# COMP6771

# Advanced C++ Programming

## Week 9

## Runtime Polymorphism

Dynamic polymorphism or Late binding

# Key concepts

- **Inheritance**
  - To be able to create new classes by inheriting from existing classes.
  - To understand how inheritance promotes software reusability.
  - To understand the notions of base classes and derived classes.
- **Polymorphism**
  - **Static: determine which method to call at compile time**
  - **Dynamic polymorphism: determine which method to call at run time**
    - **function call is resolved at run time**
    - **Closely related to polymorphism**
    - Supported via virtual functions

# Tenets of C++

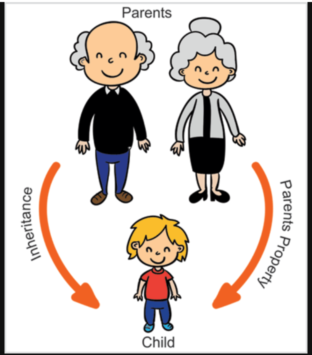
- Don't pay for what you don't use
  - C++ Supports OOP
    - No runtime performance penalty
  - C++ supports generic programming with the STL and templates
    - No runtime performance penalty
  - Polymorphism is extremely powerful, and we need it in C++
    - Do we need polymorphism at all when using inheritance?
      - Answer: sometimes
      - But how do we do so, considering that we don't want to make anyone who doesn't use it pay a performance penalty

# ~~Thinking about programming~~

- Represent concepts with classes
- Represent relations with inheritance or composition
  - **Inheritance:** A is also a B, and can do everything B does
    - "is a" relationship
    - A dog **is an** animal
  - **Composition (data member):** A contains a B, but isn't a B itself
    - "has a" relationship
    - A person **has a** name
  - Choose the right one!

# Inheritance

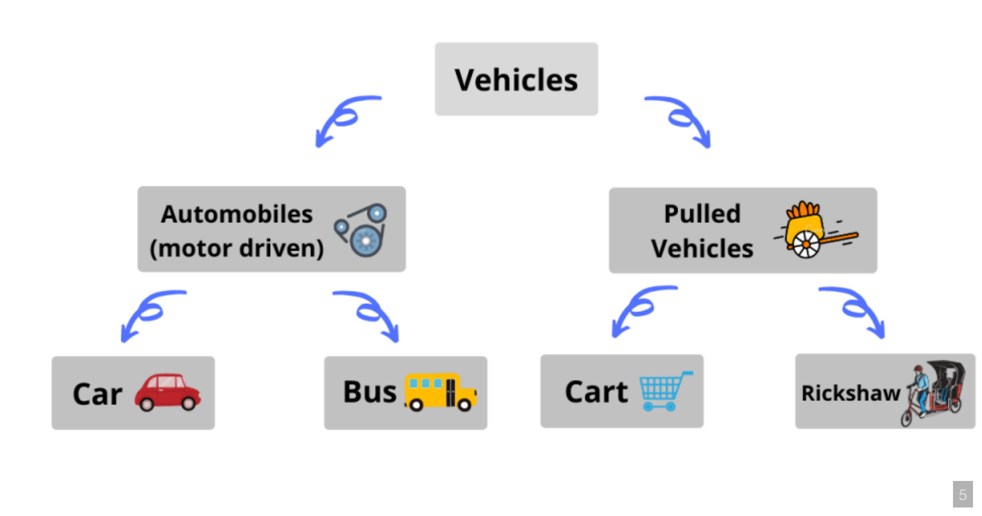
- Represent concepts with classes
- Represent relations with inheritance or composition
  - **Inheritance:** A is also a B, and can do everything B does
    - "is a" relationship
    - A dog **is an** animal
  - **Composition (data member):** A contains a B, but isn't a B itself
    - "has a" relationship
    - A person **has a** name
  - Choose the right one!




Inheritance is relation between two or more classes where child/derived class inherits properties from existing base/parent class.

Why:

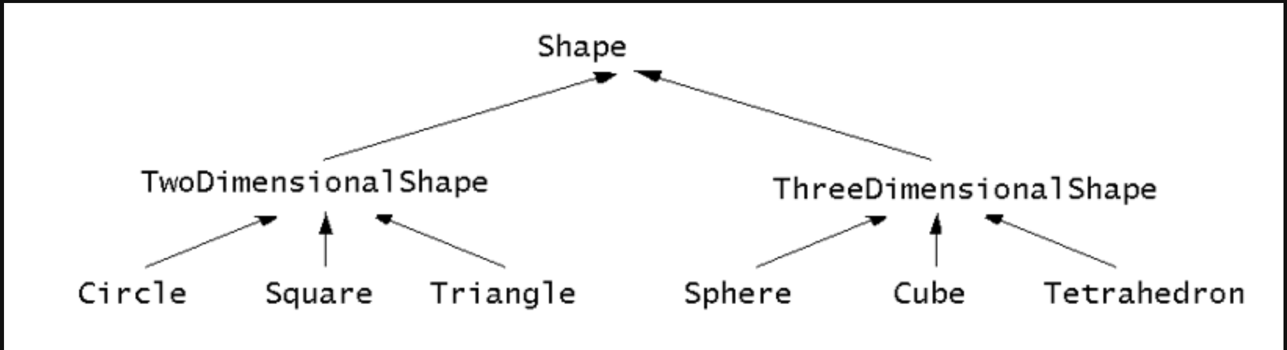
code reusability & data protection



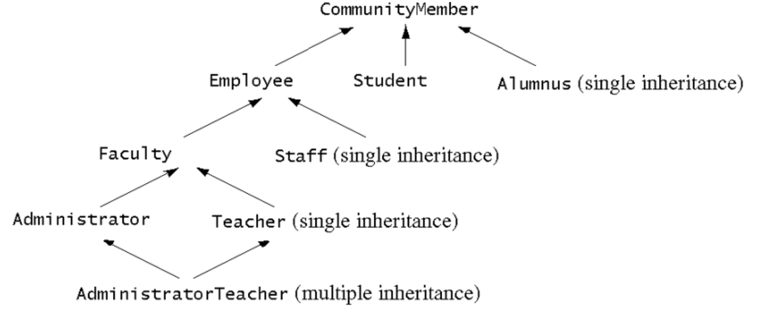
# Examples

•Often an object from a derived class (subclass) “is an” object of a base class (superclass)

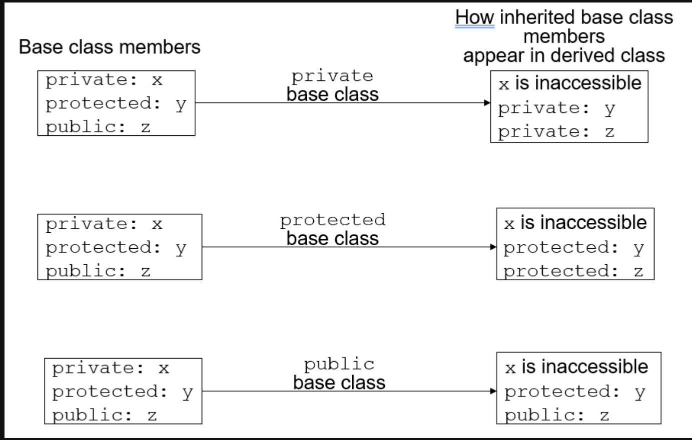
| Base class | Derived classes |
|---|---|
| Student | GraduateStudent<br>UndergraduateStudent |
| Shape | Circle<br>Triangle<br>Rectangle |
| Loan | CarLoan<br>HomeImprovementLoan<br>MortgageLoan |
| Employee | FacultyMember<br>StaffMember |
| Account | CheckingAccount<br>SavingsAccount |



# Inheritance in C++




- Single vs Multiple
- To inherit off classes in C++, we use "class DerivedClass: public BaseClass"
- Visibility can be one of:
  - **public**
    - object of derived class can be treated as object of base class (not vice-versa)
    - (generally use this unless you have good reason not to)
    - If you don't want public, you should (usually) use composition
  - **protected**
    - allow derived to know details of parent
  - **private**
    - not inaccessible
- Visibility is the maximum visibility allowed
  - If you specify ": private BaseClass", then the maximum visibility is private
    - Any BaseClass members that were public or protected are now private



# Inheritance vs Access

```
class Grade
private members:
  char letter;
  float score;
  void calcGrade();
public members:
  void setScore(float);
  float getScore();
  char getLetter();
```

```
class Test : public Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;
public members:
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this:

```
private members:
  int numQuestions:
  float pointsEach;
  int numMissed;
public members:
  Test(int, int);
  void setScore(float);
  float getScore();
  char getLetter();
```

```
class Grade
private members:
  char letter;
  float score;
  void calcGrade();
public members:
  void setScore(float);
  float getScore();
  char getLetter();
```

```
class Test : protected Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;
public members:
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `protected` class access, it looks like this:

```
private members:
  int numQuestions:
  float pointsEach;
  int numMissed;
public members:
  Test(int, int);
protected members:
  void setScore(float);
  float getScore();
  float getLetter();
```

```
class Grade
private members:
  char letter;
  float score;
  void calcGrade();
public members:
  void setScore(float);
  float getScore();
  char getLetter();
```

```
class Test : private Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;
public members:
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `private` class access, it looks like this:

```
private members:
  int numQuestions:
  float pointsEach;
  int numMissed;
  void setScore(float);
  float getScore();
  float getLetter();
public members:
  Test(int, int);
```

# Syntax and memory layout

This is very important, as it guides the design of everything we discuss this week

| BaseClass object |
| --- |
| int_member_<br>string_member_ |

| | SubClass object |
| --- | --- |
| BaseClass subobject | int_member_<br>string_member_ |
| SubClass subobject | vector_member_<br>ptr_member_ |

```cpp
class BaseClass {
 public:
  int get_int_member() { return int_member_; }
  std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int int_member_;
  std::string string_member_;
}
```

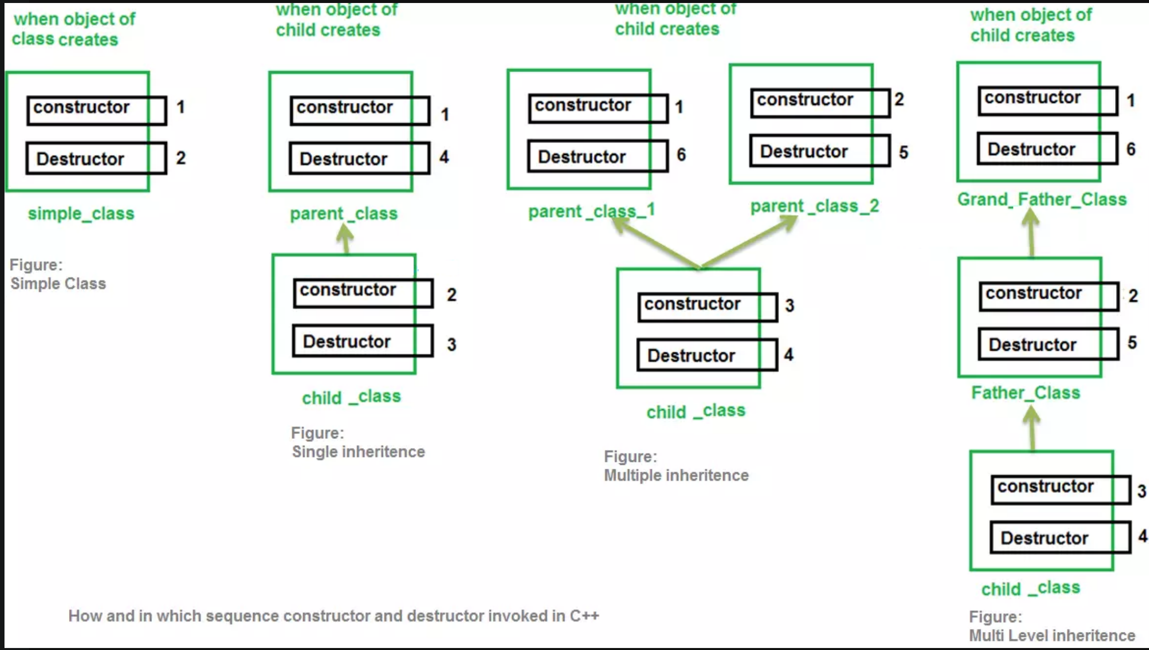
```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  std::vector<int> vector_member_;
  std::unique_ptr<int> ptr_member_;
}
```

# Constructors and Destructors

## Single

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

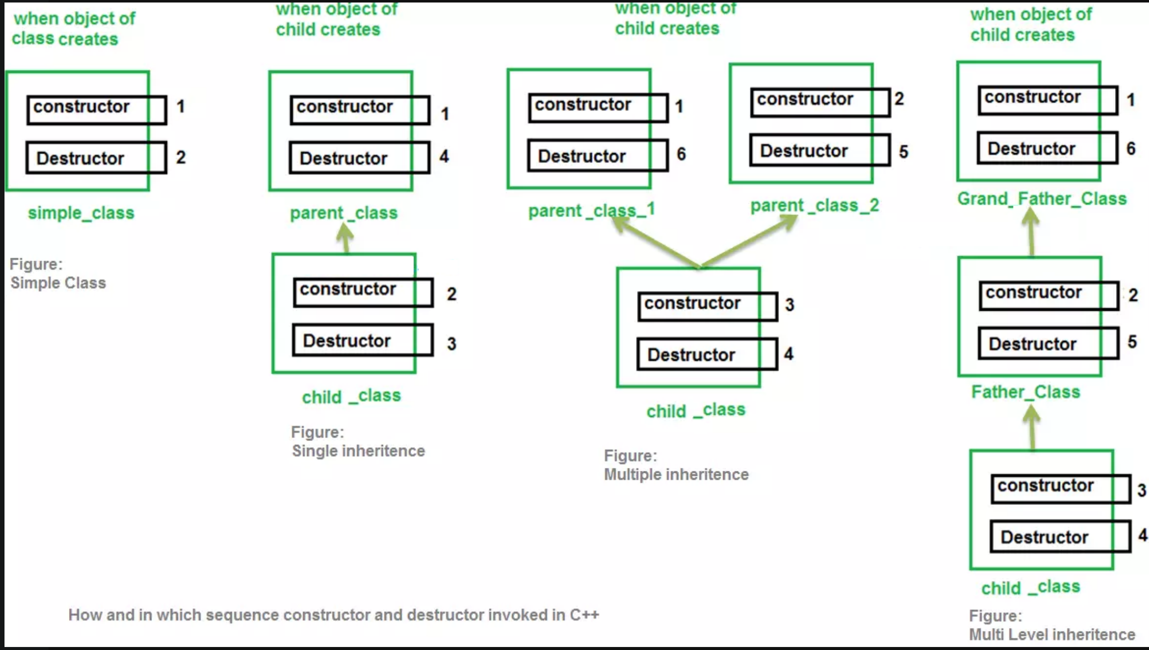
```cpp
#include <iostream>

class base {
public:
    base() { std::cout << "Constructing base\n"; }
    ~base() { std::cout << "Destructing base\n"; }
};

class derived: public base {
public:
    derived() { std::cout << "Constructing derived\n"; }
    ~derived() { std::cout << "Destructing derived\n"; }
};

int main()
{
    derived ob;

    // do nothing but construct and destruct ob

    return 0;
}
```



# Constructors and Destructors

## Multilevel

```cpp
#include <iostream>

class base {
public:
  base() { std::cout << "Constructing base\n"; }
  ~base() { std::cout << "Destructing base\n"; }

};

class derived1 : public base {
public:
  derived1() { std::cout << "Constructing derived1\n"; }
  ~derived1() { std::cout << "Destructing derived1\n"; }
};

class derived2: public derived1 {
public:
  derived2() { std::cout << "Constructing derived2\n"; }
  ~derived2() { std::cout << "Destructing derived2\n"; }
};

int main()
{
  derived2 ob;

  // construct and destruct ob

  return 0;
}
```



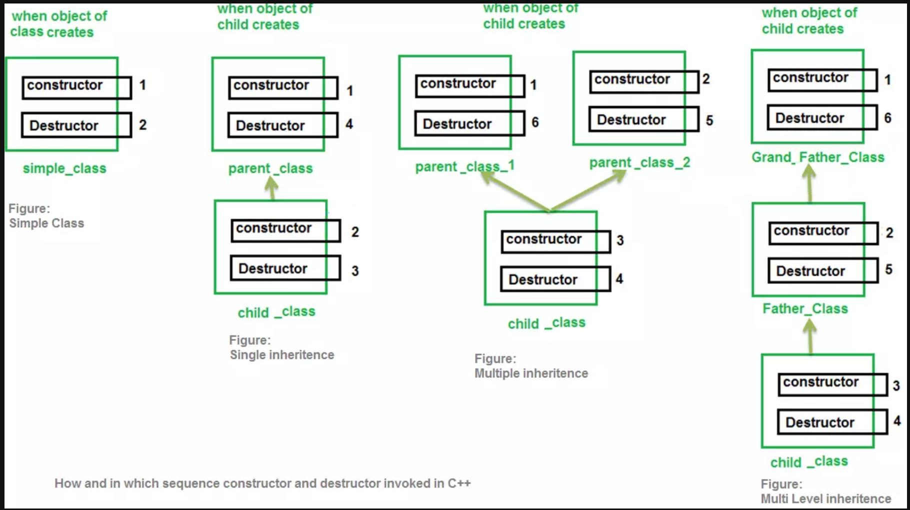
# Constructors and Destructors

## Multiple

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

class base1
public:
  base1() { std::cout << "Constructing base1\n"; }
  ~base1() { std::cout << "Destructing base1\n"; }
};

class base2 {
public:
  base2() { std::cout << "Constructing base2\n"; }
  ~base2() { std::cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  derived() { std::cout << "Constructing derived\n"; }
  ~derived() { std::cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

   // construct and destruct ob

  return 0;
```

Constructors are called in order of derivation, left to right, as specified in derived's inheritance list.

Destructors are called in reverse order, right to left.




Problem: what if base classes have member variables/functions with the same name?

Solutions:

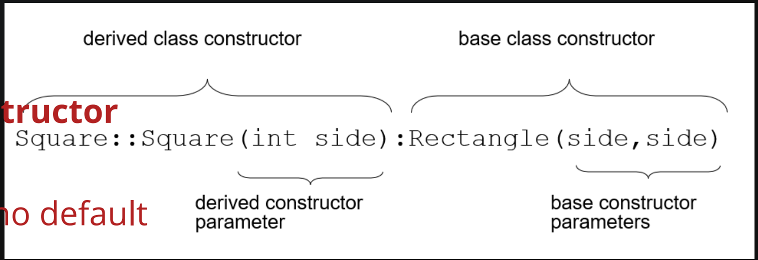
–Derived class redefines the multiply-defined function

–Derived class invokes member function in a particular base class using scope resolution operator ::

**Passing Arg to constructor**

can be inline too

Must be if base has no default



# Redefining Base Function

1. Redefining function: function in a derived class that has the same name and parameter list as a function in the base class.
2. Typically used to replace a function in base class with different actions in derived class.
3. Not the same as overloading – with overloading, parameter lists must be different.
4. Objects of base class use base class version of function; objects of derived class use derived class version of function.

```
//base class
class GradeActivity{
protected:
        char letter;
    double score;
    void determineGrade();
public:
        GradeActivity() //default constr.
         {letter=' '; score=0.0;}
    void setScore(double s){ // mutator
         score=s;
          determineGrade();}
   double getScore() const
        {return score;}
    char getLetterGrade() const
        {return letter;}
}
```

```
//derived class
#ifndef CURVEACTIVITY_H
#define CURVEACTIVITY_H

class CurveActivity : public GradeActivity{
protected:
        char rawScore;
    double percenrage;
    void determineGrade();
public:
        CurveActivity():GradeActivity() //default constr
        {rawScore=0.0; percentage=0.0;}
    void setScore(double s){ // mutator
         rawScore=s;
        GradeActivity::setScore(rawScore*percentage);}
    void setPercentage(double c) const
        {percentage=c;}
    //accessor function
    double getPercentage() const
        {return percentage;}
    double getRawScore() const
        {return rawScore;}
}
```

```
int main()
{
double numscore, per;
CurvedActivity exam;
std::cout<<"Enter raw score";
std::cin>>numscore;
std::cout<<"%age";
std::cin>>per;
exam.setPercentage(per);
exam.setScore(numscore);

std::cout<<exam.getRawScore();
std::cout<<exam.getScore();
std::cout<<exam.getLetterGrade();

}
```

# Problem: Redefining Base Function

BaseClass

void X();

Void Y();

DerivedClass : public BaseClass

Void Y(); //redefined

Object D invokes function X() in BaseClass. Function X() invokes function Y() in BaseClass, not function Y() in DerivedClass, because function calls are bound at compile time. This is static binding.

```
DerivedClass D;
D.X();
```

# Problem: Redefining Base Function

```cpp
#include <iostream>

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         std::cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
      int area () {
         std::cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
```

```cpp
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
   // store the address of Rectangle
   shape = &rec;
  // call rectangle area.
   shape->area();
  // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();
   return 0;
}
```

```
Parent class area :
Parent class area :
```

```cpp
// Main function for the program
int main() {

   Rectangle rec(10,7);
   Triangle  tri(10,5);

   rec.area();

   tri.area();
   return 0;
}
```

```
Rectangle class area :
Triangle class area :
```

## Example from Past

```
int main() {
    int num_desserts = 24 + 35;  // + operator used for addition
    cout << num_desserts << endl;
    string str1 = "We can combine strings ";
    string str2 = "that talk about delicious desserts";
    string str = str1 + str2; // + operator used for combining two strings
    cout << str << endl;
    return 0;
}
```

# Polymorphism and values

- Polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Polymorphism allows reuse of code by allowing objects of related types to be treated the same.

  How many bytes is a BaseClass instance?
- How many bytes is a DerivedClass instance?
- One of the guiding principles of C++ is "You don't pay for what you don't use"
  - Let's discuss the following code, but pay great consideration to the memory layout

```cpp
class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

demo901-poly.cpp

```cpp
void print_class_name(BaseClass base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

# The object slicing problem

- If you declare a BaseClass variable, how big is it?
- How can the compiler allocate space for it on the stack, when it doesn't know how big it could be?
- The solution: since we care about performance, a BaseClass can only store a BaseClass, not a SubClass
  - If we try to fill that value with a SubClass, then it just fills it with the BaseClass subobject, and drops the SubClass subobject

```cpp
class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

```cpp
void print_class_name(BaseClass base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

demo901-poly.cpp

# Polymorphism and References

- How big is a reference/pointer to a BaseClass
- How big is a reference/pointer to a SubClass
- Object slicing problem solved (but still another problem)
- One of the guiding principles of C++ is "You don't pay for what you don't use"
  - How does the compiler decide which version of GetClassName to call?
    - When does the compiler decide this? Compile or runtime?
  - How can it ensure that calling GetMember doesn't have similar overhead

```
class BaseClass {
 public:
  int get_member() { return member_; }
  std::string get_class_name() {
    return "BaseClass";
  };

 private:
  int member_;
}
```

```
class SubClass: public BaseClass {
 public:
  std::string get_class_name() {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

```
void print_class_name(BaseClass& base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

demo902-poly.cpp

# Virtual functions

- How does the compiler decide which version of GetClassName to call?
- How can it ensure that calling GetMember doesn't have similar overhead
  - Explicitly tell compiler that GetClassName is a function designed to be modified by subclasses
  - Use the keyword "virtual" in the base class:
    - function in base class that expects to be redefined in derived class
    - supports dynamic binding: functions bound at run time to function that they call.
    - At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.
    - It ensures that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
    - Without virtual member functions, C++ uses static (compile time) binding.
    - Use the keyword "override" in the subclass

```
class BaseClass {
 public:
  int get_member() { return member_; }
  virtual std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int member_;
}
```

```
class SubClass: public BaseClass {
 public:
  std::string GetClassName() override {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

demo903-virt.cpp

```
void print_stuff(const BaseClass& base) {
  std::cout << base.get_class_name()
            << ' ' << base.get_member()
            << '\n';
}

int main() {
  BaseClass base_class;
  SubClass subclass;
  print_class_name(base_class);
  print_class_name(subclass);
}
```

# Override

- While override isn't required by the compiler, you should **always** use it
- Override fails to compile if the function doesn't exist in the base class. This helps with:
  - Typos
  - Refactoring
  - Const / non-const methods
  - Slightly different signatures

```cpp
class BaseClass {
 public:
  int get_member() { return member_; }
  virtual std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  // This compiles. But this is a
  // different function to the
  // BaseClass get_class_name.
  std::string get_class_name() const {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

# Virtual functions

So what happens when we start using virtual members?

```
class BaseClass {
 public:
  virtual std::string get_class_name() {
    return "BaseClass";
  };

 ~BaseClass() {
   std::cout << "Destructing base class\n";
 }
}

class SubClass: public BaseClass {
 public:
  std::string get_class_name() override {
    return "SubClass";
  }

 ~SubClass() {
   std::cout << "Destructing subclass\n";
 }
}
```

```
void print_stuff(const BaseClass& base_class) {
  std::cout << base_class.get_class_name()
        << ' ' << base_class.get_member()
        << '\n';
}

int main() {
  auto subclass = static_cast<std::unique_ptr<BaseClass>>(
    std::make_unique<SubClass>());
  std::cout << subclass->get_class_name();
}
```

demo904-virt.cpp

# Rules for Virtual Function

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism. Base class pointer can point to the objects of base class as well as to the objects of derived class.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

```cpp
#include<iostream>
class base {
public:
        virtual void print() {
                std::cout << "print base class\n";
        }

        void show() {
                std::cout << "show base class\n";
        }
};
class derived : public base {
public:
        void print() {
                std::cout << "print derived class\n";
        }

        void show() {
                std::cout << "show derived class\n";
        }
};

int main()
{
        base *bptr;
        derived d;
        bptr = &d;
        // Virtual function, binded at runtime
        bptr->print();
        // Non-virtual function, binded at compile time
        bptr->show();
    base b1;
    b.print();
    base b2=derived();
    b2.print();
}
```

```cpp
#include <iostream>

class Shape {
  protected:
        int width, height;

  public:
        Shape( int a = 0, int b = 0) {
        width = a;
        height = b;
}
  virtual int area() {
        cout << "Parent class area :" <<endl;
        return 0;
}
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
      int area () {
         std::cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
```

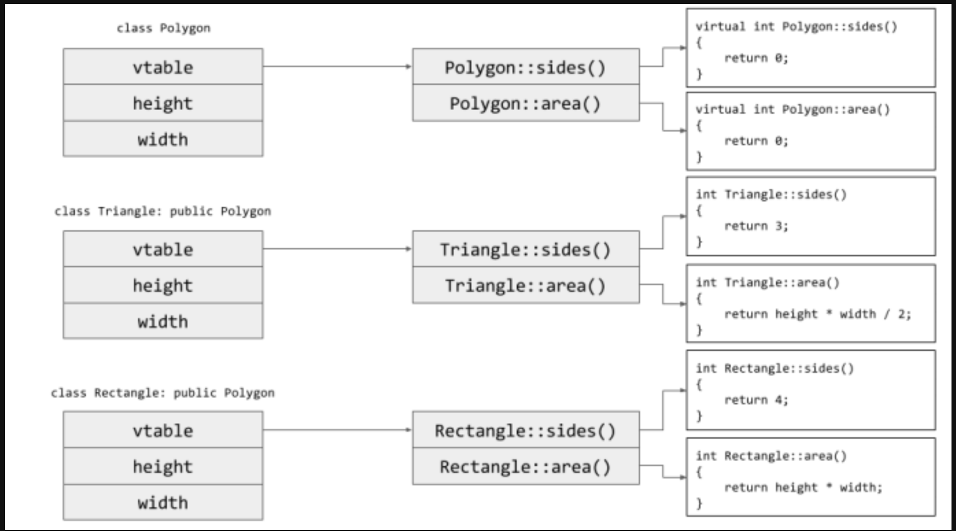
```cpp
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
   // store the address of Rectangle
   shape = &rec;
  // call rectangle area.
   shape->area();
  // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();
   return 0;
}
```

```
Rectangle class area
Triangle class area
```

# VTables

- Each class has a VTable stored in the data segment
    - A vtable is an array of function pointers that says which definition each virtual function points to for that class
- If the VTable for a class is non-empty, then every member of that class has an additional data member that is a pointer to the vtable
- When a virtual function is called **on a reference or pointer type**, then the program actually does the following
    1. Follow the vtable pointer to get to the vtable
    2. Increment by an offset, which is a constant for each function
    3. Follow the function pointer at vtable[offset] and call the function

Another example here



# Final

- Specifies to the compiler "this is not virtual for any subclasses"
- If the compiler has a variable of type SubClass&, it now no longer needs to look it up in the vtable
- This means static binding if you have a SubClass&, but dynamic binding for BaseClass&

```cpp
class BaseClass {
 public:
  int get_member() { return member_; }
  virtual std::string get_class_name() {
    return "BaseClass"
  };

 private:
  int member_;
}
```

```cpp
class SubClass: public BaseClass {
 public:
  std::string get_class_name() override final {
    return "SubClass";
  }

 private:
  int subclass_data_;
}
```

# Types of functions

| Syntax | Name | Meaning |
|---|---|---|
| virtual void fn() = 0; | pure virtual | Inherit interface only |
| virtual void fn() {} | virtual | Inherit interface with optional implementation |
| void fn() {} | nonvirtual | Inherit interface and mandatory implementation |

Note: nonvirtuals can be hidden by writing a function with the same name in a subclass

**DO NOT DO THIS**

# Why We Need Poly

```cpp
class Shape{
public:
void draw(){ cout<<"Shape"<<endl;};
};

class Traingle: public Shape
{
public: void draw(){cout<<"Triangle"<<endl;}
};

class Rectangle: public Shape
{
public: void draw (){cout<<"Rectangle"<<endl;}
};

void pre_draw1(Shape1&);
void pre_draw2(Shape2&);
// ...
void pre_drawN(ShapeN&);

int main(){
std::vector<Shape1> v1 = get_shape1_vector();
std::vector<Shape2> v2 = get_shape2_vector();
// ...
std::vector<ShapeN> vN = get_shapeN_vector();

for(Shape1& s : v1)
    s.draw();
for(Shape2& s : v2)
    s.draw();
// ...
for(ShapeN& s : vN)
    s.draw();
// Suppose we need to modify
for(Shape1& s : v1) {
    pre_draw1(s);
    s.draw();
}
for(Shape2& s : v1) {
    pre_draw2(s);
    s.draw();
}
// ...
for(ShapeN& s : v1) {
    pre_drawN(s);
    s.draw();
}
}
```

```cpp
int main(){
    Traingle tObj;
    tObj->draw();
    Rectangle rObj;
    rObj->draw();
}
```

```cpp
class Shape{
public:
virtual void draw(){ cout<<"Shape"<<endl;};
};

class Traingle: public Shape
{
public: void draw(){cout<<"Triangle"<<endl;}
};

class Rectangle: public Shape
{
public: void draw (){cout<<"Rectangle"<<endl;}
};

void pre_draw(Shape*);

int main(){
std::vector<Shape*> v = get_shape_vector();
for(Shape* s : v)
    s->draw();

    // To modify
for(Shape* s : v) {
pre_draw(s);
s->draw();
}
```

To add new shapes later. simply need to define the new type, and the virtual function. we simply need to add pointers to it into the array and they will be processed just like objects of every other compatible type.

Besides defining the new type, we have to create a new array for it. And need to create a new pre_draw function as well as need to add a new loop to process them.

# Abstract Base Classes (ABCs)

- Might want to deal with a base class, but the base class by itself is nonsense
  - What is the default way to draw a shape? How many sides by default?
  - A function takes in a "Clickable"
- Might want some default behaviour and data, but need others
  - All files have a name, but are reads done over the network or from a disk
- If a class has at least one "abstract" (pure virtual in C++) method, the class is abstract and cannot be constructed
  - It can, however, have constructors and destructors
  - These provide semantics for constructing and destructing the ABC subobject of any derived classes

# Pure virtual functions

- Virtual functions are good for when you have a default implementation that subclasses may want to overwrite
- Sometimes there is no default available
- A pure virtual function specifies a function that a class **must** override in order to not be abstract

```
class Shape {
  // Your derived class "Circle" may forget to write this.
  virtual void draw(Canvas&) {}

  // Fails at link time because there's no definition.
  virtual void draw(Canvas&);

  // Pure virtual function.
  virtual void draw(Canvas&) = 0;
};
```

# Creating polymorphic objects

- In a language like Java, everything is a pointer
  - This allows for code like on the left
  - Not possible in C++ due to objects being stored inline
    - This then leads to slicing problem
- If you want to store a polymorphic object, use a pointer

```cpp
// Java-style C++ here
// Don't do this.

auto base = std::vector<BaseClass>();
base.push_back(BaseClass{});
base.push_back(SubClass1{});
base.push_back(SubClass2{});
```

```cpp
// Good C++ code
// But there's a potential problem here.
// (*very* hard to spot)

auto base = std::vector<std::unique_ptr<BaseClass>>();
base.push_back(std::make_unique<BaseClass>());
base.push_back(std::make_unique<Subclass1>());
base.push_back(std::make_unique<Subclass2>());
```

# Inheritance and constructors

- Every subclass constructor must call a base class constructor
  - If none is manually called, the default constructor is used
  - A subclass cannot initialise fields defined in the base class
  - Abstract classes must have constructors

```
class BaseClass {
 public:
  BaseClass(int member): int_member_{member} {}

 private:
  int int_member_;
  std::string string_member_;
}

class SubClass: public BaseClass {
 public:
  SubClass(int member, std::unique_ptr<int>&& ptr): BaseClass(member), ptr_member_(std::move(ptr)) {}
  // Won't compile.
  SubClass(int member, std::unique_ptr<int>&& ptr): int_member_(member), ptr_member_(std::move(ptr)) {}

 private:
  std::vector<int> vector_member_;
  std::unique_ptr<int> ptr_member_;
}
```

# Destructing polymorphic objects

- Which constructor is called?
- Which destructor is called?
- What could the problem be?
    - What would the consequences be?
- How might we fix it, using the techniques we've already learnt?

```cpp
// Simplification of previous slides code.

auto base = std::make_unique<BaseClass>();
auto subclass = std::make_unique<Subclass>();
```

# Destructing polymorphic objects

- Whenever you write a class intended to be inherited from, **always** make your destructor virtual
- Remember: When you declare a destructor, the move constructor and assignment are not synthesized

```
class BaseClass {
  BaseClass(BaseClass&&) = default;
  BaseClass& operator=(BaseClass&&) = default;
  virtual ~BaseClass() = default;
}
```

Forgetting this can be a hard bug to spot

# Static and dynamic types

- Static type is the type it is declared as
- Dynamic type is the type of the object itself
- Static means compile-time, and dynamic means runtime
  - Due to object slicing, an object that is neither reference or pointer **always** has the same static and dynamic type

Quiz - What's the static and dynamic types of each of these?

```
int main() {
  auto base_class = BaseClass();
  auto subclass = SubClass();
  auto sub_copy = subclass;
  // The following could all be replaced with pointers
  // and have the same effect.
  const BaseClass& base_to_base{base_class};
  // Another reason to use auto - you can't accidentally do this.
  const BaseClass& base_to_sub{subclass};
  // Fails to compile
  const SubClass& sub_to_base{base_class};
  const SubClass& sub_to_sub{subclass};
  // Fails to compile (even though it refers to at a sub);
  const SubClass& sub_to_base_to_sub{base_to_sub};
}
```

# Static and dynamic binding

- Static binding: Decide which function to call at compile time (based on static type)
- Dynamic binding: Decide which function to call at runtime (based on dynamic type)
- C++
  - Statically typed (types are calculated at compile time)
  - Static binding for non-virtual functions
  - Dynamic binding for virtual functions
- Java
  - Statically typed
  - Dynamic binding

# Up-casting

- Casting from a derived class to a base class is called up-casting
- This cast is always safe
  - All dogs are animals
- Because the cast is always safe, C++ allows this as an implicit cast
- One of the reasons to use auto is that it avoids implicit casts

```
auto dog = Dog();

// Up-cast with references.
Animal& animal = dog;
// Up-cast with pointers.
Animal* animal = &dog;
```

# Down-casting

- Casting from a base class to a derived class is called down-casting
- This cast is not safe
  - Not all animals are dogs

```
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with references.
// Neither of these compile.
// Why not?
Dog& dog_ref{animal_dog};
Dog& dog_ref{animal_cat};
```

# How to down cast

- The compiler doesn't know if an Animal happens to be a Dog
  - If you **know** it is, you can use **static_cast**
  - Otherwise, you can use **dynamic_cast**
    - Returns null pointer for pointer types if it doesn't match
    - Throws exceptions for reference types if it doesn't match

```
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with references.
Dog& dog_ref{static_cast<Dog&>(animal_dog)};
Dog& dog_ref{dynamic_cast<Dog&>(animal_dog)};
// Undefined behaviour (incorrect static cast).
Dog& dog_ref{static_cast<Dog&>(animal_cat)};
// Throws exception
Dog& dog_ref{dynamic_cast<Dog&>(animal_cat)};
```

```
auto dog = Dog();
auto cat = Cat();
Animal& animal_dog{dog};
Animal& animal_cat{cat};

// Attempt to down-cast with pointers.
Dog* dog_ref{static_cast<Dog*>(&animal_dog)};
Dog* dog_ref{dynamic_cast<Dog*>(&animal_dog)};
// Undefined behaviour (incorrect static cast).
Dog* dog_ref{static_cast<Dog*>(&animal_cat)};
// returns null pointer
Dog* dog_ref{dynamic_cast<Dog*>(&animal_cat)};
```

# Covariants

- Read more about covariance and contravariance
- **If a function overrides a base, which type can it return?**
  - If a base specifies that it returns a LandAnimal, a derived also needs to return a LandAnimal
- Every possible return type for the derived must be a valid return type for the base

```
class Base {
  virtual LandAnimal& get_favorite_animal();
};

class Derived: public Base {
  // Fails to compile: Not all animals are land animals.
  Animal& get_favorite_animal() override;
  // Compiles: All land animals are land animals.
  LandAnimal& get_favorite_animal() override;
  // Compiles: All dogs are land animals.
  Dog& get_favorite_animal() override;
};
```

# Contravariants

- **If a function overrides a base, which types can it take in?**
  - If a base specifies that it takes in a LandAnimal, a LandAnimal must always be valid input in the derived
- Every possible parameter to the base must be a possible parameter for the derived

```
class Base {
  virtual void use_animal(LandAnimal&);
};

class Derived: public Base {
  // Compiles: All land animals are valid input (animals).
  void use_animal(Animal&) override;
  // Compiles: All land animals are valid input (land animals).
  void use_animal(LandAnimal&) override;
  // Fails to compile: Not All land animals are valid input (dogs).
  void use_animal(Dog&) override;
};
```

# Default arguments and virtuals

- Default arguments are determined at compile time for efficiency's sake
- Hence, default arguments need to use the **static** type of the function
- Avoid default arguments when overriding virtual functions

```cpp
class Base {
public:
  virtual ~Base() = default;
  virtual void print_num(int i = 1) {
    std::cout << "Base " << i << '\n';
  }
};

class Derived: public Base {
public:
  void print_num(int i = 2) override {
    std::cout << "Derived " << i << '\n';
  }
};

int main() {
  Derived derived;
  Base* base = &derived;
  derived.print_num(); // Prints "Derived 2"
  base->print_num(); // Prints "Derived 1"
}
```

demo905-default.cpp

# Construction of derived classes

- Base classes are always constructed before the derived class is constructed
  - The base class ctor never depends on the members of the derived class
  - The derived class ctor may be dependent on the members of the base class

```
class Animal {...}
class LandAnimal: public Animal {...}
class Dog: public LandAnimals {...}

Dog d;

// Dog() calls LandAnimal()
  // LandAnimal() calls Animal()
    // Animal members constructed using initialiser list
    // Animal constructor body runs
  // LandAnimal members constructed using initialiser list
  // LandAnimal constructor body runs
// Dog members constructed using initialiser list
// Dog constructor body runs
```

# Virtuals in constructors

If a class is not fully constructed, cannot perform dynamic binding

```
class Animal {...};
class LandAnimal: public Animal {
  LandAnimal() {
    Run();
  }

  virtual void Run() {
    std::cout << "Land animal running\n";
  }
};
class Dog: public LandAnimals {
  void Run() override {
    std::cout << "Dog running\n";
  }
};

// When the LandAnimal constructor is being called,
// the Dog part of the object has not been constructed yet.
// C++ chooses to not allow dynamic binding in constructors
// because Dog::Run() might depend upon Dog's members.
Dog d;
```

# Destruction of derived classes

Easy to remember order: Always opposite to construction order

```
class Animal {...}
class LandAnimal: public Animal {...}
class Dog: public LandAnimals {...}

auto d =  Dog();

// ~Dog() destructor body runs
  // Dog members destructed in reverse order of declaration
    // ~LandAnimal() destructor body runs
    // LandAnimal members destructed in reverse order of declaration
  // ~Animal() destructor body runs
// Animal members destructed in reverse order of declaration.
```

# Virtuals in destructors

- If a class is partially destructed, cannot perform dynamic binding
- Unrelated to the destructor itself being virtual

```cpp
class Animal {...};
class LandAnimal: public Animal {
  virtual ~LandAnimal() {
    Run();
  }

  virtual void Run() {
    std::cout << "Land animal running\n";
  }
};
class Dog: public LandAnimals {
  void Run() override {
    std::cout << "Dog running\n";
  }
};

// When the LandAnimal constructor is being called,
// the Dog part of the object has already been destroyed.
// C++ chooses to not allow dynamic binding in destructors
// because Dog::Run() might depend upon Dog's members.
auto d = Dog();
```

# Feedback