

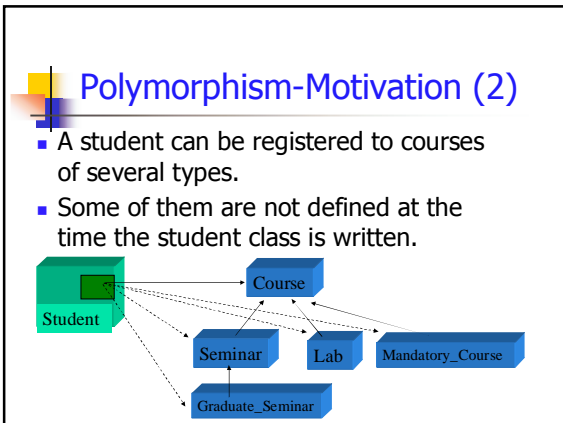
Polymorphism, the 3rd pillar of OOP: Motivation

- ### Polymorphism-Motivation (1)
- Assume that a company has several types of employees.
 - Assume that we want to hold the entire employee information in an array.
 - Impossible, because each type has a different size.

Solution by using pointers

- The size of a pointer is constant, no matter to what type it points.

```
Employee *array_of_employees[SIZE];
```



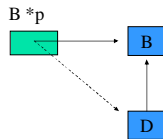
Polymorphism-Motivation(3)

- The need for **dynamic binding/dispatch (late binding)**.
- Sometimes the actual type of the object that a pointer will point to is not known at compile time.

```
cout << "Enter the type of the course (S/L/M): ";
cin >> course_type;
```

Dynamic (Late) Binding

- It happens when an object is instantiated dynamically and assigned to a pointer.
- The type of the pointer is fixed in code.
- The type of the object is determined in runtime.



Example - Shape

```

class Circle : public Shape
{
public:
    Circle(Point center,int r)
        :Shape(center),radius(r){}
    // ...
    void Draw();
private:
    int radius;
};

class Rectangle : public Shape
{
public:
    Rectangle(Point base,int wid, int len)
        :Shape(base), width(wid),length(len){}
    // ...
    void Draw(); // overriding
    void PrintCorners(); // not in Shape
private:
    int width;
    int length;
};
  
```

Example – cont'd

```

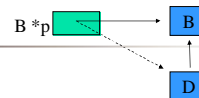
int main()
{
    Shape * sp;
    sp = new Rectangle(Point(0,0),3,4);
    sp->Draw();
    sp->PrintCorners();
}
  
```

Polymorphism

Shape::Draw()

Compilation error

Dynamic Binding



Possible problems:

- p can not invoke functions in D that are not present in B.
- If a function is present in both D and B, the version in B will be invoked.

Solutions:

- Virtual functions
- dynamic_cast

Virtual Functions

Virtual Functions

- A virtual function is a function that the linkage to it from a base class pointer is dynamic (happens at runtime).
- This means that the actual function to be invoked, is determined by the run time type of the object (as opposed to its compile time type).

Virtual functions - syntax

- The virtual keyword appears in the interface, not in the implementation.

```
class B{
    //...
    virtual function_header;
    //...
};
```

Example - Shape

```
class Shape
{
public:
    Shape(Point base): basePoint(base){}
    virtual ~Shape(){} // virtual destructor
    virtual char* ShapeType(){return "Shape";}
    virtual int GetArea() { return -1; } // error
    virtual int GetPerim() { return -1; } // error
    virtual void Draw() {}
protected:
    Point basePoint;
};
```

A non-virtual function A virtual function

More on
virtual d'tors,
later on.

Example - cont'd

```
class Circle : public Shape
{
public:
    Circle(Point center,int r)
        :Shape(center),radius(r){}
    virtual char* ShapeType(){return "Circle";}
    virtual int GetArea() { return 3.14 * radius * radius; }
    virtual int GetPerim() { return 6.28 * radius; }
    virtual void Draw();
private:
    int radius;
};
```

Example - cont'd

```
class Rectangle : public Shape
{
public:
    Rectangle(Point base,int wid, int len)
        :Shape(base), width(wid),length(len){}
    virtual char* ShapeType(){return "Rectangle";}
    virtual int GetArea() { return length * width; }
    virtual int GetPerim() {return 2*length + 2*width; }
    virtual int GetLength() { return length; }
    virtual int GetWidth() { return width; }
    virtual void Draw();
private:
    int width;
    int length;
};
```

Example - cont'd

```
class Square : public Shape
{
public:
    Square(Point base, int s)
        :Shape(base), side(s){}
    virtual char* ShapeType(){return "Square";}
    virtual int GetSide() { return side; }
    virtual int GetPerim() {return 4 * side;}
    virtual int GetArea() {return side * side;}
    virtual void Draw();
private:
    int side;
};
```

Example - cont'd

```
int main()
{
    int choice;
    int x,y,radius, width,length;
    bool fquit = false;
    Shape * sp;
    while ( !fquit )
    {
        cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
        cin >> choice;
        switch (choice)
        {
            case 0: fquit = true; break;
            case 1:
                cout<<"Base Point: ";
                cin>>x>>y;
                cout<<"Radius: ";
                cin>>radius;
                sp = new Circle(Point(x,y),radius);
                break;
        }
    }
}
```

client code

Object
created by
polymorphism

Example – cont'd

client code

```
case 2:
    cout<<"Base Point: ";
    cin>>x>>y;
    cout<<"Width: ";
    cin >> width;
    cout<<"Length: ";
    cin >> length;
    sp = new Rectangle(Point(x,y),width,length);
    break;

case 3:
    cout<<"Base Point: ";
    cin>>x>>y;
    cout<<"Width: ";
    cin >> width;
    sp = new Square(Point(x,y),width);
    break;

default:
    cout<<"Please enter a number between 0 and 3"<<endl;
    continue;
    break;
} // end of switch statement
```

Example – cont'd

client code

```
if( !fquit )
{
    sp->Print();
    sp->Draw();
}
delete sp;
sp = 0;
cout << "\n";
} // end of while statement
return 0;
}
```

Non-polymorphic call

Polymorphic call

Comments on the example

- The type of the object sp points to, is determined at runtime.
- The function Print() is not virtual, but invokes the virtual function ShapeType()

```
void Shape::Print()
{
    cout<<ShapeType()<<" Base Point: ";
    cout<<basePoint;
}
```

Properties of Virtual functions

- The polymorphic invocations of the virtual functions are done only for dynamically allocated objects. They will not work when the object is allocated in the stack.
- static functions can not be virtual.
- It will call only functions with same signature. It is not enough that the name will be the same

Properties of Virtual functions

- It is a bad practice to override a function that is not defined a virtual (it will confuse clients).
- Corollary: Every overridable function should be declared a virtual.
- The "virtual-ness" is inherited.
- Operators can be virtual too.
- D'tors should always be virtual unless we do not want clients to inherit from our class (to be explained soon).

Virtual c'tors and d'tors

- A c'tor can not be virtual, because when we create an object we know its type exactly, and we want its own c'tor to be invoked.
- D'tors should be almost always virtual.
- A failure to do that may result in a memory leak.
- A non virtual d'tor is a sign of a non-extendable class.

Example - Student

```
class Person
{
public:
    Person( int s, const string& n = "" );
    //...
private:
    int ssn;
    string name;
};

class Course
{
public:
    Course(string name, const Person& teacher);
    // ...
private:
    string courseName;
    Person& instructor;
};
```

Example – cont'd

```
class Student : public Person
{
public:
    Student( int s, const string & n = "", double g = 0.0 );
    //...
private:
    double gpa;
    Course **courses;
};
```

Client code:

```
Person *ps = new Student(123456789, "Izzet Morakas", 4.0);
// ...
delete ps;
```

About the example

- If the d'tor of person is not virtual then the actual d'tor invoked will be always ~Person().
- However, if p actually points to a Student, we want ~Student() be invoked.
- Failure to invoke it, will cause the course array remain in memory forever.

The vtable

- When a function is not virtual a simple "jump" statement is generated by the compiler to jump to the start of the (compile time known) function.
- When a function is virtual the jump involves lookup in a table, that is an array of function pointers.
- For every class that contains at least one virtual function a table (vtable) is allocated to contains these (function) pointers.

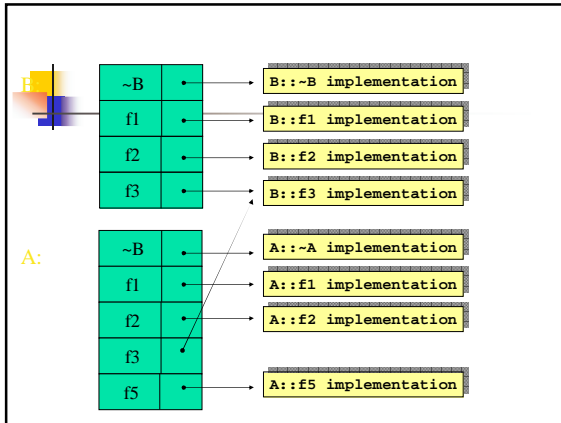
The vtable and vptr

- Every instance of a class having a vtable, contains a pointer to the start of this table.
- This pointer is called **vptr**.
- The code generated by the compiler is something like:
vtable = *vptr;
Jump *(vtable[C]); // C is known to compiler.
- The location of vptr is known only to compiler.
- Its existence can be "felt" when using sizeof().

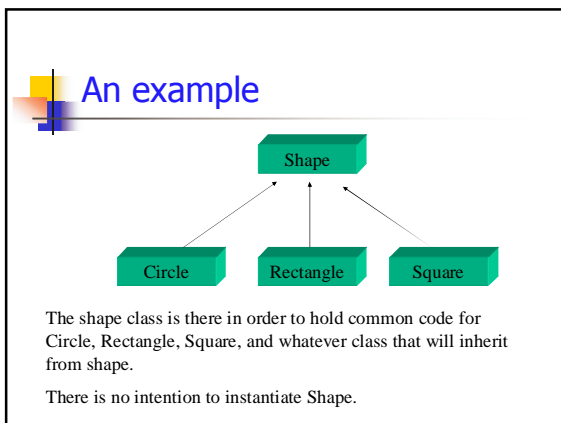
Example

```
class B{
public:
    //...
    virtual ~B();
    virtual void f1(...);
    virtual int f2(...);
    virtual double f3(...);
    void f4(...);
    //...
};

class A: public B{
public:
    //...
    virtual ~A();
    virtual void f1(...);
    virtual int f2(...);
    virtual int f5(...);
    //...
};
```



Abstract Classes



- ### Abstract Classes
- In Java a class may be defined as **abstract**.
 - This imposes a restriction on this class: it can not be instantiated,
 - but the subclasses can be instantiated.
 - In C++ this is done by declaring at least one function that is "pure virtual":


```
virtual ret_val func_name(parameters) = 0;
```
 - "pureness" is inherited too.

- ### I/F inheritance vs. Implementation inheritance
- A summary:
 - A non-virtual function is fully inherited, i.e. its interface and its implementation. We should not override its implementation.
 - A virtual function inherits its I/F, and we can choose to override its implementation.
 - A pure virtual function inherits its I/F only, we have to implement it in all **concrete** classes.

- ### Pure Virtual D'tors
- If a D'tor is pure virtual, it should be implemented (even as empty), because it is invoked by the D'tors of the subclasses.


```
virtual ~func_name() = 0 {block};
```

Multiple Inheritance

Multiple Inheritance - Syntax

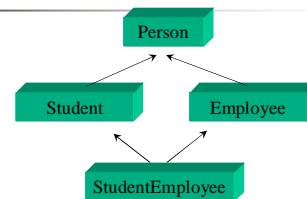
- A class can inherit from many classes

```
class B1
{...};
class B2
{...};
class B3
{...};
class D: public B1, public B2, private B3
{...};
```

Problems

- Variables with the same name in some of the base classes.
- Functions with same signature in some of the base classes.
- Common ancestor (to be explained soon).
- Q: Order of base class c'tor invocations?
- Q: How memory is organized?

Common Ancestor-A Diamond



Example-StudentEmployee

```
class Person
{
public:
    Person( const string & n, const string & t )
        : name( n ), ptype( t ) { }
    virtual ~Person() { }
    const string & getName() const
        { return name; }
    const string getPtype() const
        { return ptype; }
private:
    string name;
    string ptype;
};
```

Example – cont'd

```
class Student : public Person
{
public:
    Student( const string & n, int h )
        : Person( "Student", n ), hours( h ) { }
    int getHours() const // credit hours taken
        { return hours; }
private:
    int hours;
};

class Employee : public Person
{
public:
    Employee( const string & n, int h )
        : Person( "Employee", n ), hours( h ) { }
    int getHours() const // vacation hours left
        { return hours; }
private:
    int hours;
};
```

Example – cont'd

```
class StudentEmployee : public Student, public Employee
{
public:
    StudentEmployee( const string & n, int h1, int h2 );
    // ...
};
```

Do we need two hours parameters ?

About the example

- Problems:
 - The variable hours appears in both base classes.
 - The function getHours() appears in both base classes.
 - The object name appears in the common ancestor.

About the example

- Solutions:
 - Do not define the same names. If we do not have control on the names, then refer them as Employee::hours and Student::hours.
 - Same solution for getHours().
 - Virtual inheritance.

Virtual Inheritance

```
class Student : virtual public Person
{ ... };

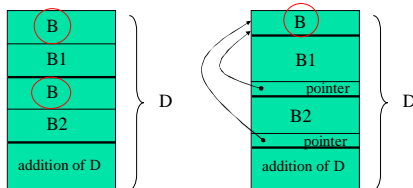
class Employee : virtual public Person
{ ... };
```

- The keyword virtual means that the data members inherited from Person are hold separately from the data members of the class itself.
- When there is a common ancestor, the compiler knows to allocate only one instance of these variables.

Memory Layout

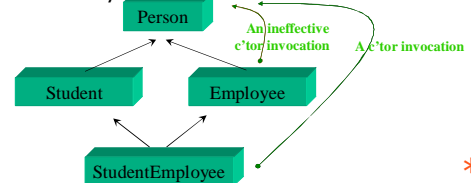
Non-virtual inheritance

Virtual Inheritance



C'tors in Virtual Inheritance

- The sub-sub-class c'tor can invoke the c'tor of the common ancestor.
- In this case the compiler ignores the invocations to the c'tors by sub-classes.



StudentEmployee

```
class Person
{
public:
    Person( const string & n, const string & t )
        : name( n ), ptype( t ) { }
    virtual ~Person() { }
    const string & getName() const
        { return name; }
    const string & getPtype() const
        { return ptype; }
private:
    string name;
    string ptype;
};
```

*

StudentEmployee

```
class Student : virtual public Person
{
public:
    Student( const string & n, int h )
        : Person( "Student", n ), hours( h ) { }
    int getCreditHours() const // credit hours taken
        { return hours; }
private:
    int hours;
};

class Employee : virtual public Person
{
public:
    Employee( const string & n, int h )
        : Person( "Employee", n ), hours( h ) { }
    int getVacationHours() const // vacation hours left
        { return hours; }
private:
    int hours;
};
```

Will not be invoked when creating StudentEmployee

*

StudentEmployee

```
class StudentEmployee : public Student, public Employee
{
public:
    StudentEmployee( const string & n, int ch, int vh )
        : Person( "StudentEmployee", n )
        , Student( "ignored", ch ), Employee( "ignored", vh )
        { ... }
};
```

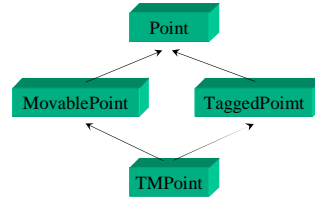
C'tors invoked not necessarily in this order, but in the order of inheritance

Invocation to the grand-parent class c'tor

*

Another example - TMPoint

- A movable point with a tag.



*

```
class Point
{ //...
private:
    int x,y;
};

class MovablePoint: virtual public Point
{ //...
private:
    int deltax,deltay;
};

class TaggedPoint: virtual public Point
{ //...
private:
    static int defaultTag;
    static int count;
    char *tag;
    //...
};

class TMPoint: public TaggedPoint, public MovablePoint
{ //...
};
```

*

```
TMPoint::TMPoint() {}

TMPoint::TMPoint(char *name)
: TaggedPoint(name)
{}

TMPoint::TMPoint(int a, int b, int dx, int dy)
: Point(a,b), MovablePoint(a,b,dx,dy)
{}

TMPoint::TMPoint(int a, int b, char *name, int dx, int dy)
: Point(a,b), TaggedPoint(a,b,name), MovablePoint(a,b,dx,dy)
{}

ignored
```

apply default ctors of all base classes

apply default ctors of MovablePoint and Point

apply default ctor of MovablePoint

*

Interfaces ?

A short comparison to Java

- Java does not support multiple inheritance
- However it supports what is called **interfaces**.
- An interface is like a class but it does not have an implementation.
- Remember: class=I/F+implementation.

Comparison to Java-cont'd

- In Java, a class may inherit from at most one parent-class, but
- may implement many interfaces.
- The implementation of multiple interfaces does not involve all the problems on multiple inheritance.
- What do we do in C++ instead?

Interfaces C++ style

- For each interface we define a class with all functions pure virtual.
- We inherit from all these classes.
- Convention: interface classes' names start with the capital letter "I".

Example

```
class Printable
{
public:
    virtual ~Printable(){}
    virtual void Print(ostream & out = cout) const = 0;
};

class Serializable
{
public:
    virtual ~Serializable(){}
    virtual void Serialize(ostream & out = cout) const = 0;
    virtual void Deserialize(istream & in = cin) = 0;
};
```

Example – cont'd

Multiple Inheritance

```
class Person: public Printable, public Serializable
{
public:
    // functions implemented in .cpp
    virtual void Print(ostream & out = cout) const;
    virtual void Serialize(ostream & out = cout) const;
    virtual void Deserialize(istream & in = cin);
    // ...
};
```

A last word on Multiple Inheritance

- Refrain from using it,
- except if you want to implement interfaces.

RTTI and casting

Run Time Type Identification RTTI

- We saw that the run time type of an object pointed by a pointer can be different from its compile time type.
- The Run Time Type is a Compile Time Type.
- We can determine the Run Time Type of an object by using the `typeid()` function.

Run Time Type Identification RTTI

- The `typeid()` function returns a `type_info`.
- It can be invoked as:
 - `typeid(<object reference>)`
 - `typeid(<Class Name>)`
- `type_info` has:
 - A `const char *name()` method
 - An overloaded `==` operator
- If `(typeid(object) == typeid(Student))`

Example

```
Shape *sp;
//
// sp is initialized at runtime
// to a derived class object
//
sp->Print();
cout<<"This is a "<<typeid(*sp).name()+6<<endl;
cout<<"Area: "<<sp->GetArea()<<endl;
cout<<"Perimeter: "<<sp->GetPerim()<<endl;
sp->Draw();
```

polymorphic calls

RTTI

skip "class "

Casting in C++

- The normal C-style casting is still available (how not?),
- but there is a C++ style casting superseding it.
- There are 4 types of C++ style cast operations:
 - `static_cast`
 - `dynamic_cast`
 - `const_cast`
 - `reinterpret_cast`

C++ cast syntax

(type) expression C style cast

↓

castType<type>(expression) C++ style cast

castType is one of:

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

Example:

```
b = (double) a;
```

↓

```
b = static_cast<double>(a);
```

static_cast

- Semantics similar to C-style cast.
- It is done at compile time, (therefore static).
- Only meaningful cast are accepted.
- The unaccepted casts are, among others:
 - a pointer to a normal type or vice versa.
 - an object to another type for which there is no copy ctor, = operator
 - a const to a non const.

dynamic_cast

- Used to cast a super-class to a sub-class.
- It is done at run time, and the run time type is checked to "match" the requested type, i.e.:
- is the run time type a requested type?

```
void f(Shape* sh)
{
    Rectangle *pr = static_cast<Rectangle*>(sh);
    // ...
}
```

dynamic_cast (cont'd)

```
void f(Shape* sh)
{
    Rectangle *pr = dynamic_cast<Rectangle*>(sh);
    if (pr != NULL) {
        // ...
    }
}
```

- If a dynamic_cast on a pointer fails the pointer receives NULL.
- If a dynamic_cast on a reference fails this is a run time error.
- dynamic_cast can be used on objects having a vtable (otherwise use a static_cast).

dynamic_cast

```
Shape *sp;
//
// sp is initialized at runtime
// to a derived class object
//
if (typeid(*sp) == typeid(Rectangle))
    dynamic_cast<Rectangle*>(sp)->PrintCorners();
else if (typeid(*sp) == typeid(Square))
    dynamic_cast<Square*>(sp)->PrintCorners();
```

The function is defined for Square and Rectangle only.

const_cast

- Casts a const to a non-const.
- Usually needed to pass a const to a function receiving a non-const (and we do not have control on the function)

```
class B{...};
void g(B*)
{ /* ... */ }

void f(const B& x)
{
    // ...
    g(const_cast<B*>(&x));
    // ...
}
```

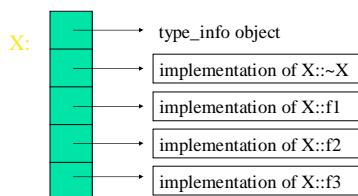
reinterpret_cast

- Used to cast:
 - An object ptr to a unrelated object ptr.
 - An object ref to a unrelated object ptr.
 - An int to a ptr and vice-versa.
 - A function ptr to another function ptr.
- Caution: Use sparingly.

Summary

The Price of Polymorphism

- The space used by vtables and vptrs.
- The time in function call overhead.



The Java approach

- In Java all the functions are virtual and by default can be overridden.
- Use the same approach in C++ (unless you have very special reasons, see next slide):
 - Define all functions as virtual.
 - Refrain from defining automatic objects (on stack).
 - Then, all the invocations will be polymorphic.
 - Do not use multiple inheritance except for interfaces.

Exceptions to the Java approach

- When performance is a major issue, and
- The client programmers are aware of all the implications.