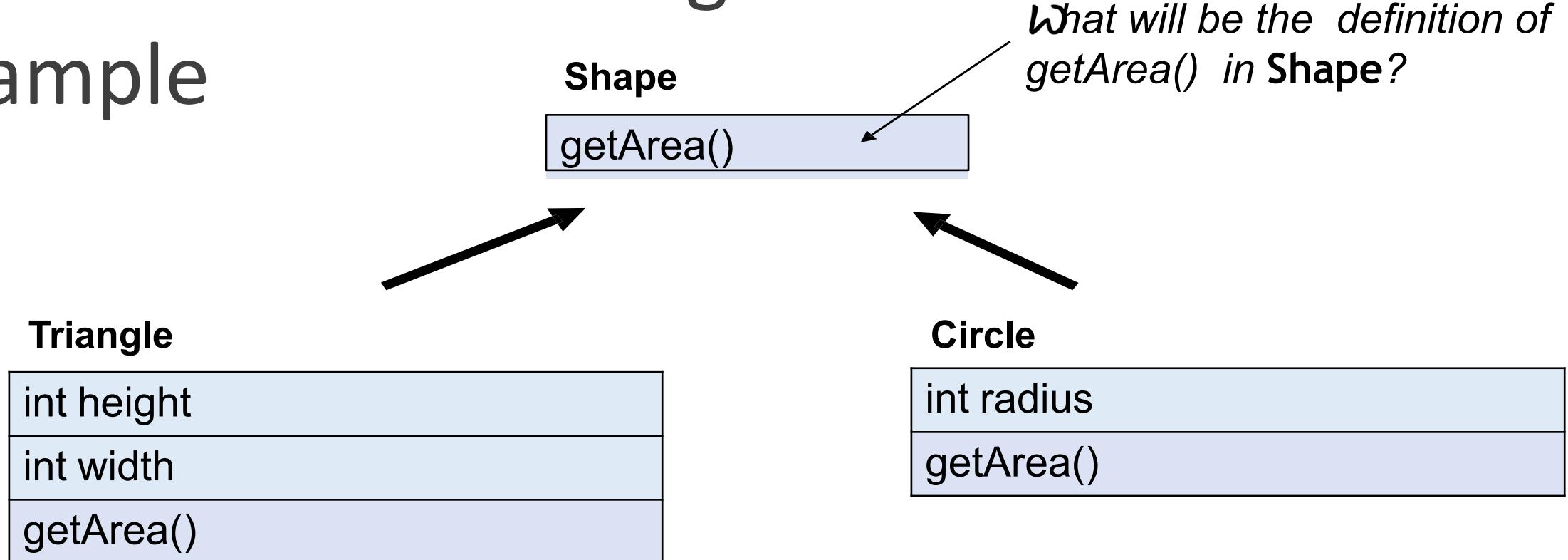
Consider the following example



Consider the following example

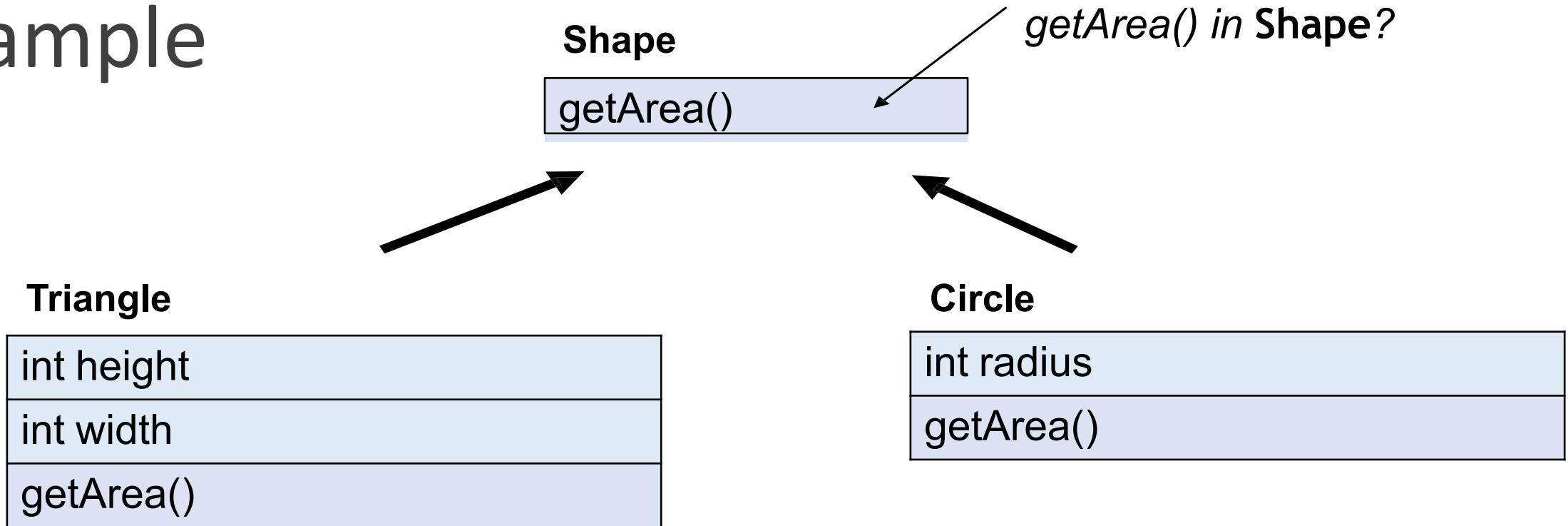


Consider the following example

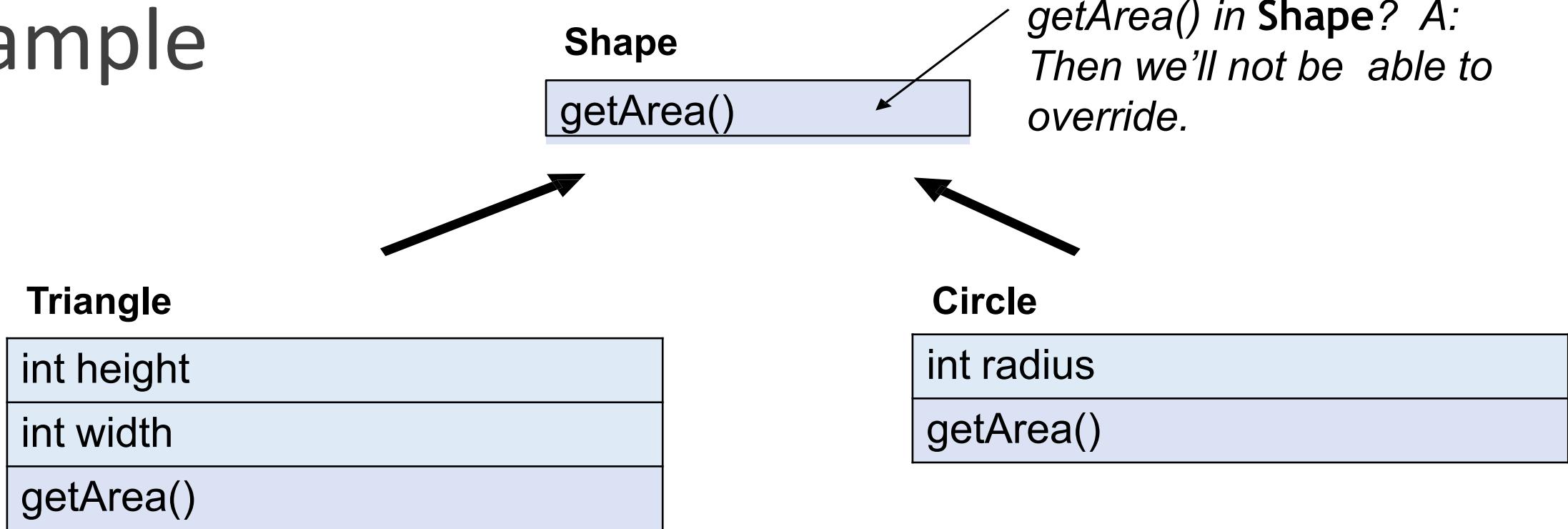


Consider the following example

*Q: Should we omit
getArea() in Shape?*



Consider the following example



Q: Should we omit getArea() in Shape? A: Then we'll not be able to override.

The Classes

```
class Shape
{
public:
    virtual double getArea()
    {

    }
};
```

```
class Triangle : public Shape
{
public:
    int height;
    int base;
    double getArea()
    {
        return 0.5 * base * height;
    }
};
```

```
class Circle : public Shape
{
public:
    int radius;
    double getArea()
    {
        return 3.1416 * radius;
    }
};
```

The Main Function

```
void printArea(Shape * obj)
{
    cout << "Area is " << obj->getArea() << endl;
}

int main()
{
    Triangle t1;
    t1.height = 10;
    t1.base = 10;
    printArea(&t1);

    Circle c1;
    c1.radius = 10;
    printArea(&c1);
}
```

The Main Function

```
void printArea(Shape * obj)
{
    cout << "Area is " << obj->getArea() << endl;
}

int main()
{
    Triangle t1;
    t1.height = 10;
    t1.base = 10;
    printArea(&t1);

    Circle c1;
    c1.radius = 10;
    printArea(&c1);

    Shape s1;
    printArea(&s1);
}
```

What will be the output?

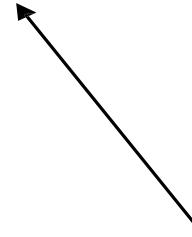
Revisiting the Shape Class

```
class Shape
{
public:
    virtual double getArea()
    {

    }
};
```

Pure Virtual Function

```
class Shape
{
public:
    virtual double getArea() = 0;
};
```



*Notice the syntax
This is known as 'pure specifier'*

We cannot have a definition of `getArea()` here, but we cannot omit the function from this class either (Because then we'll not be able to override).

So we keep it as 'Pure Virtual' function.

Pure Virtual Function

A function that doesn't yet have a definition in a class. It may be implemented later in its class hierarchy (one of the derived classes).

e.g: the `getArea()` in the `Shape` Class doesn't have a definition, but in `Triangle/Circle` class, it has a definite definition. Therefore, `getArea()` is written as pure virtual function in `Shape` Class.

Note:

- Destructor can be virtual or pure virtual. **But a constructor can never be virtual or pure virtual.**

Abstract Class

A class containing at least one pure virtual function.

- Cannot be instantiated (object cannot be created), but can be used as a base class
- In C++, abstract class can have constructor.

Which one is abstract class in the previous example?

Abstract Class

A class containing at least one pure virtual function.

- Cannot be instantiated (object cannot be created), but can be used as a base class
- In C++, abstract class can have constructor

Which one is abstract class in the previous example?

- Shape Class

Revisiting the Main Function

```
void printArea(Shape * obj)
{
    cout << "Area is " << obj->getArea() << endl;
}

int main()
{
    Triangle t1;
    t1.height = 10;
    t1.base = 10;
    printArea(&t1);

    Circle c1;
    c1.radius = 10;
    printArea(&c1);

    Shape s1; } What will happen now that Shape class is abstract?
    printArea(&s1);
}
```

Note on Abstract Class

If you don't override a pure virtual function in a derived class, that derived class becomes abstract too.

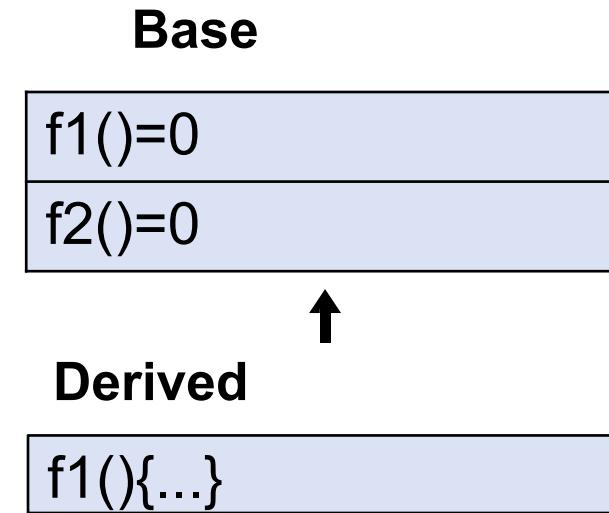
Base

f1()=0

f2()=0

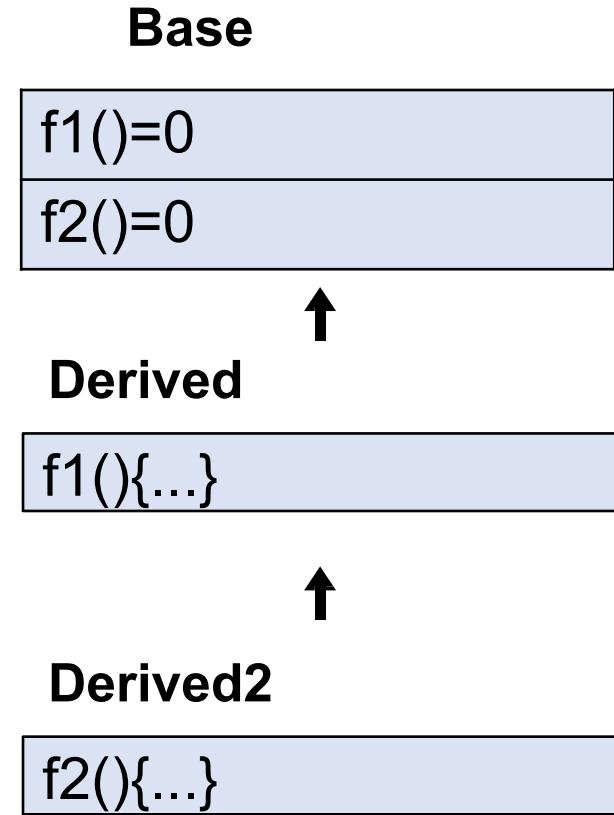
Note on Abstract Class

If you don't override a pure virtual function in a derived class, that derived class becomes abstract too.



Note on Abstract Class

If you don't override a pure virtual function in a derived class, that derived class becomes abstract too.



Can you identify
- *Abstract Class?*
- *Concrete Class?*

Abstract Class

A class with at least one pure virtual function

Concrete Class

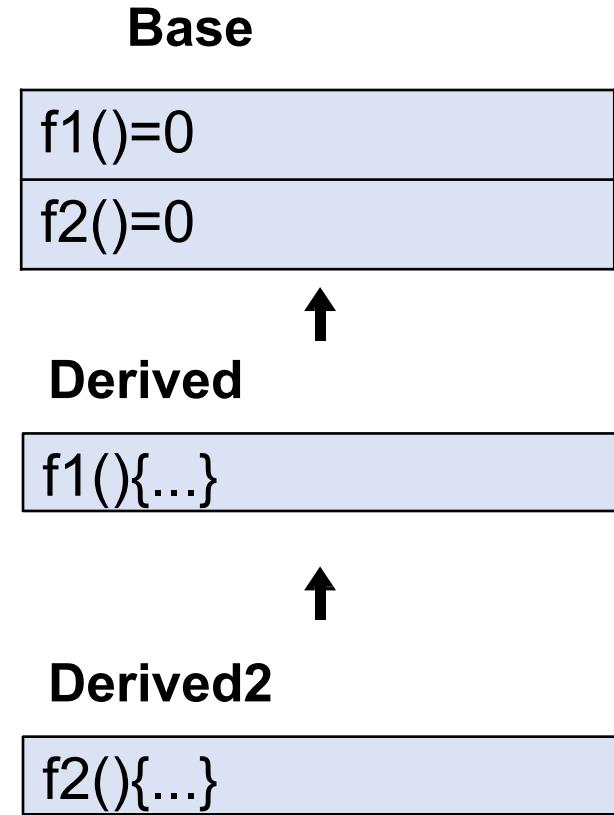
A class with no pure virtual function

Interface

A class with only pure virtual functions

Note on Abstract Class

If you don't override a pure virtual function in a derived class, that derived class becomes abstract too.



Can you identify
- *Interface?*
- *Abstract Class?*
- *Concrete Class?*

Note on Abstract Class

If you don't override a pure virtual function in a derived class, that derived class becomes abstract too.

```
class Base
{
public:
    virtual void f1() = 0;
    virtual void f2() = 0;
};

class Derived : public Base
{
public:
    void f1()
    {
        cout << "f1 is implemented" << endl;
    }
    //f2() is not implemented here
    //So this class is also an abstract class
};
```

Note on Abstract Class

...Cont.

```
class Derived2 : public Derived
{
public:
    void f2()
    {
        cout << "f2 is implemented" << endl;
    }
};

int main()
{
    //Base b1;           //Compiler Error - Base class is abstract
    //Derived d1;         //Compiler Error - Derived class is abstract
    Derived2 d2;        //OK
    d2.f1();            //OK
    d2.f2();            //OK
}
```

Thank you

The Diamond Problem and Virtual Base Classes

CSE 205 - Week 11 Class 1

Lec Raiyan Rahman

Dept of CSE, MIST

raian@cse.mist.ac.bd

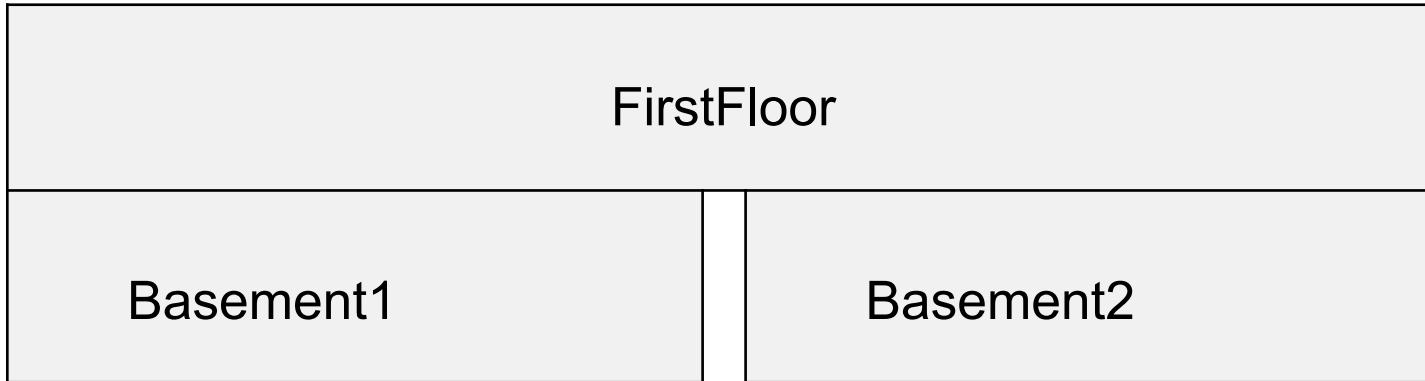
CC: Lec Saidul Hoque Anik



Let's do a quick recap of Multiple Inheritance

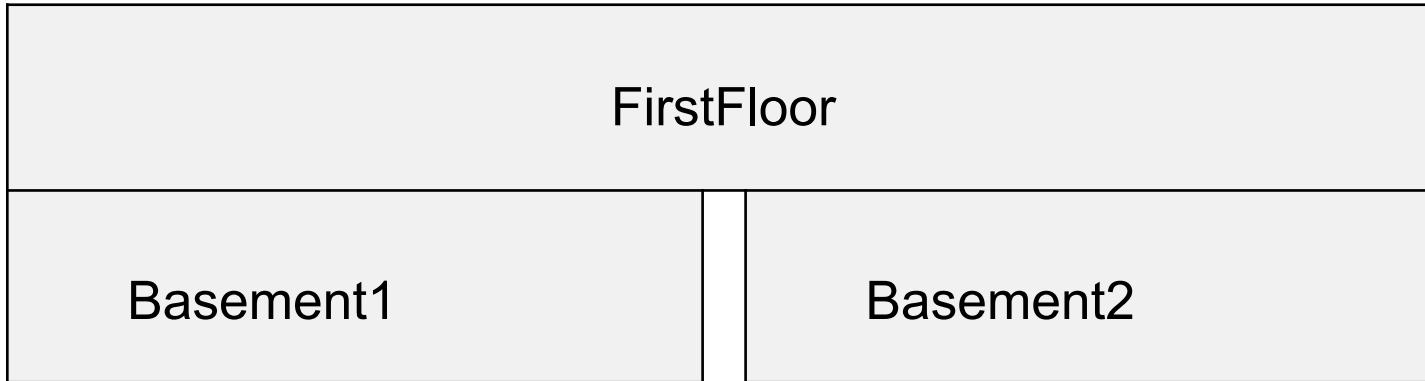
Hierarchical Inheritance/Multiple Inheritance

Derived Class with multiple base classes



Hierarchical Inheritance

What will be the order of constructor?



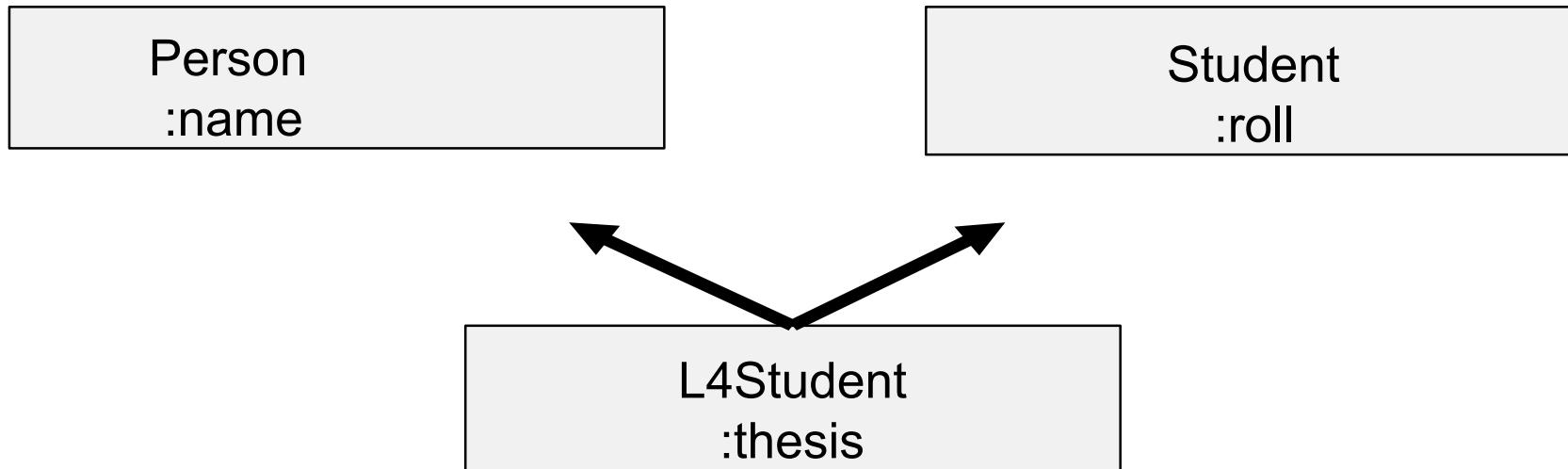
Hierarchical Inheritance

What will be the order of destructor?



Hierarchical Inheritance

Derived Class with multiple base classes



Hierarchical Inheritance

Derived Class with multiple base classes

```
class Person
{
    char name[100];
public:
    Person(char * n)
    {
        strcpy(name, n);
    }
    void printName()
    {
        cout << name << endl;
    }
};
```

```
class Student
{
    int roll;
public:
    Student(int r)
    {
        roll = r;
    }
    void printRoll()
    {
        cout << roll << endl;
    }
};
```

Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first



Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first

*This will be called
after first*

Hierarchical Inheritance

Derived Class with multiple base classes

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first

This will be called after first

This will be called at last

Hierarchical Inheritance

What will be the order of destructor?

```
class L4Student : public Person, public Student
{
    int thesis;
public:
    L4Student(char * name, int roll, int th): Person(name), Student(roll)
    {
        thesis = th;
    }
    void printMark()
    {
        cout << thesis << endl;
    }
};
```

This will be called first

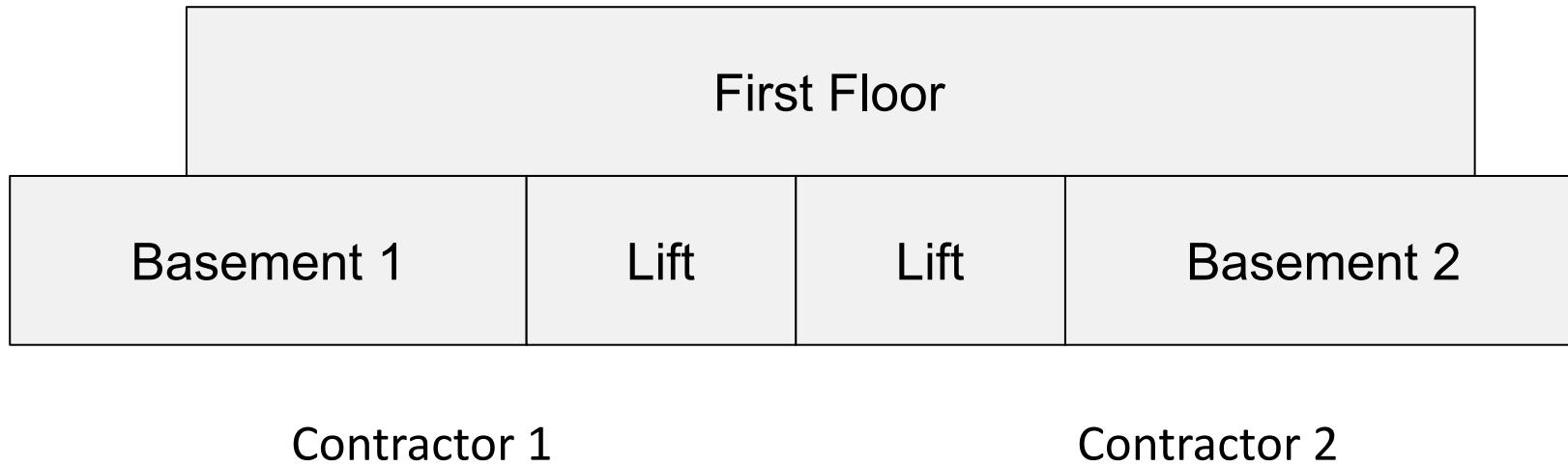
This will be called after first

This will be called at last

Now, Let's consider this scenario in Multiple Inheritance

Hierarchical Inheritance/Multiple Inheritance - What if?

Derived Class with multiple base classes with common attributes in Base classes

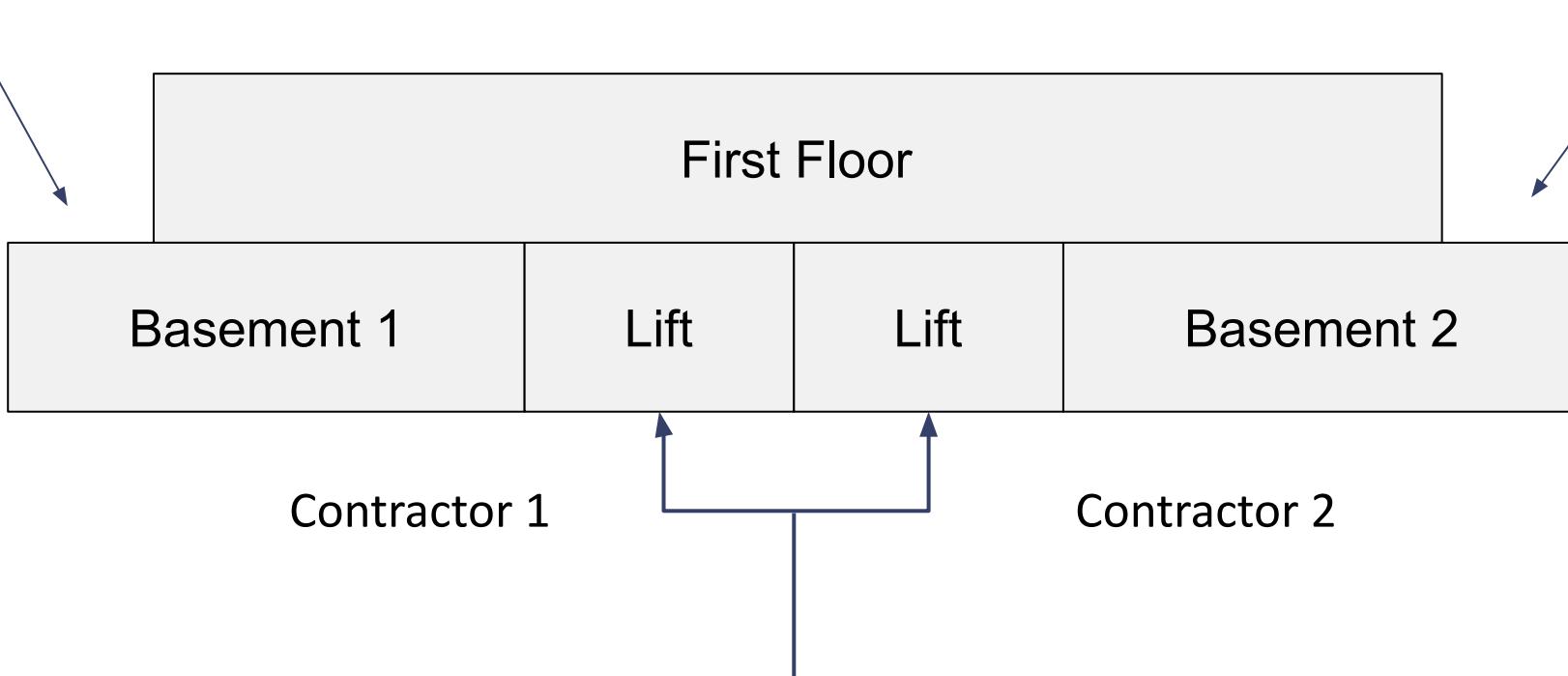


Hierarchical Inheritance/Multiple Inheritance - What if?

What are the obvious drawbacks of this scenario?

We're also exceeding allotted space!

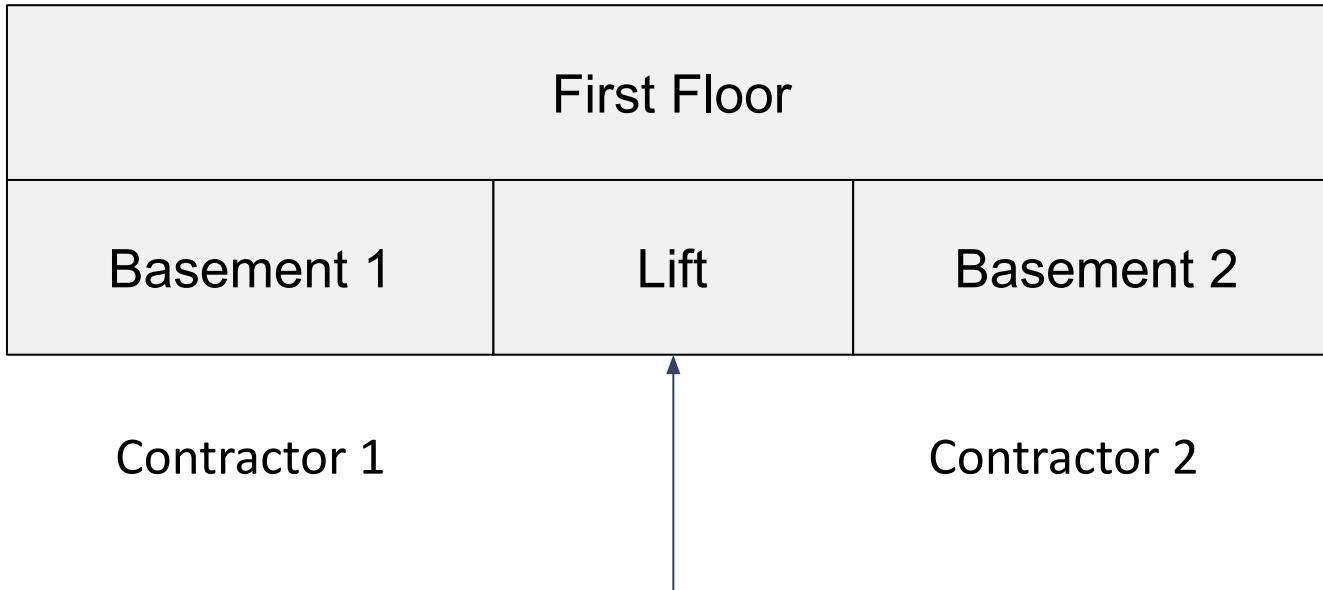
Another wasted space.



We're wasting valuable
time and resources
building the same thing
twice!

Hierarchical Inheritance/Multiple Inheritance - Solution

Derived Class with multiple base classes with common attributes in Base classes



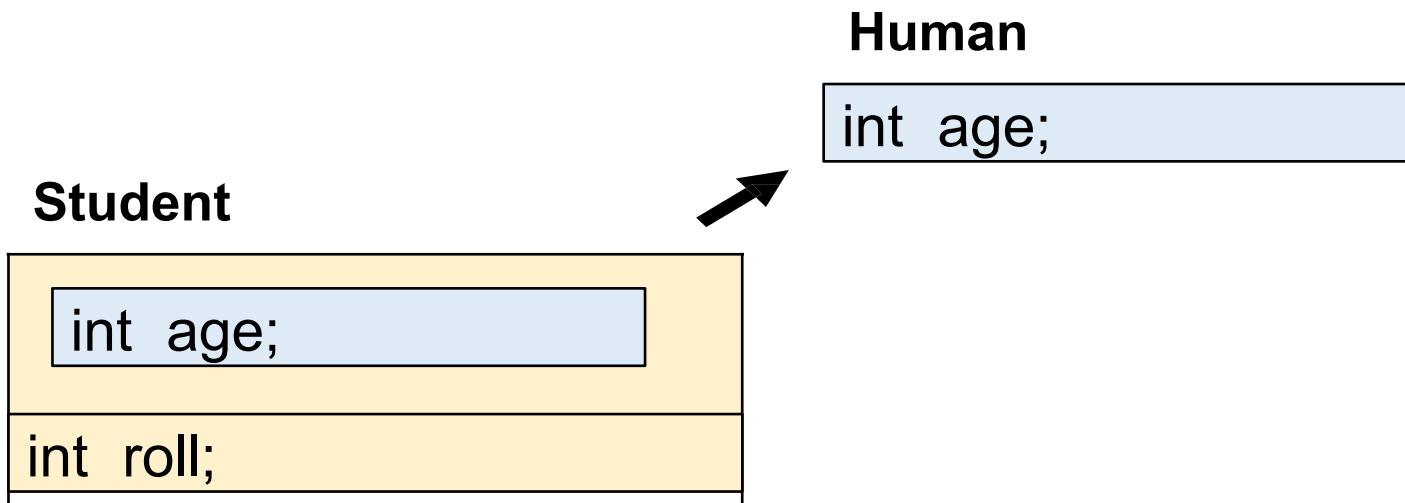
Who builds the common part then?
We'll soon see.

The Diamond Problem

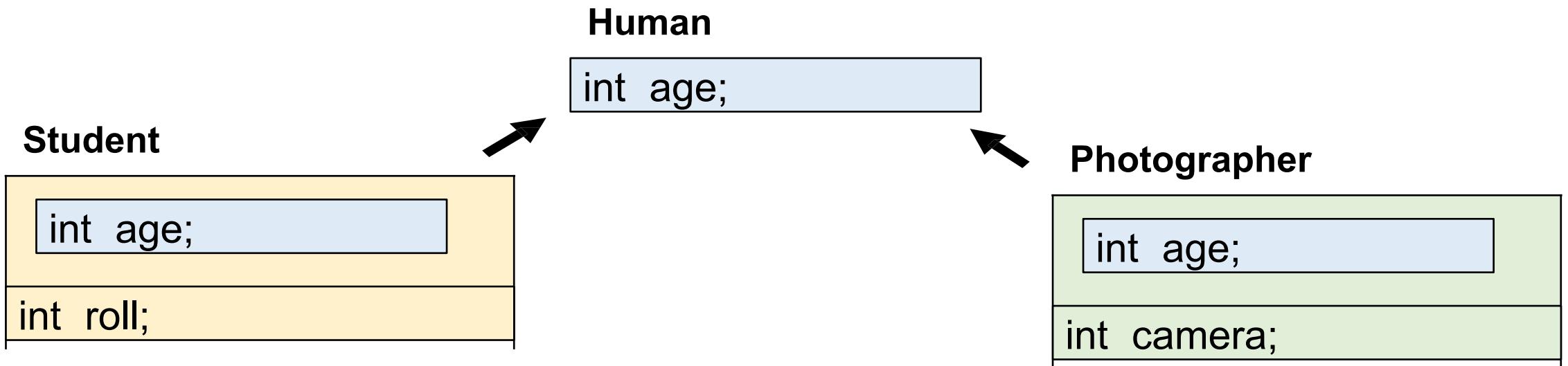
Human

```
int age;
```

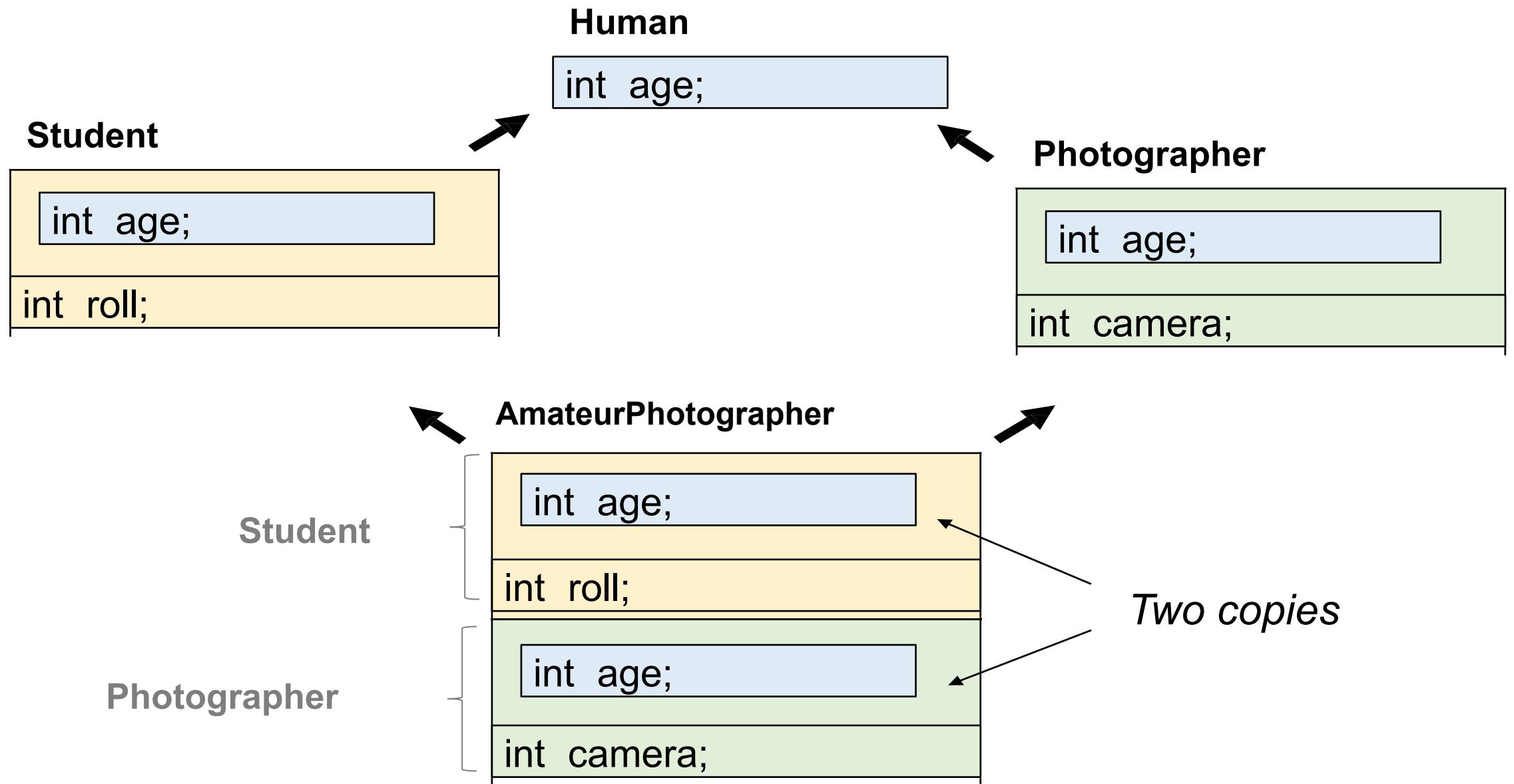
The Diamond Problem



The Diamond Problem



The Diamond Problem



The Diamond Problem

The diamond problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.

If there is a variable (age variable) in A that B and C have assigned different values, then which version of the variable does D inherit?

that of B, or that of C?

If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

The Diamond Problem

```
class Human
{
public:
    int age;
};

class Student : public Human
{
public:
    int roll;
};

class Photographer : public Human
{
public:
    int camera;
};
```

The Diamond Problem

```
class AmaturePhotographer : public Student, public Photographer
{
public:
    void printDetails()
    {
        cout << "Age is: " << age << endl;
        cout << "Roll is: " << roll << endl;
        cout << "Number of camera: " << camera << endl;
    }
};
```

Which age is used?



The Diamond Problem

```
class AmaturePhotographer : public Student, public Photographer
{
public:
    void printDetails()
    {
        cout << "Age is: " << age << endl;
        cout << "Roll is: " << roll << endl;
        cout << "Number of camera: " << camera << endl;
    }

};

int main()
{
    AmaturePhotographer person1;
    person1.age = 20; ←
    person1.roll = 50;
    person1.camera = 2;
    person1.printDetails();
}
```

Which age is used?

Which age is used?

The Diamond Problem

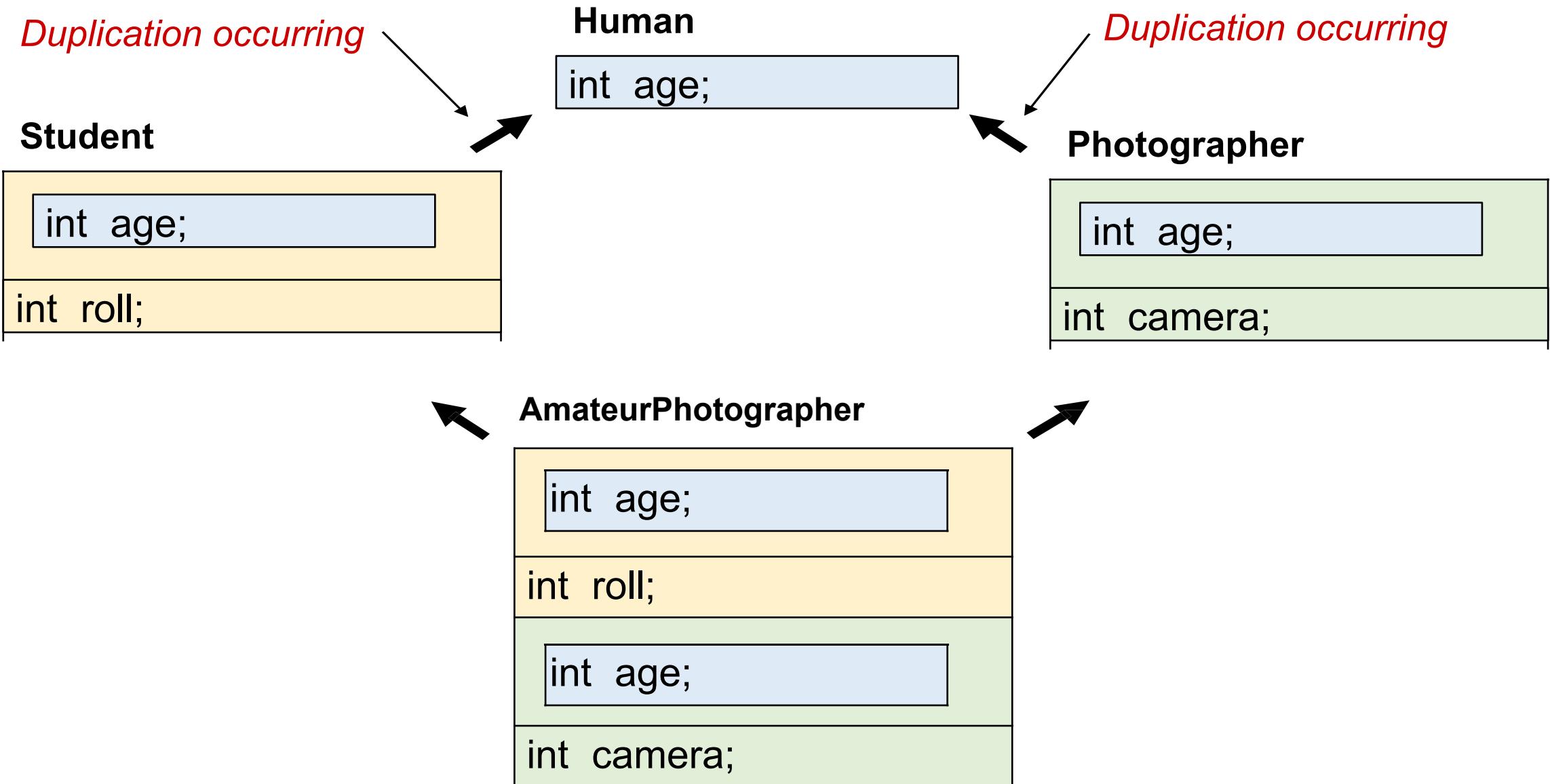
We can always use scope resolution operator

```
class AmaturePhotographer : public Student, public Photographer
{
public:
    void printDetails()
    {
        cout << "Age is: " << Student::age << endl;
        cout << "Roll is: " << roll << endl;
        cout << "Number of camera: " << camera << endl;
    }
};

int main()
{
    AmaturePhotographer person1;
    person1.Photographer::age = 20;
    person1.roll = 50;
    person1.camera = 2;
    person1.printDetails();
}
```

What if wrong scope is used?

We'll have to prevent duplication



Virtual Inheritance

```
class Student : virtual public Human
{
public:
    int roll;
};
```

```
class Photographer : virtual public Human
{
public:
    int camera;
};
```

*This is to ensure that always
a single copy is created*

Virtual Inheritance

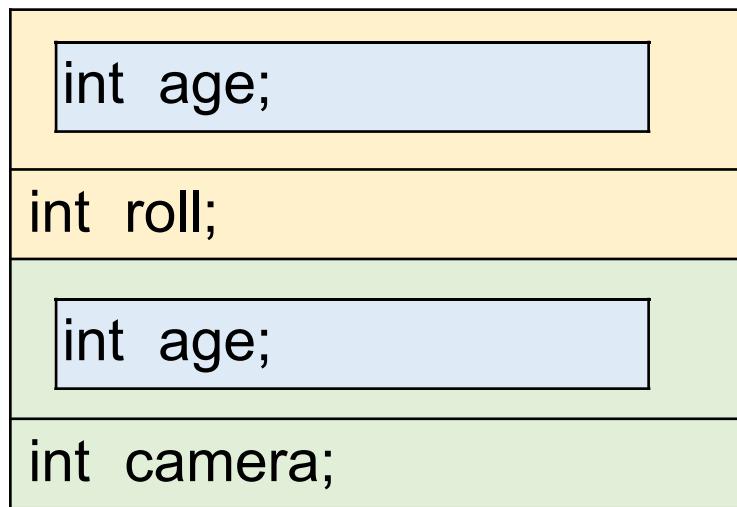
```
class Student : virtual public Human
{
public:
    int roll;
};

class Photographer : virtual public Human
{
public:
    int camera;
};
```

This is now virtual base class

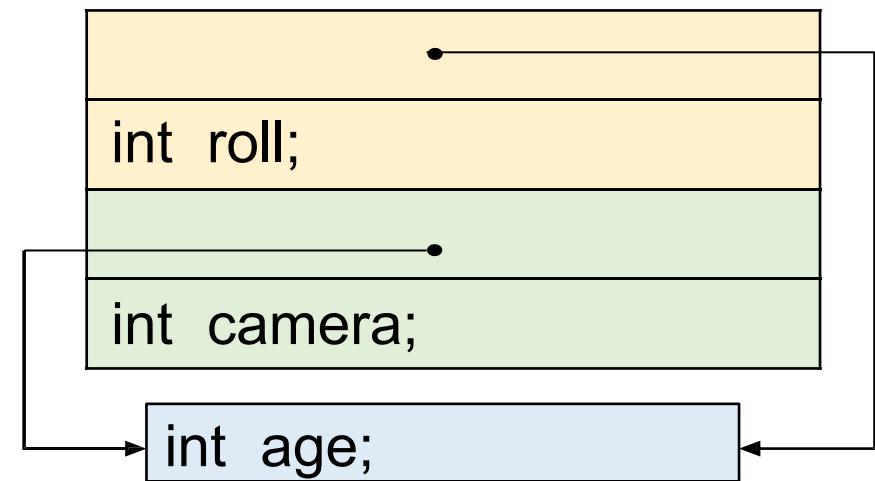


AmateurPhotographer



Normal Inheritance

AmateurPhotographer



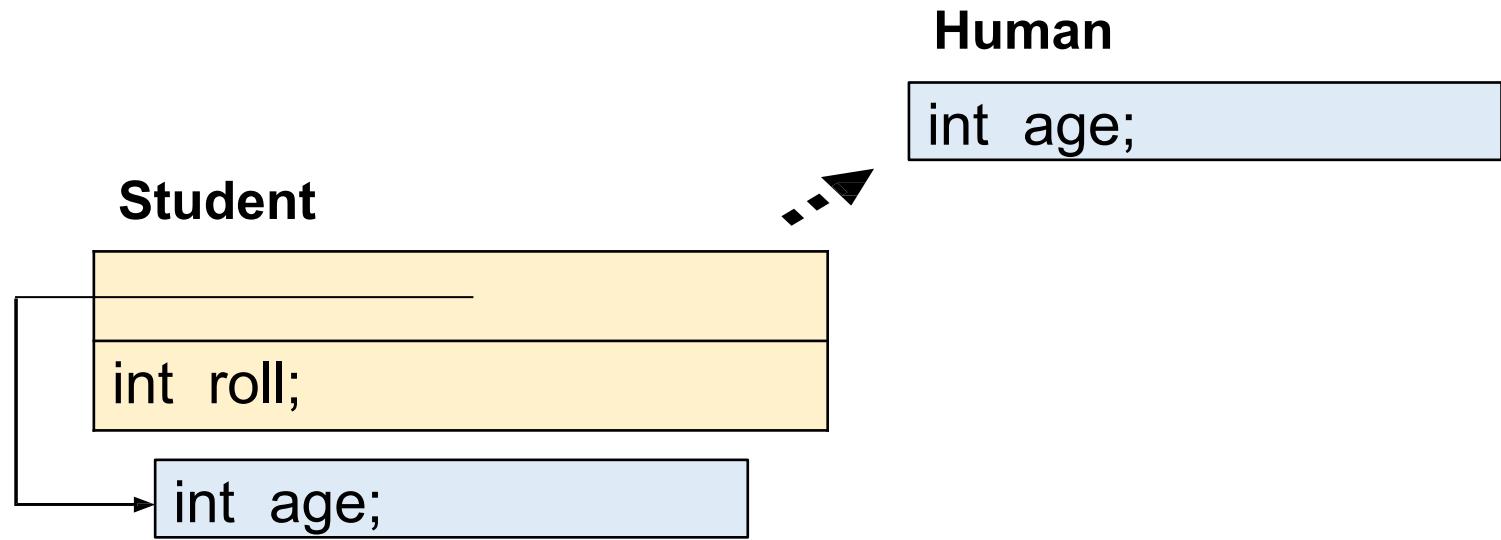
Virtual Inheritance

Virtual Inheritance

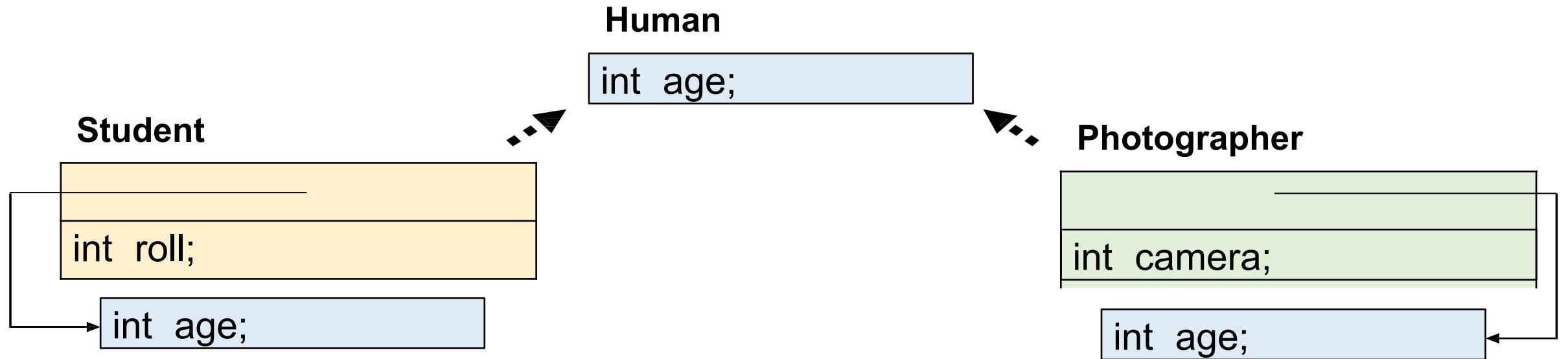
Human

```
int age;
```

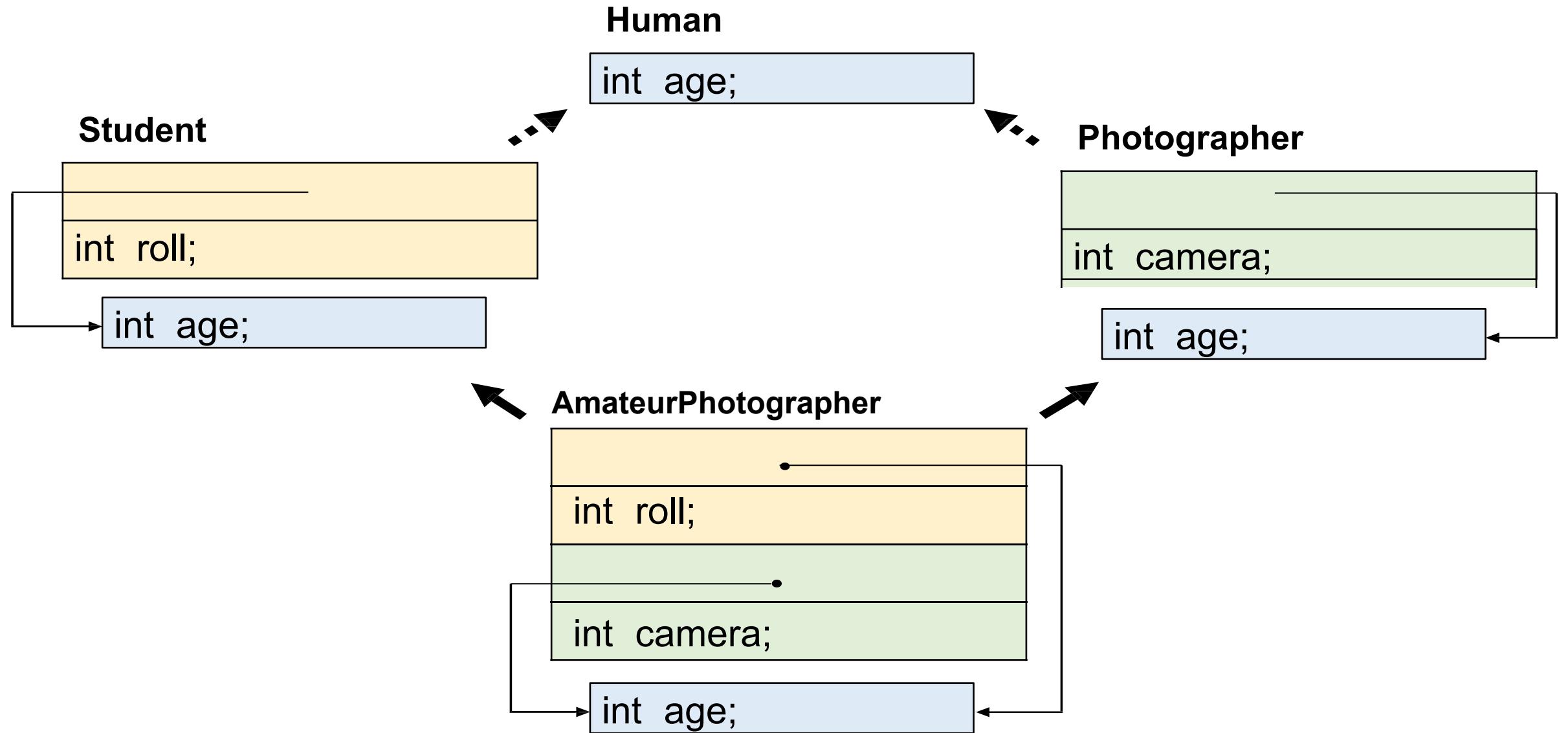
Virtual Inheritance



Virtual Inheritance



Virtual Inheritance



Constructor in Virtual Inheritance

```
class Human
{
public:
    int age;
    Human(int a)
    {
        cout << "Human is created" << endl;
        age = a;
    }
};
```

Constructor in Virtual Inheritance

```
class Photographer : virtual public Human
{
public:
    int camera;
    Photographer(int a, int c): Human(a)
    {
        cout << "Photographer is created" << endl;
        camera = c;
    }
};
```

Constructor in Virtual Inheritance

```
class AmateurPhotographer : public Student, public Photographer
{
public:
    AmateurPhotographer(int a, int r, int c) : Student(a, r),
                                                Photographer(a, c),
                                                Human(a)
    {
        cout << "AmateurPhotographer is created" << endl;
    }
};

int main()
{
    //Student s1;
    //Photographer p1;
    AmateurPhotographer ap1(1, 2, 3);
}
```

Virtual Inheritance - Syntax

So, we need to do two things while implementing virtual inheritance to overcome Diamond Problem:

1. Inherit the common base class (Human) as virtual in the first level (while writing the class definition for Student and Photographer).
2. While writing the constructor for the **most derived** class (the class at the bottom of the inheritance sequence, i.e. AmateurPhotographer), pass variables to the virtual base class (i.e. Human) as if it was a direct base class of AmateurPhotographer.

Constructor in Virtual Inheritance

A few things to note

One, Virtual base classes are always created before non-virtual base classes

```
class AmateurPhotographer : public Student, public Photographer
{
```

Constructor in Virtual Inheritance

A few things to note

One, Virtual base classes are always created before non-virtual base classes

```
class AmateurPhotographer : public Student, public Photographer
{
```

Ctor Seq: Human > Student > Photographer

Constructor in Virtual Inheritance

Even if we had a standalone unrelated class (for example, a new class “Employee”) to the Virtual Base class (Human), the virtual base class will be called first.

Then, the rest of the classes will be called as per the inheritance sequence.

Constructor in Virtual Inheritance

A few things to note

One, Virtual base classes are always created before non-virtual base classes

```
class Employee
{
public:
    Employee()
    {
        cout <<"Employee created" << endl;
    }
};

class AmateurPhotographer : public Employee, public Student,
                           public Photographer
{
public:
    AmateurPhotographer(int a, int r, int c) : Student(a, r),
                                                Photographer(a, c),
                                                Human(a)
```

Ctor Seq: Human > **Employee** > Student > Photographer

Constructor in Virtual Inheritance

A few things to note

Two, the *most* derived class is responsible for constructing the virtual base class

```
int roll;
Student(int a, int r) : Human(a)
{
    cout << "Student is created" << endl;
}

...
Photographer(int a, int c) : Human(a)

...
AmateurPhotographer(int a, int r, int c) : Student(a, r),
                                                Photographer(a, c),
                                                Human(a)
// constructor p1,
AmateurPhotographer ap1(1, 2, 3);
```

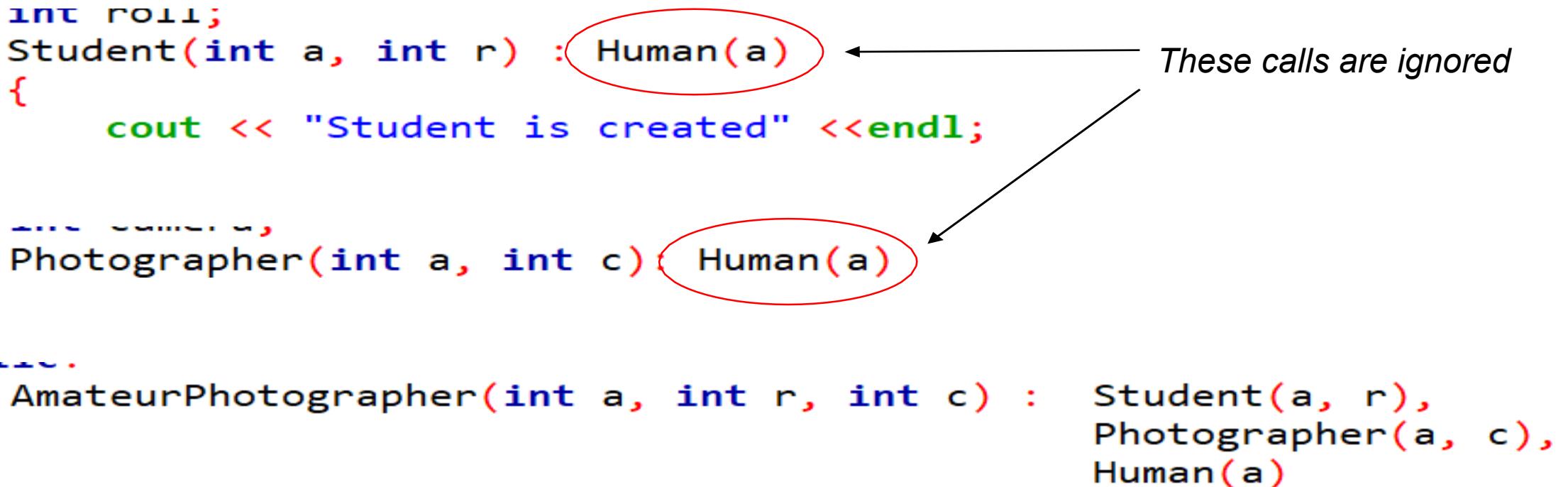
This constructor is called

Constructor in Virtual Inheritance

A few things to note

Two, the *most* derived class is responsible for constructing the virtual base class

```
int roll;  
Student(int a, int r) : Human(a) {  
    cout << "Student is created" << endl;  
}  
...  
Photographer(int a, int c) : Human(a) {  
    cout << "Photographer is created" << endl;  
}  
...  
AmateurPhotographer(int a, int r, int c) : Student(a, r),  
                                         Photographer(a, c),  
                                         Human(a)  
{  
    cout << "AmateurPhotographer is created" << endl;  
}  
  
AmateurPhotographer ap1(1, 2, 3);
```



These calls are ignored

Constructor in Virtual Inheritance

A few things to note

Three, a virtual base class is always considered a direct base of its most derived class.

This is why the most derived class is responsible for its construction.

```
AmateurPhotographer(int a, int r, int c) : Student(a, r),  
Photographer(a, c),  
Human(a)
```

References

- <https://docs.microsoft.com/en-us/cpp/cpp/multiple-base-classes>
- <http://www.learncpp.com/cpp-tutorial/128-virtual-base-classes/>

Thank you

Operator Overloading

CSE 205 - Week 11 Class 2 & 3

Lec Raiyan Rahman

Dept of CSE, MIST

raian@cse.mist.ac.bd

CC: Lec Saidul Hoque Anik



Remember Function Overloading?

Same function name, different definition **based on parameter**.

```
void printMax(int a, int b);
```

```
void printMax(double a, double b);
```

But Can We Do This?

```
int a = 10;  
int b = 20;  
  
int res;  
  
res = a + b;  
  
cout<<res; //30
```

```
Point P1(1, 1);  
Point P2 (2, 2);
```

```
Point res;  
  
res = P1 + P2;  
  
cout<<res; //(3, 3)
```

Solution: Operator Overloading

Same operator, different definition **based on parameter**.

```
int main()
{
    int a = 2;
    int b = 3;
}

}
```

```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

Operator Overloading

Same operator, different definition **based on parameter**.

```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;      What are the parameters of + ?
}                      This is like calling a function:  
                      int operator+ (int left, int right)
```

```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

'operator' is a keyword

Operator Overloading

Same operator, different definition **based on parameter**.

```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;

    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2;
}
```

```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

Operator Overloading

Same operator, different definition **based on parameter**.

```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;

    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2; //p3 = (4, 5)
}
```

```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

Operator Overloading

Same operator, different definition **based on parameter**.

```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;

    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2;
```

```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

what will be the overloaded function?
operator+ (Point left, Point right)

Operator Overloading

```
class Point
{
public:
    int x;
    int y;
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ")";
    }
};
```

Point p3 = p1 + p2;

Operator Overloading

```
class Point
{
public:
    int x;
    int y;
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ")";
    }
};

Point operator+(Point left, Point right)
{
    int new_x = left.x + right.x;
    int new_y = left.y + right.y;
    Point temp(new_x, new_y);
    return temp;
}

Point p3 = p1 + p2;
```

Operator Overloading

What if we wanted this?

```
Point p3 = p1 + 10;
```

Operator Overloading

```
Point operator+(Point left, int right)
{
    int new_x = left.x + right;
    int new_y = left.y + right;
    Point temp(new_x, new_y);
    return temp;
}
```

```
Point p3 = p1 + 10;
```

Now, Let's take a closer look at different kinds of
Operator Overloading

Operator Overloading

- Enables C++ operators to work with class objects.
- Done by writing an 'operator' function. Eg. operator+ will overload + operator.
- Default operators of any class: ',', '=' and '&'

Operator Overloading Restrictions

C++ Operators that can be overloaded:

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+ =	- =	/ =	% =	^ =	& =
=	* =	<< =	>> =	[]	()
->	->*	new	new []	delete	delete []

Pointer to member operator

C++ Operators that cannot be overloaded:

::	*	.	:	sizeof
----	---	---	---	--------

Operator Overloading Restrictions

- Precedence of operator cannot be changed (order of evaluation)
 $(p1 + (p2 / p3))$ will not be $((p1 + p2) / p3)$
- Associativity of an operator cannot be changed (left-to-right)
 $((A + B) + C)$ cannot be changed into $(A + (B + C))$
- Number of operands cannot be changed
 - Unary operator remains unary, binary operator remains binary
 - Default parameter cannot be passed
- New operator can not be created
- No overloading of built in type
 - Cannot change how two integers are added (Will produce syntax error)

Operator Overloading Placement

- Operator function as Member vs. Non-member function:
Any operator can be non-member function **except**:

()	[]	->	Any assignment op
-----	----	----	-------------------

- Operator function as Member function:
Leftmost operand must be an object
(If leftmost operand is different, should make it non-member)
- Operator function as Non-member function:
Must be friend of the class if private member access is required

Bottom Line: Consider making it a member function if you're dealing with private attributes.

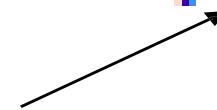
Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```



*This is thus a member function of
Point which we'll have to overload*

Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```



*The coordinates of p1 will come
from the member variable*

Arithmetic Operator Overloading

When we write this:

```
p1 + p2;
```

Compiler executes this:

```
p1.operator+(p2);
```

*The coordinates of p2 will come
from the function argument*

Arithmetic Operator Overloading

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }

    Point operator+(Point rightPoint)
    {
        int new_x = x + rightPoint.x;
        int new_y = y + rightPoint.y;
        Point ret(new_x, new_y);
        return ret;
    }
};
```

```
int main()
{
    Point p1(2, 3);
    Point p2(10, 20);
    Point p3 = p1 + p2;
    p3.display(); //12, 22
}
```

Similar Arithmetic Operators

+	-	*	/	%
---	---	---	---	---

Relational Operators

==	!=	>	<	>=	<=
----	----	---	---	----	----

- Must return a bool value (true/false)

Relational Operators

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }
    bool operator==(Point rightpt)
    {
        if ((x == rightpt.x) && (y == rightpt.y))
            return true;
        else
            return false;
    }
};
```

Relational Operators

...cont.

```
int main()
{
    Point p1(2, 3);
    Point p2(2, 3);

    if (p1 == p2)
        cout << "Both are equal" << endl;
    else
        cout << "Both are not equal" << endl;
}
```

Compound Assignment Operators

<code>+ =</code>	<code>- =</code>	<code>* =</code>	<code>/ =</code>	<code>% =</code>
<code>& =</code>	<code> =</code>	<code>^ =</code>	<code><<=</code>	<code>>>=</code>

- **Changes the left hand operator**
- Should be overloaded as member function

Point p1(1, 2), p2(10, 10);

...

`p1 += p2; //p1 = (11, 12); equivalent to p1 = p1 + p2`

Compound Assignment Operators

`+ =` Implementation as member function

```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }

    Point operator+=(Point obj)
    {
        this->x = this->x + obj.x;
        this->y = this->y + obj.y;
        return *this;
    }
};
```

```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Compound Assignment Operators

`+ =` Implementation as non-member function

```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }

    friend Point operator+=(Point&t, Point obj);
};

Point operator+=(Point&t, Point obj)
{
    t.x = t.x + obj.x;
    t.y = t.y + obj.y;
    return t;
}
```

```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Increment/Decrement Operator

++	--
----	----

- These operators can be prefix/postfix

`+ + | p1 ;`

`p1 + + ;`

Increment/Decrement Operator

++	--
----	----

- These operators can be prefix/postfix

Prefix

Postfix

Will they return the same thing?

Prefix Increment Operator

Implementation as member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }
}
```

```
Point operator++()
{
    this->x++;
    this->y++;
    return *this;
}
```

```
int main()
{
    Point p1(2, 3);
    ++p1;
    p1.display();
}
```

Postfix Increment Operator

Implementation as member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }

    Point operator++(int a)
    {
        //value of a is ignored
        Point copyObj = *this;
        this->x++;
        this->y++;
        return copyObj;
    }
};
```

p1++;

```
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

Prefix Increment Operator

Implementation as non-member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }

    friend void operator++(Point &obj);
};

void operator++(Point &obj)
{
    obj.x++;
    obj.y++;
}
```

~~++p1;~~

```
int main()
{
    Point p1(2, 3);
    ++p1;
    p1.display();
}
```

Postfix Increment Operator

Implementation as non-member function

```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y << endl;
    }
    friend Point operator++(Point &obj, int a);
};

Point operator++(Point &obj, int a)
{
    //value of a is ignored
    Point copyObj = obj;
    obj.x++;
    obj.y++;
    return copyObj;
}
```

p1++;

```
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

Assignment Operator =

- Must be a member function
- Receives the new value as argument, modifies this
- Should return `*this` to support `x = y = z;`

A Practical use of = overloading

```
class String
{
    char * p;
    int len;
public:
    String()
    {
        len = 0;
        p = 0;
    }
    String(char * arr, int l)
    {
        len = l;
        p = new char[len];
        for (int i = 0; i<len; i++)
            p[i] = arr[i];
    }
    void display()
    {
        for (int i = 0; i<len; i++)
            cout << p[i];
        cout << endl;
    }
    ~String()
    {
        delete [] p;
    }
};
```

```
int main ()
{
    String s;
    String dummy("abcde", 5);
    s=dummy;
}
```

A Practical use of = overloading

```
String &operator=(String newStr)
{
    len = newStr.len;
    p = new char[len];
    for (int i = 0; i<len; i++)
        p[i] = newStr.p[i];
    return *this;
}
```

Subscript Operator []

- **Must** be a member function
- Takes only one explicit parameter, the index
- The index can also be other than int

Subscript Operator []

Expectation

```
int main()
{
    String s1("abcde", 5);

    cout << s1[2] << endl; //expecting 'c'
}
```

Subscript Operator [] overloading

Overloaded as a member function of String

```
char operator[](int index)
{
    return p[index];
}
```

Subscript Operator [] overloading

Different type of index

```
int main()
{
    String s1("abcde", 5);

    cout << s1[2] << endl; //expecting 'c'

    cout << s1['a'] << endl; //expecting '0'
    cout << s1['e'] << endl; //expecting '4'
    cout << s1['p'] << endl; //expecting '-1'

}
```

Subscript Operator [] overloading

Implementation

```
int operator[](char ch)
{
    for (int i = 0; i<len; i++)
        if (p[i] == ch)
            return i;
    return -1;
}
```

Making [] work as lvalue

Design the operator[]() in such a way that the [] can be used on both the left and the right side of an assignment operator.

```
int main()
{
    String s1("abcde", 5);

    s1[0] = 'A';      //Assigning value using []
    s1.display();     //ABCDE
}
```

Making [] work as lvalue

Implementation

```
char& operator[](int index)
{
    return p[index];
}
```

“Now that [] operator returns a reference to the array element at ‘index’, It can be used on the left side of an assignment operator to modify an element of the array. Of course, it can still be used on the right side as well.”

- Teach yourself C++ by Herb Schildt (Page 225)

But What if ?

How to overload the + operator so that the following code works?

```
int main()
{
    Point p1(10, 10);

    Point p2 = 5 + p1;

    p2.display();           //p2 = (15, 15)
}
```

Solution

We must declare the operator+ function as non-member in this case.

```
Point operator+(int a, Point p)
{
    int x_ = a + p.x;
    int y_ = a + p.y;
    Point ret(x_, y_);
    return ret;
}
```

Overloading the () operator

Example:

Suppose you are required to calculate the value of y for the following line equations, where the values of x are from 1 to 5.

$$y = 4x + 3$$

$$y = 7x - 2$$

$$y = 2x + 5$$

Will you write three separate functions?

Overloading the () operator

```
class LineEquation
{
    int m;
    int c;
public:
    LineEquation(int a, int b)
    {
        m = a;
        c = b;
    }

    int operator()(int x)
    {
        return m * x + c;
    }
};
```

y = 4x + 3
y = 7x - 2
y = 2x + 5

Overloading the () operator

- Enabling the object to act like a function
$$\text{obj1(param1, param2,...)}$$
- Must be a member function of the class
- It can have any number of parameters and any return type
- The object works like a programmable function

Overloading the () operator

$$\begin{aligned}y &= 4x + 3 \\y &= 7x - 2 \\y &= 2x + 5\end{aligned}$$

```
int main()
{
    LineEquation line1(4, 3);
    LineEquation line2(7, -2);
    LineEquation line3(2, 5);

    cout << "Points of line1:" << endl;
    for (int i = 1; i<5; i++)
    {
        cout << "(" << i << ", " << line1(i) << ")" << endl;
    }

    //similar for line2 and line3
}
```

Functor (Function Object)

Yes, you've read it right. Functor.

- Functor is a C++ class that acts like function.
- It's a class where operator () is defined.
- line1, line2, line3 in the previous example are Functors.

Functor vs Function Pointer

- Functors are more efficient than Function Pointer. Function pointer may require runtime pointer dereferencing.
- Functor can contain state

Conversion Function

When we want to convert an object of one type to another

Conversion Function

When we want to convert an object of one type to another

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};
```

Conversion Function

When we want to convert an object of one type to another

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};

int main()
{
    Subject cse205(80, 80);
```

Conversion Function

When we want to convert an object of one type to another

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

}
```

Conversion Function

When we want to convert an object of one type to another

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks;      //160
}
```

Conversion Function

Syntax: operator type() { return value; }

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }

    operator int()
    {
        return partI + partII;
    }
};
```

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks; //160
}
```

Conversion Function

Syntax: `operator type() { return value; }`

- `operator` and `return` are keywords
 - `type` is the target type we'll be converting our object to
 - `value` is the value of the object after the conversion has been performed.
-
- Returns a value of type `type`
 - No parameter can be specified
 - Conversion function must be a member function

Overloading new and delete

```
void * operator new (size_t count);  
void operator delete (void * ptr);
```

Overloading new and delete

```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void * operator new (size_t sz)
    {
        cout << "mem allocated" << endl;
        void * p = malloc(sz);
        return p;
    }
    void operator delete (void * p)
    {
        cout << "mem deallocated" << endl;
        free(p);
    }
};
```

```
int main()
{
    Point * pt = new Point(1,2);
    pt->display();
}
```

Overloading new [] and delete []

```
void * operator new [] (size_t count);  
void operator delete [] (void * ptr);
```

Overloading new [] and delete []

```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }

    void * operator new (size_t sz)
    {
        cout << "mem allocated" << endl;
        void * p = malloc(sz);
        return p;
    }

    void * operator new [] (size_t sz)
    {
        cout << "array mem allocated" << endl;
        void * p = malloc(sz);
        return p;
    }
}
```

Overloading new [] and delete []

Cont...

```
void operator delete (void * p)
{
    cout << "mem deallocated" << endl;
    free(p);
}

void operator delete [](void * p)
{
    cout << "array mem deallocated" << endl;
    free(p);
}

void display()
{
    cout << "(" << x << ", " << y << ")" << endl;
}

int main()
{
    Point * pt = new Point[2] {Point(1,2), Point(3,4)};
    pt[0].display();
    pt[1].display();
    delete [] pt;
}
```

References

- www.cs.bu.edu/fac/gkollios/cs113/Slides/lecture12.ppt
- Teach Yourself C++, 3rd Ed. By Herb Schildt (Chapter 6)
- www.tutorialspoint.com/cplusplus/cpp_overloading.htm
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading

Thank you

Type Introspection

CSE 205 - Week 12 Class 2 & 3

Lec Raiyan Rahman

Dept of CSE, MIST

raian@cse.mist.ac.bd

CC: Lec Saidul Hoque Anik



Type Introspection

- Ability to inspect a type
- Retrieve its various qualities

The typeinfo library

To obtain the type of object

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

int main()
{
    int a;
    cout << typeid(a).name() << endl;
    return 0;
}
```



Returns a reference to an object of type: `type_info`
Its `name()` function returns a pointer to the name of the type.

Non-Polyomophic Language | Polymorphic Language

Type of each object is known at compile time.

Eg. : C

Type of each object can be unknown at compile time.

(Remember Late Binding?)

Eg.: C++

Run Time Type Identification

```
class Base
{
    // ...
};

class Derived : public Base
{
    // ...
};

Base * ptr;
```

Run Time Type Identification

```
class Base
{
    // ...
};

class Derived : public Base
{
    // ...
};

Base * ptr;

ptr = new Base();
```

Run Time Type Identification

```
class Base
{
    // ...
};

class Derived : public Base
{
    // ...
};

Base * ptr;

ptr = new Derived();
```

The type of object ptr must be determined during runtime

RTTI

In C++, it's possible to identify the type of object a pointer is pointing to during Run time using RTTI mechanism.

This feature is only available for **polymorphic classes** (A class hierarchy where at least one virtual function is present). For non-polymorphic class, pointer to derived class will return the base type.

In practice, this is not a limitation because base classes must have a virtual destructor to allow objects of derived classes to perform proper cleanup if they are deleted from a base pointer.

Non-Polymeric Class

No virtual function is present

Polymorphic Class

At least one virtual function is present

Example

```
#include <iostream>      // cout
#include <typeinfo>      // for 'typeid'

class Person
{
public:
    virtual ~Person() {}

class Employee : public Person
{
```

Example

```
#include <iostream>          // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person person;
```

Example

```
#include <iostream>          // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person person;

std::cout << typeid(person).name() << std::endl;
```

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person person;
```

```
std::cout << typeid(person).name() << std::endl;
                                         // Person (statically known at compile-time)
```

Example

```
#include <iostream>          // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Employee employee;

std::cout << typeid(employee).name() << std::endl;
```

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Employee employee;

std::cout << typeid(employee).name() << std::endl;
// Employee (statically known at compile-time)
```

Example

```
#include <iostream>      // cout
#include <typeinfo>       // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person* ptr = &employee;
```

```
std::cout << typeid(ptr).name() << std::endl;
```

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person* ptr = &employee;
```

```
std::cout << typeid(ptr).name() << std::endl;
```

// Person (statically known at compile-time)*

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person* ptr = &employee;
```

```
std::cout << typeid(*ptr).name() << std::endl;
```

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person* ptr = &employee;
```

```
std::cout << typeid(*ptr).name() << std::endl;
```

// Employee (Looked up dynamically at run-time
// because it is the dereference of a
// pointer to a polymorphic class)

Example

```
#include <iostream>           // cout
#include <typeinfo>           // for 'typeid'

class Person
{
public:
    virtual ~Person() {}
};

class Employee : public Person
{
};
```

```
Person& ref = employee;
```

```
std::cout << typeid(ref).name() << std::endl;
```

// Employee (references can also be polymorphic)

References

- https://en.wikipedia.org/wiki/Run-time_type_information
- Teach Yourself C++ by Herbert Schildt (Chapter 12.1)

Thank you

Understanding Streams & File I/O in C++

CSE 205

[Lec Raiyan Rahman](#)

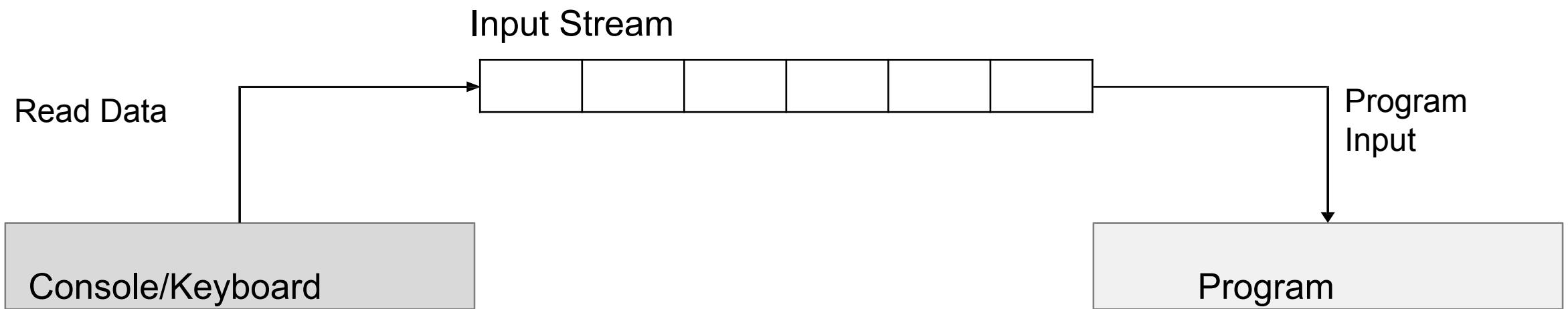
Dept of CSE, MIST

raian@cse.mist.ac.bd

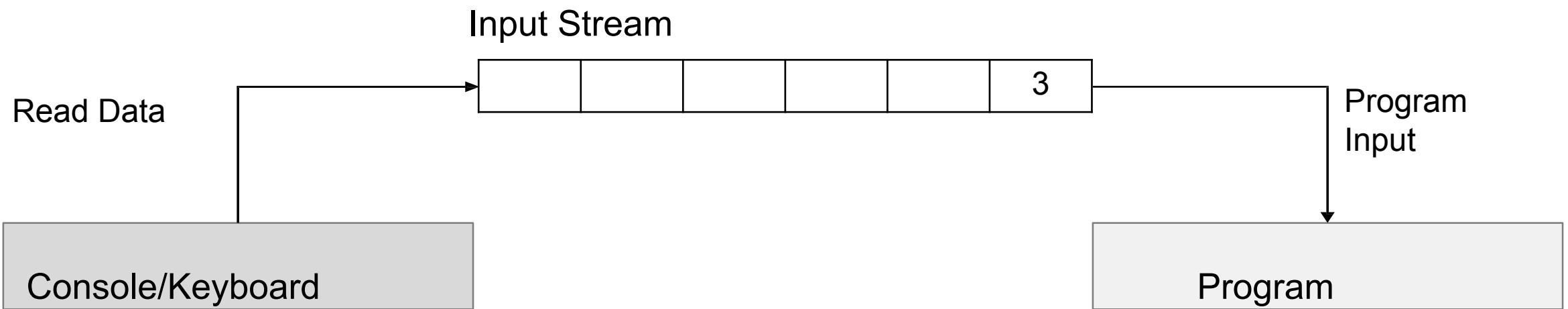
CC: Lec Saidul Hoque Anik



Understanding Input Stream

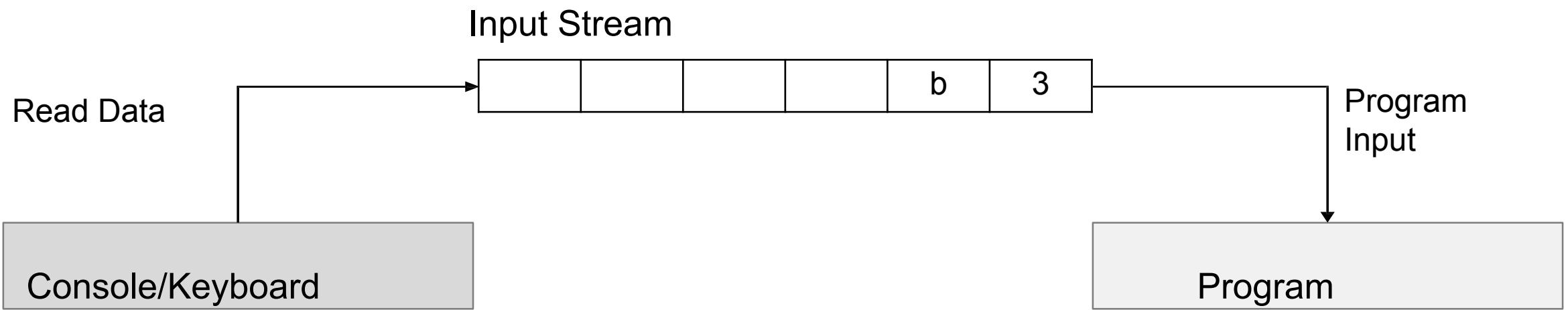


Understanding Input Stream



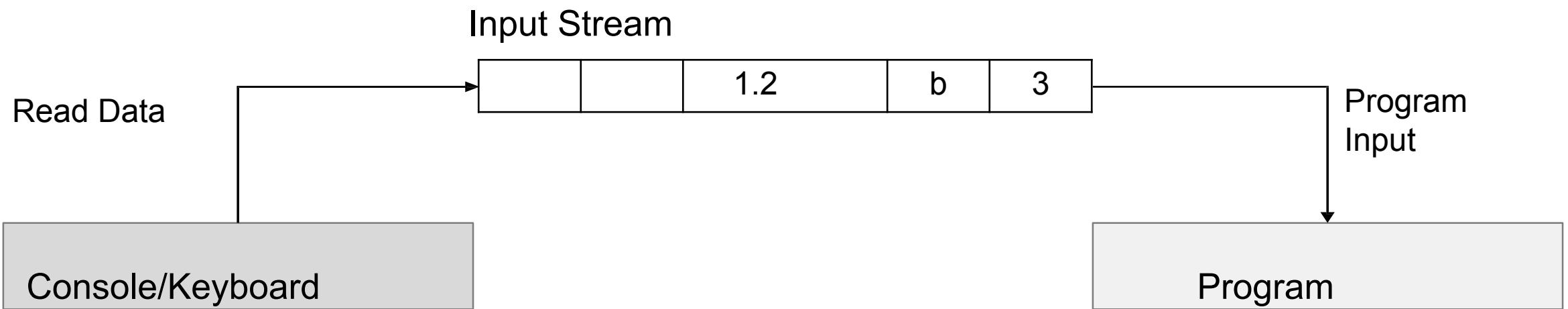
Typed: 3 in console

Understanding Input Stream



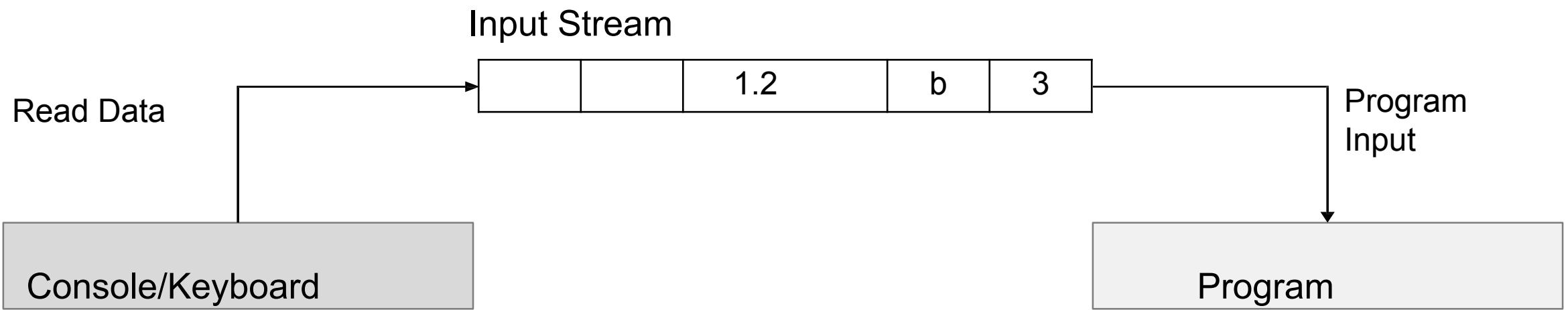
Typed: b in console

Understanding Input Stream



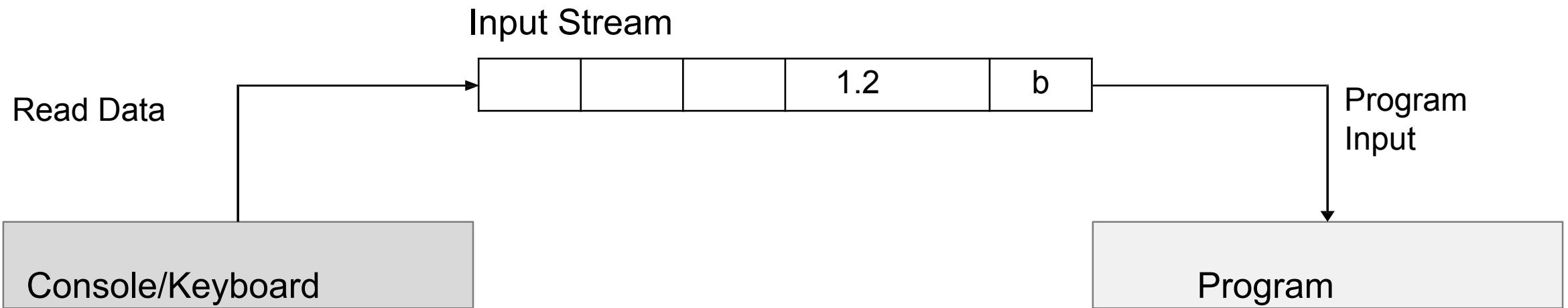
Typed: 1.2 in console

Understanding Input Stream



```
int a;
```

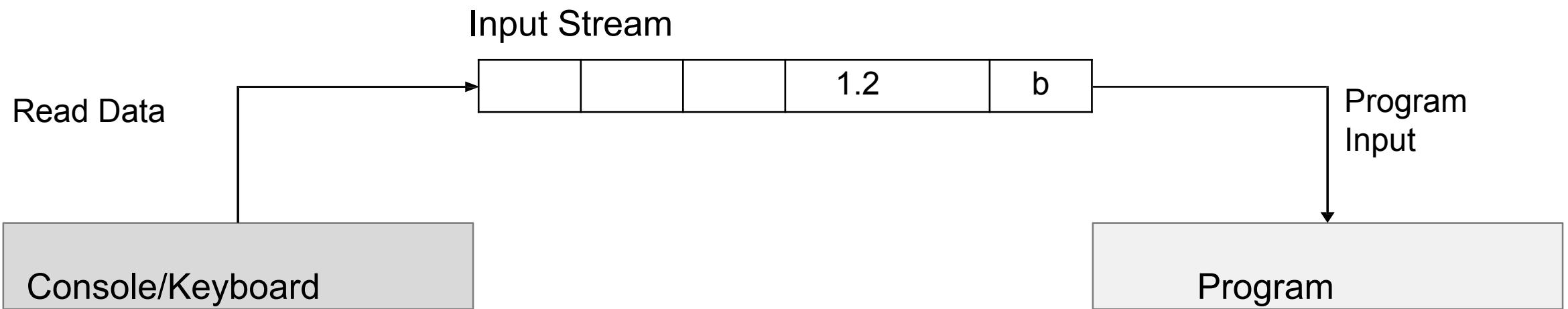
Understanding Input Stream



```
int a;  
cin >> a;
```

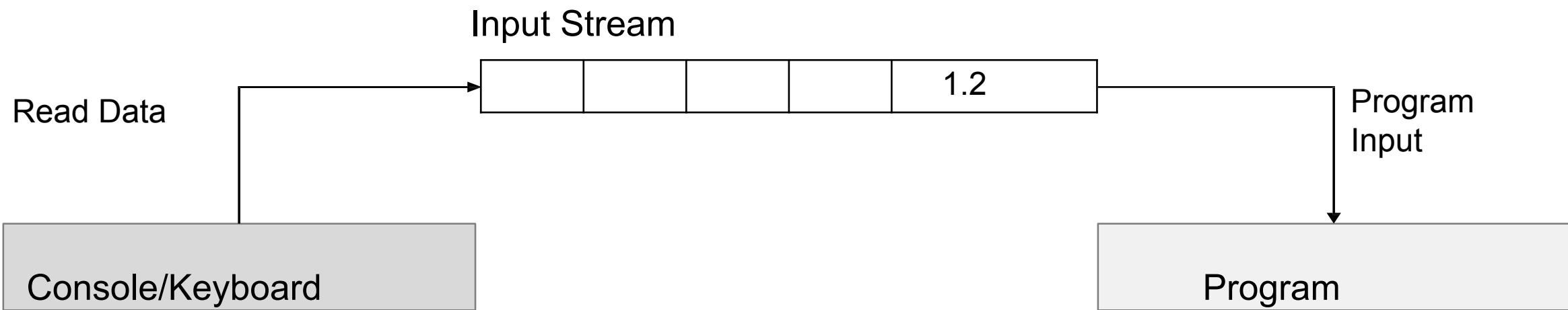
//Now a = 3

Understanding Input Stream



```
char ch;
```

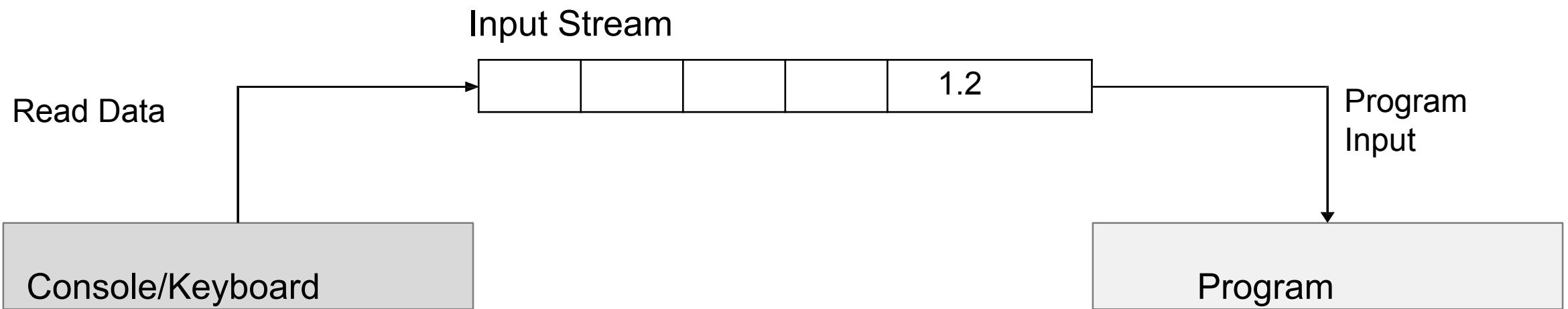
Understanding Input Stream



```
char ch;  
cin >> ch;
```

```
//ch = 'b'
```

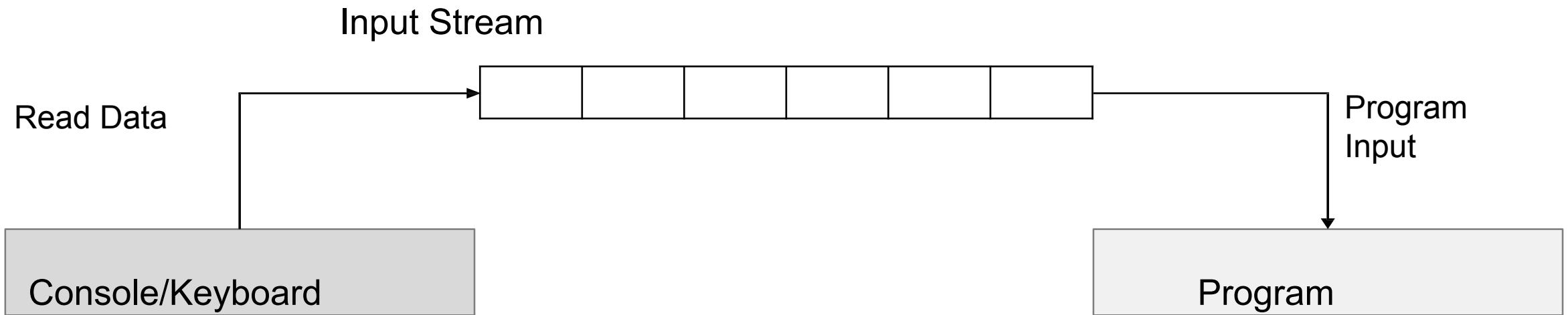
Understanding Input Stream



```
float f;
```

Understanding Input Stream

Different source, same interface

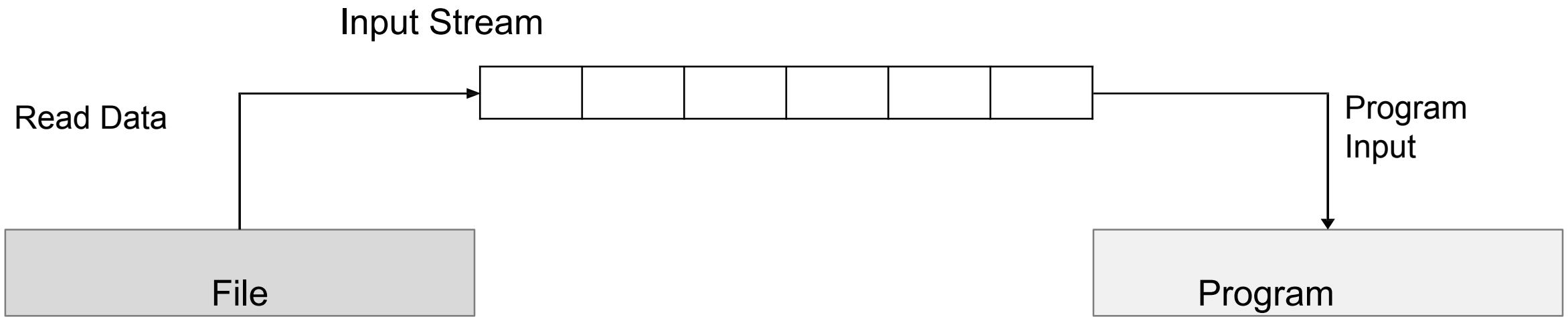


*Standard Input Stream Default
Source: Keyboard*

```
float f;  
cin >> f;  
//f = 1.2
```

Understanding Input Stream

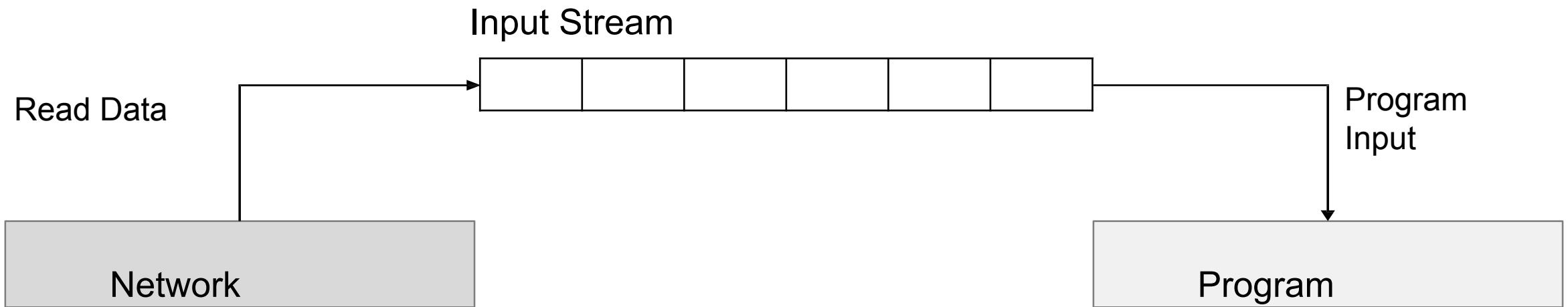
Different source, same interface



*File Input Stream
Source: Disk File*

Understanding Input Stream

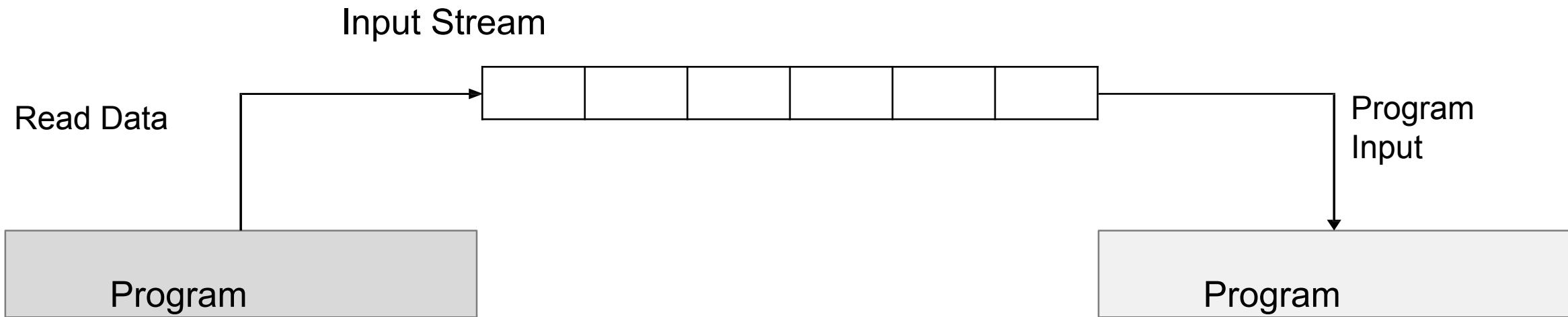
Different source, same interface



*Network Input Stream
Source: Socket*

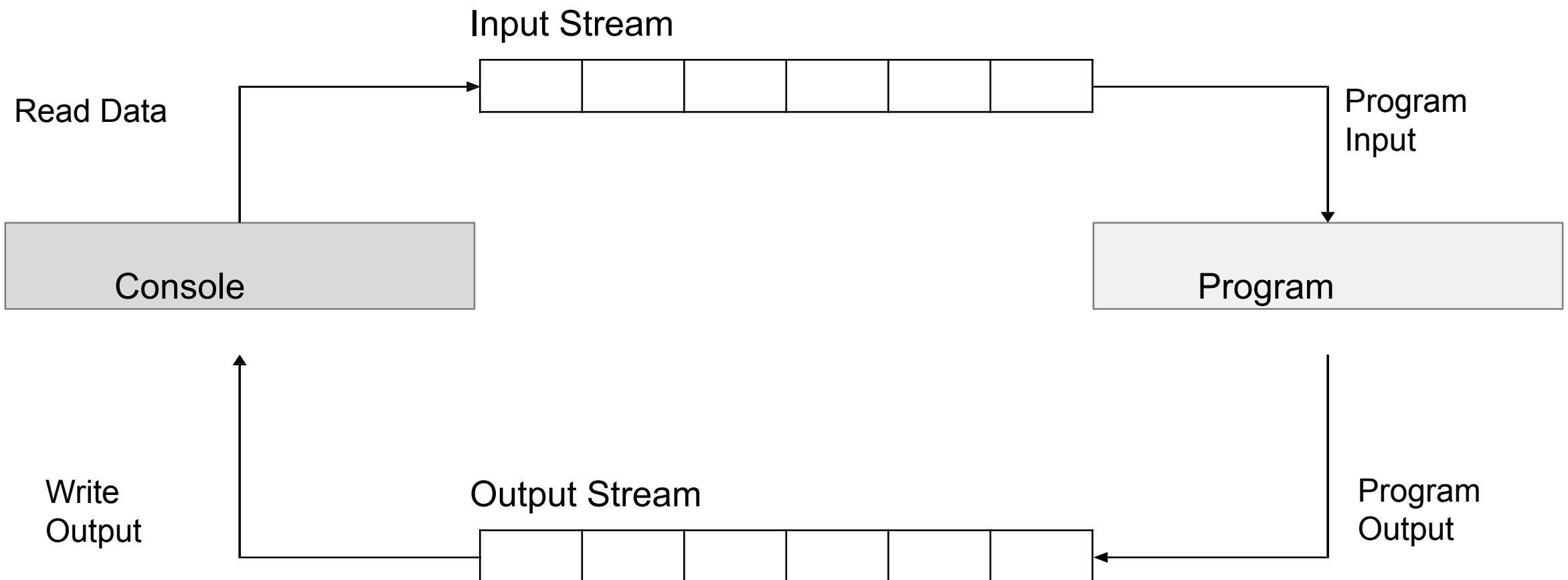
Understanding Input Stream

Different source, same interface

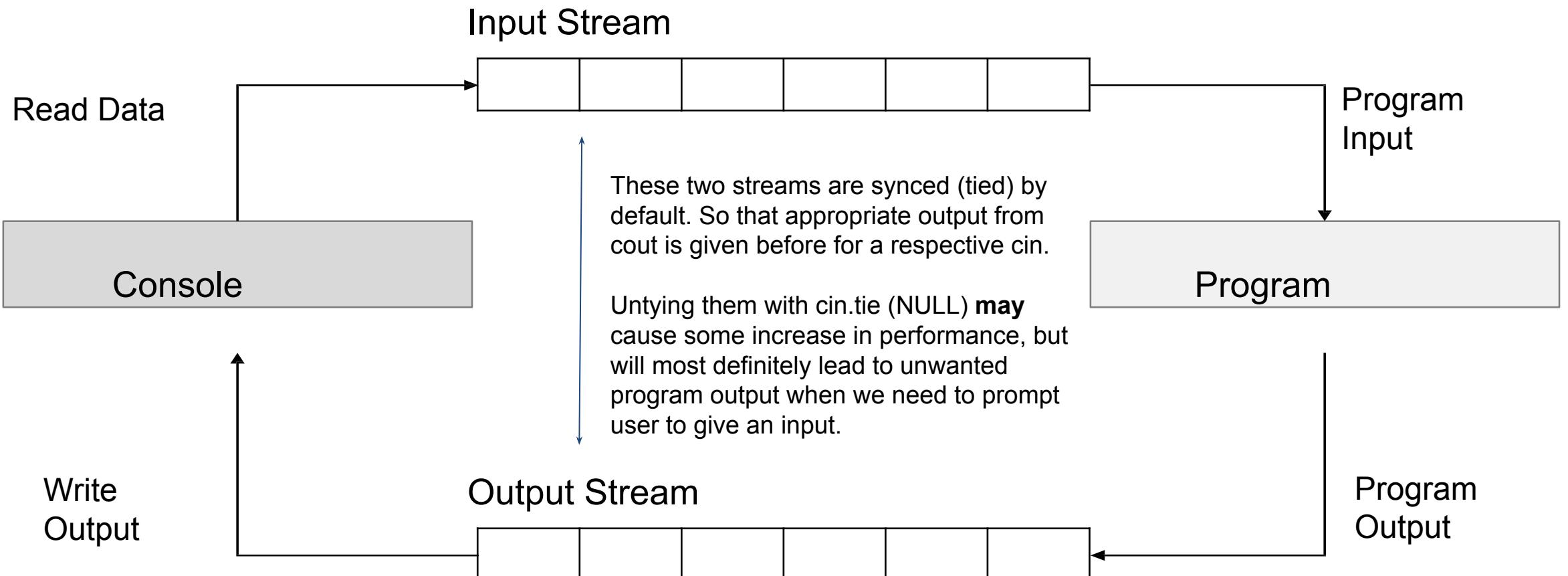


*String Stream
Source: String*

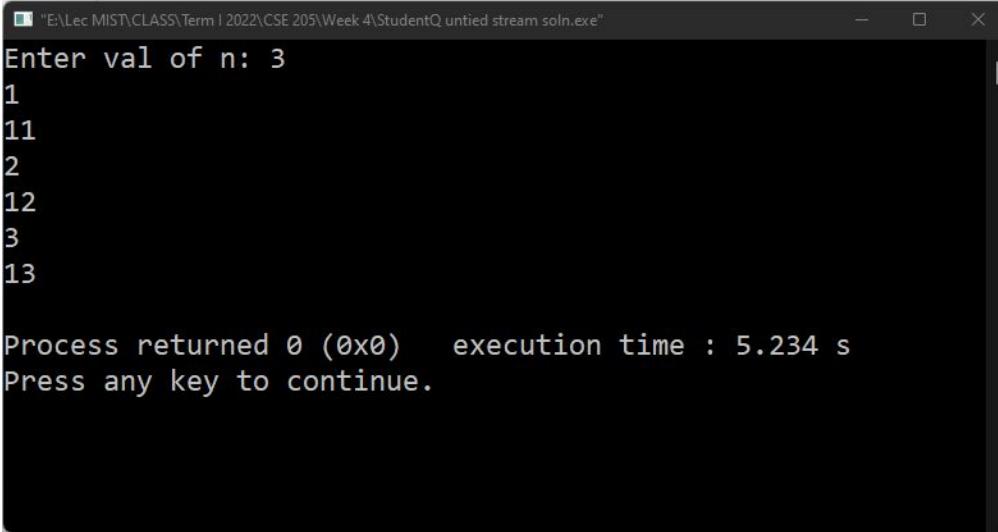
IO Stream



IO Stream



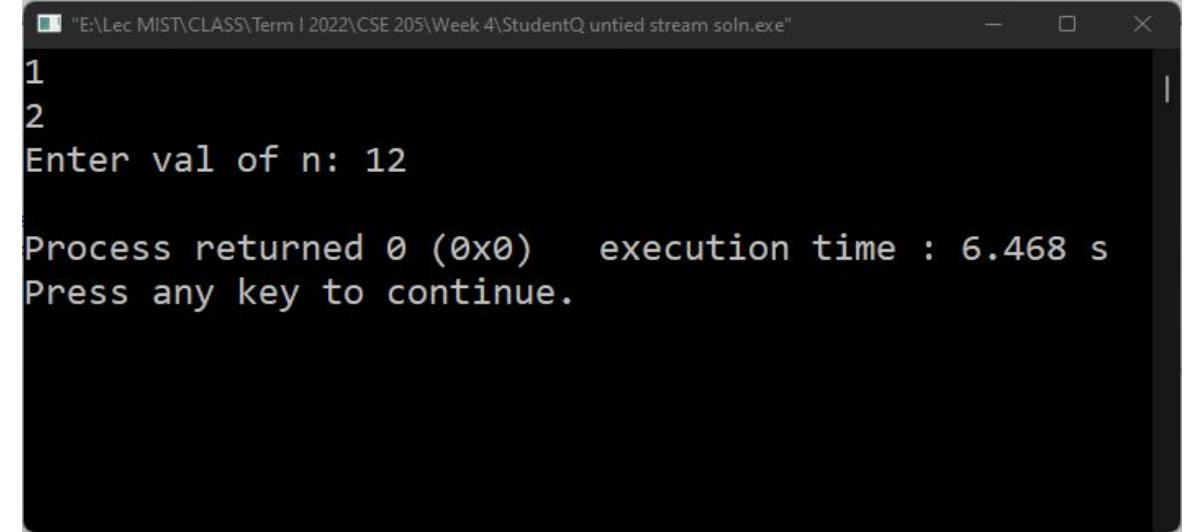
Tied vs Untied Cin and Cout streams



```
"E:\Lec MIST\CLASS\Term I 2022\CSE 205\Week 4\StudentQ untied stream soln.exe"
Enter val of n: 3
1
11
2
12
3
13

Process returned 0 (0x0) execution time : 5.234 s
Press any key to continue.
```

When streams are **tied**



```
"E:\Lec MIST\CLASS\Term I 2022\CSE 205\Week 4\StudentQ untied stream soln.exe"
1
2
Enter val of n: 12

Process returned 0 (0x0) execution time : 6.468 s
Press any key to continue.
```

When streams are **not tied**. See how the “Enter val of n” is printed later even though we put the cout first in our program.

We can use std::flush to forcefully print the current cout buffer to the screen. Which would be done automatically with the corresponding cin, if we hadn't untied the streams.

Syncing the C and C++ streams

The I/O streams of both are **synced by default**. That is we use the same target resource for both printf() & cout, scanf() & cin. This has a higher I/O time but ensures compatibility.

We may delete this synchronization using **ios_base::sync_with_stdio(false)**; in hopes of runtime efficiency.

However, that will have unforeseen consequences when using both streams in the same program. For example, variable taken as a input with scanf() may not be the same as one with cin even if they are of the same data type.

But, you should be just fine if you use only cout and cin throughout the program.

Note: These are techniques used in competitive programming to speed up large I/O operations. No need to worry about it in regular programs.
For further reading, [click here](#).

Standard Streams

C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**.

cerr vs clog

clog and cerr, both are associated with stderr, but it differs from cerr in the sense that the streams in **clog** **are buffered** and not automatically tied with cout.

Buffered output is more efficient than unbuffered output. In the case of buffered output, all the output is saved into a variable and written to disk all at once. For unbuffered output, we have to keep writing to disk.

Buffered output isn't preferred for critical errors. In case of system crash, there can come a situation where the output was still in buffer and wasn't written to disk and the error message cannot be retrieved. We cannot afford to lose error data in case of system crash so we keep writing the critical errors to disc even though it is slower.

Reading a line

cin by default terminates a string input when a **whitespace is found**

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

int main()
{
    char c[100];
    cin >> c;
    //Hello World
    cout << c << endl;
    //Hello
}
```

Reading a line

cin.get() terminates a string input when a newline is found

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

int main()
{
    char c[100];
    cin.get(c, 100);
    // "Hello Wor     ld      !"
    cout << c;
    // "Hello Wor     ld      !"
}
```

How many characters to read from stream/buffer



Reading a line

getline() reads a line into a string

```
int main()
{
    string s;
    getline(cin, s);
    cout << s;
}
```

Formatted I/O

Formatting using flags

	Flags	Remark
basefield	ios::oct	
	ios::dec	
	ios::hex	
adjustfield	ios::left	left justified
	ios::right	right justified
	ios::internal	fill between +/- or hex notation
floatfield	ios::scientific	2E10
	ios::fixed	2.12
Others	ios::boolalpha ios::showbase ios::showpoint ios::showpos ios::skipws ios::uppercase	show true/false instead of 1/0 show 0x or 0 at beginning show trailing zeros 25 will be +25 skip beginning whitespace for cin 2e10 will be 2E10

Formatted I/O

To set a flag, use `setf()`. To unset a flag, use `unsetf()`

```
using namespace std;

int main()
{
    bool b = 0;
    cout.setf(ios::boolalpha);
    cout << b << endl; //false
    cout.unsetf(ios::boolalpha);
    float a = 1.2;
    cout << b << endl; //0
    cout.setf(ios::showpoint);
    cout << a << endl; //1.20000
    cout.unsetf(ios::showpoint);
    cout << a; //1.2
}
```

More formatting

Using width(), precision(), fill()

```
int main()
{
    float f = 1.0/3.0;
    cout << f << endl;

    cout.precision(10);
    cout << f << endl;
}
```

More formatting

Using width(), precision(), fill()

```
int main()
{
    cout.width(10);
    cout.fill('-');
    cout.setf(ios::right);
    cout << "Hello";      //-----Hello
}
```

Manipulators

Another way of formatting

```
int main()
{
    int i = 50;
    cout << i << endl;
    cout << i*2 << endl;
}
```

Manipulators

Another way of formatting

```
int main()
{
    int i = 50;
    cout << i << endl;
    cout << i*2 << endl;
}
```

This is called manipulator

Manipulators

Another way of formatting

```
int main()
{
    int i = 12;
    cout << "Hex value of " << i << " is ";
    cout << hex << i;
}
```



This is called manipulator

Manipulators

Another way of formatting

```
int main()
{
    bool b = 0;
    cout << boolalpha << b << endl; //false

    int i = 12;
    cout << "Hex value of " << i << " is ";
    cout << hex << i; //c
}
```

Manipulators

Another way of formatting

Manipulator	Purpose	Input/Output
boolalpha	Turns on boolalpha flag.	Input/Output
dec	Turns on dec flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on fixed flag.	Output
flush	Flush a stream.	Output
hex	Turns on hex flag.	Input/Output
internal	Turns on internal flag.	Output
left	Turns on left flag.	Output
nobooalpha	Turns off boolalpha flag.	Input/Output
noshowbase	Turns off showbase flag.	Output
noshowpoint	Turns off showpoint flag.	Output
noshowpos	Turns off showpos flag.	Output

Table 20-1. *The C++ Manipulators*

Manipulators

Another way of formatting

Manipulator	Purpose	Input/Output
<code>noskipws</code>	Turns off <code>skipws</code> flag.	Input
<code>nounitbuf</code>	Turns off <code>unitbuf</code> flag.	Output
<code>nouppercase</code>	Turns off <code>uppercase</code> flag.	Output
<code>oct</code>	Turns on <code>oct</code> flag.	Input/Output
<code>resetiosflags (fmtflags f)</code>	Turn off the flags specified in <i>f</i> .	Input/Output
<code>right</code>	Turns on <code>right</code> flag.	Output
<code>scientific</code>	Turns on <code>scientific</code> flag.	Output
<code>setbase(int base)</code>	Set the number base to <i>base</i> .	Input/Output
<code>setfill(int ch)</code>	Set the fill character to <i>ch</i> .	Output
<code>setiosflags(fmtflags f)</code>	Turn on the flags specified in <i>f</i> .	Input/output
<code>setprecision (int p)</code>	Set the number of digits of precision.	Output
<code>setw(int w)</code>	Set the field width to <i>w</i> .	Output
<code>showbase</code>	Turns on <code>showbase</code> flag.	Output
<code>showpoint</code>	Turns on <code>showpoint</code> flag.	Output
<code>showpos</code>	Turns on <code>showpos</code> flag.	Output
<code>skipws</code>	Turns on <code>skipws</code> flag.	Input
<code>unitbuf</code>	Turns on <code>unitbuf</code> flag.	Output
<code>uppercase</code>	Turns on <code>uppercase</code> flag.	Output
<code>ws</code>	Skip leading white space.	Input

Table 20-1. The C++ Manipulators (continued)

File IO

Writing in a file

File Output Stream

Writing in a file

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

int main()
{
    ofstream of;
    of.open("file.txt");
    of << "This is the first line";
    of << "\nThis is 2nd line";

    of.close();
}
```

input.txt

```
This is the first line
This is 2nd line
```

If the stream is already associated with a file (i.e., it is already *open*), calling this function fails.

So, we should always close the filestream after using it, especially if we want to open it again in our prog in a different mode later.

File Output Stream

Writing in a file

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ofstream of;

    of.open("file.txt");
    of << "This is the 3rd line" << endl;
    of.close();
}
```

input.txt

This is the 3rd line

Each time we are opening a file, the file is emptied, and output is written from beginning.

File Output Stream

To append to a file, we'll need to open it in append mode

*Append mode opens the file keeping the contents intact, and only writes new content at the end (So, it **appends**).*

```
of.open("file.txt", ios::app);
```



Other file opening modes

member constant	stands for	access
in	input	File open for reading: the <i>internal stream buffer</i> supports input operations.
out	output	File open for writing: the <i>internal stream buffer</i> supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The <i>output position</i> starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Using this mode would be sufficient for what we regularly do.

Other file opening modes

member constant	stands for	access
in	input	File open for reading: the <i>internal stream buffer</i> supports input operations.
out	output	File open for writing: the <i>internal stream buffer</i> supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The <i>output position</i> starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Only diff is, special characters such as \n will not be automatically converted

Optional Reading: ios::ate vs ios::app

ios::ate "sets the stream's position indicator to the end of the stream on opening."

ios::app "set the stream's position indicator to the end of the stream before each output operation."

This means the difference is that ios::ate puts your position to the end of the file when you open it. ios::app instead puts it at the end of the file every time you flush your stream.

If for example you two programs that write to the same log file ios::ate will override anything that was added to the file by the other program since your program opened it. ios::app will instead jump to the end of file each time your program adds a log entry.

File Input Stream

For writing, we used ‘ofstream’.

For reading, we’ll be using ‘ifstream’

File Input Stream

Reading a single line from file

```
ifstream ifs;  
ifs.open("file.txt");  
  
string line;  
getline(ifs, line);  
cout << line;
```

File Input Stream

Reading word by word line from file

A stream returns false when EOF (end of file) is reached.
So, the while loop will continue until there are no more words
left in the file we're reading.

```
string line;
while (ifs >> line)
{
    cout << line << endl;
}
```

File Input Stream

Reading all lines from file

```
string line;
while (getline(ifs, line))
{
    cout << line << endl;
}
```

A function to check End Of File

peek() reads the next character without extracting it

```
while (ifs.peek() != EOF)
{
    getline(ifs, line);
    cout << line << endl;
}
```

Notice that getline() is keeping the newline character to itself, so we have to explicitly put << endl at the back.

A function to check End Of File

If we want to keep the whitespace, we'll have to use a manipulator called 'noskipws'

```
char ch;  
while (ifs.peek() != EOF)  
{  
    ifs >> noskipws >> ch;  
    cout << ch;  
}
```

Now, let's say we want to save an object to a file.

But objects aren't of fixed length like basic Data Types.
The first question is, how do we know how much data
(size) to save for each object?

Writing objects in a file

We'll use `read()` and `write()` binary function do achieve “unformatted” writing.

Binary I/O Functions	Description
<code>read()</code>	This binary function is used to perform <i>file input operation</i> i.e. to read the objects stored in a file.
<code>write</code>	This binary function is used to perform <i>file output operation</i> i.e. to write the objects to a file, which is stored in the computer memory in a binary form . <i>Only the data member of an object are written and not its member functions</i>

Note: The content of the file created using the `write` and `read` function may not be in a human-readable form because we have written an object to the file using binary functions.

Writing objects in a file

Unformatted writing. The write() function

```
ostream& write (const char* s, streamsize n);
```

Inserts the first *n* characters of the array pointed by s into the stream.

Writing objects in a file

Unformatted writing. The write() function

```
struct Student
{
    char name[15];
    int roll;
    void display()
    {
        cout << name << " " << roll << endl;
    }
};
```

Writing objects in a file

Unformatted writing. The write() function

```
int main()
{
    Student s1;
    s1.roll = 50;
    strcpy(s1.name, "student1");

}
```

Writing objects in a file

Unformatted writing. The write() function

```
int main()
{
    Student s1;
    s1.roll = 50;
    strcpy(s1.name, "student1");

    ofstream out("record.txt", ios::out);

}
```

Writing objects in a file

What is the size of s1?

```
struct Student
{
    char name[15];
    int roll;
    void display()
    {
        cout << name << " " << roll << endl;
    }
};
```

Writing objects in a file

What is the size of s1?

```
int main()
{
    Student s1;
    s1.roll = 50;
    strcpy(s1.name, "student1");

    ofstream out("record.txt", ios::out);
}
```

```
s1
{
    char name[15];
    int roll;
};
```


Writing objects in a file

Unformatted writing. The write() function

```
int main()
{
    Student s1;
    s1.roll = 50;
    strcpy(s1.name, "student1");

    ofstream out("record.txt", ios::out);
    out.write((char *) &s1, sizeof(s1));
}
```

Compare this with the write() function's signature

ostream& write (const char* s, streamsize n);

Writing objects in a file

Unformatted writing. The write() function

```
int main()
{
    Student s1;
    s1.roll = 50;
    strcpy(s1.name, "student1");

    ofstream out("record.txt", ios::out);
    out.write((char *)&s1, sizeof(s1));
}

}
```

each byte of s1 is read by the write function using the char *, passed as the first argument.

s1

sizeof s1 is 20 bytes.
It's 15 byte of char + 1 byte padding + 4 byte of int (roll).
This is then saved to the file.

Reading objects from a file

Unformatted reading of blocks of data. The `read()` function.

```
istream& read (char* s, streamsize n);
```

Extracts `n` characters from the stream and stores them in the array pointed to by `s`.

So, it does the opposite job of `write()`.

Reading objects from a file

Unformatted reading of blocks of data. The read() function.

```
int main()
{
    Student s1;
    ifstream in("record.txt", ios::in);
    in.read((char *) &s1, sizeof (s1));

}
```

Reading objects from a file

Unformatted reading of blocks of data. The read() function.

```
int main()
{
    Student s1;
    ifstream in("record.txt", ios::in);
    in.read((char *) &s1, sizeof (s1));
    s1.display();
}
```

Task

- Follow the same read/write routine for an **array of objects**.
- Design a record book manager, which will be able to add, edit and delete records of students dynamically. After each operation, the changes will be saved in the file.

Hint:

Use `read()` and `write()` for array of objects: [Tutorial](#)

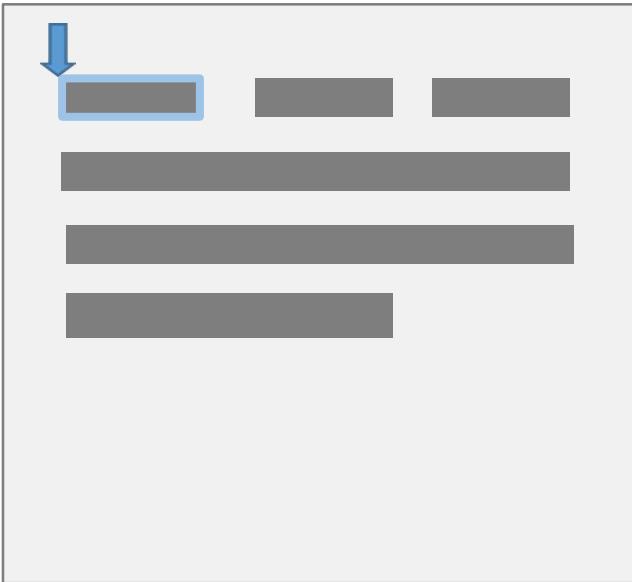
Question

Can we follow the read/write mechanism when our class/struct contains dynamic data types such as string?

Sequential File Access

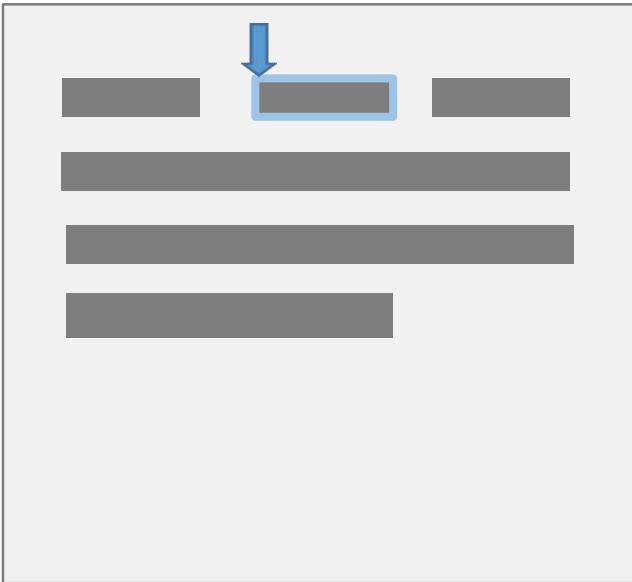
Up until now, our file reading process was sequential

```
ifstream in;  
in.open("input.txt");
```



Sequential File Access

Up until now, our file reading process was sequential

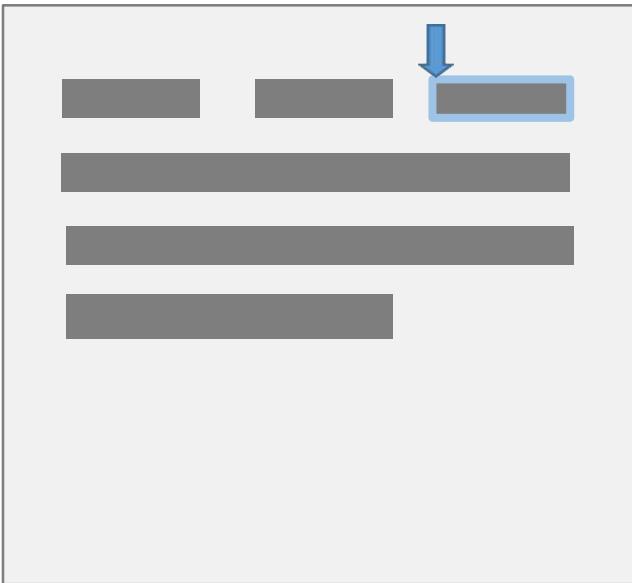


```
string      first_word;  
in  >> first_word;
```

```
//'in' is now pointing  
//at the second word
```

Sequential File Access

Up until now, our file reading process was sequential



```
string      second_word;  
in  >> second_word;  
//‘in’ is now pointing  
//at the third word
```

Random Access in file

Let's write A to Z in a txt file

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

int main()
{
    ofstream out;
    out.open("alphabet.txt", ios::out);
    for (int i = 0; i<26; i++)
    {
        out << (char) (i + 'A');
    }
    out.close();
}
```

Random Access in file

If we now open the file and read a character, 'A' will be displayed.

```
ifstream in;  
in.open("alphabet.txt");  
  
char ch;  
in >> ch;  
cout << ch; //A  
  
in.close();
```

How a file is read

alphabet.txt



ABCDEFGHIJKLMNOPQRSTUVWXYZ

When we are opening a file, by default the read pointer (or the 'get' pointer) points to the beginning of the file.

How a file is read

alphabet.txt



ABCDEFGHIJKLMNOPQRSTUVWXYZ

When

`in>>ch;`

is executed, one character is read, and the pointer advances one byte (single character)

Random Access in file: Functions to remember!

Accessing from the middle of a file instead of sequential access

Functions:

- `tellg()` - file pointer position of where next char will be read
- `tellp()` - file pointer position of where next char will be written
- `seekg()` - Move the reading file pointer
- `seekp()` - Move the writing file pointer

Introduction to Random Access

alphabet.txt



ABCDEFGHIJKLMNOPQRSTUVWXYZ

In order to move the get pointer, we use a function called seekg()

If we write `in.seekg (5, ios::beg);` The pointer will advance 5 character (5 byte) from the beginning.

Introduction to Random Access

alphabet.txt



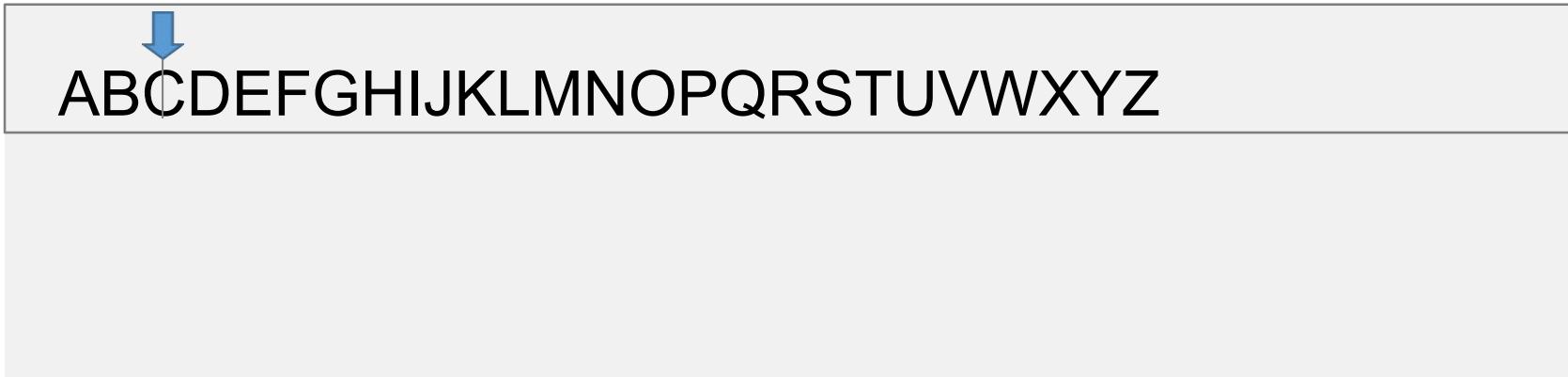
ABCDEFGHIJKLMNOPQRSTUVWXYZ

In order to move the get pointer, we use a function called seekg()

If we write `in.seekg(5, ios::beg);` The pointer will advance 5 character (5 byte) from the beginning.

Introduction to Random Access

alphabet.txt

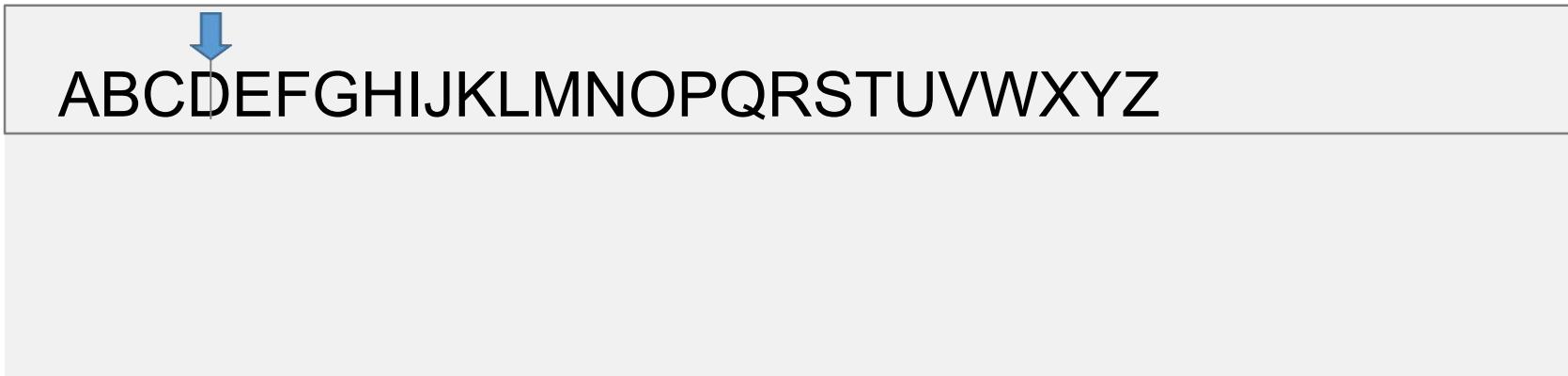


In order to move the get pointer, we use a function called seekg()

If we write `in.seekg(5, ios::beg);` The pointer will advance 5 character (5 byte) from the beginning.

Introduction to Random Access

alphabet.txt



In order to move the get pointer, we use a function called seekg()

If we write `in.seekg(5, ios::beg);` The pointer will advance 5 character (5 byte) from the beginning.

Introduction to Random Access

alphabet.txt



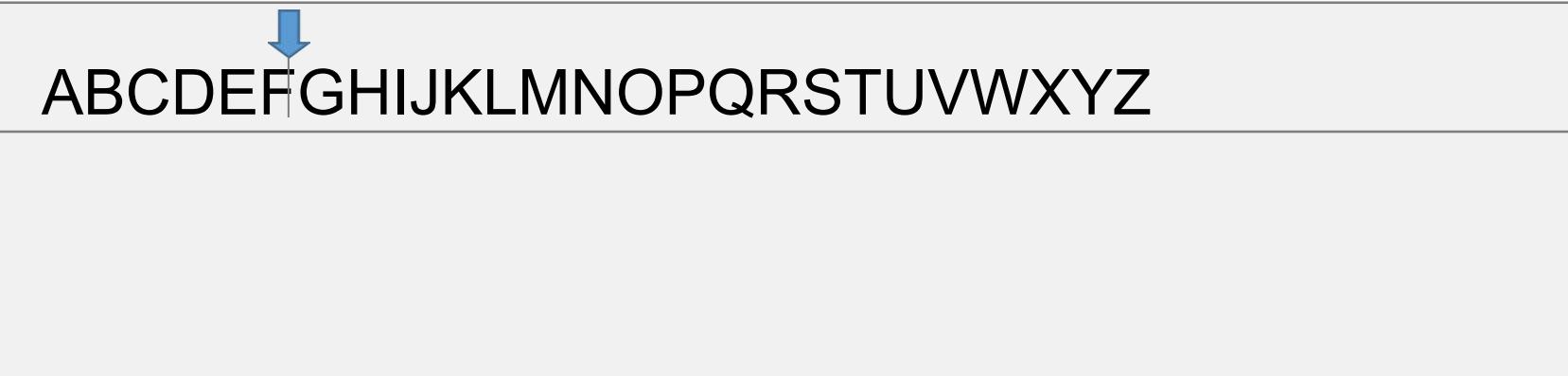
ABCDEFGHIJKLMNOPQRSTUVWXYZ

In order to move the get pointer, we use a function called seekg()

If we write `in.seekg(5, ios::beg);` The pointer will advance 5 character (5 byte) from the beginning.

Introduction to Random Access

alphabet.txt



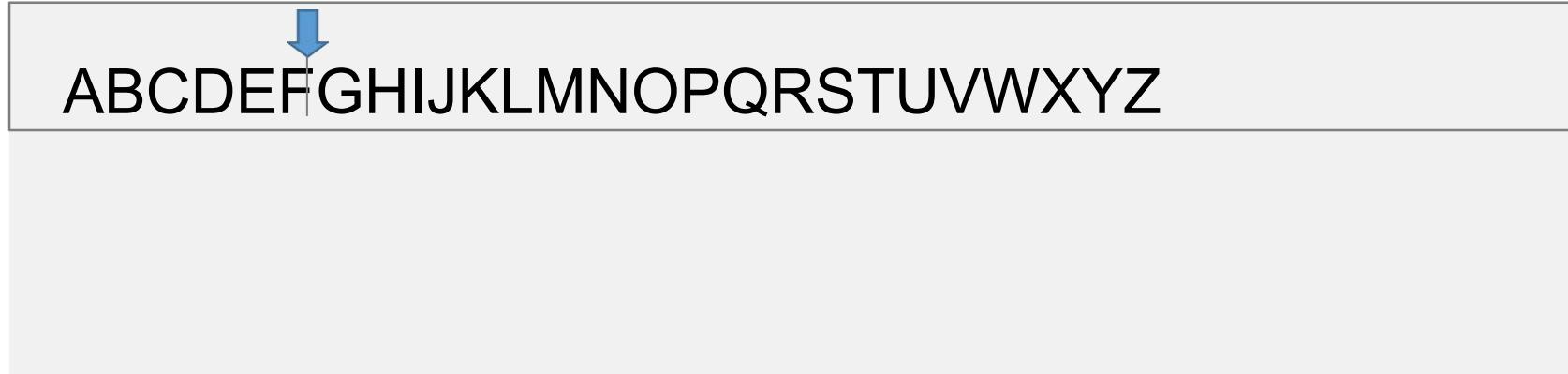
ABCDEF**G**HJKLMNOPQRSTUVWXYZ

What will be the output of the following code now?

```
in  >> ch; cout    << ch;
```

Introduction to Random Access

alphabet.txt

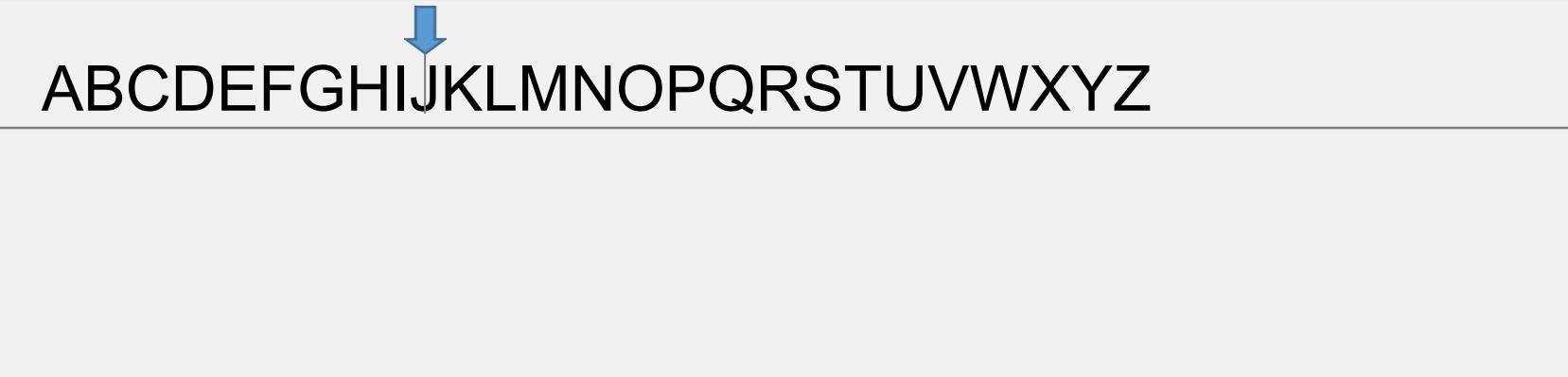


What will be the pointer position if we write the following?

```
in.seekg(10, ios::beg);
```

Introduction to Random Access

alphabet.txt



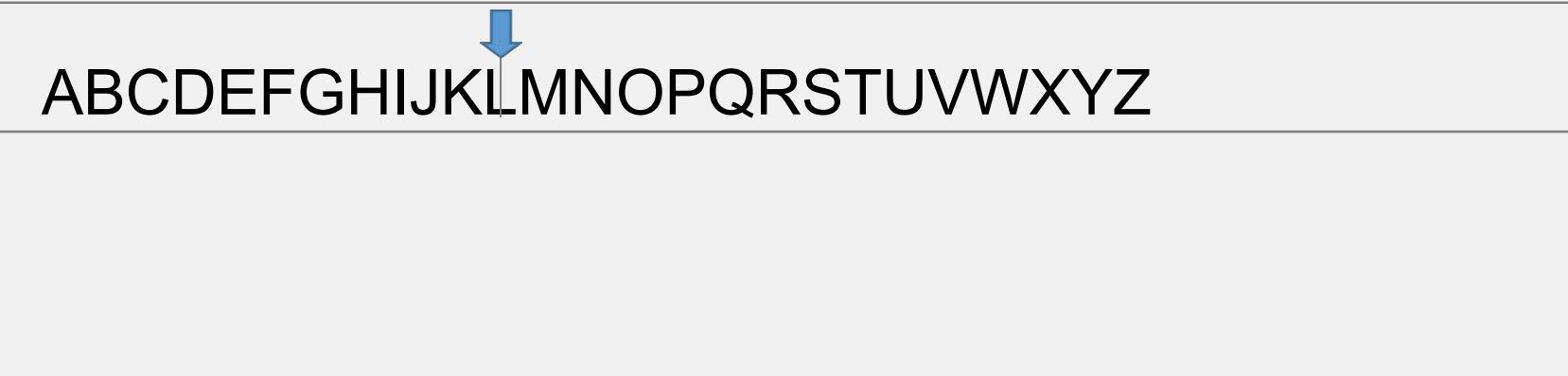
```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What will be the pointer position if we write the following?

```
in.seekg(10, ios::beg);
```

Introduction to Random Access

alphabet.txt

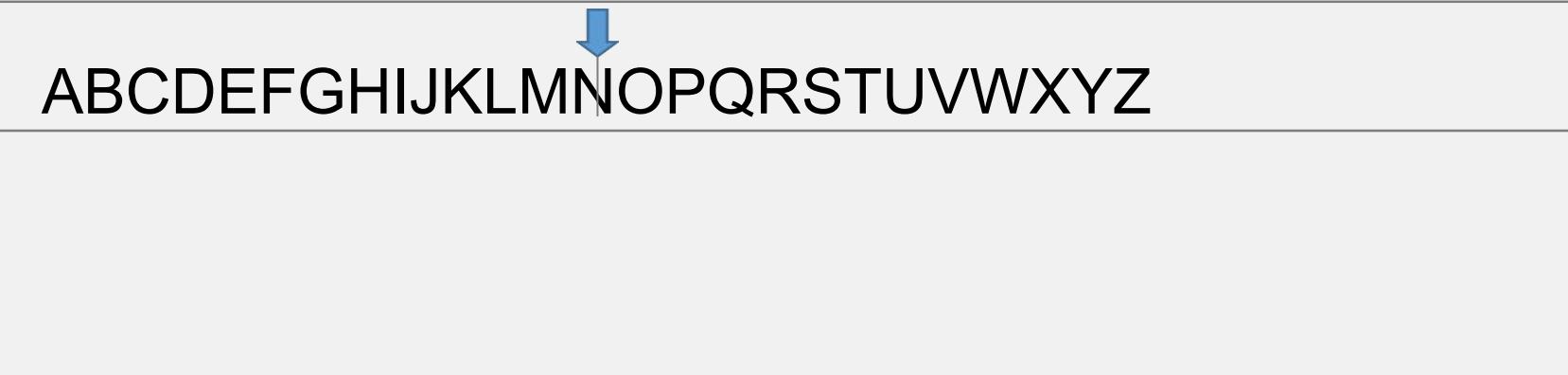


In order to move the pointer relative to the current position, we'll use `ios::cur`.

```
in.seekg(2, ios::cur);
```

Introduction to Random Access

alphabet.txt



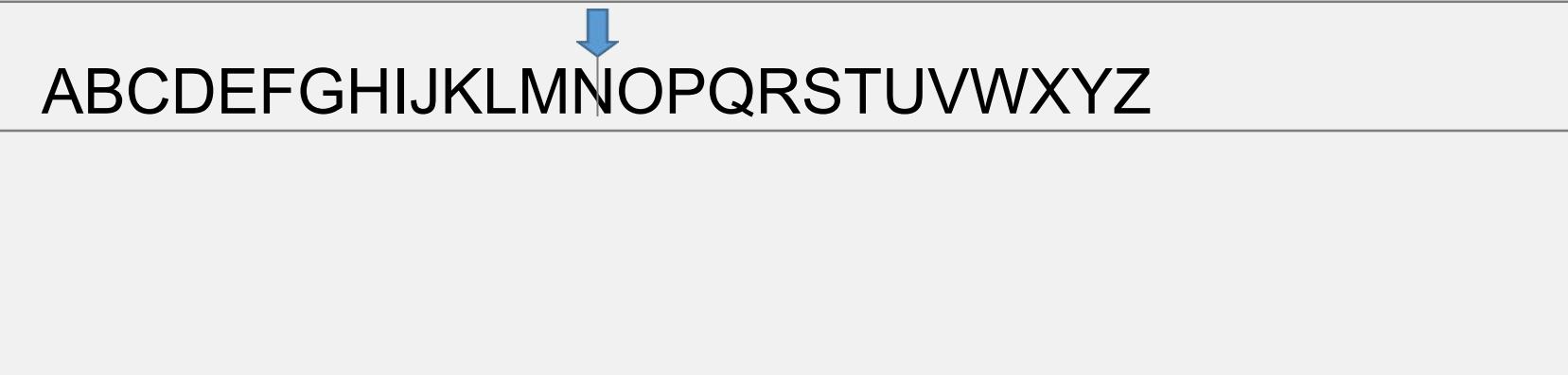
ABCDEFGHIJKLMNOPQRSTUVWXYZ

In order to move the pointer relative to the current position, we'll use `ios::cur`.

```
in.seekg(2, ios::cur);
```

Introduction to Random Access

alphabet.txt



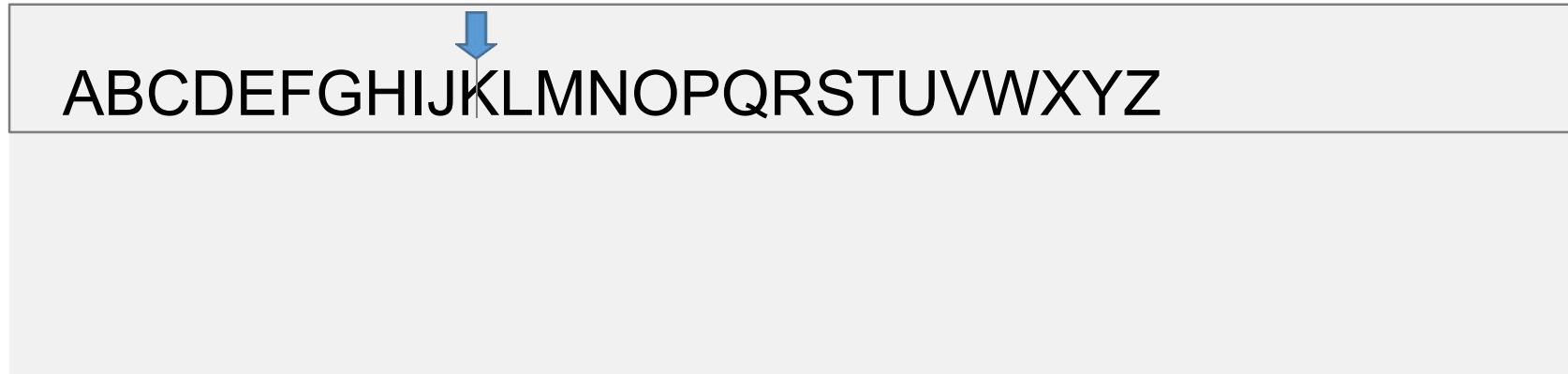
ABCDEFGHIJKLMNOPQRSTUVWXYZ

In order to move backward, use negative number.

```
in.seekg(-3, ios::cur);
```

Introduction to Random Access

alphabet.txt

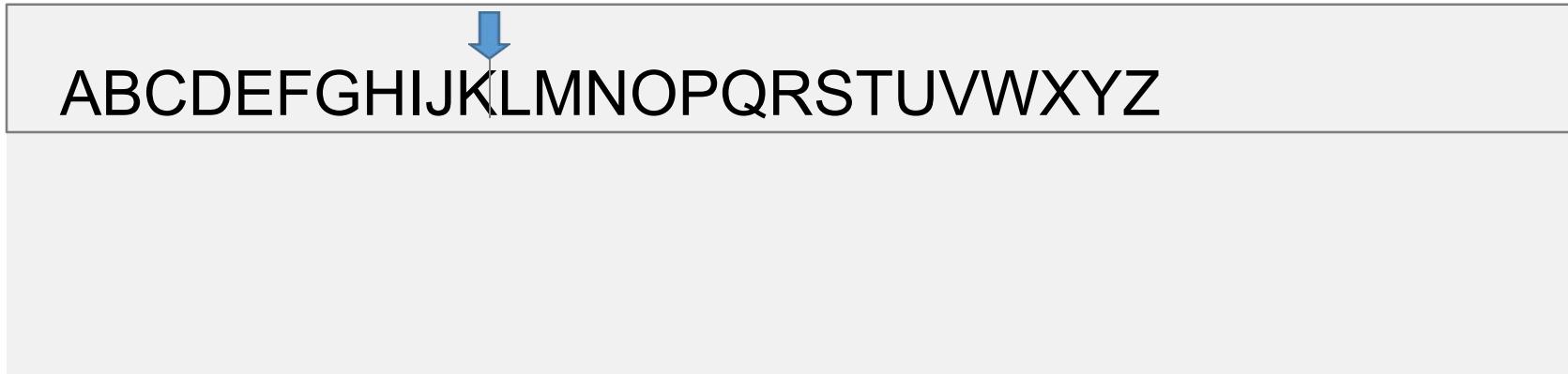


In order to move backward, use negative number.

```
in.seekg(-3,ios::cur);
```

Introduction to Random Access

alphabet.txt



To navigate from the end, we use `ios::end`.

```
in.seekg(-5, ios::end);
```

Introduction to Random Access

alphabet.txt

ABCDEFGHIJKLMNOPQRSTUVWXYZ

A diagram showing a file named "alphabet.txt". Inside the file, the letters of the alphabet are listed from A to Z. A blue arrow points directly to the letter 'V', indicating it is the current character being accessed.

To navigate from the end, we use `ios::end`.

```
in.seekg(-5, ios::end);
```

Introduction to Random Access

alphabet.txt

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

To find out how many bytes you've moved away from the beginning, use tellg()

```
int offset = in.tellg();
cout << offset;
```

Introduction to Random Access

alphabet.txt

ABCDEFGHIJKLMNOPQRSTUVWXYZ

A diagram illustrating a file named "alphabet.txt". Inside the file, the letters of the English alphabet are listed sequentially from A to Z. A blue arrow points directly to the letter 'W', highlighting it. The letters are displayed in a large, bold, black font.

How can we find out the size of a file?

We can use `tell()` !

Finding the file size

To find the size of a file:

- Open it
- Move the read pointer at the end
- Call tellg() to find out the offset, which is the filesize in byte

```
in.seekg(0, ios::end);
int filesize = in.tellg();
cout << filesize << " bytes";
```

Random Access in file: summary

Accessing from the middle of a file instead of sequential access

Functions:

- `tellg()` - file pointer position of where next char will be read
- `tellp()` - file pointer position of where next char will be written
- `seekg()` - Move the reading file pointer
- `seekp()` - Move the writing file pointer

`p` = `put`/`ofstream`

`g` = `get`/`ifstream`

Redirecting File Streams

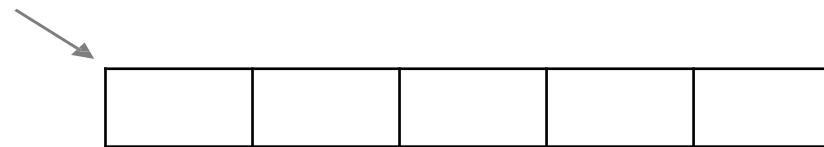
rdbuf() function

Remember Stream?

Every stream is associated with a Buffer/queue called **streambuf**.

The address of this **streambuf** is found using rdbuf() function.

cout



Redirecting File Streams

`rdbuf()` function

Remember Stream?

Every stream is associated with a Buffer/queue called **streambuf**

The address of this **streambuf** is found using `rdbuf()` function.

`cout`



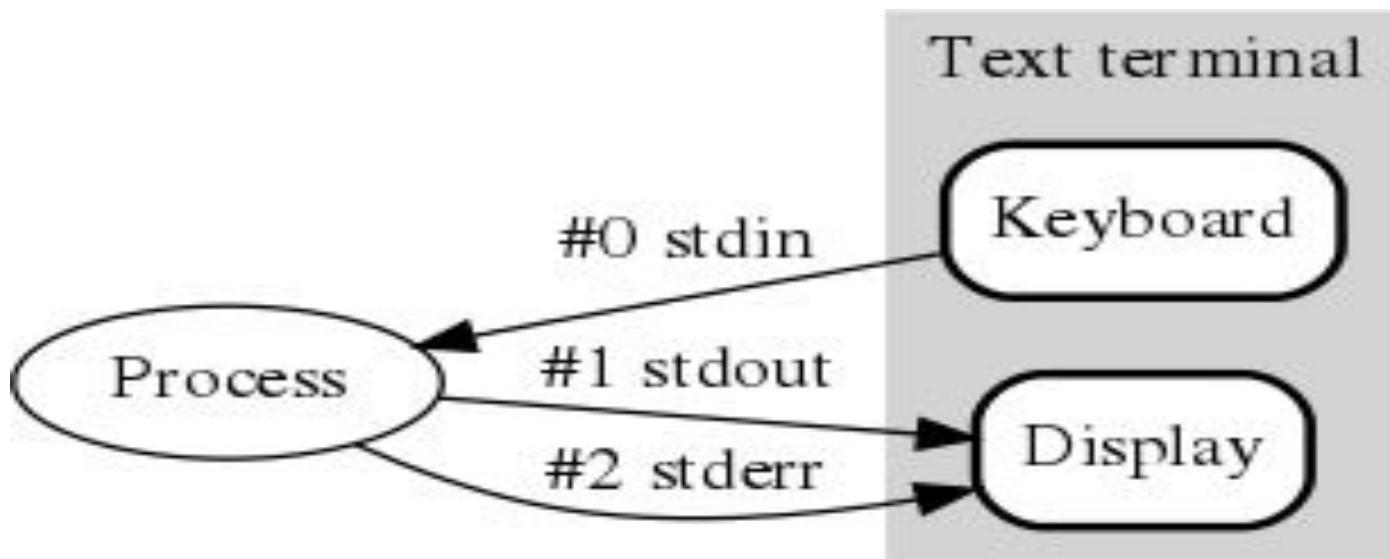
The address of this buffer is found using
`cout.rdbuf();`

The `rdbuf()` function returns a pointer to the internal `filebuf` object. We can use this to redirect the stream to our desired location. It can also be used to pass a stream.

Redirecting File Streams

rdbuf() function

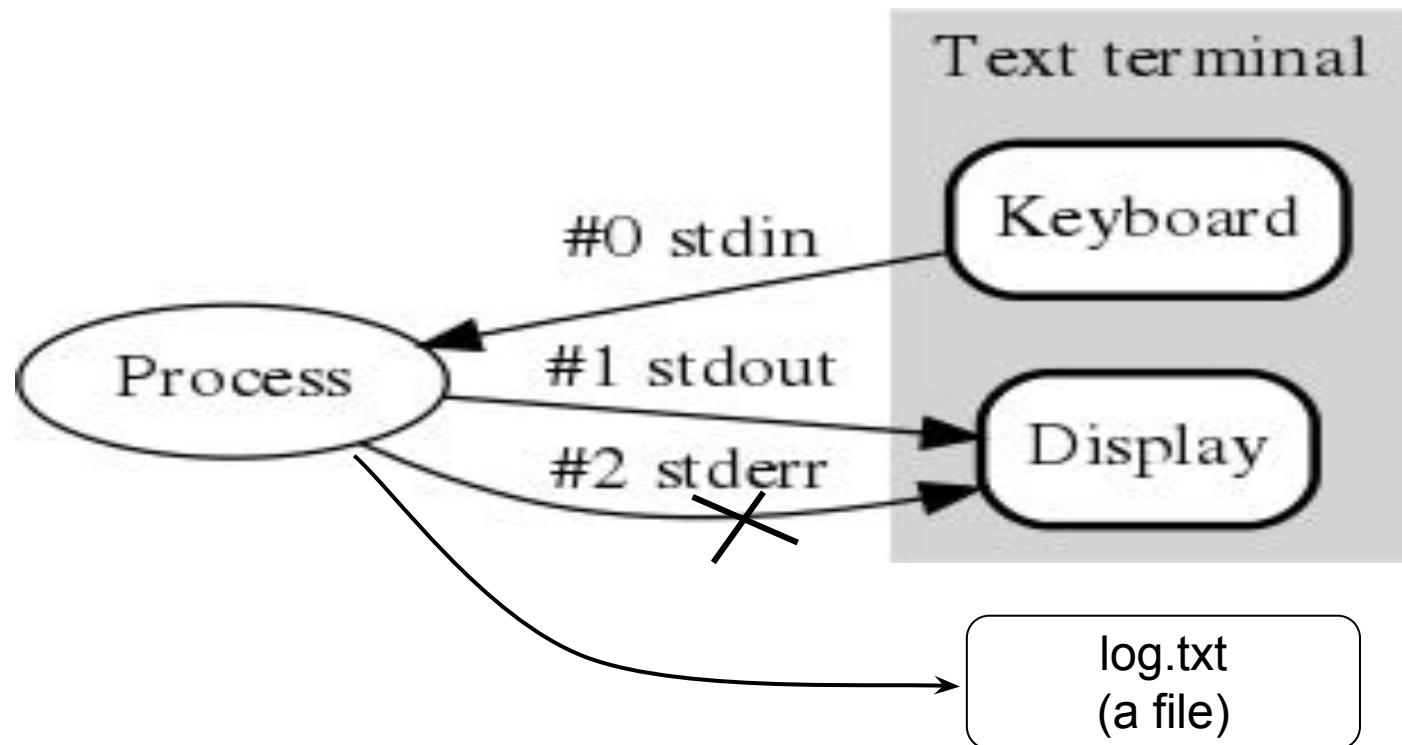
A program usually has three stream buffers.



Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”



Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”

```
ofstream error("log.txt", ios::app);
```

To change the default stream to a file stream of your own -

1. open a new file stream

Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”

```
ofstream error("log.txt", ios::app);
streambuf * ferror = error.rdbuf();
```

To change the default stream to a file stream of your own -

1. open a new file stream
2. assign this new buffer's address to a pointer (streambuf *)

Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”

```
ofstream error("log.txt", ios::app);
streambuf * ferror = error.rdbuf();
cerr.rdbuf(ferror);
```

To change the default stream to a file stream of your own -

1. open a new file stream
2. assign this new buffer's address to a pointer (streambuf *)
3. Now, pass this pointer to “cerr” using rdbuf() function.

Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”

```
ofstream error("log.txt", ios::app);
streambuf * ferror = error.rdbuf();
cerr.rdbuf(ferror);

cout << "This is OK";
cerr << "This is Error";
```

To change the default stream to a file stream of your own -

1. open a new file stream
2. assign this new buffer's address to a pointer (streambuf *)
3. Now, pass this pointer to “cerr” using rdbuf() function.
4. Then use cout and cerr accordingly.

Redirecting File Streams

rdbuf() function

We want to redirect cerr to a file stream called “log.txt”

```
ofstream error("log.txt", ios::app);
streambuf * ferror = error.rdbuf();
cerr.rdbuf(ferror);

int i;
cin >> i;
while (i != 0)
{
    if (i > 0)
        cout << "OK" << endl;
    else
        cerr << "Error " << i << " is a negative number" << endl;
    cin >> i;
}
```

Now, Let's Talk about CSE 205 and the Term Final

Topics Covered

1. Inheritance
 - a. Base and Derived Classes (Basic Concept)
 - b. Access Specifier (Public, Protected and Private)
 - c. Private Inheritance and Delegate Functions
 - d. Modifying Functions, Variables and Visibility in Inheritance
 - e. Multiple/Hierarchical and Multilevel Inheritance
2. Runtime Polymorphism
 - a. Object SLicing
 - b. Early and Late Binding & it's implementation.
 - c. Virtual Destructors

Topics Covered

3. Abstract Class and Interfaces
 - a. Introductory Concept
 - b. Pure Virtual Functions
 - c. Function overriding vs function overloading
 - d. Diamond Problem
 - e. Virtual base class and Virtual Inheritance
4. Operator Overloading
 - a. Intro to op. overloading
 - b. Arithmetic, Relational, Compound, Increment/Decrement, Assignment op. overloading
 - c. Subscript op. overloading
 - d. Functors (overloading "()")
 - e. Conversion function
 - f. Overloading new and delete

Topics Covered

5. Type Introspection
6. Understanding Streams and File I/O in C++
 - a. Understanding how I/O Streams work in C++
 - b. Std Streams, cerr vs clog
 - c. I/O Formatting
 - d. Creating Stream enabled class
 - e. File I/O
 - f. Reading and Writing objects from/to Files
 - g. Random file access (seekg())
 - h. Redirecting file streams (stderr to log.txt for example)
7. Templates in C++
8. Exception Handling

Thank you

Templates in C++

CSE 205 - Week 13

[Lec Raiyan Rahman](#)

Dept of CSE, MIST

raian@cse.mist.ac.bd

CC: Lec Sanjida Nasreen Tumpa



How we know things to be!

```
int abs (int n){  
    return (n < 0) ? -n : n;  
}  
long labs (long n){  
    return (n < 0) ? -n : n;  
}  
float fabs (float n){  return (n < 0) ? -n :  
    n;  
}
```

```
main()  
{  
    int i=-3;  
    cout<<abs (i);  
  
    long l=-7; cout<< labs (l);  
  
    float f=-2.0; cout<<fabs (f);  
}
```

Templates in C++

- It might be the case that we want to write a portion of code for which we don't know what exact data type the variable may be.
- So rather than writing different block of code for each of the data type, we can use the concept of "Templates" in C++.
- We can provide the data type later and that portion of the code will later be automatically generated based on our given data type.
- It's almost as if we are passing the data type we want as a function parameter (it works exactly like that actually!).

Templates in C++

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Making a Dynamic Dynamic Array List!

Passing the data type “int” to the class. That way, The “DynamicArray” class will know we want “list” to be a pointer of int type. We’ll later use it to make a DAL of ints!

```
6  template <class T> //  
7  class DynamicArray {  
8  private:  
9    T * list; //a poi: Step 2  
10   int max_size; //t  
11   int length; //to
```

Step 1

```
70  int main() {  
71  
72    DynamicArray <int> list;  
73
```

Note: We could also pass float, char or a user-defined data type like “students” or “library” to the class. That way we’d have a DAL of floats, chars, students or libraries respectively.

Using C++

Template

Generic Function/
Template Function

```
template <class T>  
  
T abs(T n)  
{ return (n < 0) ? -n : n; }
```

```
main()  
{  
    int i=-3;  
    cout<<abs (i);  
  
    long l=-7; cout<< labs (l);  
  
    float f=-2.0; cout<<fabs (f);  
}
```

Generic Function

- Programming that works regardless of type is called generic programming.
- A generic function defines a general set of operations that will be applied to various types of data.
- Another type of polymorphism, known as parametric polymorphism.
- Give the user the ability to reuse code in a simple, type-safe manner that allows the compiler to automate the process of **type instantiation**.

Function Template

Syntax

A generic function is created using the keyword **template**.

- Also called **Template Function**

```
template <class identifier>return_type function_name (arguments of type identifier)  
{ ... }
```

```
template <class identifier>
```

```
return_type function_name (arguments of type identifier) { ... }
```

- If Second form is used, no other statement can occur between the template statement and the start of the generic function definition
- Function return type is not taken into account to select the template function (just like function overload).

Function Template

Syntax

- Instead of using the keyword **class**, keyword **typename** can also be used to specify a generic type in a template definition

```
template <class identifier>
```

```
return_type function_name (arguments of type  
identifier) { ... }
```

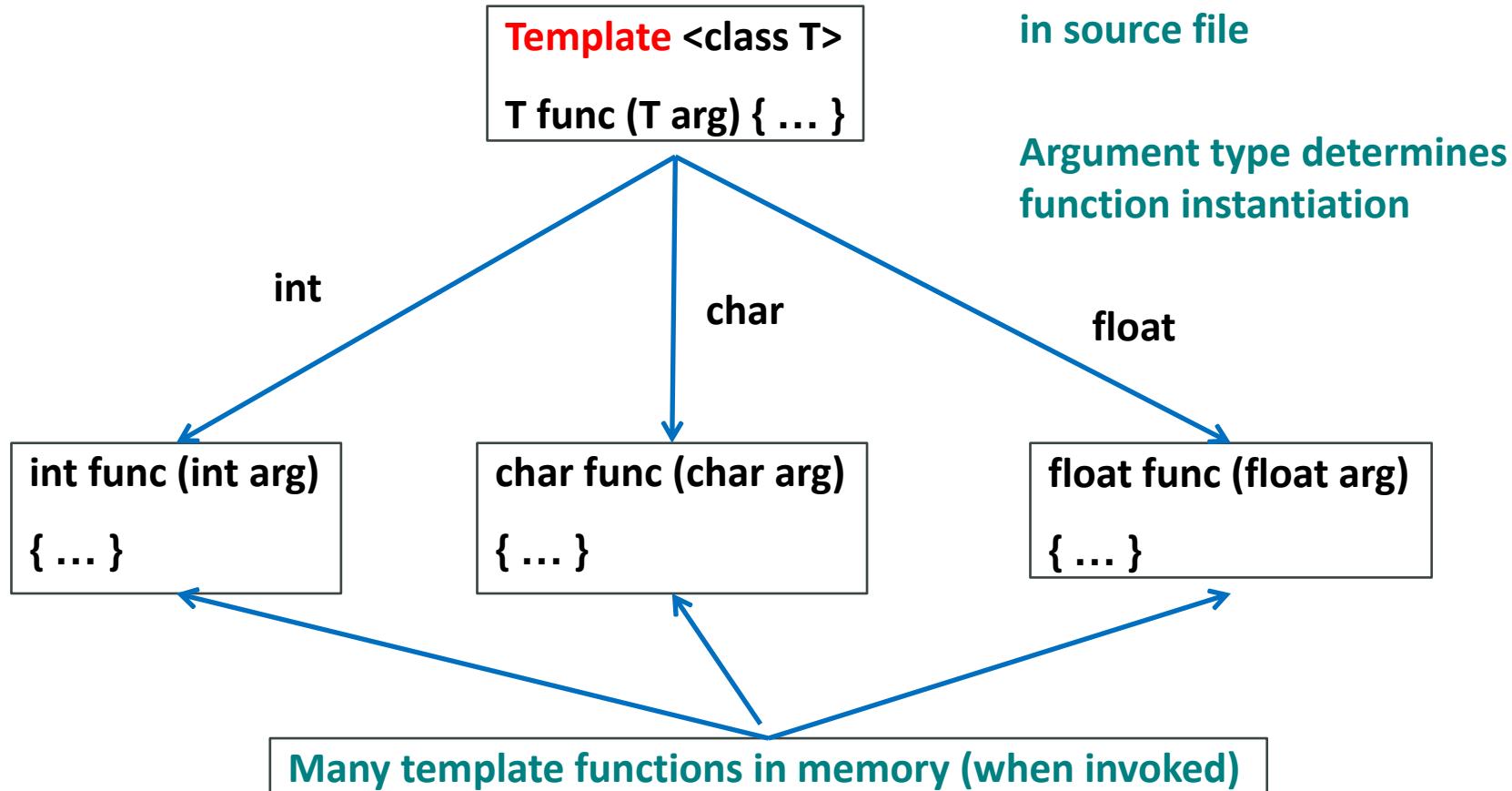
```
template <typename identifier>
```

```
return_type function_name (arguments of type  
identifier) { ... }
```

What Compiler Does

- Function template does not cause the compiler to generate any code (**similar to the way a class does not create anything in memory**).
- Compiler simply remembers the template for possible future use.
- Code generation does not take place until the function is actually called (invoked).
- Compiler generates a template function (specific version of a function template) depending on the function signature (that's why we call parametric polymorphism).

Function Template Example



One function template
in source file

Argument type determines
function instantiation

Function Template with Multiple

Ar

```
template<class T1, class T2> void copy(T1 a[], T2 b[], int n){ //two arguments  
    template for(int i=0; i<n; i++)  
        a[i] = b[i];  
}
```

```
main(){  
    char c[50] = {'a', 'b', 'd', 'e', 'f'};  
    int i[50] = {10, 20 30, 40, 50};  
    float f[50] = {1.1, 2.2, 3.3, 4.4, 5.5};  
    copy (i,f,5); // ok, but loss of data  
    copy (c,f,5); // ok, but loss of data  
    copy (f,i,5); // ok, int to float  
    copy (f,c,5); //ok, char to float  
}
```

Function Template with Multiple Arguments

```
template <class T> void swap( T& x, T& y){  
    T tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```

```
main(){  
    int x, y;  
    char c;  
    double m, n;  
    swap( x, y );  
    swap( m, n );  
    swap( x, m ); // error  
}
```

Generic Function vs Function Overloading

- When functions are **overloaded**, one can have different actions performed within the body of each function but **generic function must perform the same general version for all actions.**
- A generic function can be overloaded. In that case the overloaded function **overrides** the generic function relative to that specific version.

Generic Function vs Function Overloading

```
template <class X>
void func(X n){
    cout << n << endl;
}
```

```
template <class X, class Y>
void func(X a, Y b){
    cout <<a<<"  " << b << endl;
}
```

```
main(){
    func(10);
    func(20, 30);
}
```

Generic Function vs Function Overloading

```
template <class X>
void func(X n){
    cout << "Inside Generic Function" << endl;
}

int func(int n){
    cout<<"Inside Specific Function: "<<n;
}

main(){
    int i=10;
    func(i);
    double d=5.5;
    func(d);
}
```

Generic Class

- A generic class defines the algorithm used by the class but the actual type of data is specified when object of that class are created.
- The *identifier* is a template argument that essentially stands for an arbitrary type.
- The template argument can be used as a type name throughout the class definition.

```
template <class identifier >
```

```
class classname { ... };
```

```
classname<type> ob; //Object Instantiation
```

Generic Class

```
// template stack implementation
template <class TYPE>
class stack {
    TYPE* s;
    int top;
public:
    stack (int size =100){
        s = new TYPE[size]; top=0;
    }
    ~stack() { delete []s; }
    void push (TYPE c)
    { s[top++] = c; }
    TYPE pop()
    { return s[--top];}
};
```

```
void main(){
    //100 char stack
    stack<char>
    stk_ch;

    //200int stack
    stack<int>
    stk_int(200);

    //20 float stack
    stack<float>
    stk_float(20);
}
```

Thank you

**COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING**

Dept of CSE, MIST

Exception Example 1

```
double divide(double a,  
double b){  
    double result=a/b;  
    return result;  
}
```

What if **b=0??**

```
main(){  
    double a,b,c;  
    cin>>a>>b;  
    c=divide(a,b);  
    cout<<c;  
}
```

This is divide by
zero exception

Exception Example 2

```
class Numlist{  
    int* a;  
    int size;  
public:  
    Numlist(int s) {a=new int[s]; size=s; }  
    void putAt(int index, int v){a[index]=v;}  
    int getAt(int index){ return a[index];}  
};
```

```
main(){  
    Numlist list(100);    int index, val;  
    cin>> index>>val;  
    list.putAt(index,val);  
    cout<< list.getAt(index);  
}
```

What if

index<0 ?

index>100 ?

**Array Index
out of bound
Exception**

Exception Example 3

```
class Numlist{  
    int* a;  
    int size;  
public:  
    Numlist(int s) {a=new int[s]; size=s; }  
    void putAt(int index, int v){a[index]=v;}  
    int getAt(int index){ return a[index];}  
};
```

```
main(){  
    Numlist list(100);    int index, val;  
    cin>> index>>val;  
    list.putAt(index,val);  
    cout<< list.getAt(index);  
}
```

What if
size > memory available?

**Memory Allocation
Exception**

Exception Example 4

```
void manipList (Numlist* list, int op){  
    int index;  
    cin>>index;  
    if (op==1){ int val; cin>>val;  
        list->putAt(index,val); }  
    else if  
(op==2){cout<<list->getAt(index);}  
    return; }
```

```
main(){  
    Numlist* list;  
    int option;  
    cout<<"Put/Get: ";  
    cin>>option;  
    manipList (list,option);  
}
```

What if
**list is not
initialized?**

**Null Pointer
Exception**

Exception

- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as, an attempt to **divide by zero**.
- **Exceptions provide a way to transfer control from one part of a program to another.**
- C++ exception handling is built upon three keywords: **try, catch, and throw**.

Exception

- **try:** program statements to be monitored for exceptions are contained in try block. A function called from within a try block can throw exception too
- **Throw:** If an exception occurs within the **try block**, it is thrown
- **Catch:** when an exception is thrown, it is caught using **catch**, and processed. There can be more than one catch statement associated with a try

```
try{  
    //try block  
    throw exception;  
}  
catch(type1 arg){  
    //catch block  
}  
catch(type2 arg){  
    //catch block  
}  
catch(typeN arg){  
    //catch block  
}
```

throw statement

- **throw** is used to signal the fact that an exception has occurred; also called *raise*
- Can throw any valid expression/ object.
- **Syntax:**
throw expression;

Default catch statement

- If an exception is thrown for which there is **no applicable catch statement**, an **abnormal program termination** might occur
- In some circumstances you will want an **exception handler** to catch **all exceptions** instead of just **a certain type**
- To solve above situations, we can use default **catch statement**

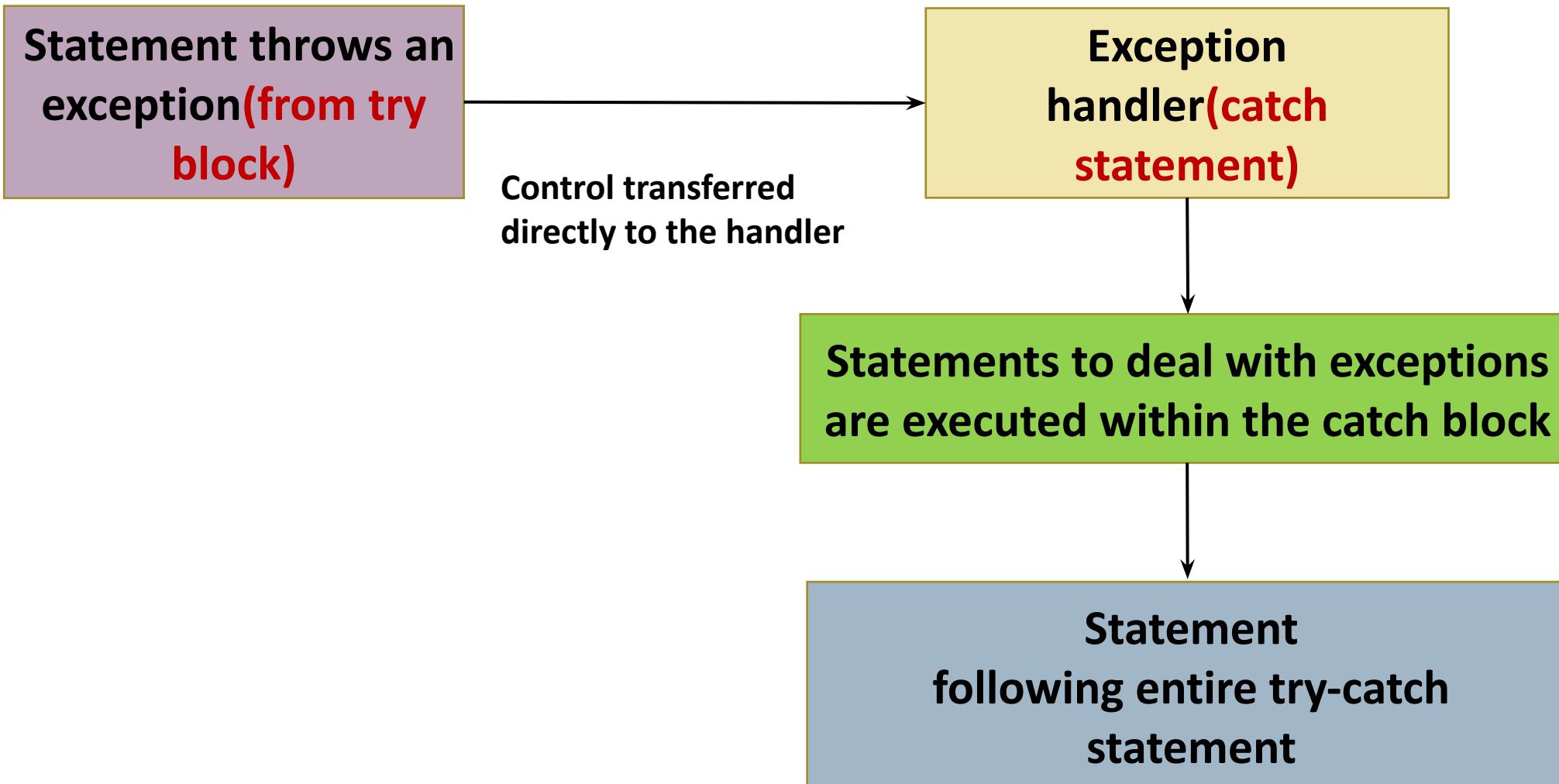
- **Syntax:**

```
catch(...){  
    // process all exceptions  
}
```

Execution of try-catch

- Once exception is thrown, **control passed** to the **catch** expression and **try block is terminated**
- **Catch** is **not called**
- Program execution is transferred to it
- After the execution of the catch statement, program continues with the **statement following** the **catch**

Execution of try-catch



Exception

An exception is an unusual, often unpredictable event, detectable by software or hardware, that requires special processing occurring at runtime.

- If without handling,
 - Program crashes
 - Falls into unknown state

Using try-catch

```
main(){
    cout<<“start\n”;
    try{
        cout<<“Inside try block”<<endl;
        throw 10;
        cout<<“This will not be executed”<<endl;
    }
    catch(int i ){
        cout<<“exception occurred”<<endl;
        cout<<“number is: ”<<i<<endl;
    }
}
```

OUTPUT

```
start
Inside try block
exception occurred
number is: 10
```

Using try-catch

```
main(){  
    cout<<“start\n”;  
    try{  
        cout<<“Inside try block”<<endl;  
        throw 10;  
        cout<<“This will not be  
executed”<<endl;  
    }  
    catch(double i ){  
        cout<<“exception occurred”<<endl;  
        cout<<“number is: ”<<i<<endl;  
    }  
}
```

OUTPUT

start
Inside try block
Abnormal Program Termination

This program produces such output
because the **integer exception** will not
be caught by a **double catch
statement**

SOLUTION??
Write another catch statement with
integer argument

Exception thrown from a stmt outside try block

```
void Xtest(int test){  
    cout<<"inside Xtest: "<<test<<endl;  
    if(test>0)  
        throw test;  
}
```

```
int main (){  
    cout<<"start\n";  
    try{  
        cout<<"inside a try block\n";  
        Xtest(0);  
        Xtest(1);  
        Xtest(2);  
    }  
    catch(int i){  
        cout<<"caught number: "<<i<<endl;  
    }  
    cout<<"end";  
}
```

Exception Handling

- An exception handler is a section of program code that is designed to execute when a particular exception occurs
 - Resolve the exception
 - Lead to known state, such as exiting the program

Exception Handling(divide by zero exception)

```
double divide(double a, double b)
{
    if(b!=0){
        double result=a/b;
        return result;
    }
    else{
        throw -1;
    }
}
```

```
main(){
    double a,b,c;
    cin>>a>>b;
    try{
        c=divide(a,b);
        cout<<c;
    }
    catch (int i){
        cout<<"Division Exception:
        "<<i;
    }
}
```

Exception Handling(divide by zero exception)

```
double divide(double a,  
             double b){  
    if(b!=0){  
        double result=a/b;  
        return result;  
    }  
    else{  
        throw "Divide by zero";  
    }  
}
```

```
main(){  
    double a,b,c;  
    cin>>a>>b;  
    try{  
        c=divide(a,b);  
        cout<<c;  
    }  
    catch (const char* message){  
        cout<<"Exception: ";  
        puts(message);  
    }  
}
```

Exception Handling

```
class MyException{  
    char message[30];  
public:  
    MyException ( char* m){  
        strcpy (message, m);  
    }  
    char* getMessage(){ return  
message; }};
```

```
double divide(double a, double b){  
    if(b!=0){ double result=a/b;  
        return result; }  
    else{  
        MyException Ex("Divide By Zero");  
        throw Ex; }  
}
```

```
main(){  
    double a,b,c;  
    cin>>a>>b;  
    try{  
        c=divide(a,b);  
        cout<<c;  
    }  
    catch (-----? e){  
        cout << "Error: ";  
        cout << e.getMessage();  
    }  
}
```

Exception Handling

```
class MyException{  
    char message[30];  
public:  
    MyException ( char* m){  
        strcpy (message, m);  
    }  
    char* getMessage(){ return  
message; }};
```

```
double divide(double a, double b)  
{  
    if(b!=0){ double result=a/b;  
        return result; }  
    else{  
        MyException Ex("Divide By Zero");  
        throw Ex ; }  
}
```

```
main(){  
    double a,b,c;  
    cin>>a>>b;  
    try{  
        c=divide(a,b);  
        cout<<c;  
    }  
    catch (MyException e){  
        cout << "Error: ";  
        cout << e.getMessage();  
    }  
}
```

Anonymous object

- In certain cases, we need a variable only temporarily.
- This is done by creating objects like normal, but **omitting the variable name**.

Example:

ComplexNumber **A(10, 2)**//normal object

ComplexNumber(10, 2) // Anonymous Object

Multiple Exception

```
#define MAX 3
class StackFull { };
class StackEmpty{ };
class Stack
{
    int S[MAX], top;
public:
Stack () { top= -1; }
void push(int v) {
    if (top>= MAX-1)
        throw StackFull();
    S[++top] = v;    }
int pop() {
    if (top<0)
        throw StackEmpty();
return S[top--];    }
};
```

```
int main() {
    Stack st;
try{
    st.push(10);
    st.push(20);
    st.push(30);
    st.push(40);
cout<<st.pop()<<endl;
cout<<st.pop()<<endl;
}
catch(StackFull F) {
    cout<<"Exception: Stack is Full" <<endl;
}
catch(StackEmpty E) {
    cout<<"Exception: Stack is Empty" <<endl;
}
```

Nested try catch

```
try{  
    try{  
        try{  
            }  
            catch (dataType varname){  
                //Exception Handler  
            }  
        }  
        catch (dataType varname){  
            //Exception Handler  
        }  
    }  
    catch (...){  
        //Default Exception Handler  
    }
```

Rethrow an exception

- An exception is rethrown to allow multiple handlers access to the exception
- Perhaps, one exception handler manages one aspect of an exception, and a second handler copes with another
- An exception can **only** be **rethrown** from within a **catch block**(or **from any function called from within that block**)
- When an exception is rethrown, it is **not re-caught** by the **same catch statement**
- It will propagate to an outer catch exception

Rethrow an exception example

```
void Xhandler(){
```

```
try{
```

```
    throw "hello";
```

```
}
```

```
catch(const char *c){
```

```
    cout<<"caught char* inside
```

```
xhandler\n";
```

```
    throw;
```

```
}
```

```
}
```

```
int main (){
```

```
    cout<<"start\n";
```

```
try{
```

```
    Xhandler();
```

```
}
```

```
catch(const char *c){
```

```
    cout<<"caught char* inside main\n";
```

```
}
```

```
    cout<<"end";
```

```
}
```

Restricting Exception in a Function

- You can **restrict** the **type** of **exceptions** that a function can throw back to its caller
- You can in fact **prevent** a function from **throwing** any exception
- To do so, a **throw** clause is to be added to the function definition

```
Ret-type func_name(arg list)throw(type list)
{
    //body
}
```

Restricting Exception in a Function

```
// can throw anything  
void func();
```

```
// promises not to throw  
void func() throw();
```

```
// promises to only throw int  
void func() throw(int);
```

```
// throws only char or int  
void func() throw(char,int);
```

By default, a function can throw anything it wants.

- A throw clause in a function's signature
 - Limits what can be thrown
 - A promise to calling function
 - If other type thrown, standard library function **unexpected()** is called
- A throw clause with no types
Says nothing will be thrown
- Can list multiple types
Comma separated

Source of Exceptions

- Exceptions Thrown by user code, using *throw* statement.
- Exceptions Thrown by the Language
for example while using: new
- Exceptions Thrown by Standard Library Routines

Exception Thrown by New

```
#include<iostream>
#include<new>
using namespace std;
int main(){
    int *p;
    try{
        p = new int [100000];
    }
    catch (bad_alloc xa){
        cout<<"allocation Failure";
    }
    delete [ ] p;
}
```

Good Programming Style with C++ Exceptions

- Don't use exceptions for normal program flow
 - Only use where normal flow isn't possible
- Don't let exceptions leave main or constructors
 - Violates "normal" initialization and termination
- Always throw some type
 - So the exception can be caught
 - Resources may have been allocated when exception thrown. **Catch handler should delete space allocated by new and close any opened files**
- Use exception specifications widely
 - Helps caller know possible exceptions to catch

*Thank
you!*