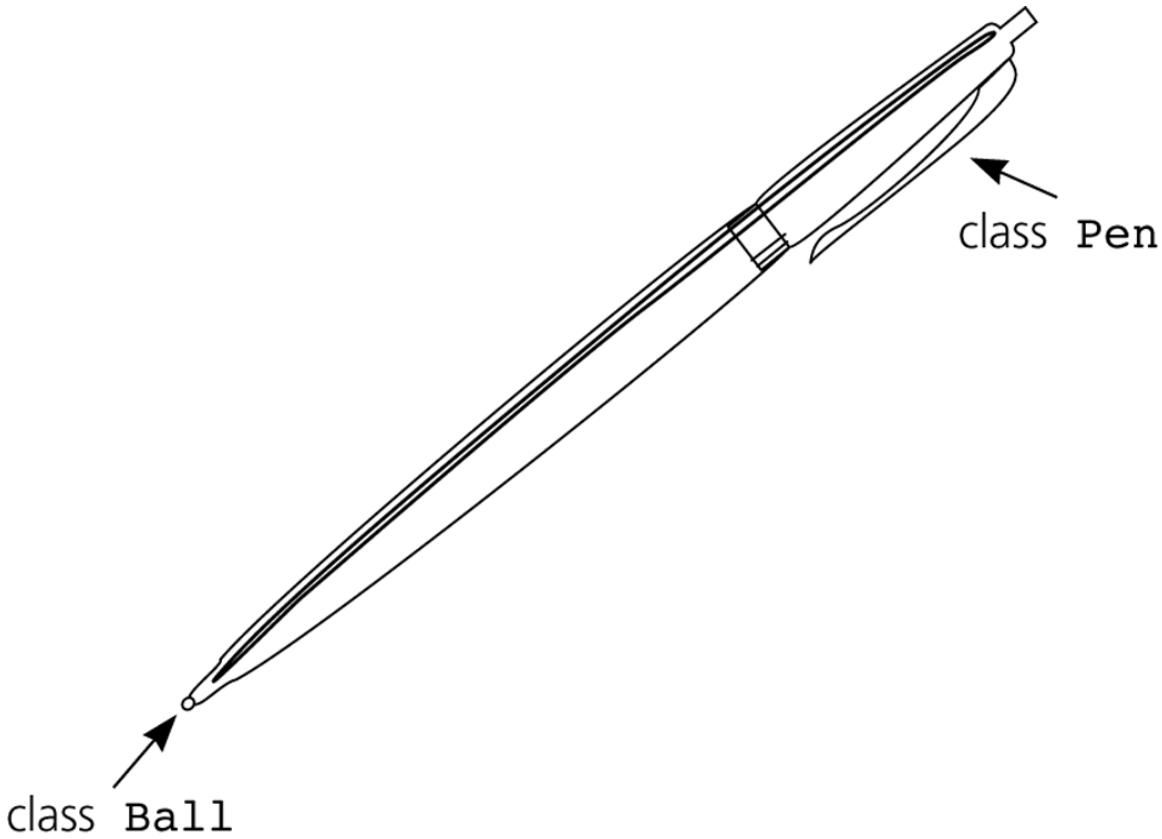
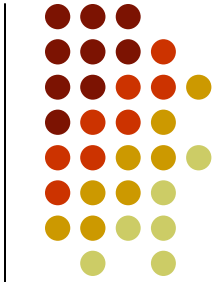
Has-a Relationships



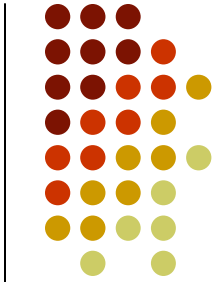
A pen “has a” or “contains a” ball

Has-a Relationships



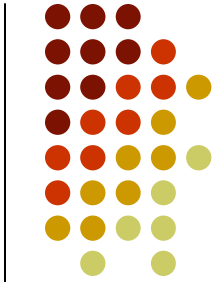
- Has-a relationship
  - Also called **containment**
  - Cannot be implemented using inheritance
    - Example: To implement the has-a relationship between a pen and a ball
      - Define a data field `point` – whose type is `Ball` – within the class `Pen`
    - Examples: when implementing Stack or Queue using ADT List we used containment!

Has-a Relationships in C++



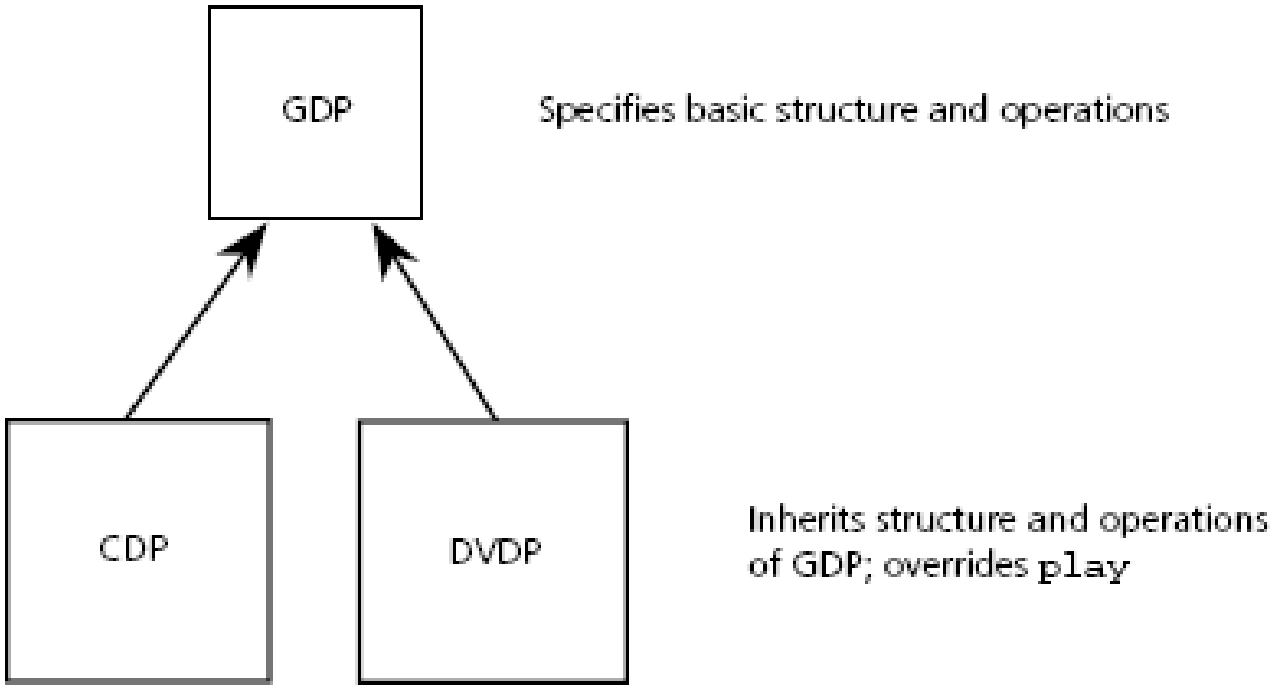
- To implement the has-a relationship in C++, we can use 2 different ways:
  - **containment** – but the objects are directly contained in the containing class (not just references to them) – you have to use initialization list to initialize them
  - **private inheritance** – seldom used, allows overriding the methods of contained class

Abstract Classes



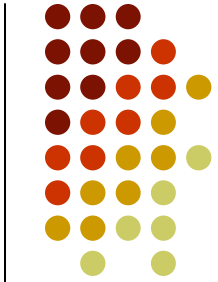
- Example
  - CD player and DVD player
    - Both involve an optical disk
    - Operations
      - Insert, remove, play, record, and stop such discs

Abstract Classes



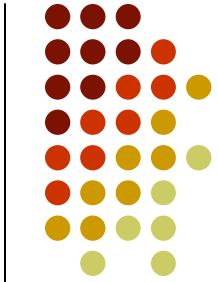
CDP and DVDP have an abstract base class GDP

Abstract Classes



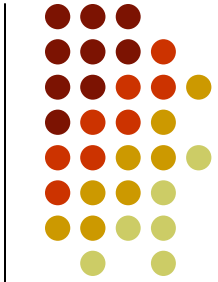
- Abstract classes
  - An abstract class is used only as the basis for subclasses
    - It defines a minimum set of methods and data fields for its subclasses
  - An abstract class has no instances
  - An abstract class should, in general, omit implementations except for the methods that
    - Provide access to private data fields
    - Express functionality common to all of the subclasses

Abstract Classes



- Abstract classes (Continued)
  - A class that contains at least one abstract method must be declared as an abstract class
  - A subclass of an abstract class must be declared abstract if it does not provide implementations for all abstract methods in the superclass

Abstract Base Class – Example in C++

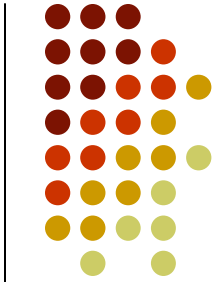


```
class Shape // abstract base class
{
 public:
  virtual void move(int dx, int dy) =0; // pure virtual function
  virtual void show() const =0;         // pure virtual function
};

class Circle : public Shape // concrete class
{
  int xcenter,ycenter,radius;
 public:
  Circle(int xc, int yc, int r) : xcenter(xc), ycenter(yc), radius(r) {}
  void move(int dx, int dy) { xcenter += dx; ycenter += dy; }
  void show() const {
    cout <<"Circle centered at ("<<xcenter<<","<<ycenter<<") with radius "
         <<radius<<"."<<endl;
  }
};

class Square : public Shape // concrete class
{
  int xll,yll,xur,yur;
 public:
  Square(int x1, int y1, int x2, int y2)
    : xll(x1), yll(y1), xur(x2), yur(y2) {}
  void move(int dx, int dy) { xll += dx; yll += dy; xur += dx; yur += dy; }
  void show() const {
    cout <<"Square with lower left corner at ("<<xll<<","<<yll
         <<") and upper right corner at ("<<xur<<","<<yur<<")."<<endl;
  }
};
```

Abstract Base Class - Example

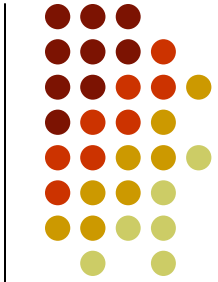
```
int main() {
//  Shape s; // compiler error, Shape is abstract

  Circle c(10,20,5);
  Square s(0,0,15,10);

  Shape *array[2]={&c,&s};

  cout <<"Before move:"<<endl;
  for (int i=0; i<2; i++)
    array[i]->show();

  for (int i=0; i<2; i++)
    array[i]->move(5,-5);

  cout <<endl<<"After move:"<<endl;
  for (int i=0; i<2; i++)
    array[i]->show();

  cin.get();
  return 0;
}
```

Generic programming



- ADTs we have design so far where holding references/pointers to Object
- in most practical application, we want to store in ADT only particular type; for instance: *Integer* or *String*
- typecasting from *Object* to desired type is troublesome
- one option would be rewrite the code of ADT each time storing different type (*Integer* instead of *Object*, etc.), but that's not a good design (changes are not localized)

Generic programming – example



- solution: use templates
- Example: (implementation of Stack with linked list)

```
public class Node <T> {
  private T item;
  private Node<T> next;

  public Node(T newItem) {
    item = newItem;
    next = null;
  } // end constructor

  public Node(T newItem, Node<T> nextNode) {
    item = newItem;
    next = nextNode;
  } // end constructor
```

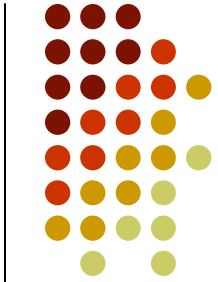
Generic programming – example

```
  public void setItem(T newItem) {
    item = newItem;
  } // end setItem

  public T getItem() {
    return item;
  } // end getItem

  public void setNext(Node<T> nextNode) {
    next = nextNode;
  } // end setNext

  public Node<T> getNext() {
    return next;
  } // end getNext
} // end class Node
```

Generic Stack

```
public class StackReferenceBased <T> {
  private Node<T> top;

  public StackReferenceBased() {
    top = null;
  }  // end default constructor

  public boolean isEmpty() {
    return top ==  null;
  }  // end isEmpty

  public void push(T newItem) {
    top = new Node<T>(newItem, top);
  }  // end push

  public T pop() throws StackException {
    if (!isEmpty()) {
      Node<T> temp = top;
      top = top.getNext();
      return temp.getItem();
    }
    else {
      throw new StackException("StackException on " +
                               "pop: stack empty");
    }  // end if
  }  // end pop
```

Generic Stack (continued)



```
  public void popAll() {
    top = null;
  }  // end popAll

  public T peek() throws StackException {
    if (!isEmpty()) {
      return top.getItem();
    }
    else {
      throw new StackException("StackException on " +
                               "peek: stack empty");
    }  // end if
  }  // end peek
}  // end StackReferenceBased
```

- how to use:

```
Stack<Character> S=new Stack<Character>();
S.push(new Character('a'));
```

Generic programming in Java and C++

- Java: the substituted type has to be a class (hence it's derived from *Object*)
- C++: different syntax:

```
template <typename T>
class Node <T> {
private:
  T item;
  Node<T> next;
public:
  Node(T newItem) {
    item = newItem;
    next = NULL;
  } // end constructor
```

etc..

- T can be any type also *int* or *char*
- there are other specifics of generic programming in Java and C++ (not discussed in the course, see textbook or other material)