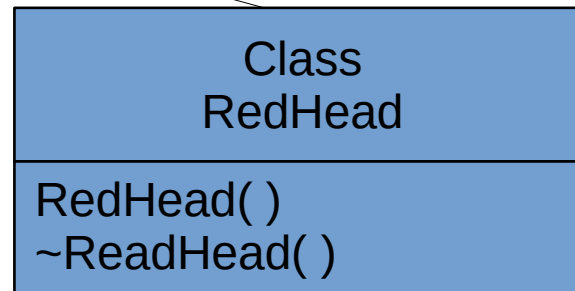
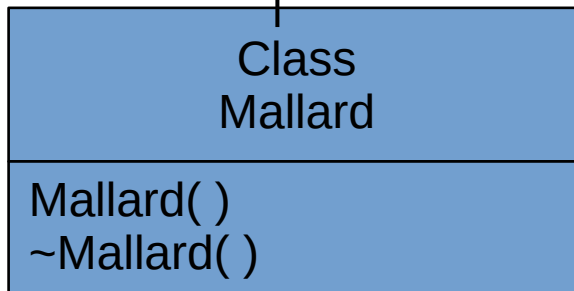
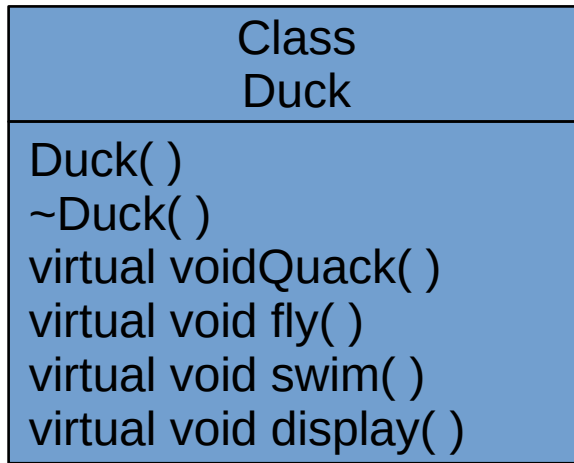


# ISA vs HASA

## When to use

# Consider the code/program diagrammed below



# Duck.h, Duck.cpp

```
class Duck {  
public:  
  
    Duck(std::string);  
    Duck( );  
    virtual ~Duck( );  
  
    virtual void quack( );  
    virtual void fly( );  
    virtual void swim( );  
    virtual void display( );  
  
private:  
    std::string kind;  
};
```

```
Duck::Duck( ) {  
    kind = "generic duck";  
}  
  
Duck::Duck(std::string k) {  
    kind = k;  
}  
  
Duck::~~Duck( ){ }  
  
void Duck::quack( ) {std::cout << "quack" << std::endl;}  
  
void Duck::fly( ) {std::cout << "fly" << std::endl;}  
  
void Duck::swim( ) {std::cout << "swim" << std::endl;}  
  
void Duck::display( ) {std::cout << kind << std::endl;}
```

# Mallard.h, Mallard.cpp

```
#ifndef MALLARD_H_  
#define MALLARD_H_  
#include "Duck.h"  
  
class Mallard : public Duck {  
public:  
  
    Mallard( );  
    virtual ~Mallard( );  
  
};  
#endif /* MALLARD_H_ */
```

```
#include <iostream>  
#include "Mallard.h"
```

```
Mallard::Mallard( ) : Duck("Mallard") { }  
Mallard::~~Mallard( ) { }
```

RedHead.h and RedHead.cpp are very similar. Just change all of the “Mallard” and “MALLARD” to “RedHead” and “READHEAD”.

# DuckSim.cpp

```
#include <iostream>
#include "Duck.h"
#include "Mallard.h"
#include "RedHead.h"
```

```
int main (int argc, char *argv[]) {
```

```
    Duck* ducks[4];
```

```
    ducks[0] = new Mallard( );
    ducks[1] = new RedHead( );
    ducks[2] = new Mallard( );
    ducks[3] = new Duck( );
```

```
        for (int i = 0; i < 4; i++) {
            ducks[i]->display( );
            ducks[i]->quack( );
            ducks[i]->fly( );
            std::cout << std::endl;
        }
    }
```

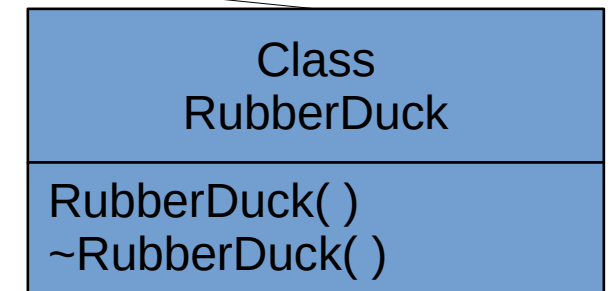
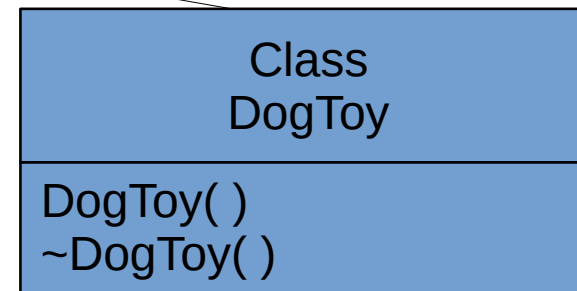
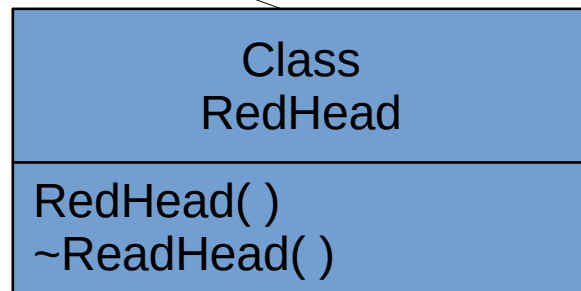
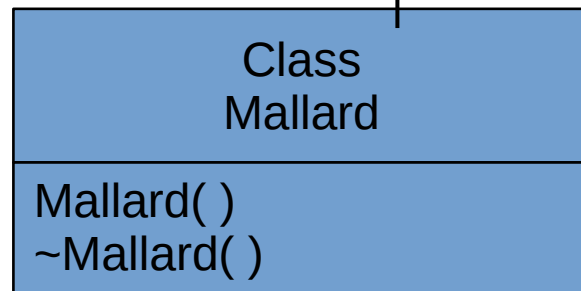
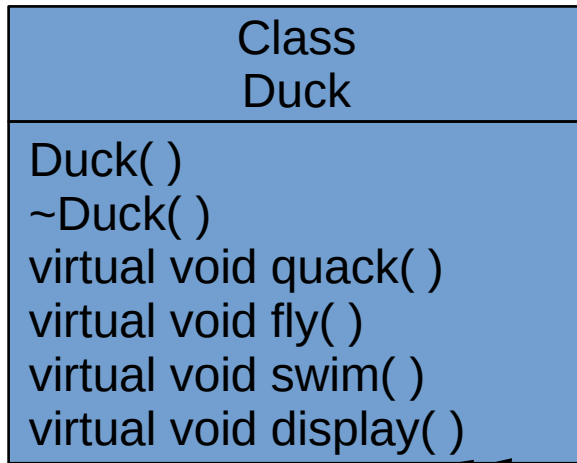
Mallard  
quack  
fly

RedHead  
quack  
fly

Mallard  
quack  
fly

generic duck  
quack  
fly

# Let's add more ducks!



```
#include <iostream> DuckSim.cpp with more ducks!
```

```
#include "Duck.h"
```

```
#include "Mallard.h"
```

```
#include "RedHead.h"
```

```
#include "DogToy.h"
```

```
#include "Rubber.h"
```

```
for (int i = 0; i < 5; i++) {  
    ducks[i]->display( );  
    ducks[i]->quack( );  
    ducks[i]->fly( );  
    std::cout << std::endl;  
}
```

```
int main (int argc, char *argv[]) {  
    }  
}
```

```
Duck* ducks[5];
```

```
ducks[0] = new Duck( );
```

```
ducks[1] = new Mallard( );
```

```
ducks[2] = new RedHead( );
```

```
ducks[3] = new Rubber( );
```

```
ducks[4] = new DogToy( );
```

generic duck  
quack  
fly

Mallard  
quack  
fly

RedHead  
quack  
fly

Rubber  
quack  
fly

DogToy  
quack  
fly

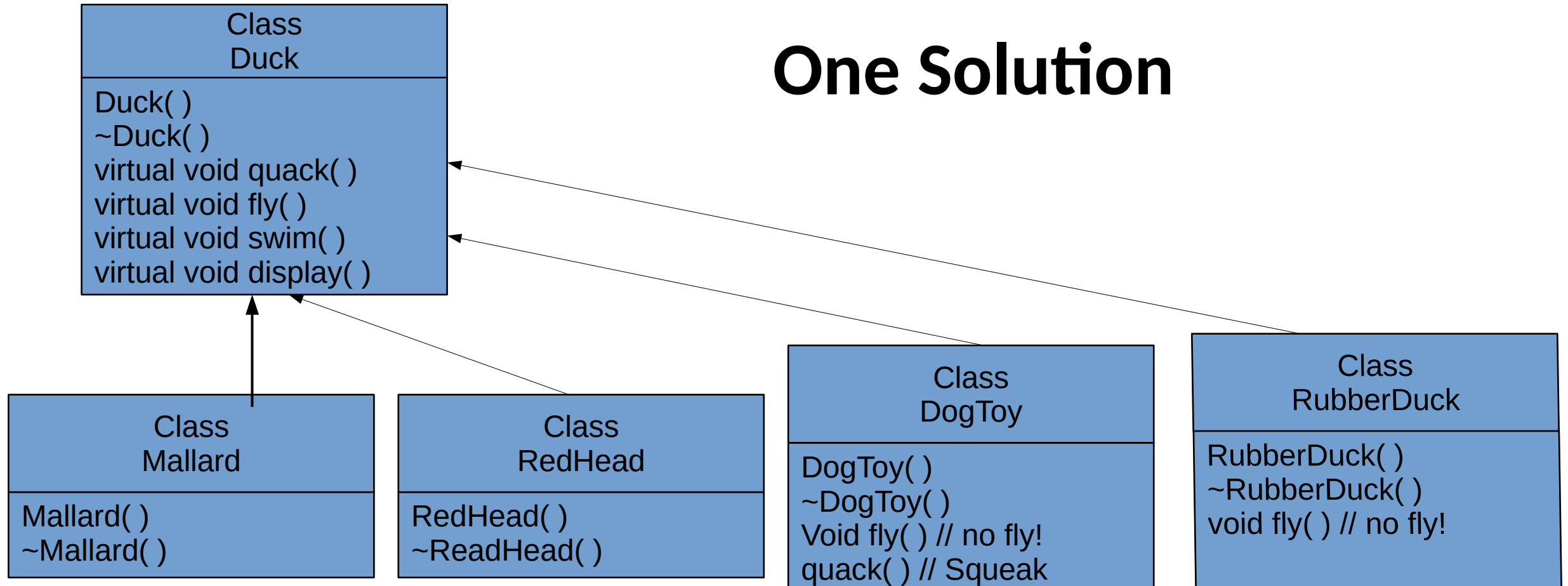
It's always good to test your code against reality

- I tried to find a Mallard and RedHead duck and bring in to show how they worked, but I didn't have any around the house.
  - But I think we'll all agree (based on the pictures in the previous slide) that they fly. And I've heard them quack, so all seems good.
- I do have a rubber duck, however. Let's test it.



The original code works about as well as lots of code I've written, which is to say, not as well as I would like.

## One Solution



# But this also has problems.

- Fly and quack are part of the base class Duck
- Fly and quack should be attributes of every duck
- We can get around the different properties of flying and quacking in the base class by using overriding, but it seems strange for something that is supposedly a property of ducks

# And more problems . . .

- And what about good old DogToy duck?
  - It squeaks when you first get it
  - Then it meets up with friendly dog
  - It gets chewed, shaken, and becomes very silent
  - How do we handle that?
  - The same is true if we had a duckling (true that it's quack will change, not that it falls victim dog)
- Suddenly something very complicated is starting to look kind of messy.
- Homework 3 has the solution, and we'll talk about the right way to do it after you complete homework 3.



# Access levels and encapsulation in C++

# File extensions

- C++ programs, like C programs, have *header* files and *code* files
- C++ header files have the extension of .h, just like C. You can also use .hpp
- .h/.hpp files often declare symbols and declare classes. With C++, as with C, they should not be included/expanded multiple times during a compilation of a file
- GCC recognizes .C, .cc, .cpp, .CPP, .c++, .cp, or .cxx as files.
  - Other compilers may use a subset of these
  - Some IDEs may want one format or another
  - Let's look at a .h/.hpp file for a class.

# Include files

```
#ifndef POLYGON_H_  
#define POLYGON_H_  
#include <string>  
Using namespace std;  
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    ~Polygon( );  
    float getArea( );  
    string getName( );  
  
protected:  
    int numSides;  
    float lenSides;  
  
};  
#end /* POLYGON_H_ */
```

Include files must  
only be  
expanded once

.h files *declare*  
classes.

Include string.h, just  
like C

Brings a bunch of names into scope.  
Shouldn't use in .h files – can cause bad  
things to happen if the .h is included in a .cpp  
that uses this.

Declare attributes  
(fields) and methods

# Include files

```
#ifndef POLYGON_H_  
#define POLYGON_H_  
#include <string>  
Using namespace std;  
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    ~Polygon( );  
    virtual float getArea( );  
    string getName( );  
  
protected:  
    int numSides;  
    float lenSides;  
  
};  
#end /* POLYGON_H_ */
```

The class name (Polygon).  
File name should be the  
same, but this is not  
required.

*public* says the methods and fields are visible  
to inheriting classes and code outside the  
class. *public* access level provides the  
highest visibility.

*protected* says the  
declared items are visible  
in inheriting classes but  
not to other classes

The is *private* access level  
that says the declared  
items are only visible in  
this class.

# Include files

```
#ifndef POLYGON_H_  
#define POLYGON_H_  
#include <string>  
Using namespace std;  
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    ~Polygon( );  
    float getArea( );  
    string getName( );  
  
protected:  
    int numSides;  
    float lenSides;  
  
};  
#end /* POLYGON_H_ */
```

The prototype for the *constructors*. These will be discussed later.

The prototype for the *destructor*. This will be discussed later.



# Some terminology

```
#ifndef POLYGON_H_  
#define POLYGON_H_  
#include <string>  
Using namespace std;  
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    ~Polygon( );  
    float getArea( );  
    string getName( );  
  
protected:  
    int numSides;  
    float lenSides;  
  
};  
#end /* POLYGON_H_ */
```

Signature is  
*Polygon+int+float*

Signature is  
getArea+void

In OO languages, methods are identified by their signatures. The signature is the name + the argument types. *The return value is not part of the signature.*

# The Square class

```
#include "Polygon.h"  
#include <string>
```

Because Square is derived from Polygon, we need to include Polygon.h so the compiler knows what is available from Polygon for this class.

```
class Square : public Polygon {  
public:  
    Square(float);  
    ~Square( );  
  
    float getArea( );  
  
};  
#endif
```

This says that Square is derived from Polygon. *public* says that everything in Polygon that can be reached through a Square object is as public as it would have been reached through a Polygon object.

Defines its own getArea class that overrides Polygon's getArea class

# How access levels work in C++

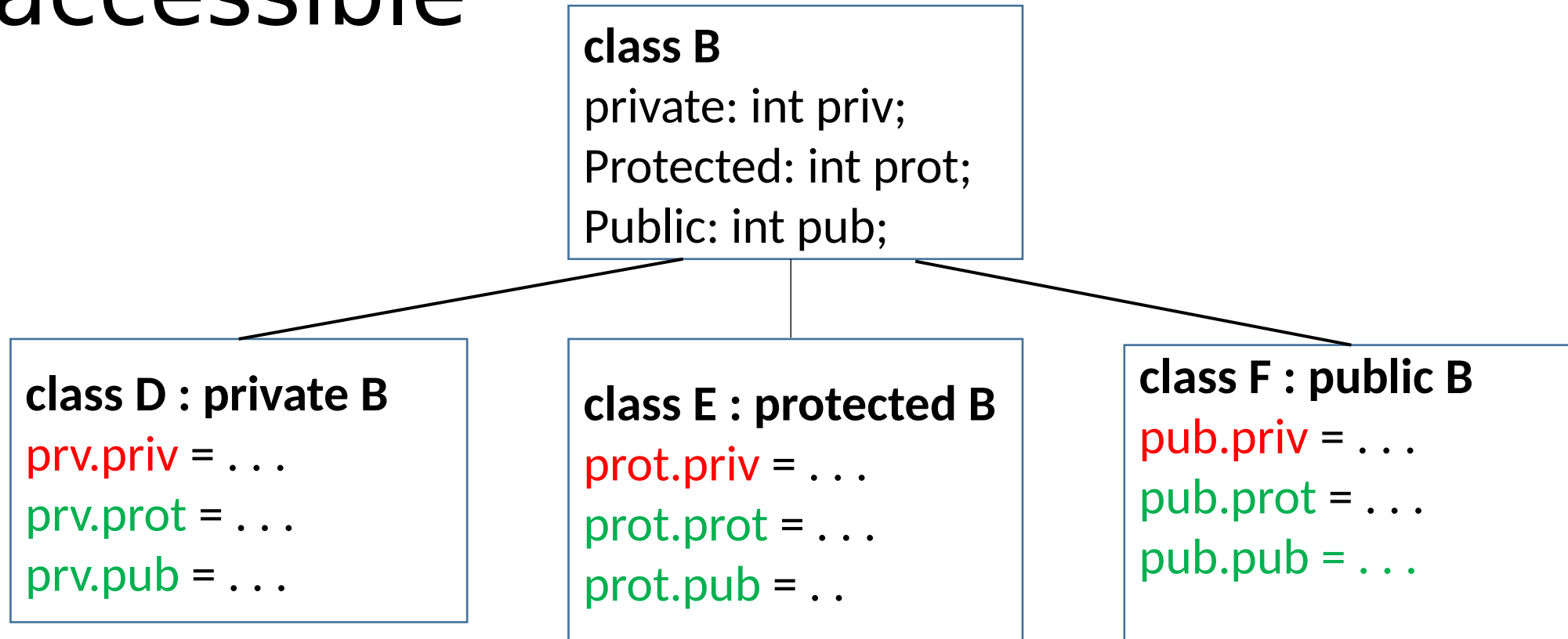
- private: accessible only within the class where the object or method is declared. This is the default for class inheritance and data members (class variables).
  - *In a class, members are by default private; in a struct, members are by default public (§16.2.4). (the C++ Programming Language, Stroustrup)*
- protected: accessible within the class the object is declared in, and classes that extend or inherit from this class.
  - Inheritance can be direct or indirect
  - For example, if B is the base class, and class D extends B, and class E extends D, within class E protected variables of B can accessed
- public: accessible within the class where it is declared and everywhere else that someone can access the object the variable is in
- Access level can be modified by an inheriting class *for that class*



# Access levels and inheritance

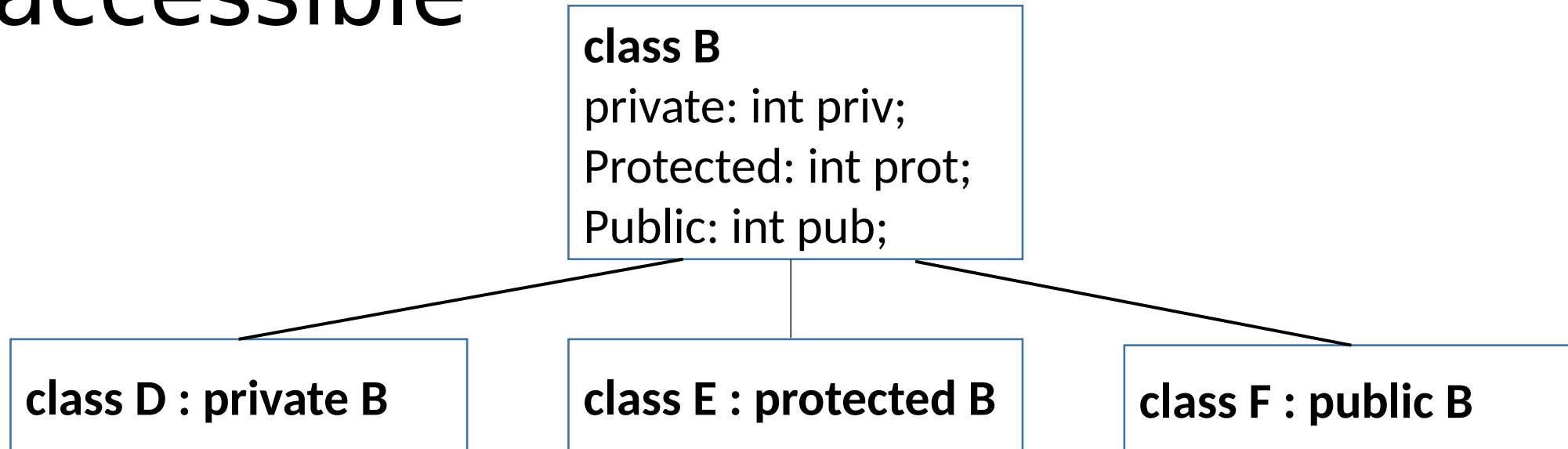
- When inheriting we can always make something more private, but we cannot make it less private.
- Let a variable  $v$  be declared with access level  $l$  in class  $B$
- Let a class  $X$  inherit  $B$  with inheritance access level  $z$
- Then accesses of  $v$  through  $X$  will be at the level of  $\min(l, z)$ , where an access level is smaller if it is more private.
- The default for methods, fields and inheritance is *private*.
  - Always make things private unless you have a reason to make them public.
  - It's good practice to say private: and not rely on defaults. It is a form of documentation.

# Green is can access, Red is not accessible



Shows access *within* the inheriting class

# Green is can access, Red is not accessible



```
main:  
B base = new B( );  
D prv = new D( );  
E prot = new E( );  
F pub = new F( );
```

← Typo fixed

```
base.priv = ...  
base.prot = ...  
base.pub = ...
```

```
prv.priv = ...  
prv.prot = ...  
prv.pub = ...
```

```
prot.priv = ...  
prot.prot = ...  
prot.pub = ...
```

```
pub.priv = ...  
pub.prot = ...  
pub.pub = ...
```

Show access *outside* of  
the class

# Some more terminology

- *Methods* are any function that is declared within a class and can access class information
- All functions in Java are methods
- C++ can declare functions outside of classes
- A method is said to be *bound* to its class
  - In Java and C++ this happens at compile time and it is called *static* binding

# .cpp files for classes

```
#include "Polygon.h"
```

```
#include <math.h>
```

Include the .h file for the class being defined

```
Polygon::Polygon(int n, float l) :  
    numSides(n), lenSides(l) { }
```

Include other needed .h files

```
Polygon::Polygon( ) { };
```

```
Polygon::~~Polygon( ) { };
```

The *constructor* definition. It tells how to initialize the object.

```
float Polygon::getArea( ) {  
    float p = numSides * lenSides;  
    float apo = numSides / 2. * tan(180. / numSides);  
    return apo * p / 2.;  
}
```

The *destructor* definition. The destructor is executed after the object is *deleted*, i.e., destroyed.

.cpp files  
*define*  
classes



# .cpp files for classes

```
#include "Polygon.h"  
#include <math.h>
```

```
Polygon::Polygon(int n, float l) :  
    numSides(n), lenSides(l) { }
```

```
Polygon::Polygon( ) { };  
Polygon::~~Polygon( ) { };
```

```
float Polygon::getArea( ) {  
    float p = numSides * lenSides;  
    float apo = numSides / 2. * tan(180. / numSides);  
    return apo * p / 2.;  
}
```

This is the default constructor. Put this in your code for now – we'll discuss the more in a few weeks.

This is the definition of the `getArea( )` method bound to the class, and objects of the class.

# .cpp files for classes

```
#include "Polygon.h"  
#include <math.h>
```

```
Polygon::Polygon(int n, float l) :  
    numSides(n), lenSides(l) { }
```

```
Polygon::Polygon( ) { };
```

```
Polygon::~~Polygon( ) { };
```

```
float Polygon::getArea( ) {  
    float p = numSides * lenSides;  
    float apo = numSides / 2. * tan(180. / numSides);  
    return apo * p / 2.;  
}
```

Since the declaration and definition of a class are in different files, and the name of the file has not meaning, definitions of methods, constructors and destructors are always preceded by *Classname::*

# How to compile a C++ program

- Use the compilation method of your IDE (Interactive Development Environment)
  - Eclipse (works with Java and C++ with the right plugins)
  - Vendor specific IDE, Netbeans (Java), Visual Studio (Microsoft)
  - For this course I recommend you use an IDE
    - It is a good thing to know
    - It will make you more productive
    - With Java, you don't have a debugger if you are not using an IDE
- Use the command line

# A common problem with command line compiles

\$ gcc main.cpp

```
/tmp/cc298fP3.o: In function `main':
main.cpp:(.text+0x21): undefined reference to `operator new(unsigned long)'
main.cpp:(.text+0x34): undefined reference to `Square::Square(float)'
main.cpp:(.text+0x42): undefined reference to `operator new(unsigned long)'
main.cpp:(.text+0x55): undefined reference to `Triangle::Triangle(float)'
main.cpp:(.text+0x63): undefined reference to `operator new(unsigned long)'
main.cpp:(.text+0x7b): undefined reference to `Polygon::Polygon(int, float)'
main.cpp:(.text+0x89): undefined reference to `operator new(unsigned long)'
main.cpp:(.text+0x9c): undefined reference to `Pentagon::Pentagon(float)'
main.cpp:(.text+0xbe): undefined reference to `Polygon::getArea()'
main.cpp:(.text+0xde): undefined reference to `Polygon::getArea()'
main.cpp:(.text+0xf0): undefined reference to `std::cout'
main.cpp:(.text+0xf5): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)'
main.cpp:(.text+0x102): undefined reference to `std::ostream::operator<<(int)'
main.cpp:(.text+0x10f): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)'
main.cpp:(.text+0x11f): undefined reference to `std::ostream::operator<<(float)'
main.cpp:(.text+0x124): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)'
main.cpp:(.text+0x12c): undefined reference to `std::ostream::operator<<(std::ostream& (*)(std::ostream&))'
main.cpp:(.text+0x144): undefined reference to `std::cout'
main.cpp:(.text+0x149): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)'
main.cpp:(.text+0x15f): undefined reference to `std::ostream::operator<<(float)'
main.cpp:(.text+0x164): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)'
main.cpp:(.text+0x16c): undefined reference to `std::ostream::operator<<(std::ostream& (*)(std::ostream&))'
main.cpp:(.text+0x17e): undefined reference to `operator delete(void*)'
main.cpp:(.text+0x194): undefined reference to `operator delete(void*)'
main.cpp:(.text+0x1aa): undefined reference to `operator delete(void*)'
main.cpp:(.text+0x1c0): undefined reference to `operator delete(void*)'
/tmp/cc298fP3.o: In function `__static_initialization_and_destruction_0(int, int)':
main.cpp:(.text+0x1fb): undefined reference to `std::ios_base::Init::Init()'
main.cpp:(.text+0x20a): undefined reference to `std::ios_base::Init::~Init()'
/tmp/cc298fP3.o:(.eh_frame+0x13): undefined reference to `__gxx_personality_v0'
collect2: error: ld returned 1 exit status
smidkiff@SMIDKIFFLT1:~/me/Documents/Work/Courses/30862Building/LecturesPowerPoint/L1/Code/Polygon$
```

# A common problem with command line compiles

```
gcc main.cpp
```

```
...
```

```
main.cpp:(.text+0x21): undefined reference to `operator new(unsigned long)'
```

```
main.cpp:(.text+0x34): undefined reference to `Square::Square(float)'
```

```
main.cpp:(.text+0x42): undefined reference to `operator new(unsigned long)'
```

```
main.cpp:(.text+0x55): undefined reference to `Triangle::Triangle(float)'
```

```
main.cpp:(.text+0x63): undefined reference to `operator new(unsigned long)'
```

```
main.cpp:(.text+0x7b): undefined reference to `Polygon::Polygon(int, float)'
```

# Easily solved

```
$ g++ main.cpp Pentagon.cpp Polygon.cpp Square.cpp  
Triangle.cpp  
$
```

# Interfaces that provide encapsulation

- Only make attributes and methods public if you want other people to call, read and write them directly
  - Public methods arguments cannot be easily changed in the future
  - Public attributes cannot be easily changed in the future
  - Because they can be accessed directly they are part of the interface, and the interface is a promise that must be kept.
- Use protected when you want an inheriting class to access the attribute or method but not allow them to be accessed through the object.
  - Useful when using different classes to logically organize functionality while solving a larger problem.
  - Often make the most derived class not be inheritable (we'll discuss how to do this later)
    - This keeps other programmers classes from accessing the protected attributes and methods.

# Creating objects

- C++ allows objects to be created in two ways
  1. By allocating the object on the heap (with *new*) and returning a pointer to the object
  2. By allocating the object on the runtime stack
- Objects can be accessed in three ways
  1. Through a pointer: typically done with objects created with *new*
  2. By using the name of the object variable: done with objects allocated on the runtime stack
  3. Through a *reference*: This is a discussion we will defer for later when our brains have grown sufficiently to handle it (so no breakfast club until we are past references. And Java synchronization. And to be safe, graduation.)



# Consider the following C++ class declaration . . .

```
#include <string>
```

```
class DOB {
```

```
public:
```

```
    DOB(int, int, int);
```

```
    ~DOB( );
```

```
    void print( );
```

```
private:
```

```
    std::string monthToStr[12];
```

```
    int day;
```

```
    int month;
```

```
    int year;
```

```
};
```

# . . . Definition . . .

```
#include "DOB.h"
```

```
#include <iostream>
```

```
DOB::DOB(int m, int d, int y) : month(m), day(d), year(y) {  
    monthToStr[0]="Jan"; monthToStr[1]="Feb"; monthToStr[2]="Mar";  
    monthToStr[3] = "Apr"; monthToStr[4]="May"; monthToStr[5]="Jun";  
    monthToStr[6]="Jul"; monthToStr[7] = "Aug"; monthToStr[8]="Sep";  
    monthToStr[9]="Oct"; monthToStr[10]="Nov"; monthToStr[11] = "Dec";  
}
```

```
DOB::~~DOB( ) { };
```

```
void DOB::print( ) {  
    std::cout << monthToStr[month] << " " << day << ", " << year << std::endl;  
}
```

# . . . and main function

```
#include "DOB.h"
```

```
int main (int argc, char *argv[]) {
```

```
    DOB *dp = new DOB(3,4,2020);  
    DOB dv(1,2,2020);
```

```
    dp->print( );
```

```
    dv.print( );
```

```
}
```

Create a DOB object on the heap.  
Assign its address to *dp*

Create a DOB variable *dv* on the stack. The value of *dv* is the created object

# Consider the C++ class and main file below

```
#include "DOB.h"
```

```
int main (int argc, char *argv[]) {
```

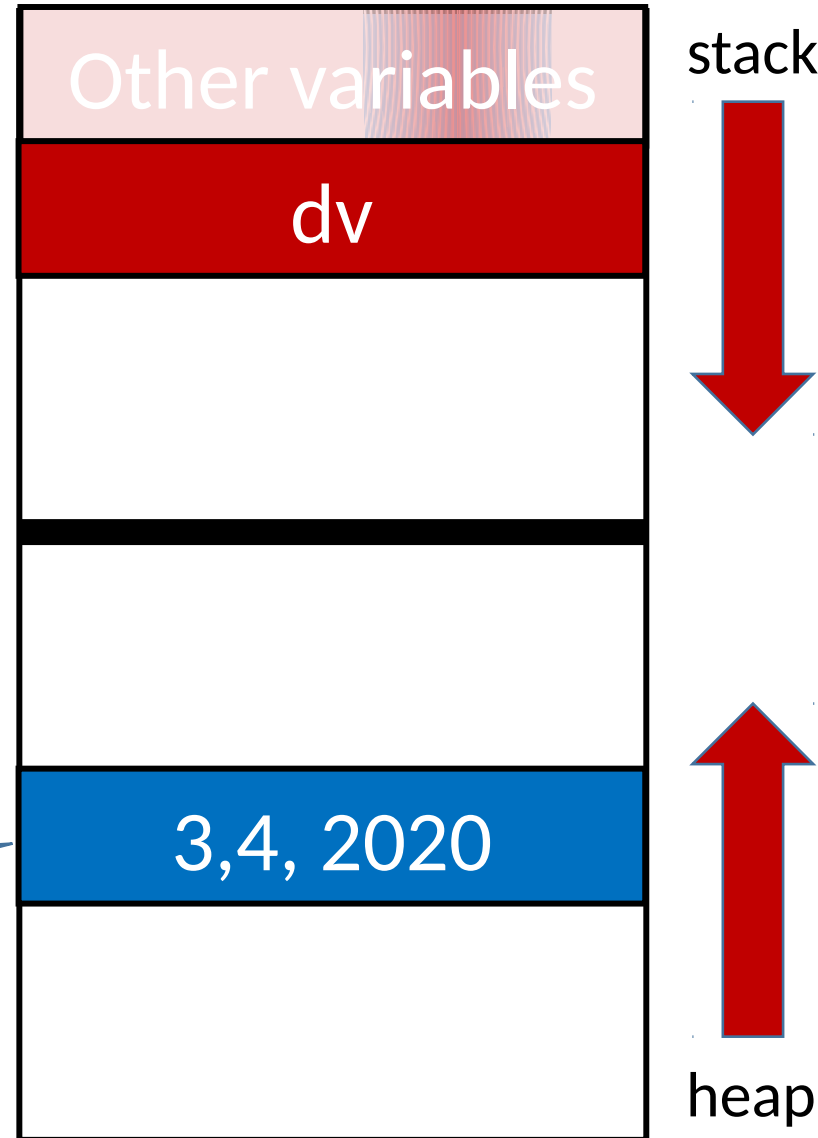
```
    DOB *dp = new DOB(3,4,2020);
```

```
    DOB dv(1,2,2020);
```

```
    dp->print( );
```

```
    dv.print( );
```

```
}
```

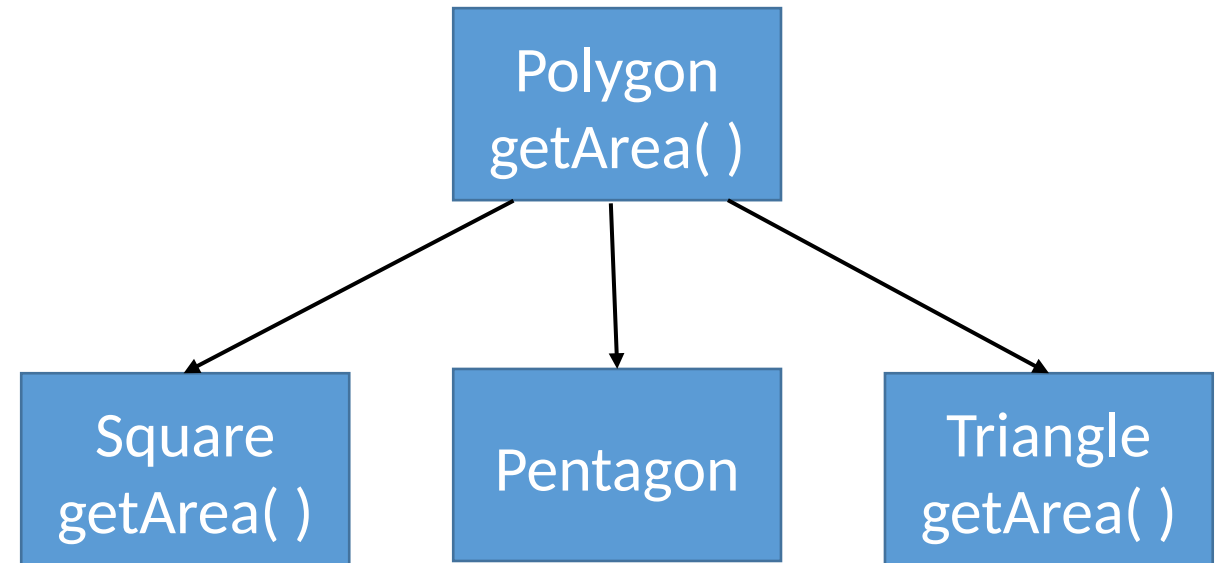


# Polymorphism

- Polymorphism is the ability of an object to act like two types of objects simultaneously.
- In practice, this means
  - Being referred to as an object of one type
  - Acting like the type of object it actually is
- Let's go back to our polygon example
- **Polymorphism only works with functions that are *virtual* (general OO), in C++ for functions that are virtual and accessed via a pointer or reference.**
- By default, functions are non-virtual

# Polymorphism

- What we would like to do is to create Square, Pentagon, Triangle and Polygon objects and have each define a `getArea( )` method.
- Then, we would like to take a vector of these objects and find the total area in all of them
- Code to do this follows.



# The Polygon class

```
#include <string>
```

```
class Polygon {
```

```
public:
```

```
    Polygon( );
```

```
    Polygon(int,float);
```

```
    ~Polygon( );
```

```
    virtual float getArea( );
```

```
protected:
```

```
    int numSides;
```

```
    float lenSides;
```

```
};
```

```
#include "Polygon.h"
```

```
#include <math.h>
```

```
Polygon::Polygon(int n,float l) :  
    numSides(n), lenSides(l) { }
```

```
Polygon::Polygon( ) { };
```

```
Polygon::~~Polygon( ) { };
```

```
float Polygon::getArea( ) {  
    float p = numSides*lenSides;  
    float apo = numSides / 2. * tan(180. / numSides);  
    return apo*p / 2.;  
}
```

# The Triangle class

```
#include "Polygon.h"  
#include <string>
```

```
class Triangle : public Polygon {  
public:  
    Triangle(float);  
    ~Triangle( );  
  
    virtual float getArea( );  
  
};
```

```
#include "Triangle.h"  
#include <math.h>
```

```
Triangle::Triangle(float l) : Polygon(3, l) { }  
  
Triangle::~~Triangle( ) { };  
  
float Triangle::getArea( ) {  
    return 0.433 * lenSides * lenSides;  
}
```

Defines its own getArea method that overrides Polygon's getArea method.



# The Square class

```
#include "Polygon.h"  
#include <string>
```

```
class Square : public Polygon {  
public:  
    Square(float);  
    ~Square( );  
  
    virtual float getArea( );  
  
};  
#endif
```

```
#include "Square.h"  
#include <math.h>
```

```
Square::Square(float l) : Polygon(4, l) { }  
  
Square::~~Square( ) { };  
  
float Square::getArea( ) {  
    return lenSides * lenSides;  
}
```

Defines its own getArea method that overrides Polygon's getArea method.

# The Pentagon class

```
#include "Polygon.h"  
#include <string>
```

```
class Pentagon : public  
Polygon {  
public:  
    Pentagon(float);  
    ~Pentagon( );  
};
```

```
#include "Pentagon.h"  
#include <math.h>
```

```
Pentagon::Pentagon(float l) : Polygon(5, l) { }  
Pentagon::~~Pentagon( ) { }
```

Uses the `getArea` method in the base `Polygon` class

# The array p of object pointers

// elided includes of .h files and iostream

```
int main (int argc, char *argv[]) {
```

```
    Polygon* p[4];
```

```
    float area = 0;
```

```
    p[0] = new Square(4.0); p[1] = new Triangle(4.0);
```

```
    p[2] = new Polygon(8, 4.0); p[3] = new Pentagon(4.0);
```

```
    for (int i=0; i <4; i++) {
```

```
        area += p[i]->getArea( );
```

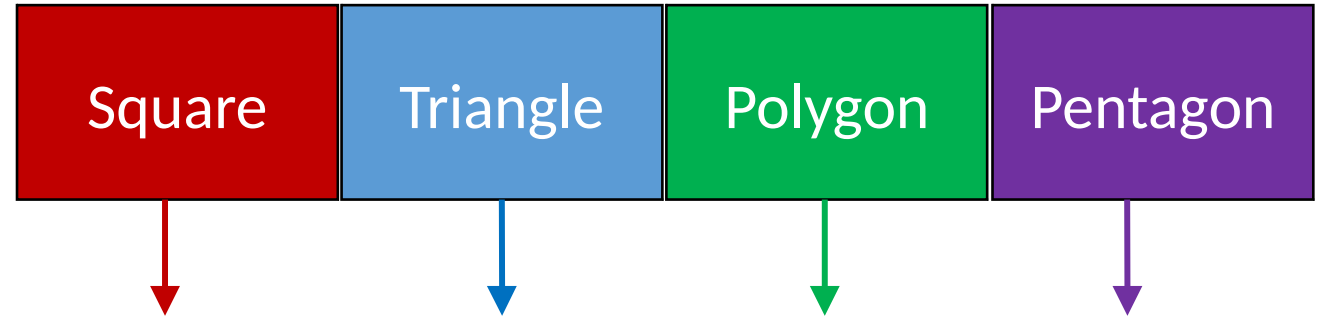
```
        std::cout << "area of poly " << i << " = " << p[i]->getArea( ) << std::endl;
```

```
    }
```

```
    std::cout << "total area = " << area << std::endl;
```

```
}
```

The Polygon \*p[4] vector



# The array p of object pointers

// elided includes of .h files and iostream

```
int main (int argc, char *argv[]) {
```

```
Polygon* p[4];
```

```
float area = 0;
```

```
p[0] = new Square(4.0); p[1] = new Triangle(4.0);
```

```
p[2] = new Polygon(8, 4.0); p[3] = new Pentagon(4.0);
```

```
for (int i=0; i < 4; i++) {
```

```
    area += p[i]->getArea( );
```

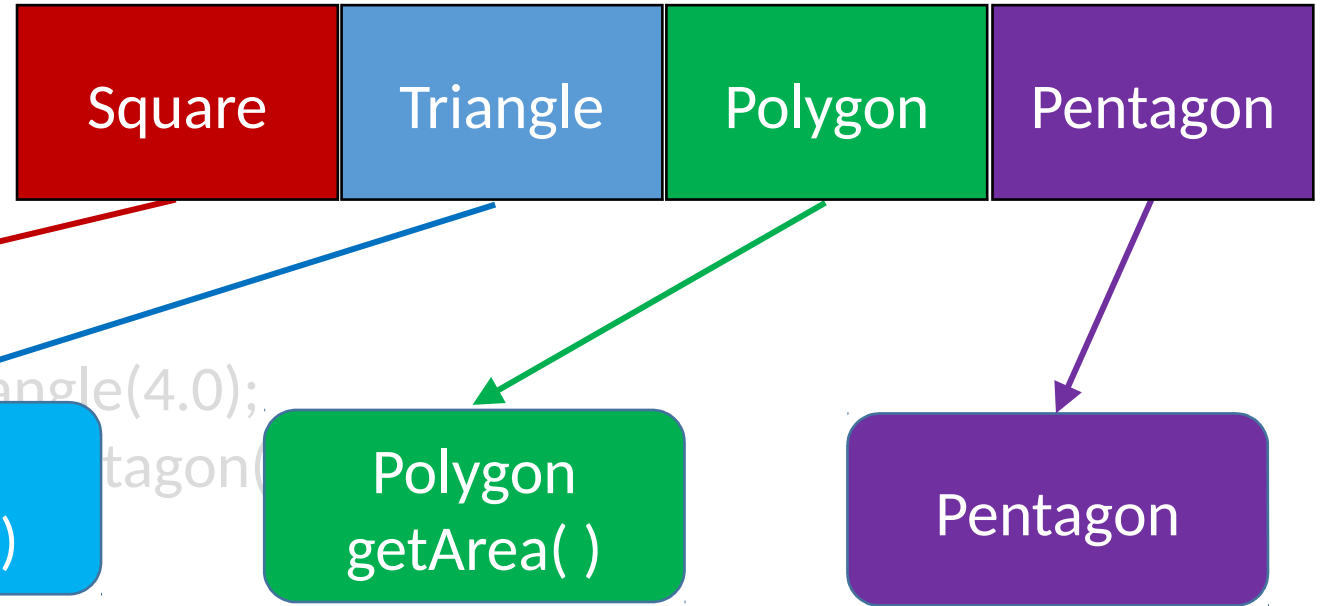
```
    std::cout << "area of poly " << i << " = " << p[i]->getArea( ) << std::endl;
```

```
}
```

```
std::cout << "total area = " << area << std::endl;
```

```
}
```

9/21/18



# The array p of object pointers

// elided includes of .h files and iostream

```
int main (int argc, char *argv[]) {
```

```
Polygon* p[4];
```

```
float area = 0;
```

```
p[0] =
```

Square object  
getArea( )

```
p[2] =
```

Triangle  
getArea( )

Polygon  
getArea( )

Pentagon

```
for (int i=0; i <4; i++) {
```

```
    area += p[i]->getArea( );
```

```
    std::cout << "area of poly " << i << " = " << p[i]->getArea( ) << std::endl;
```

```
}
```

***As we access each object through a pointer, we invoke the right  
getArea( ) for that object***

```
std::cout << "total area = " << area << std::endl;
```

```
}
```

9/21/20

# How does C++ know which method to call?

- For the Polygon object, calls Polygon's getArea( )
- For the Triangle object, calls Triangle's getArea( )
- For the square object, calls Square's getArea( )
- For the Pentagon object, calls **Polygon's** getArea( )
- This happens because of the properties of *virtual functions*
  - A virtual function is one that supports polymorphism
  - By default, functions in C++ are non-virtual. Functions must be declared as virtual in the .h file to be virtual
  - Virtual functions in C++ are called through a *virtual function table* (VFT). Each class has a VFT.

# Let's see how the VFT's are constructed for our example

```
#include <string>
```

```
class Polygon {  
public:  
    virtual float getArea( );  
};
```

Destructors are in the VFT, but I've left them out to keep things smaller

Polygon's VFT

0

getArea( )'s entry in the VFT

Code for Polygon's getArea( )

- Note that a VFT is constructed for each class
- Therefore, we'll look at the classes 1 by 1 to see how their VFT is constructed.
- Only virtual functions are in a VFT, so we can ignore variables and non-virtual functions
- Polygon has 1 virtual function, so the VFT has one entry (position 0).

# The VFT for Triangle

```
#include "Polygon.h"  
#include <string>
```

```
class Triangle : public Polygon {  
public:  
    virtual float getArea( );
```

- Because Triangle is derived from Polygon, the compiler starts out with Polygon's VFT,
- It then replaces entries in Polygon's VFT for those functions in Triangle that *override* a Polygon function
- Overriding occurs when the signatures “match”, where match will be precisely defined later

Polygon's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Polygon's  
getArea( )

Triangle's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Triangle's  
getArea( )



# The VFT for Square

```
#include "Polygon.h"
```

```
#include <string>
```

```
class Square : public Polygon {  
public:
```

```
    virtual float getArea( );
```

- Because Square is derived from Polygon, the compiler starts out with Polygon's VFT,
- It then replaces entries in Polygon's VFT for those functions in Square that override a Polygon function
- Overriding occurs when the signature "match", where match will be precisely defined later

Polygon's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Polygon's  
getArea( )

Square's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Square's  
getArea( )

# The VFT for Square

```
#include "Polygon.h"  
#include <string>
```

```
class Pentagon : public Polygon {  
public:
```

- Because Pentagon is derived from Polygon, the compiler starts out with Polygon's VFT,
- It then replaces entries in Polygon's VFT for those functions in Square that override a Polygon function – but there are none! (again, ignoring the destructor)

Polygon's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Polygon's  
getArea( )

Pentagon's VFT

0  
getArea( )'s entry in  
the VFT

Code for  
Polygon's  
getArea( )

# What happens in a virtual function call through a pointer, at runtime

- If the pointer/reference has type T, the class T is examined for a function with the same name and matching signature as what is being called.
- If no match, error
- If a match then
  1. The VFT entry for the function, ***in the class of the object pointed to***, is loaded into the a register
  2. The function is called through the address in the register

# More concretely . . .

- If the pointer/reference has type Polygon, the class Polygon is examined for a function (getArea) with the same name and a matching signature as what is being called.
- If no matching getArea found, error
- If a match then
  1. The VFT entry for getArea, in the class of the Square object pointed to, is loaded into the a register
  2. The function is called through the address in the register

# What function is called

Let the function call be  $p \rightarrow f(args)$

Let  $p$  be a pointer to a  $B$  object

Let the type of the object  $p$  points to be a  $D$  object

Let  $f(args')$  be the function that matches  $f(args)$  and is visible in  $B$

If ( $f(args')$ ) is not private and is not static) {

  If ( $f(args')$  is *virtual* in class  $B$ ) {

    Find the position  $k$  of  $f(args')$  in  $B$ 's *virtual function table (VFT)*

    Call  $f(args')$  through position  $k$  of  $D$ 's VFT

  } else {

    Call  $B$ 's  $f(args')$

← Note new code

  }

} else {

  Call  $B$ 's  $f(args')$

← Note args' instead of args

}

# In C++, not all calls to virtual functions exhibit polymorphism

- Two conditions must be met for polymorphism to happen
  1. The object the function is called on must be accessed via a reference (which we haven't learned about yet) or a pointer.
  2. The function must be declared virtual.
- 2 is because non-virtual functions are called directly, like regular C functions.
  - This gives the programmer control over how they want functions called.
- 1 we discuss next

# Polymorphic call of a virtual function: VFTs

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );
```

```
    virtual float getArea( );
```

```
    virtual float perimeter( );
```

```
protected:  
    int numSides;  
    float lenSides;  
};
```

Polygon VFT

getArea( )

Polygon's  
getArea( )  
code

Remove this!

getArea( )

perimeter( )

```
class Square : public Polygon {  
public:
```

```
    Square(float);  
    virtual ~Square( );
```

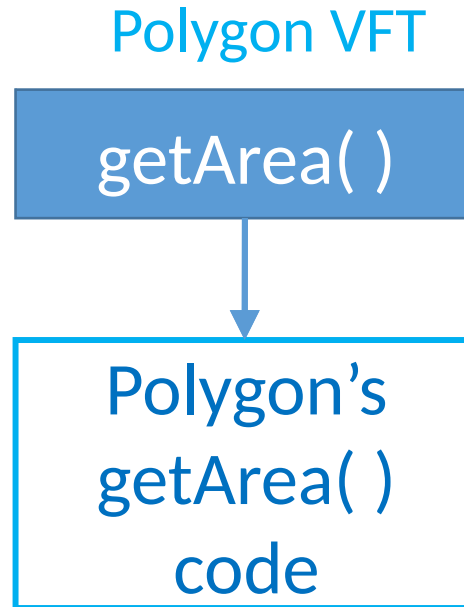
```
    float getArea( );  
    virtual float perimeter( );
```

Square's getArea( ) code

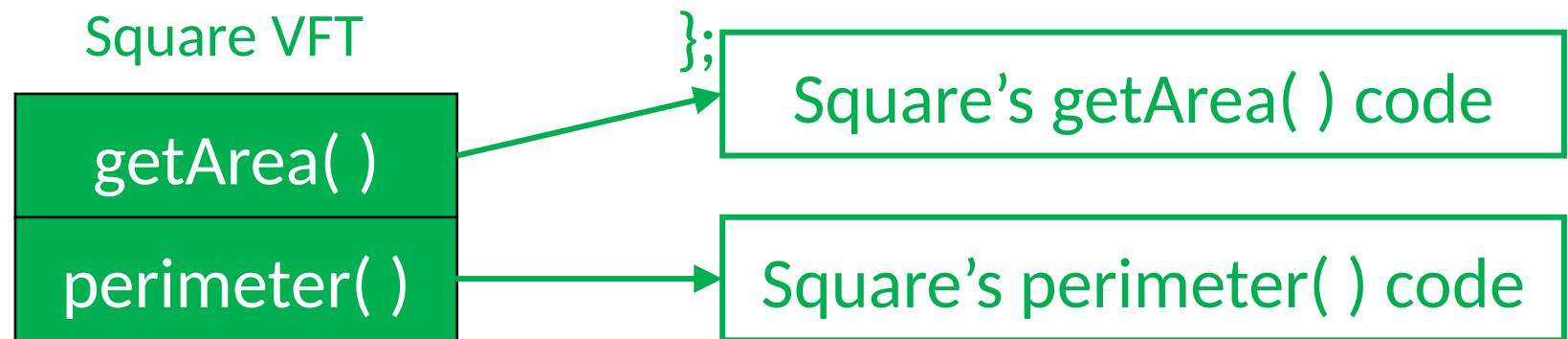
Square's perimeter( ) code

# Polymorphic call of a virtual function: VFTs

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );  
  
    virtual float getArea( );  
protected:  
    int numSides;  
    float lenSides;  
};
```



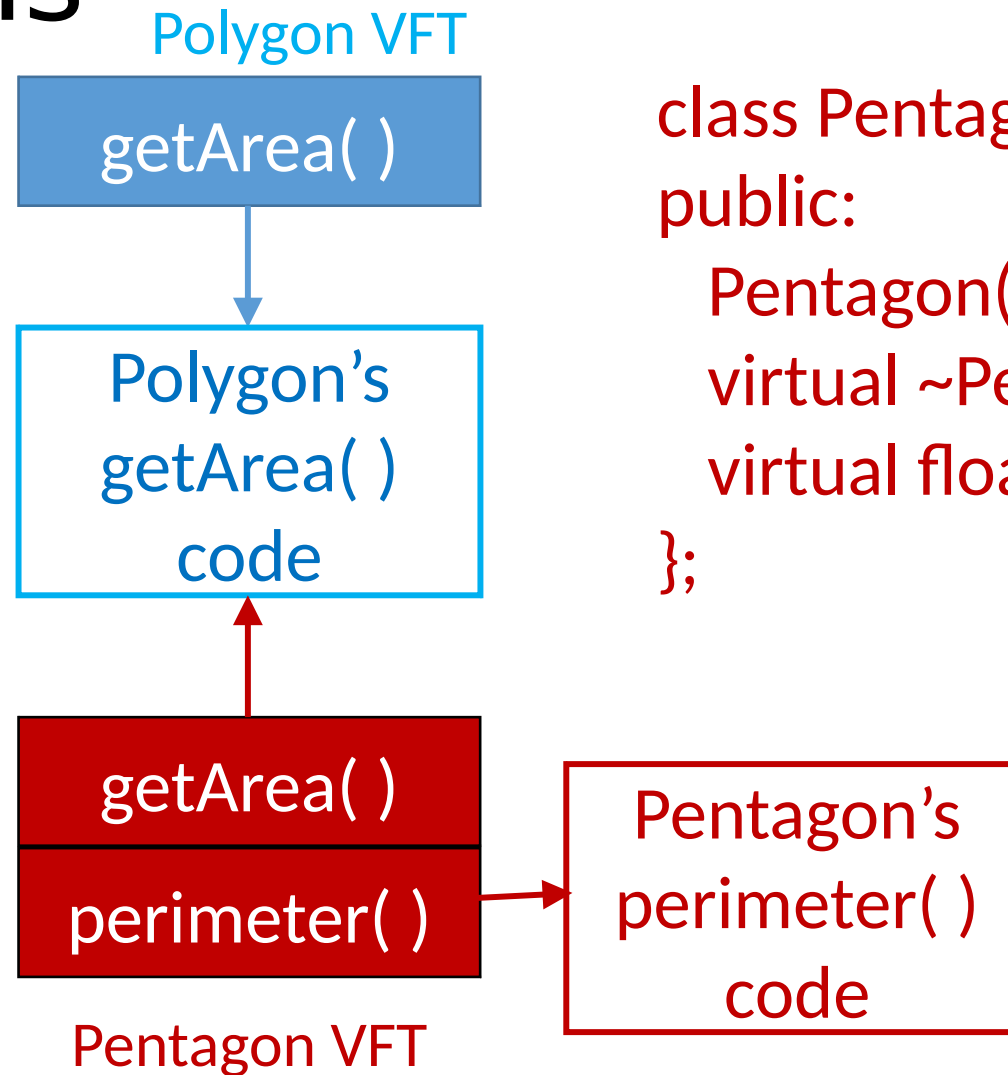
```
class Square : public Polygon {  
public:  
    Square(float);  
    virtual ~Square( );  
  
    float getArea( );  
    virtual float perimeter( );  
};
```





# Polymorphic call of a virtual function: VFTs

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );  
  
    virtual float getArea( );  
  
protected:  
    int numSides;  
    float lenSides;  
};
```



```
class Pentagon : public Polygon {  
public:  
    Pentagon(float);  
    virtual ~Pentagon( );  
    virtual float perimeter( );  
};
```

# Making the call

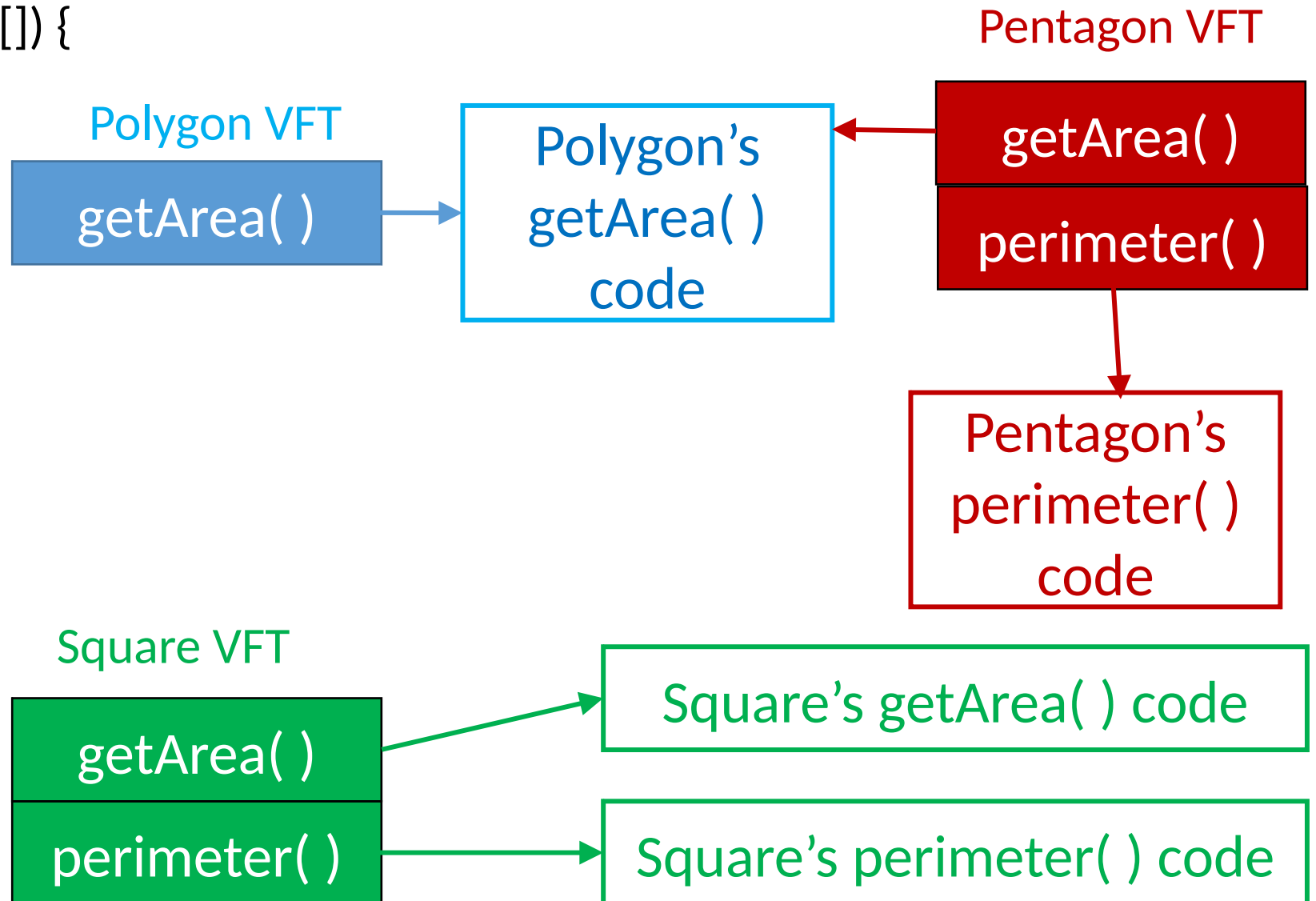
```
int main (int argc, char *argv[]) {
```

```
    Polygon* p[3];  
    float area = 0;
```

```
    p[0] = new Polygon(8, 4.0);  
    p[1] = new Square(4.0);  
    p[2] = new Pentagon(4.0);
```

```
    p[0]->getArea( );  
    p[1]->getArea( );  
    p[2]->getArea( );
```

```
}
```



# Try to call perimeter( ) via a Polygon pointer - **ERROR**

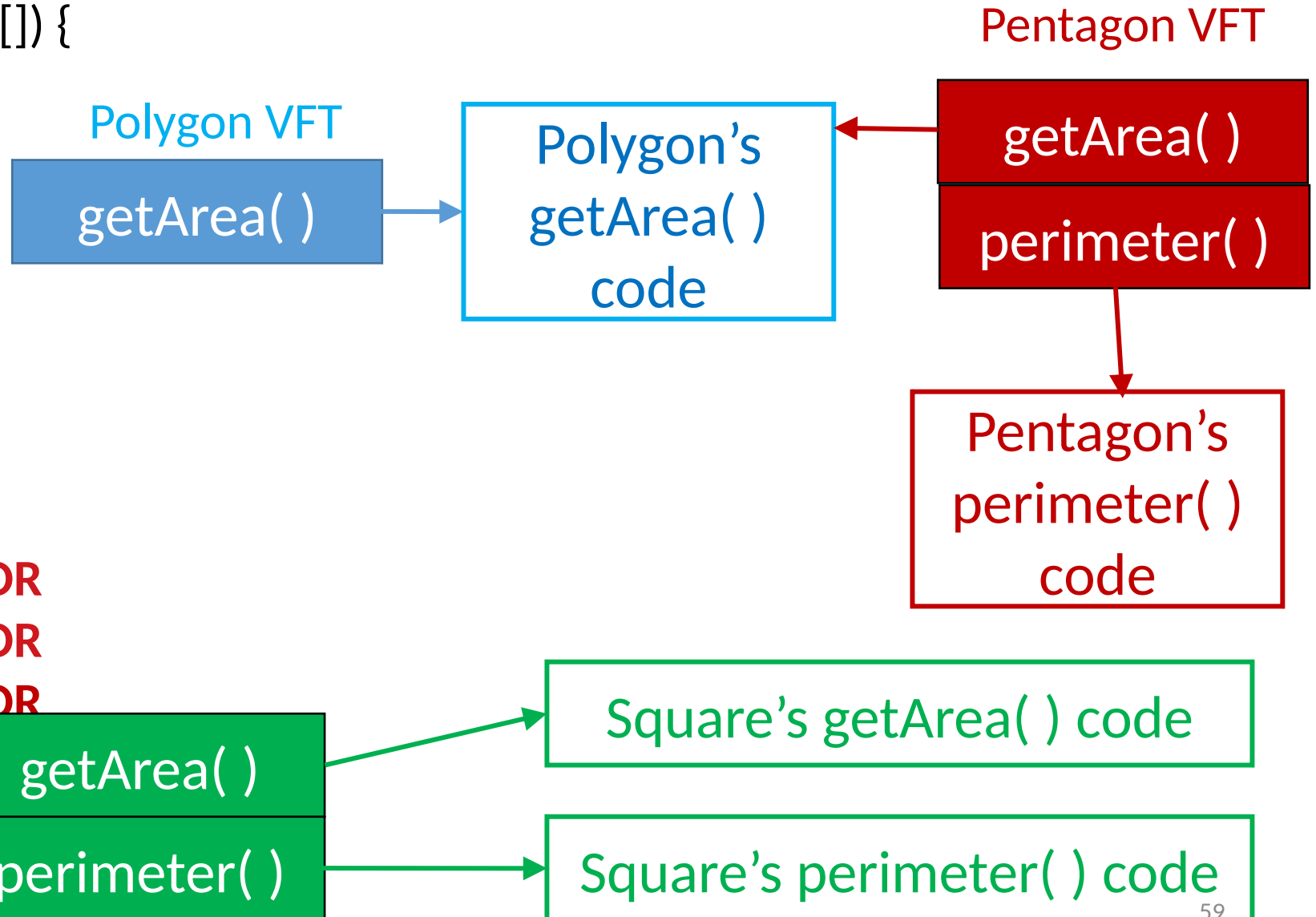
```
int main (int argc, char *argv[]) {
```

```
    Polygon* p[3];  
    float area = 0;
```

```
    p[0] = new Polygon(8, 4.0);  
    p[1] = new Square(4.0);  
    p[2] = new Pentagon(4.0);
```

```
    p[0]->perimeter( ); // ERROR  
    p[1]->perimeter( ); // ERROR  
    p[2]->perimeter( ); // ERROR
```

```
}
```



# Why does polymorphism through pointers work?

- Even when the pointer is a base type, it can point to a derived type object
- If the base type defines the function (otherwise there is an error) the function must be defined in the inherited, derived type
  - *Note that if the function is private in the base class the base class function is directly called, because there may be no visible function in the derived types*
- The compiler knows the type of the pointer, the method signature, the methods in each class (because of the .h file), and the VFT position *pos* for each method in the class (because it sets up the VFT to be loaded at runtime)
  - At runtime, the function is called through the pointer in position *pos* of the VFT of the object actually pointed to
  - This calls the pointed to object's method through a base class pointer!

# Calls to virtual functions through an object variable do not exhibit polymorphism

- This is a result of what happens when we assign a variable whose value is an object to a less derived object whose value is an object.
- We'll first consider how objects look because of inheritance
- We'll then look at what happens when we assign variables whose value is an object
- Finally, we'll see why polymorphism doesn't work with calls through variables.

# Inheritance and objects

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );
```

```
    virtual float getArea( );
```

```
protected:  
    int numSides;  
    float lenSides;
```

```
};
```

Class type information  
Int numSides;  
Float lenSidesA

Class type information  
Int numSides;  
Float lenSidesA

Class type information  
string polyName;

```
class Square : public Polygon {  
public:  
    Square(float);  
    virtual ~Square( );
```

```
    float getArea( );  
    virtual float perimeter( );  
    virtual void printName( );
```

```
private:  
    std::string polyName;
```

```
};
```

```

int main (int argc, char *argv[]) {

    Polygon p(8, 4.0);
    Square s(4.0);
    // Square s2;

    std::cout << p.getArea( ) << std::endl;
    std::cout << s.getArea( ) << std::endl;

    p = s;
    std::cout << p.getArea( ) << std::endl;
    std::cout << s.getArea( ) << std::endl;
}

```

p

Class type information  
 Int numSides = 8  
 Float lenSides = 4.0

s

Class type information  
 Int numSides = 4  
 Float lenSides = 4.0

Class type information  
 string polyName = "Square"

```
int main (int argc, char *argv[]) {  
  
    Polygon p(8, 4.0);  
    Square s(4.0);  
    // Square s2;  
  
    std::cout << p.getArea( ) << std::endl;  
    std::cout << s.getArea( ) << std::endl;  
  
    p = s;  
    std::cout << p.getArea( ) << std::endl;  
    std::cout << s.getArea( ) << std::endl;  
}
```

**Poly 35.7025**  
**Square 16**



```
int main (int argc, char *argv[]) {
```

```
    Polygon p(8, 4.0);
```

```
    Square s(4.0);
```

```
    // Square s2;
```

```
    std::cout << p.getArea( ) << std::endl;
```

```
    std::cout << s.getArea( ) << std::endl;
```

```
s = p;
```

```
p = s;
```

```
    std::cout << p.getArea( ) << std::endl;
```

```
    std::cout << s.getArea( ) << std::endl;
```

```
}
```

**State before  
the assignment**

p

Class type information

Int numSides = 8

Float lenSides = 4.0

s

Class type information

Int numSides = 4

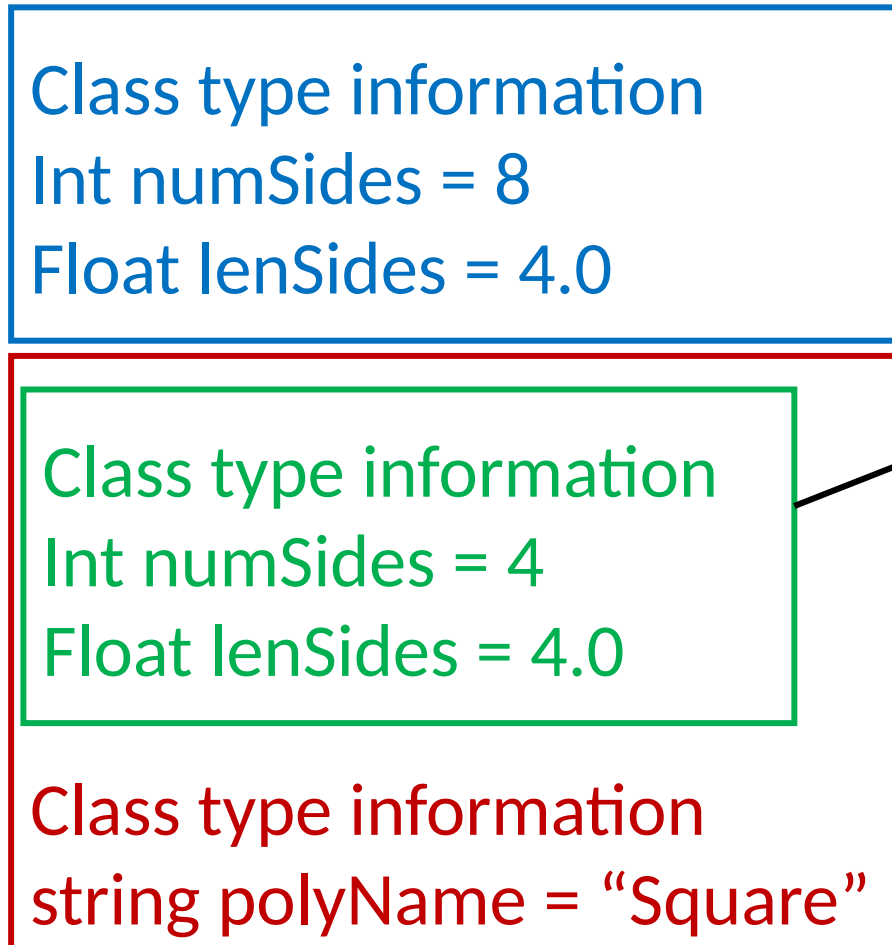
Float lenSides = 4.0

Class type information

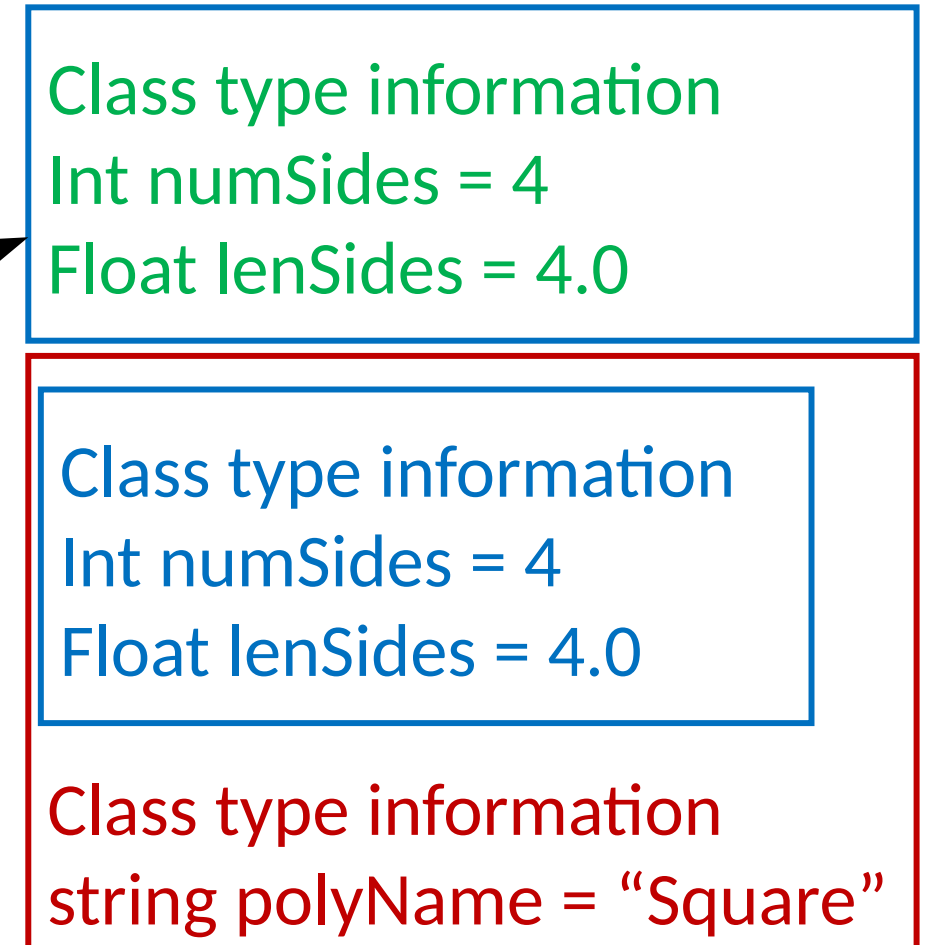
string polyName = "Square"

~~s = p;~~ p = s;


## Runtime Stack



## Runtime Stack



# What if we assigned p to s (Polygon to a Square)?

- Square *ISA* Polygon
- Therefore, the *ISA* relation  $s \text{ ISA } p$  holds
- If  $x \text{ ISA } y$ , the assignment  $y = x$  is legal. This is true for pointers as well.
- What is  $x \neg \text{ISA } y$ ?
- Then the assignment  $y = x$  is illegal. And  $s =$   *new* would be illegal.
- From the previous example we can see why this is the case.
  - Square *ISA* Polygon
  - Polygon  $\neg \text{ISA}$  Square.
  - Polygon does not have sufficient information to make a full Square object

```
int main (int argc, char *argv[]) {  
  
    Polygon p(8, 4.0);  
    Square s(4.0);  
    // Square s2;  
  
    std::cout << p.getArea( ) << std::endl;  
    std::cout << s.getArea( ) << std::endl;  
  
    p = s;  
    std::cout << p.getArea( ) << std::endl;  
    std::cout << s.getArea( ) << std::endl;  
}
```

```
Poly 35.7025  
Square 16  
Poly 25.9164  
Square 16
```

*My polygon formula  
appears to be wrong*

# In C++, only virtual functions support polymorphism

- Functions are non-virtual by default
- If a function is declared virtual in a class B, and is overridden in a directly or indirectly in derived class D, it is *still* virtual in D
- The virtual property “sticks” through inheritance
  - For a virtual function to override a function in class B, the signatures must be identical AND either
    - the return types must both be primitives, OR
    - If the return type is pointer or reference (not yet discussed) it can be a reference or pointer to a derived type of B. Often it is of type D
  - This is a little simplistic because of hiding, to be discussed a little later

# Overriding virtual and non-virtual functions

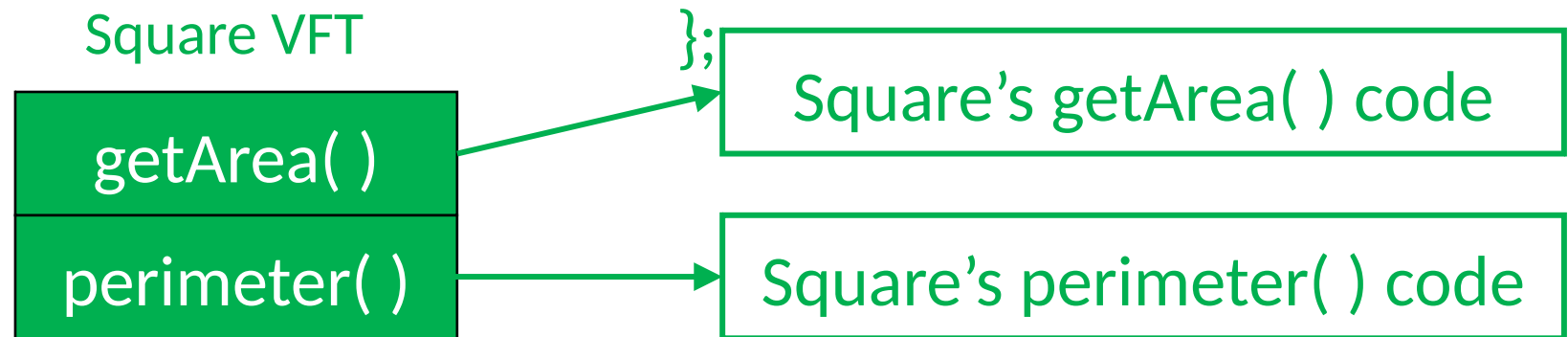
- Virtual: the derived class is not required to override, but can
- Not virtual: the derived class ***should not override it (but can)***
  - Having a non-virtual function in the base class is a way of saying “don’t override me”
  - The compiler will allow it, but you are likely ignoring the wishes of the base class developer and the interface
  - Because it is non-virtual, calls will not be made like they are for virtual functions.

# What if getArea not virtual in Polygon?

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );  
  
—virtual float getArea( );  
  
protected:  
    int numSides;  
    float lenSides;  
};
```

Polygon VFT only  
has the  
destructor entry,  
which we are  
ignoring.

```
class Square : public Polygon {  
public:  
    Square(float);  
    virtual ~Square( );  
  
    virtual float getArea( );  
    virtual float perimeter( );  
};
```



# Making the calls with Polygon::getArea( ) non virtual

```
int main (int argc, char *argv[]) {
```

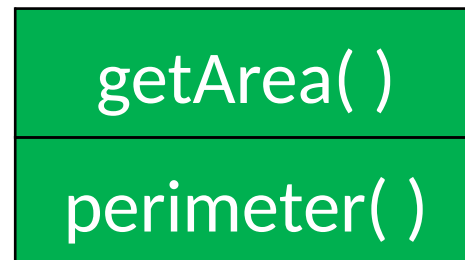
```
    Polygon* p[2];  
    float area = 0;
```

```
    p[0] = new Polygon(8, 4.0);  
    p[1] = new Square(4.0);
```

```
    std::cout << p[0]->getArea( ) << std::endl;  
    std::cout << p[1]->getArea( ) << std::endl;
```

```
}
```

Square's VFT



Polygon VFT only has  
the destructor entry,  
which we are ignoring.

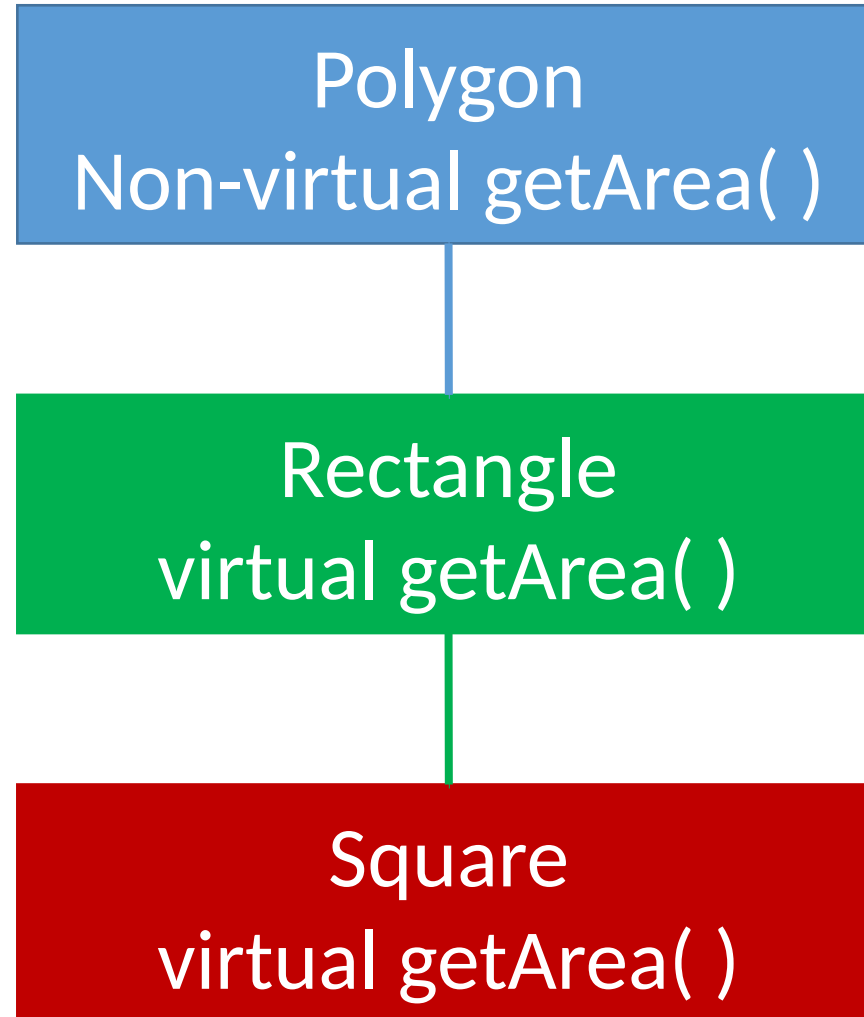
Square's getArea( ) code

Square's perimeter( ) code



# Weird behavior can arise if we override non-virtual functions

See Poly2NonVirtualBase.zip



# The .h files for this

```
class Polygon {  
public:  
    Polygon( );  
    Polygon(int,float);  
    virtual ~Polygon( );  
  
    float getArea( );  
  
protected:  
    int numSides;  
    float lenSides;  
  
};
```

```
class Rectangle :  
    public Polygon {  
public:  
    Rectangle(float, float);  
    virtual ~Rectangle( );  
  
    virtual float getArea( );  
  
private:  
    float lenSide2;  
  
};
```

```
class Square : public  
    Rectangle {  
Public:  
    Square(float);  
    virtual ~Square( );  
  
    float getArea( );  
  
};
```

# The main function and output

```
int main (int argc, char *argv[]) {
```

```
    Polygon *p = new Square(4.0);  
    Rectangle *r = new Square(4.0);
```

```
    std::cout << p->getArea( ) << std::endl;  
    std::cout << r->getArea( ) << std::endl;
```

```
}
```

Poly 25.9164

Square 16

Calls Polygon getArea

Calls Square getArea

# Why are functions non-virtual by default?

- Slighter better performance for non-virtual (no call through a pointer), but . . .
  - Makes programming more complicated
  - The performance overhead is often small to non-existent because of compiler optimizations
- Goals of C++ were to allow best performance and compatibility with C
  - All functions in C are non-virtual. C code can call non-virtual functions in a C++ program

# Some hints

- Functions in C++ should always be virtual unless you have a good reason for them not to be (C compatibility, should never be overridden, performance is paramount, etc.)
- Constructors are not virtual, and it doesn't make sense that they would be
- Destructors (~ClassName) should always be virtual) (more about this when we discuss destructors)
  - Not doing this can cause memory leaks

# Hiding



- C++ says that if a derived class overrides *or overloads*, a function *foo*, the functions named *foo* in the base class are *hidden*.
- Functions with the same name in the base class will not be visible.
- From Stroustrup: "In C++, there is no overloading across scopes - derived class scopes are not an exception to this general rule."
- Let's look at an example to make this clear.

# class Derived extends class Base

```
Base::Base( ) { }
Base::~~Base( ) { }
void Base::f(double x) {
    std::cout << "Base: " << x << std::endl;
}

Derived::Derived( ) { }
Derived::~~Derived( ) { }
void Derived::f(char x) {
    std::cout << "Derived: " << x << std::endl;
}
```

```
int main() {
    Derived* d = new Derived();
    Base* b = d;
    b->f(65.3); // okay: passes 65.3 to f(double x)
    d->f(65.3); // converts 65.3 to a char ('A' if ASCII)
                // and passes it to f(char c); It does
                // NOT call f(double x)!!
}
```

Base: 65.3

Derived: A

# If you are overriding base class functions

- then override all forms of the function if you need all forms
- You have extended the interface, and should provide valid implementations for the new interface.
- In the previous example, Derived should define both
  1. `virtual void f(char c);` // already defined
  2. `virtual void f(double x);` // not defined in Derived class, but should be



# Or, you can still call the base class

- Invoke `Base::f(65.3)` // gives 65.3, `p->Base::f(65.3)`
- Use the *using* keyword (some compilers may not support this)

```
class Derived : public Base {  
public:
```

```
    using Base::f; // this un-hides Base::f(double x). Now f(65.3) on Derived object  
                  // will call the Base class f(double x);
```

```
    void f(char c);  
};
```

# To summarize what happens in a function call

Given:

**T1 \*p = new T2(args); // T2 extends, directly or indirectly **T1****  
**p->foo(args);**

- Class T1 is examined. If there is **not a function in-scope that** matches, possibly with conversions, *foo(args)* there is an error.
- If the matching function is not virtual, then T1 class *foo(args)* is called
- If the matching function is virtual, the function in the T2 VFT in the slot for *foo(args)* is called

# Examples of how functions are called (assume all functions are virtual)

- The Base class is examined (the type of pointer b is) , *f(float)* is matched and called through the VFT of the object pointed to by b
- The Derived class (the type of pointer d) is examined, *f(char)* is matched, and called through the VFT of the object pointed to by d
- The b class is examined (the type of pointer b), *f(float)* is matched, and called through the VFT of the (Derived) object pointed to by b

```
#include "Base.h"  
#include "Derived.h"
```

```
int main() {  
    Derived* d = new Derived();  
    Base* b = d;  
    b->f(65.3);  
    d->f(65.3);  
    b = d;  
    b->f(65.3);  
}
```