Inheritance

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan

Inheritance Recap
Polymorphism

Announcements and Syllabus Check

Q: Why use dynamic memory allocation?
Q: Is while loop condition in `toVector()` redundant?

Forum: on Blackboard. Get it started, soon I will discontinue csci235.help@gmail.com

Slight modifications to Project 3:

Changed wording of copy constructor paragraph to be more clear (might have been misleading)

Added some checks to the main function

Inheritance Recap

Basic Inheritance

```cpp
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void changeCartridge();
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer


class BatchPrinter: public Printer // inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents; //Document queue
}; //end BatchPrinter
```
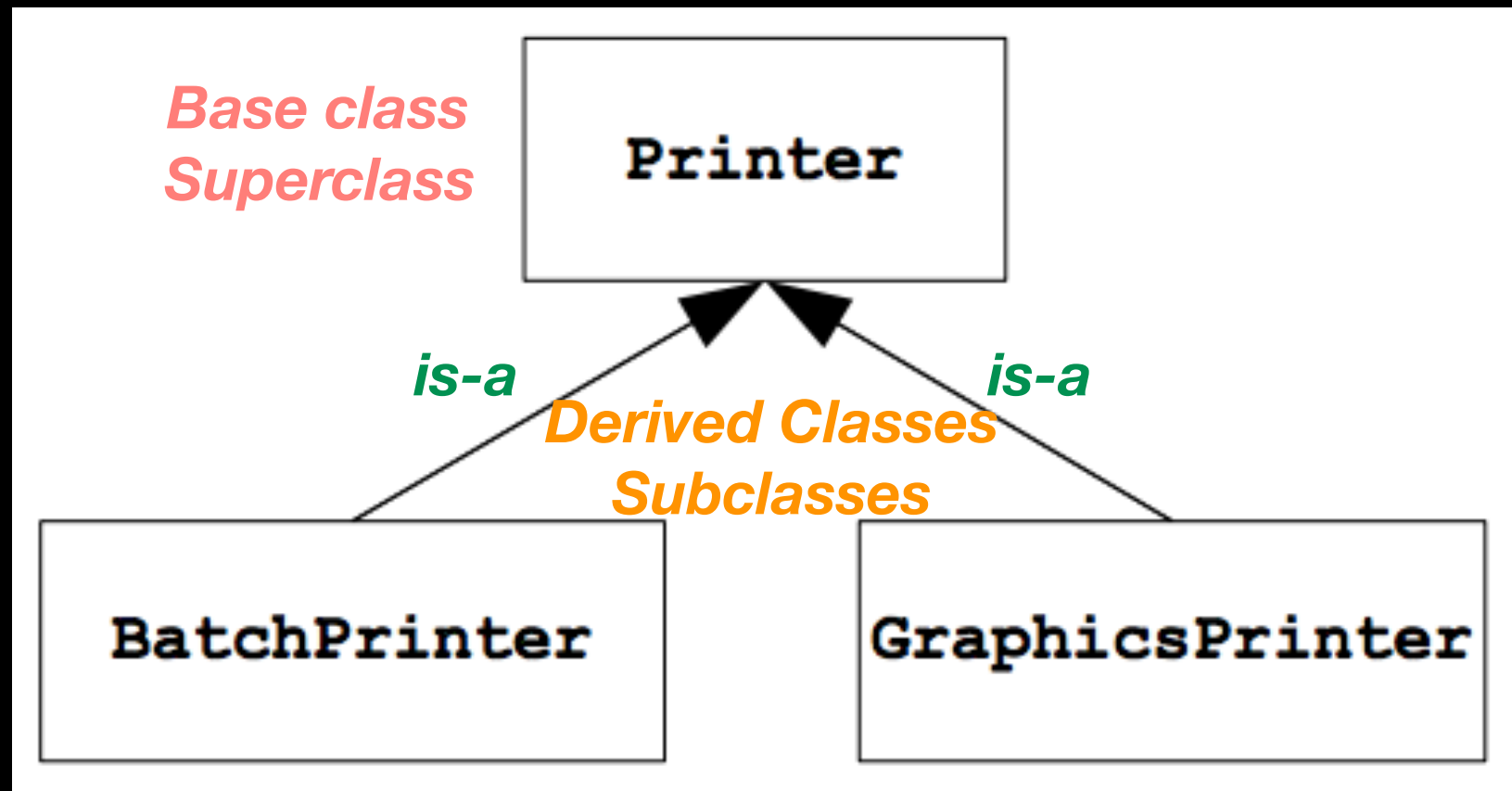
```cpp
class GraphicsPrinter: public Printer  // inherit from printer
{
public:
    //Constructor, destructor
    void changeCartridge();
    void printDocument(const Picture& picture);

private:
    //stuff here
}; //end GraphicsPrinter
```

Basic Inheritance


```
void initializePrinter(Printer& p)
BatchPrinter batch;
initizlizePrinter(batch); //legal because batch is-a printer
```

Think of argument types as specifying minimum requirements

Problem

```cpp
class BatchPrinter: public Printer // inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents; //Document queue
}; //end BatchPrinter
```

We would like to print all kinds of documents not just text
`documents` **should be able to store different types of documents**
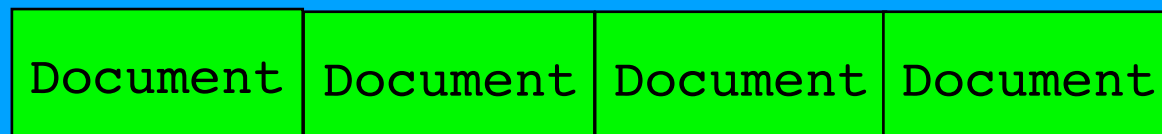
8

Generalized Document

Whatever the type of document, a printer ultimately prints a grid of pixels

Generalized `Document` should know how to convert itself into a printable format

We want `Document` to be an *interface* => not concerned with implementation details

printAllDocuments()

| Document | Document | Document | Document |

Document::convertToPixelArray()
printPixelArray()

Polymorphism

```cpp
class BatchPrinter: public Printer  // inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const Document* document);
    void printAllDocuments();
private:
    vector<Document*> documents; //Document queue
}; //end BatchPrinter
```

Abstract Class!

This function has no implementation**

```
class Document:
{
public:
    //Constructor, destructor
    virtual void convertToPixelArray() const = 0;
    virtual int getPriority() const = 0;

private:
    //stuff here
}; //end Document
```

I'll explain this next

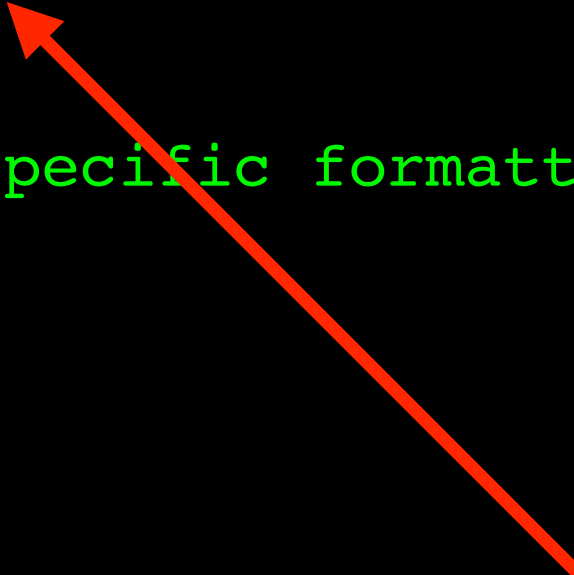```cpp
class TextDocument: public Document// inherit from Document
{
public:
    //Constructor, destructor
    virtual void convertToPixelArray() const override;
    virtual int getPriority() const override;

    void setFont(const string& font); //text-specific formatting
    void setSize(int size);

private:
    //stuff here
}; //end TextDocument
```
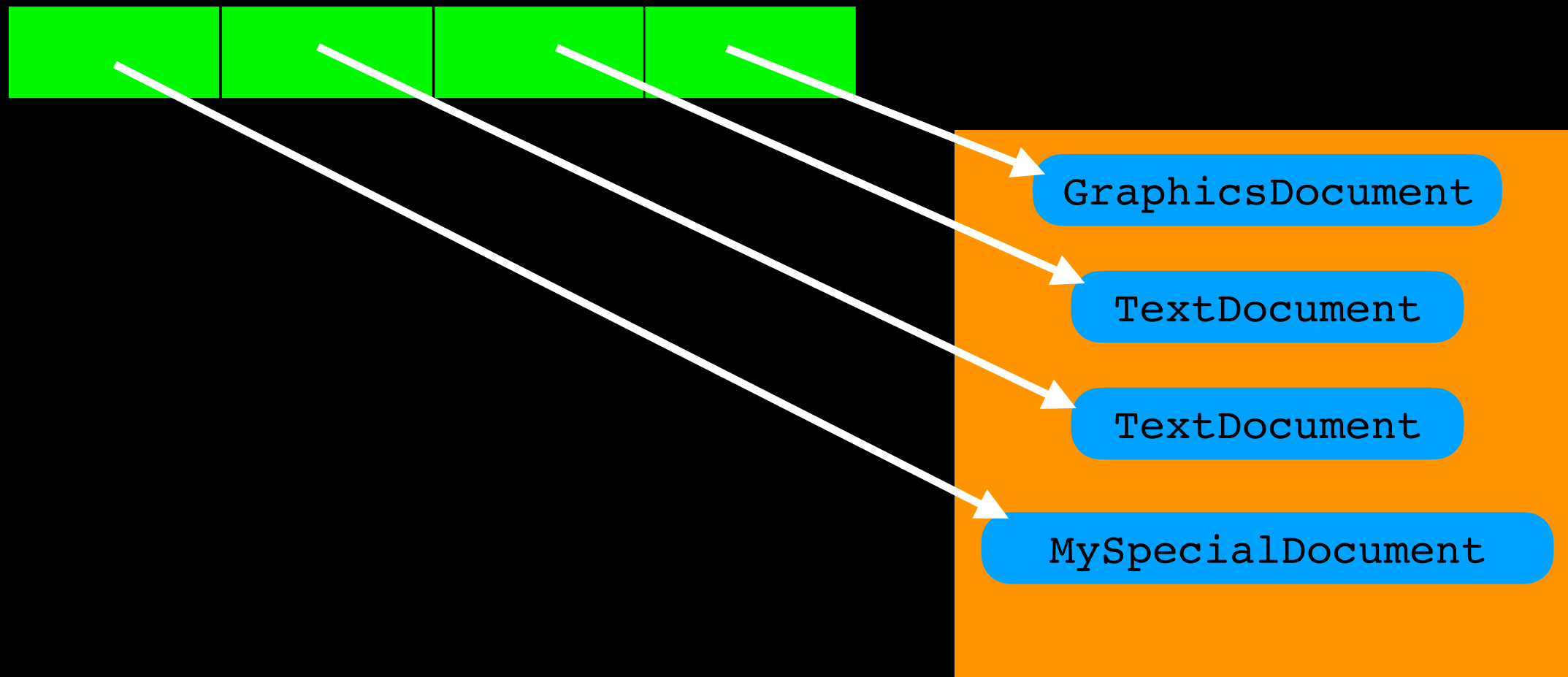
**Have implementation**

```cpp
class TextDocument: public Document

class GraphicsDocument: public Document

class PortableFormatDocument: public Document

class SpreadsheetDocument: public Document
```

But how does compiler know whose
`convertToPixelArray()` to call?
`TextDocument::convertToPixelArray?`
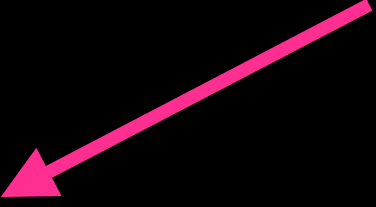`GraphicsDocument::convertToPixelArray?`

Where are we going?

I want to store all kinds of documents in my
`BatchPrinter` queue

I want to access the correct `convertToPixelArray()`
method specific to each document type

```
main()
_____

BatchPrinter myBatchPrinter;

Document* myTextDocument = new TextDocument;
Document* myGraphicsDocument = new GraphicsDocument;

//do stuff

myBatchPrinter.addDocument(myTextDocument)
myBatchPrinter.addDocument(myGraphicsDocument)


myBatchPrinter.printAllDocuments();

    myTextDocument->convertToPixelArray();
    myGraphicsDocument->convertToPixelArray();
```

TextDocument *is-a* Document
GraphicsDocument *is-a* Document
We can point to objects of derived class using pointers to base class

We store in printer queue pointers to Document
but really can access any derived class document

convertToPixelArray
is marked `virtual so`
the appropriate function call
is determined at runtime

Late Binding via Virtual Functions

Avoid statically binding function calls at compile time

Must declare functions as `virtual` for **late binding**

Polymorphism

We just saw an example of *polymorphism* (literally many forms)

With `virtual` functions the outcome of an operation is determined at execution time

With basic inheritance we were just saving ourselves the trouble of re-writing code

Abstract Class

Pure virtual function (=0) has no implementation in tis own class

Abstract class

    - Has at least one pure virtual function

    - Cannot be instantiated because does not have implementation for some/all its member functions

```
Document myDocument;        //Error!
Document* myDocument;       //Error!
```

"The curious =0 syntax was chosen over the obvious alternative of introducing a new keyword `pure` or `abstract` because at the time I saw no chance of getting a new keyword accepted. Had I suggested `pure`, Release 2.0 would have shipped without abstract classes, I chose abstract classes. Rather than risking delay and incurring the certain fights over `pure`, I used the traditional C and C++ convention of using 0 to represent 'not there' "

Bjarne Stroustrup

Polymorphism without abstraction

Superclass need not be abstract

Virtual functions in superclass need not be pure virtual

Polymorphism without Abstract Classes

```cpp
class RollerBlader()
{
public:
    //constructor, destructor
    virtual void slowDown();
    //virtual, not pure
virtual
private:
    //stuff here
}; //end RollerBlader
```

```cpp
class InexperiencedRollerBlader:
                public RollerBlader
{
public:
    //constructor, destructor
    virtual void slowDown() override;
private:
    //stuff here
}; //end InexperiencedRollerBlader
```

```cpp
void RollerBlader::slowDown()
{
    applyBreaks();
} //end slowDown
```

```cpp
void InexperiencedRollerBlader
                ::slowDown()
{
    fallDown();
} //end slowDown
```

implementation does not have **virtual** or **override** keyword

Polymorphism without Abstract Classes

main()

```
RollerBlader* firstBlader = new RollerBlader;
firstBlader->slowDown();     // applyBreaks()


RollerBlader* secondBlader = new InexperiencedRollerBlader;
secondBlader->slowDown();        // falDown()
```

Polymorphism without Abstract Classes

Need not override non-pure virtual functions

```cpp
class StuntRollerBlader: public RollerBlader
{
public:
    //constructor, destructor - note no mention of slowDown
    void frontFlip();
    void backFlip();
private:
    //stuff here
};  //end StuntRollerBlader


// stuff here


RollerBlader* blader = new StuntRollerBlader;
blader->slowDown();        // applyBreaks()
```

Warning

```cpp
class NotVirtual
{
public:
    void notAVirtualFunction();
};  //end NotVirtual


class NotVirtualDerived: public NotVirtual
{
public:
    void notAVirtualFunction();
};  //end NotVirtualDerived




NotVirtual* nv = new NotVirtualDerived;
nv->notAVirtualFunction(); // OUCH!!! calls NotVirtual's member
                           // instead of NotVirtualDerived's member
```

When using pointers to base class, to let derived classes override functions in base class must make the base class's function virtual

More design considerations

Back to `Document` class

Assume we realize all types of documents have `width` and `height` data members

Makes sense to move them into base class

Don't want client to have direct access to data members

```
class Document:
{
public:
    //Constructor, destructor
    virtual void convertToPixelArray() const = 0;
    virtual int getPriority() const = 0;

private:
    int width, height;  //Problem!!!
    //stuff here
};  //end Document
```

29

protected Access Specifier

```cpp
class Document:
{
public:
    //Constructor, destructor
    virtual void convertToPixelArray() const = 0;
    virtual int getPriority() const = 0;

protected:
    int width, height;
    //stuff here


private:
    //stuff here
}; //end Document
```

Access Specifiers

`public`
    accessible by everyone

`private`
    accessible within class and by friends

`protected`
    accessible within class, by friends and by derived classes

Inheritance accessibility

Access in Base Class	Inheritance Method	Access in Derived Class
public	public *is-a*	public
protected		protected
private		**no access**
public	protected *is-implemented-and-inherited-as*	protected
protected		protected
private		**no access**
public	private *is-implemented-as*	private
protected		private
private		**no access**

We will not discuss the details of protected and private inheritance in this course
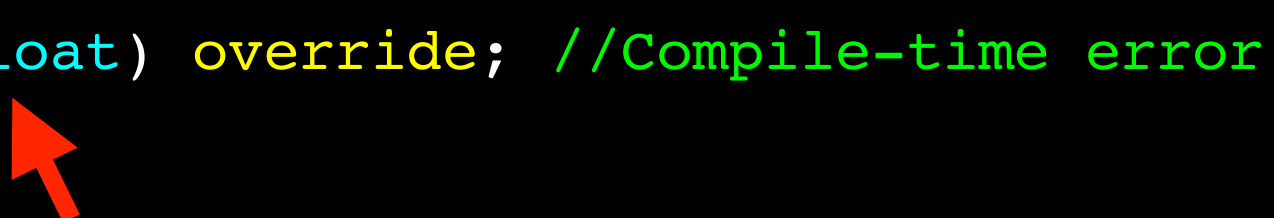
override specifier

Explicitly tell compiler you mean to override a function

Compiler will check!

Also self-documenting

```cpp
class BaseClass
{
    virtual void f(int);
};

class DerivedClass: public BaseClass
{
    virtual void f(float) override; //Compile-time error
};
```

final specifier

- Prevents inheritance
- Prevents deriving classes from overriding methods

```cpp
class A
{
    virtual void f();
};

class B : public A
{
    void f() final override; //cannot override f()
};

class C: public B final   //cannot inherit from C
{
    void f() override; //Error, f is final!    ⬅
}


class D: public C{}    //Error C is final!  ⬅
```

Programming Practice

Write `BaseClass` and `DerivedClass` and inspect behavior with different configurations
    - when do you get expected behavior?
    - when do you get errors?

Abstract `BaseClass` vs Non-abstract `BaseClass`

Understand use of `virtual`, `final` and `override` specifiers, what they mean and when appropriate together

Investigate constructor and destructor behavior (discussed next)

Recap

Polymorphism -> virtual functions

Pure vs non-pure virtual functions

Polymorphism with or without abstract classes

`override` and `final`

Need to pay **extra** attention to **destructors** when pointing to objects!!

```cpp
class BaseClass()
{
public:
   BaseClass();
   ~BaseClass();

}; //end BaseClass
```

```cpp
class DerivedClass:
             public BaseClass
{
public:
   DerivedClass();
   ~DerivedClass();

private;
   char* myString;
}; //end DerivedClass
```

```cpp
DerivedClass::DerivedClass()
{
   //allocate some memory
   myString = new char[128];
}


DerivedClass::~DerivedClass()
{
   //deallocate memory
   delete[] myString;
}
```

main()

```cpp
BaseClass* myClass = new DerivedClass;
delete myClass; //PROBLEM!!!    ←
```

BaseClass destructor is invoked.
Need to allow late binding for destructor!!!

39

```cpp
class BaseClass()
{
public:
   BaseClass();
   virtual ~BaseClass();


}; //end BaseClass

class DerivedClass:
            public BaseClass
{
public:
   DerivedClass();
   ~DerivedClass();


private:
   char* myString;
}; //end DerivedClass
```

Fix →

```cpp
DerivedClass::DerivedClass()
{
   //allocate some memory
   myString = new char[128];

}


DerivedClass::~DerivedClass()
{
   //deallocate memory
   delete[] myString;

}
```

main()
_____
```cpp
BaseClass* myClass = new DerivedClass;
delete myClass; // both destructors
                //invoked
```

Problem fixed! BOTH destructors invoked

Virtual Functions in Constructors and Destructors

Recall
- `BaseClass` constructor invoked before `DerivedClass`'
- `DerivedClass` destructor invoked before `BaseClass`'

If virtual function in constructor/destructor is called polymorphically could try to access uninitialized/deallocated data

C++ prevents this by calling virtual functions in constructors/ destructors **non-polymorphically**

```cpp
class BaseClass()
{
public:
    BaseClass()
    {
        someVirtualFunction();
    }
    virtual void someVirtualFunction()
    {
        cout << "Base" << endl;
    }
};  //end BaseClass
```

```cpp
class DerivedClass: public BaseClass
{
public:

    virtual void someVirtualFunction()
    {
        cout << "Derived" << endl;
    }

};  //end DerivedClass
```

```cpp
main()

DerivedClass myDerivedClas;
```
Standard output:
Base

42

Runtime Costs of Virtual Functions

Function call overhead
- C++ maintains *virtual function tables* that store pointers to each virtual function
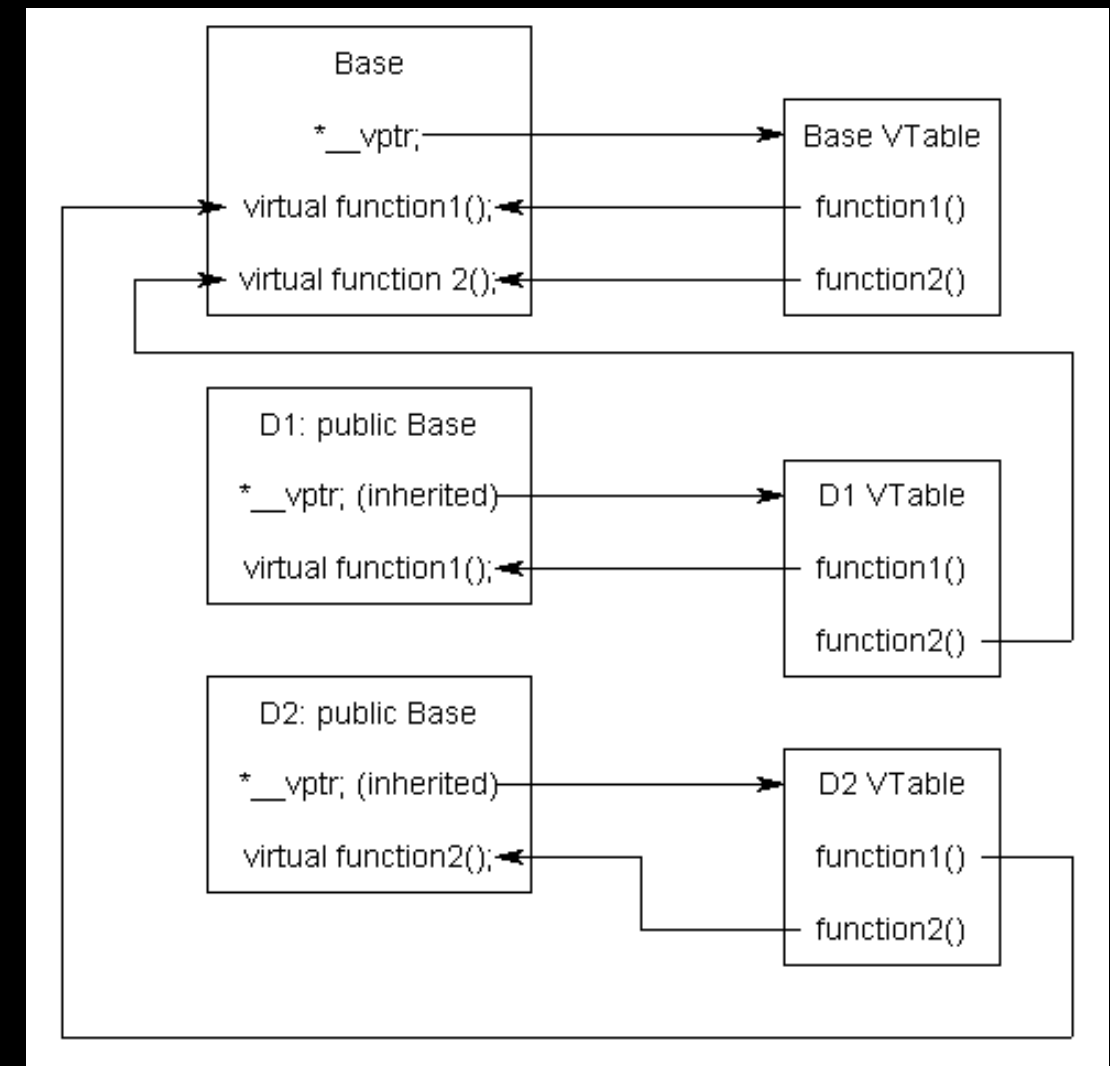- Determine which function to call at execution time by looking-up **v-table** of object being pointed to

Clever! But still
    Slower
    Extra space for v-tables

Overhead ->mark individual functions virtual to take advantage of polymorphism only when appropriate

Fully polymorphic inheritance would be overkill in most cases

Invoking Virtual Members Non-Virtually

Sometimes may need to call the `BaseClass` version of a virtual function from a `DerivedClass`

```cpp
void DerivedClass::someFunction()
{
    BaseClass::someVirtualFunction(); // no polymorphism
    //do more stuff

}
```

Copy Constructors and Assignment Operators with Inheritance

Can become complicated beasts with inheritance!!!

Must **always call explicitly** BaseClass within DerivedClass

```cpp
class Base()
{
public:
    Base();
    Base(const Base& other);
    Base& operator=(const Base& other);
    virtual ~Base();
    //other public and protected members here that will be inherited

}; //end BaseClass
```

```cpp
                    class Derived: public Base
                    {
                    public:
                        Derived();
                        Derived(const Derived& other);
                        Derived& operator=(const Derived& other);
                        virtual ~Derived();
                    private:
                        char* theString;  //a C string
                        //generic helper functions
                        void copyOther(const Derived& other);
                        void clear();
                    }; //end DerivedClass
```

Derived Implementation

```cpp
//gerenic "copy other" private member function
void Derived::copyOther(const Derived& other)
{
    theString = new char[strlen(other.theString)+1];
    strcpy(theString, other.theString);
}

// clear out private member function
void Derived::clear()
{
    delete[] theString;  //deallocate memory
    theString = NULL;     //avoid dangling pointer
}
```

Derived Incorrect Implementation

```cpp
//copy constructor
Derived::Derived(const Derived& other)
{
    copyOther(other);
}

//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```
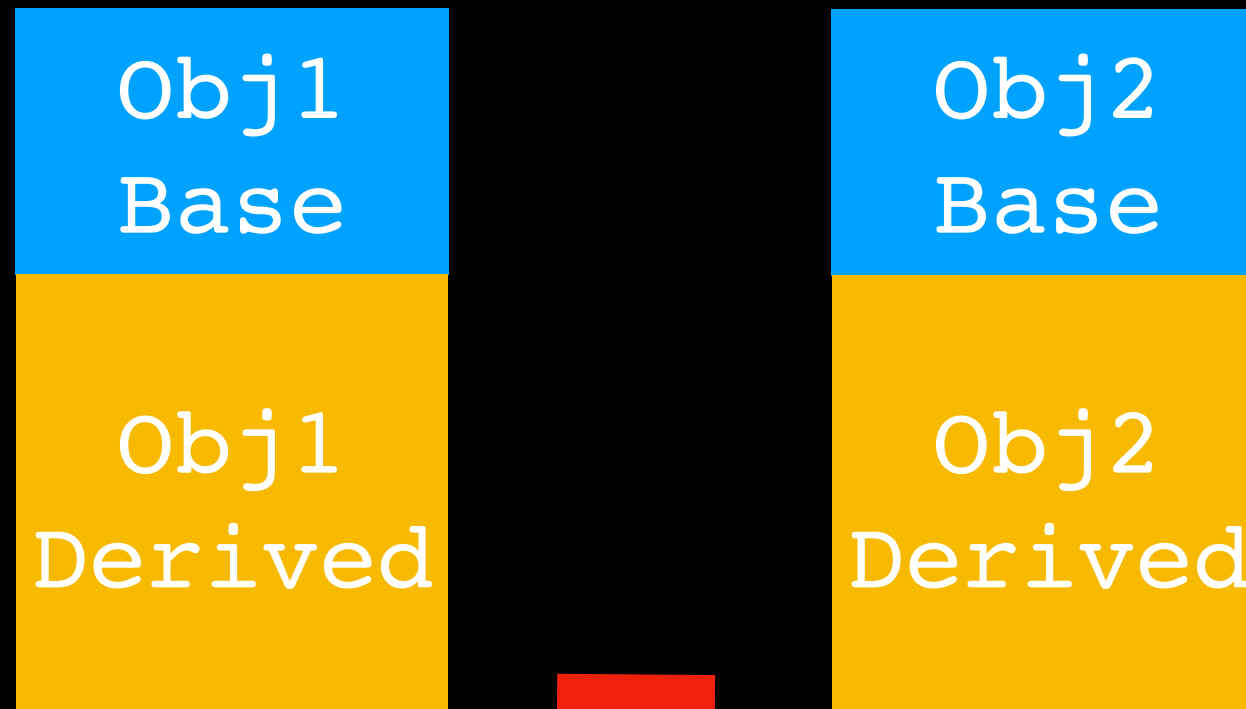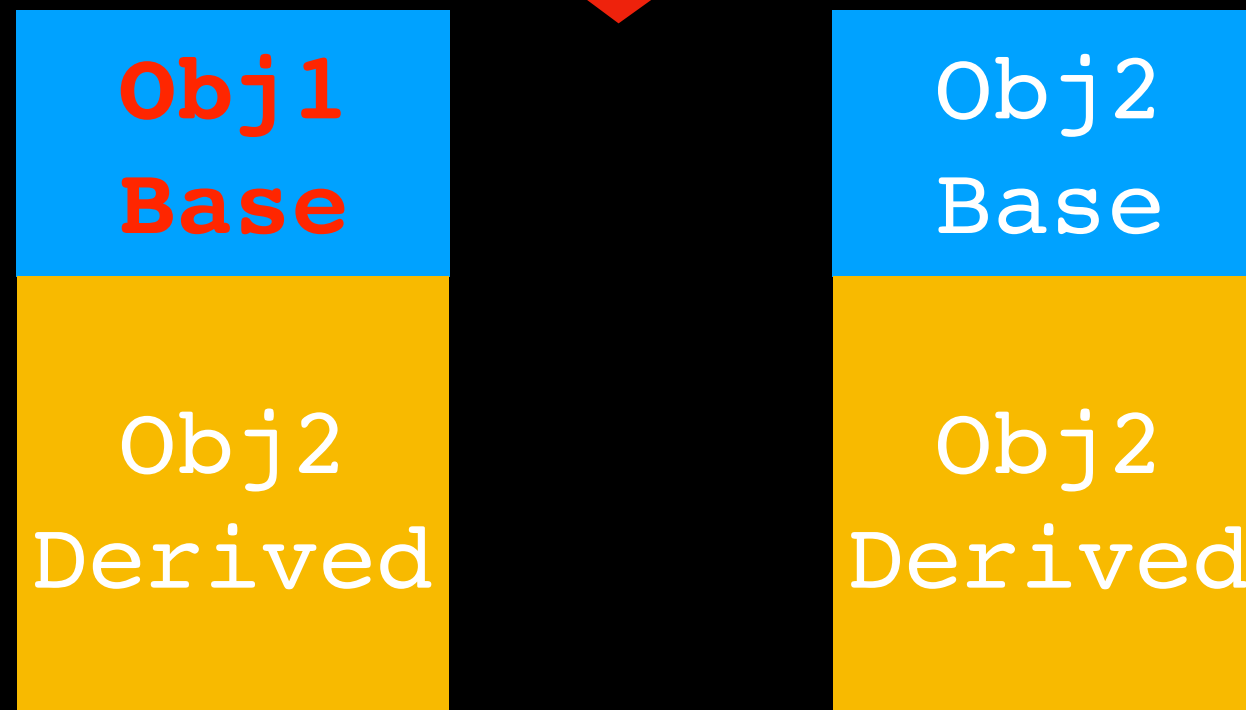
Derived Incorrect Implementation

```cpp
//copy constructor
Derived::Derived(const Derived& other)
{
    copyOther(other);                      //WRONG!!!   ⬅
}


//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        copyOther(other);             //WRONG!!!   ⬅
    }
    return *this;
}
```

49

Obj1
Base

Obj1
Derived

Obj2
Base

Obj2
Derived

**After invoking copy constructor
or assignment operator**

PROBLEM!!!
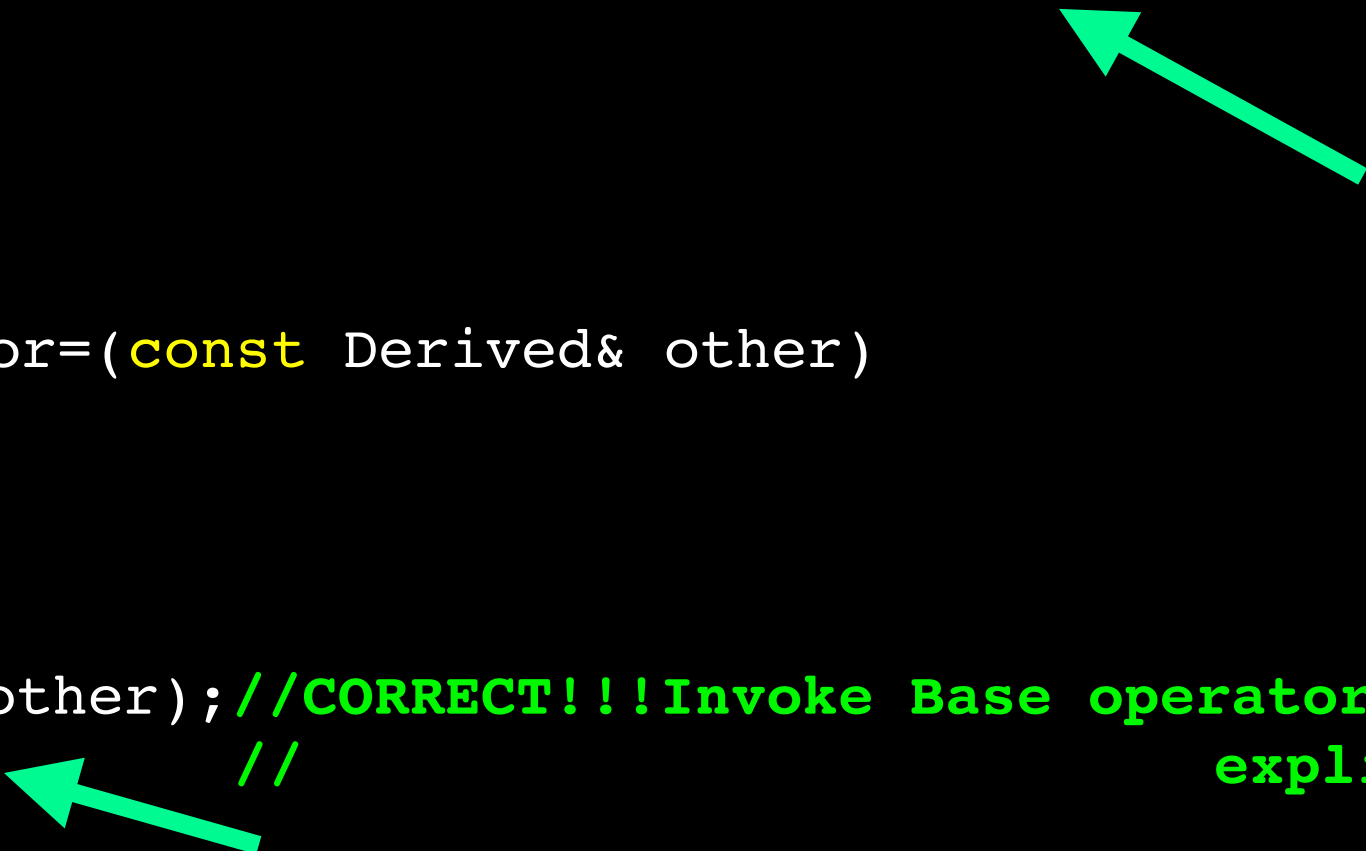
Obj1
Base

Obj2
Derived

Obj2
Base

Obj2
Derived

Derived Correct Implementation

```cpp
//copy constructor
Derived::Derived(const Derived& other): Base(other) //CORRECT!!!
{
    copyOther(other);
}

//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        Base::operator= (other);//CORRECT!!!Invoke Base operator=
                                //                         explicitly
        copyOther(other);
    }
    return *this;
}
```

Slicing

Copy ONLY `BaseClass` portion of object
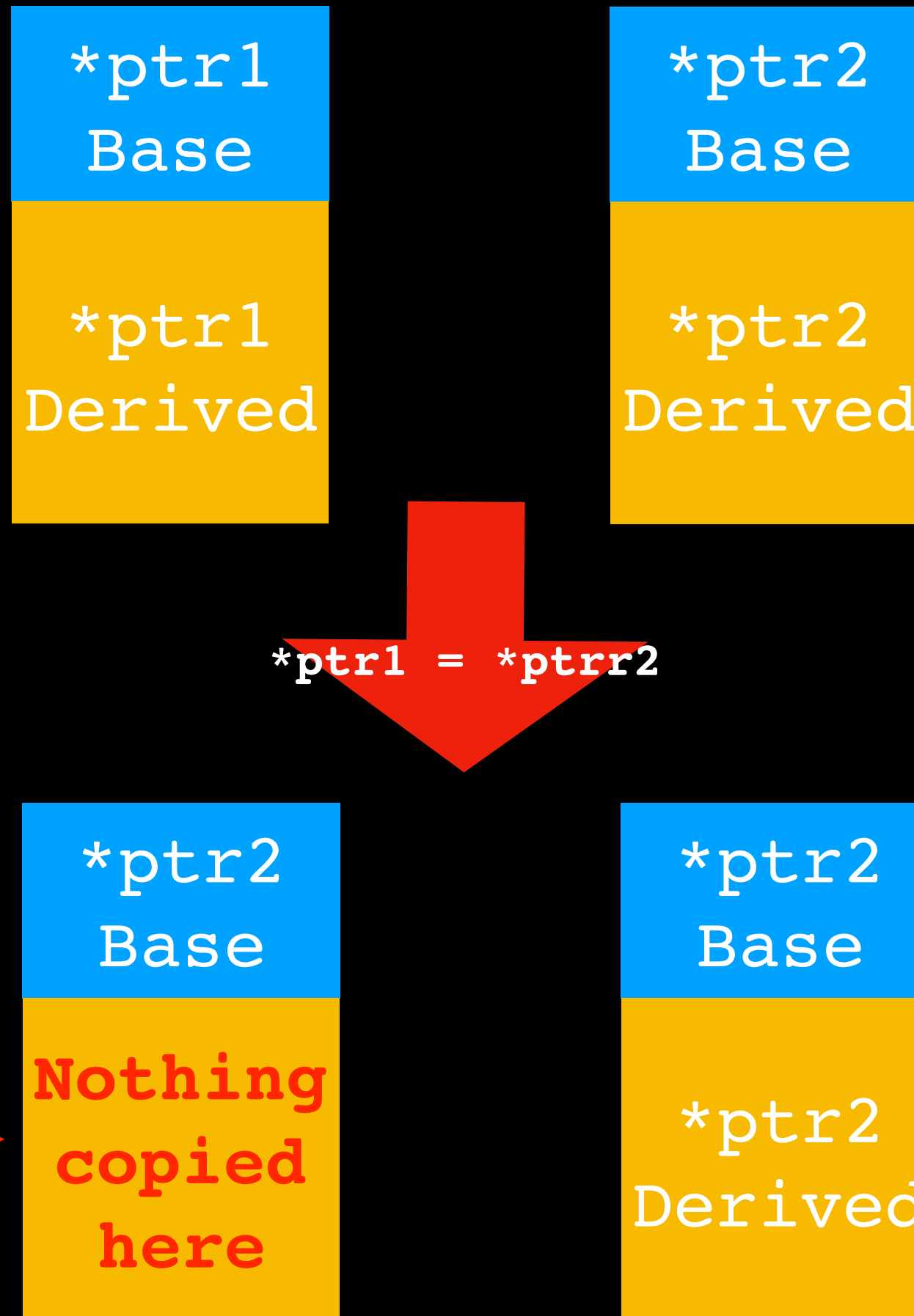
Opposite of previous case

```
Base* ptr1;
Base* ptr2 = new Derived; // pointer of type Base that points to type Derived

//do stuff

*ptr1 = *ptr2; //copy value pointed to by ptr2 into variable pointed to by
ptr1
```

Note potential problem!!!

The above expands into

```
ptr1->operator= (*ptr2);
```

Invoking the `operator=` of the `Base` loosing all data of `Derived` portion

*ptr1
Base

*ptr1
Derived

*ptr2
Base

*ptr2
Derived

*ptr1 = *ptrr2

*ptr2
Base

**Nothing
copied
here**

*ptr2
Base

*ptr2
Derived

PROBLEM!!! ➡

Slicing via Copy Constructor

```
void doSomething(Base baseObject)
{
    //do something
}


Derived myDerived;
doSomething(myDerived);
```

PROBLEM!!! Parameter `baseObject` will be initialized using `Base` copy constructor

Slicing
Ever more insidiously!!!

```
vector<Base> myBaseVector;
Base* myBasePtr = someFunction(); //pointer to Base
//ATTENTION myBasePtr could point to Derived object
myBaseVector.push_back(*myBasePtr);
```

If `someFunction` returns a pointer to an object of type `Derived` calling `push_back` on object of type `Derived` will likely **slice** the object storing only its `Base` data

Possible solution: store pointers in myBaseVector instead of objects

Casting

Forcing one datatype to be converted into another

Up-casting (`Derived` to `Base`) automatically available through inheritance

```
Base* basePtr;
Derived* derivedPtr;
//do stuff
basePtr = derivedPtr; //automatic conversion Derived is-a Base
```
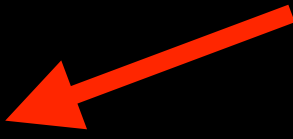
Down-casting (`Base` to `Derived`)

```
Base* basePtr = new Derived; // pointer of type Base points to Derived
//do stuff
Derived* derivedPtr = (Derived*) basePtr;
```
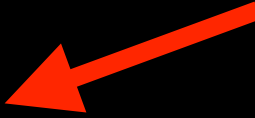
Casting

Classic C++ cast too powerful => no checks.
Could write something totally nonsensical

```
Base* basePtr;
vector<double>* myVectorPtr = (vector<double>*) basePtr;
//PROBLEM!! Makes no sense, no compiler error


const Base* basePtr = new Derived;
// do stuff
Derived* derivedPtr = (Derived*) basePtr;
//PROBLEM!!! Lost constness of Base object
//derivedPtr  is now free to modify it
```

static_cast

`static_cast` checks at compile time that cast "makes sense"

Allows:
- Converting between primitive types (e.g. `int` to `float`)
- Converting pointers or references of `Derived` type to pointers or references of `Base` type (e.g. `Derived*` to `Base*`) where target is at least as `const` as the source
- Converting pointers or references of `Base` type to pointers or references of `Derived` type (e.g. `Base*` to `Derived*`) where target is at least as `const` as the source

```cpp
Base* basePtr = new Derived;
// do stuff
Derived* derivedPtr = static_cast<Derived*>(basePtr);
```
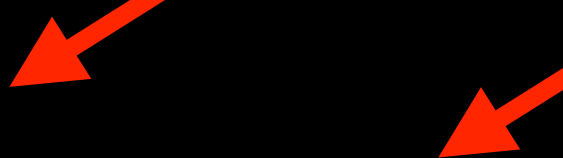
dynamic_cast

If `Base*` did not point to `Derived` object, `static_cast` would succeed

=> runtime problems

e.g. access `Derived` data members not present in `Base`

```cpp
Base* basePtr = new Base;
Derived* derivedPtr1 = (Derived*)basePtr; //BAD!!!
Derived* derivedPtr2 = static_cast<Derived*>(basePtr); //BAD!!!
Derived* derivedPtr3 = dynamic_cast<Derived*>(basePtr); //GOOD!!!
```

Will return a NULL pointer

Conclusion

Polymorphism is easy, Just put `virtual` everywhere and the compiler will take care of the rest!

Conclusion

Polymorphism is easy, Just put `virtual` everywhere and the compiler will take care of the rest!

Conclusion

Overhead! Use it only when useful/necessary

Carefully craft constructors

Make destructor virtual

Beware of Slicing (in all its forms)

Beware of casting and use level most appropriate and safe for your situation

For Project 3

- No polymorphism (if we were, which functions should we have marked virtual? In which class?)

- Pay attention to copy constructor

```cpp
public:

    LinkedSet();        // Default constructor

    LinkedSet(const LinkedSet<ItemType>& a_set); // Copy constructor

    virtual ~LinkedSet();   // Destructor should be virtual

    int getCurrentSize() const;

    bool isEmpty() const;

    bool add(const ItemType& new_entry);

    bool remove(const ItemType& an_entry);

    void clear();

    bool contains(const ItemType& an_entry) const;

    std::vector<ItemType> toVector() const;
```