INHERITANCE AND GENERALIZATION

LECTURE 10-2

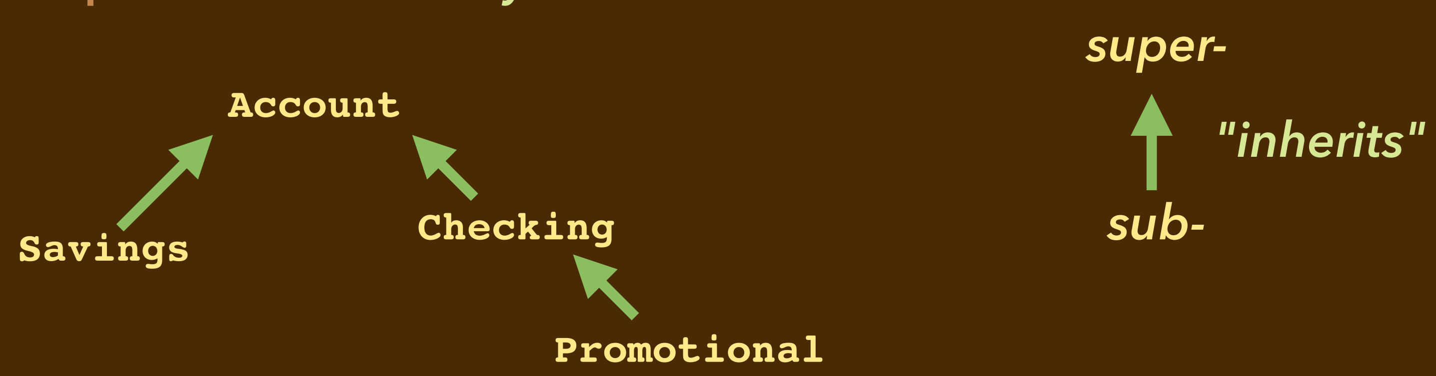
JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- INHERITANCE (CONT'D):
  - FINISH ACCOUNT EXAMPLES
  - DYNAMIC DISPATCH WITH `virtual`
  - PURELY ABSTRACT CLASSES
  - SHAPE EXAMPLE
- CLASS TEMPLATES
  - A GENERIC STACK EXAMPLE

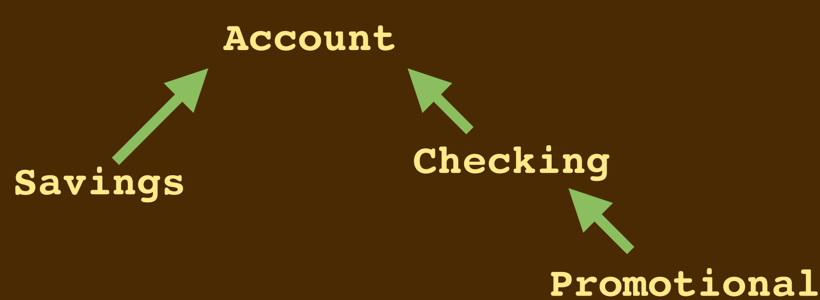
INHERITANCE

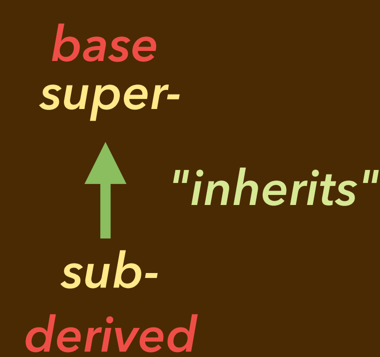
- **RECALL:** OO languages allow us to extend object classes:
  - ➡ adding instance variables enhances what they can represent.
  - ➡ adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:



INHERITANCE

- **RECALL:** OO languages allow us to extend object classes:
  - ➡ adding instance variables enhances what they can represent.
  - ➡ adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:





ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber; // used to generate account nos.
  // instance variables
  std::string name;        // description of the account
  long number;             // account no.
  double balance;          // money held
  double rate;             // monthly interest
public:
  ...
};
```

ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber;
  // instance variables
  ...
public:
  Account(std::string name, double amount, double interest);
  // getters
  double getBalance() const;
  std::string getName() const;
  long getNumber() const;
  double getRate() const;
  // methods
  void deposit(double amount);    // add money
  void gainInterest();            // each month
  double withdraw(double amount); // remove money
};
```

ACCOUNT CLASS IMPLEMENTATION (MISSING GETTERS)

```
Account::Account(std::string name, double amount, double
interest) : name {name},
            balance {amount},
            rate {interest},
            number {Account::gNextNumber++}
{ }

void Account::deposit(double amount) {
  balance += amount;
}
void Account::gainInterest() {
  deposit(rate * balance);
}
double Account::withdraw(double amount) {
  if (amount > balance) {
    amount = balance;
    balance = 0.0;
  } else {
    balance -= amount;
  }
  return amount;
}
```

SUBCLASSES OF ACCOUNT

- Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- Checking accounts accrue 1% interest, but only if balance is above $1000.

```
class Checking : public Account { ... }
```

- Promotional checking accounts accrue 0.7% interest, but give you $100 to open the account. You must stay above $100 to earn that interest.

```
class Promotional : public Checking { ... }
```

- The keyword `public` means that
  - all public members are accessible as public members in the derived class,
  - all protected members are accessible as public members in the derived class,
  - private members are only accessible if a friend.

ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
protected:
  // instance variables
  std::string name;
  long number;
  double balance;
  double rate;
public:
  // methods
  ...
};
```

Not publicly accessible, but accessible to any derived class.

ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
  // instance variables
  ...
public:
  // methods
  Account(std::string name, double amount, double interest);
  virtual double getBalance() const;
  virtual std::string getName() const;
  virtual long getNumber() const;
  virtual double getRate() const;
  virtual void deposit(double amount);
  virtual void gainInterest();
  virtual double withdraw(double amount);
};
```

Virtual keyword indicates that the code of overriding methods in subclass will get called.

EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};
```

SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We add a `penalty` instance variable.
- We *override* the `withdraw` method to charge that penalty.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We add a `penalty` instance variable.
- We *override* the `withdraw` method to charge that penalty.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We add a `level` instance variable.
- We *override* the `gainInterest` method to check that level.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We add a `level` instance variable.
- We *override* the `gainInterest` method to check that level.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

PROMOTIONAL (CHECKING) ACCOUNT

Promotional accrues less interest, has an opening gift, has lower threshold.

- They derive from **`Checking`**. No extensions or overrides.

```
class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};


Promotional::Promotional(std::string name, double amount) :
  Checking {name, amount + 100.0}
{
  rate = 0.07;
  level = 100.0;
}
```

NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **`m`** is not marked virtual, the code for **`A::m`** runs instead.
  - This is sometimes called "*static dispatch*" of the "message" **`m`**.

VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **`m`** is marked virtual, the code for **`B::m`** runs like we'd normally expect.
  - This is sometimes called "*dynamic dispatch*" of the "message" **`m`**.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
- Because the **`draw`** method is **`virtual`**, dynamic dispatch is used.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of `shapes`:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, `current->shape` is of type `Shape*`.
- Because the `draw` method is `virtual`, dynamic dispatch is used.
  - When the list node points to an `Oval` instance, `Oval::draw` is called.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
- Because the **`draw`** method is **`virtual`**, dynamic dispatch is used.
  - When the list node points to an **`Oval`** instance, **`Oval::draw`** is called.
  - When the list node points to a **`Rectangle`**, its method is called.

ABSTRACT CLASSES

- Note that the **`Account`** class shouldn't have an instance.
  - Nonetheless, it does define a few methods' useful to subclass instances:
    - The **`deposit`** and **`withdraw`** methods as defined in **`Account`** provide a default behavior that subclasses may use, or override.

- Classes not meant to be instantiated are called *abstract*.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- We can't always provide a "default" behavior in the base abstract class.
- In C++ we can designate methods as "purely virtual" with a value of 0:

```
class A {
  ...
  virtual T m(T1 v1, T2 v2, ...) = 0;
  ...
};

class B : public A {
  ...
  T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
  ...
};
```

➡ Method **m** must be defined by classes that derive from abstract **A**.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- We can't always provide a "default" behavior in the base abstract class.
- In C++ we can designate methods as "*purely virtual*" with a value of 0:

```
class A {
   ...
   virtual T m(T1 v1, T2 v2, ...) = 0;
   ...
};

class B : public A {
   ...
   T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
   ...
};
```

➡Method **m** must be defined by classes that derive from abstract **A**.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- We can't always provide a "default" behavior in the base abstract class.
- In C++ we can designate methods as "purely virtual" with a value of 0:

```
class A {
   ...
   virtual T m(T1 v1, T2 v2, ...) = 0;
   ...
};

class B : public A {
   ...
   T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
   ...
};
```

➡ Method **m** must be defined by classes that derive from abstract **A**.

EXAMPLE: SHAPE HIERARCHY

```
class Shape {
public:
  virtual double perimeter(void) const = 0;
  virtual double area(void) const = 0;
  virtual void print(void) const = 0;
  virtual double getHeight(void) const = 0;
  virtual double getWidth(void) const = 0;
  virtual void scale(double factor) = 0;
  Rectangle bounds(void);
};
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
  void scale(double factor) { radius *= factor;}
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
  void scale(double factor) { radius *= factor;}
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
  void scale(double factor) { radius *= factor;}
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
  void scale(double factor) { radius *= factor;}
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  void scale(double factor) { width *=factor; height *=factor; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  void scale(double factor) { width *=factor; height *=factor; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  void scale(double factor) { width *=factor; height *=factor; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void) const;
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  void scale(double factor) { width *=factor; height *=factor; }
  friend class Square;
};
void Rectangle::print(void) const {
  std::cout << "Here is a " << width << "x" << height;
  std::cout << " rectangle:\n" << std::endl;
  depict();
}
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void) const;
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  void scale(double factor) { width *=factor; height *=factor; }
  friend class Square;
};
void Rectangle::print(void) const {
  std::cout << "Here is a " << width << "x" << height;
  std::cout << " rectangle:\n" << std::endl;
  depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
   ...
   friend Square;
}

class Square : public Rectangle {
public:
   Square(double s) : Rectangle {s, s} { }
   void print(void) const;
};

void Square::print(void) const {
   std::cout << "Here is a " << getWidth() << "x" << getHeight();
   std::cout << " square:\n" << std::endl;
   Rectangle::depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
   ...
   friend Square;
}

class Square : public Rectangle {
public:
   Square(double s) : Rectangle {s, s} { }
   void print(void) const;
};

void Square::print(void) const {
   std::cout << "Here is a " << getWidth() << "x" << getHeight();
   std::cout << " square:\n" << std::endl;
   Rectangle::depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
  ...
  friend Square;
}

class Square : public Rectangle {
public:
  Square(double s) : Rectangle {s, s} { }
  void print(void) const;
};

void Square::print(void) const {
  std::cout << "Here is a " << getWidth() << "x" << getHeight();
  std::cout << " square:\n" << std::endl;
  Rectangle::depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
  ...
  friend Square;
}

class Square : public Rectangle {
public:
  Square(double s) : Rectangle {s, s} { }
  void print(void) const;
};

void Square::print(void) const {
  std::cout << "Here is a " << getWidth() << "x" << getHeight();
  std::cout << " square:\n" << std::endl;
  Rectangle::depict();
}
```

SHAPE PROGRAM OUTPUT

```
Here is a circle with radius 5:

  +++++++
 +++++++++
+++++++++++
+++++++++++
+++++++++++
+++++++++++
+++++++++++
+++++++++++
 +++++++++
  +++++++

Here is a 7x3 rectangle:

+++++++
+++++++
+++++++

Here is a 1x1 square:

+
```

POLYMORPHISM

- Some people say that subclassing provides a kind of ***polymorphism***
  - ***We can have a list of shapes, but the shapes can be of different types.***
  - *poly* **"multiple/many"** + *morph* **"shape/form"**
- Elements are instances of many classes derived from the same base class.

POLYMORPHISM

- Some people say that subclassing provides a kind of ***polymorphism***
  - ***We can have a list of shapes, but the shapes can be of different types.***
  - *poly* **"multiple/many"** + *morph* **"shape/form"**
- Elements are instances of many classes derived from the same base class.
- **Recall:** our container classes have had to fixate on an element type:

```
class IntStck { int* elements; ... };
class StringStck { std::string* elements; ...};
struct ShapePtr { Shape *data; };
class ShapeStck { ShapePtr* elements; ... };
```

POLYMORPHISM

- Some people say that subclassing provides a kind of ***polymorphism***
  - ***We can have a list of shapes, but the shapes can be of different types.***
  - *poly* "multiple/many" + *morph* "shape/form"
- Elements are instances of many classes derived from the same base class.
- **Recall:** our container classes have had to fixate on an element type:

```
class IntStck { int* elements; ... };
class StringStck { std::string* elements; ...};
struct ShapePtr { Shape *data; };
class ShapeStck { ShapePtr* elements; ... };
```

POLYMORPHISM

- Some people say that subclassing provides a kind of ***polymorphism***
  - ***We can have a list of shapes, but the shapes can be of different types.***
  - *poly* "multiple/many" + *morph* "shape/form"
- Elements are instances of many classes derived from the same base class.
- **Recall:** our container classes have had to fixate on an element type:

```
class IntStck { int* elements; ... };
class StringStck { std::string* elements; ...};
struct ShapePtr { Shape *data; };
class ShapeStck { ShapePtr* elements; ... };
```

- ***Wouldn't it be nice*** if we could define `Stck` once to take many forms?

```
class Stck<T> { T* elements; ... };
```

- Have this describe `Stck<int>`, `Stck<std::string>`, `Stck<ShapePtr>`???

TEMPLATE CLASSES

- C++ also provides an ability to "abstract away" the defining types of a class:
- We can define a **`class A`** with type parameters **`T1,T2,...`** :

```
class A<T1, T2...> {
   ...
   // T1 and T2 used as type names throughout its definition
   ...
};
```

- Then the client code can stamp out different **`A`** types, like so:

```
A<int,std::string> a1 = ...;
A<char,bool> a2 = ...;
```

- The definition of **`class A`** provides a *template* for *different forms* of **`A`**.

EXAMPLE: TEMPLATE STACK CLASS (SEE STCK_T.HH)

```
template <class X>
class Stck {

private:
  int capacity;
  int num_elements;
  X *elements;

public:
  Stck(const int size);
  const bool is_empty() const;
  void push(const X value);
  X pop();
  const X top() const;
  const std::string to_string() const;
  ~Stck();
};
```

EXAMPLE: TEMPLATE STACK CLASS (SEE STCK_T.HH)

```
template <class X>
class Stck {

private:
  int capacity;
  int num_elements;
  X *elements;

public:
  Stck(const int size);
  const bool is_empty() const;
  void push(const X value);
  X pop();
  const X top() const;
  const std::string to_string() const;
  ~Stck();
};
```

SOME SAMPLE TEMPLATE METHODS (ALSO IN STCK_T.HH)

```
template <class X>
Stck<X>::Stck(const int size) :
  capacity {size},
  num_elements{0},
  elements {new X[size]}
{ }

template <class X>
void Stck<X>::push(const X value) {
  elements[num_elements] = value;
  num_elements++;
}

template <class X>
X Stck<X>::pop() {
  num_elements--;
  return elements[num_elements];
}
```

USE OF TEMPLATE BY CLIENT: A NEW DC.CC

```
#include <iostream>
#include <string>
#include "Stck_T.hh"

int main() {

  Stck<int> s(100);

  std::string entry;
  do {
    std::cin >> entry;
    if (entry == "+") {
      int v1 = s.pop();
      int v2 = s.pop();
      int v = v1 + v2;
      s.push(v);
    } else if (entry == "-") {
      int v1 = s.pop();
      int v2 = s.pop();
      int v = v1 - v2;
      s.push(v);
  ...
```

USE OF TEMPLATE BY CLIENT: A DIFFERENT DC.CC

```
#include <iostream>
#include <string>
#include "Stck_T.hh"

int main() {

  Stck<double> s(100);

  std::string entry;
  do {
    std::cin >> entry;
    if (entry == "+") {
      int v1 = s.pop();
      int v2 = s.pop();
      int v = v1 + v2;
      s.push(v);
    } else if (entry == "-") {
      int v1 = s.pop();
      int v2 = s.pop();
      int v = v1 - v2;
      s.push(v);
  ...
```

NOTES ON TEMPLATES

- Templates provide something like "*generics*" (term used in **Java**).
- Notion comes from the functional prog. lang. community (e.g. **CaML**):
  - ➡*parameterized polymorhism*, e.g. `T list`
- Separate compilation in C++ makes templates tricky:
  - You must put *everything* (spec'n and impl'n) into a header.
  - Client code `#include`s the full definition, class and methods.
  - Compiler stamps out different code, code *for each* type parameterization.
- The C++ template mechanism is awkward...

  ...but generics/parametrized types are a very useful and elegant concept.