# Dynamic binding of methods

- by default in C++ the data type of a variable/parameter is used to determine which method is called through it
- the compiler can determine this at compile time, called **static binding**

```
void dosomething(shape *s) {
    s->somemethod();  // s is a shape, so call shape::somemethod
};
```

- suppose circles and squares are derived from shape, and override the inherited shape version of somemethod
- dosomething still just calls shape's version

```
circle c;
...
dosomething(&c);
```

# Dynamic binding

- dynamic binding is when the compiler inserts code so that during execution the most suitable method is called
  - under dynamic binding, s->circle::dosomething() would have been called in the previous example
- in C++ we specify we want methods dynamically bound using the virtual and override keywords

```
class shape {
  ....
  virtual void dosomething();
  ....
};
```

```
class circle: public shape {
   ....
   virtual void dosomething() override;
   ....
};
```

# Usefulness of dynamic binding

- we might have a large inheritance tree of objects descending from one base class
- we might have functions doing very similar things to/with all these objects, and want the functions to call the correct methods for each
- under static binding we'd need a special function for each class, e.g.
  - void dosomething(circle* c);
  - void dosomething(square* s);
- under dynamic binding we can have a single function
  - void dosomething(shape* s)
- avoids lots of code repetition, works on any class derived from shape

# Example: games

- games may have thousands of object types (NPCs, player characters, rocks, chairs, flowers, cars, guns, etc etc etc)
- there are many actions that can be applied to each (move the object, create the object, paint the object, interact with the object, etc etc)
- allows the main game processing cycle to be something like:
  - create a bunch of objects
  - repeat until game over:
    - detect next event that takes place & which object affected
    - call update function, passing event and object pointers

# Example: display/drawing programs

- suppose we have a program to draw things on the screen, or to allow the user to edit drawings
- potentially thousands of different kinds of shapes/objects can be drawn
- a common set of actions apply to each: redraw, rotate, resize, destroy, change colour/texture, etc
- again, the main processing cycle can be:
  - detect what the user wants to do next and to which shape
  - call an update function passing the shape and action

# Example: data storage/lookups

- our labs used linked lists and binary search trees to store/lookup data
- the parts that interacted with the user look almost identical, repeatedly:
  - ask the user what they want to do next
  - call process function to get more information from them and call the appropriate insert/lookup/etc
- with dynamic binding we could try:
  - set up a base class, DataStore, from which we derive our lists and trees
  - in the main routine user first gets to pick their desired storage type
  - process takes a DataStore* parameter, we pass it either the bstree or list and its call to insert/lookup/etc uses the correct overridden method

# Data store of key/value string pairs

```
class DataStore {
   public:
       DataStore();
      ~DataStore();
      virtual bool insert(string k, string v);
      virtual bool lookup(string k, string& v);
      virtual bool remove(string k);
      virtual void printall();
      virtual int getsize();
};

class List: virtual public DataStore {
   private:
        ...whatever we need for linked list implementation...
   public:
       ... constructors, destructors, then for each inherited method:
       virtual bool insert(string k, string v) override;
};
```

```
void Process(DataStore* s, char userCmd)
{
    if ((userCmd == 'P') && (s != NULL)) {
       s->printall();
    } else if ((userCmd == 'I') .... etc ...

}
```

# Pure virtual methods

- instead of a base class providing an actual implementation of a method they can define it as a pure virtual method
- done by assigning 0 instead of giving an implementation
- they have no implementation so ***MUST*** be overriden by descendants

```
class DataStore {
   ...
   virtual bool insert(string k, string v) = 0;  // we assign 0, DataStore never gives a body for insert
   ...
};

class List: virtual public DataStore {
   ...
   virtual bool insert(string k, string v) override; // List MUST override insert
   ...
};
```

# Abstract base classes

- classes that declare pure virtual methods (like DataStore in previous slide) are called abstract base classes
- you cannot create an instance of an abstract base class (since it has no implementation for at least one method)
- you can still use pointers to abstract base classes for dynamic binding, but what you actually pass to it will be a descendant

```
void process(DataStore* s, char cmd)
{
   ...
}
```

```
int main() {
    List *L = new List;
    ...
       process(L, 'P');
    ...
}
```