Polymorphism: "many forms"

```
int main() {
    std::vector<Account> my_accounts;

    // this is sort of OK, since CheckingAccount is
    // derived from Account (except: "slicing" will
    // occur)
    my_accounts.push_back(CheckingAccount(2000.0));

    std::cout << my_accounts.back().type() << std::endl;
    return 0;
}
```

Polymorphism: "many forms"

```
class Account {
public:
    ...
    std::string type() const { return "Account"; }
    ...
};

class CheckingAccount : public Account {
public:
    ...
    std::string type() const { return "CheckingAccount"; }
    ...
};

class SavingsAccount : public Account {
public:
    ...
    std::string type() const { return "SavingsAccount"; }
    ...
};
```

Polymorphism: "many forms"

```
// account.h

#include <string>
class Account {
public:
  Account() : balance(0.0) { }
  Account(double initial) : balance(initial) { }

  void credit(double amt)    { balance += amt; }
  void debit(double amt)     { balance -= amt; }
  double get_balance() const { return balance; }
  std::string type() const { return "Account"; }
private:
  double balance;
};

class CheckingAccount : public Account {
public:
  CheckingAccount(double initial, double atm) :
  Account(initial), total_fees(0.0),
  atm_fee(atm) { }
  void cash_withdrawal(double amt) {
    total_fees += atm_fee;
    debit(amt + atm_fee);
  }
  double get_total_fees() const {
    return total_fees;
  }
  std::string type() const {
    return "CheckingAccount";
  }

private:
  double total_fees;
  double atm_fee;
};

class SavingsAccount : public Account {
public:
  SavingsAccount(double initial, double rate) :
  Account(initial), annual_rate(rate) { }

  //Not shown here; usual compound interest calc
  double total_after_years(int years);

  std::string type() const {
    return "SavingsAccount";
  }

private:
  double annual_rate;
};
```

Polymorphism: "many forms"

```
// account_main.cpp

#include <iostream>
#include "account.h"

void print_account_type(const Account& acct) {
    std::cout << acct.type() << std::endl;
}

int main() {
    Account acct(1000.0);
    CheckingAccount checking(1000.0, 2.00);
    SavingsAccount saving(1000.0, 0.05);

    print_account_type(acct);
    print_account_type(checking);
    print_account_type(saving);

    return 0;
}
```

Polymorphism: "many forms"

Note the types:

```
void print_account_type(const Account& acct) {
    std::cout << acct.type() << std::endl;
}

int main() {
    ...
    CheckingAccount checking(1000.0, 2.00);
    ...
    print_account_type(checking);
    ...
}
```

Polymorphism: "many forms"

In `main`, `checking_acct` has type `CheckingAccount`

Passed to `print_account_type` as `const Account&`

- This is allowed; `CheckingAccount` is derived from `Account`

Usually, you may use a variable of a derived type **as though it has the base type**

- Makes logical sense; `CheckingAccount` **is-an** `Account`

Polymorphism: "many forms"

```
void print_account_type(const Account& acct) {
    std::cout << acct.type() << std::endl;
}

int main() {
    ...
    CheckingAccount checking(1000.0, 2.00);
    ...
    print_account_type(checking);
    ...
}
```

Does `acct.type()` call `Account::type()` (matching parameter's type) or `CheckingAccount::type()` (matching the original declared type)?

Polymorphism: "many forms"

```
$ g++ -o account_main account_main.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./account_main
```

```
Account
Account
Account
```

It calls `Account::type()`

Can we force `print_account_type` to call the function corresponding to the **actual** type (`CheckingAccount`) rather than the locally declared base type (`Account`)?

This requires **dynamic binding**

To use it, we declare relevant member functions as `virtual`

Polymorphism: dynamic binding

```
class Account {
public:
    ...
    virtual std::string type() const { return "Account"; }
    ...
};

class CheckingAccount : public Account {
public:
    ...
    virtual std::string type() const { return "CheckingAccount"; }
    ...
};

class SavingsAccount : public Account {
public:
    ...
    virtual std::string type() const { return "SavingsAccount"; }
    ...
};
```

Polymorphism: dynamic binding

```
// account2.h

#include <string>
class Account {
public:
  Account() : balance(0.0) { }
  Account(double initial) : balance(initial) { }

  void credit(double amt)    { balance += amt; }
  void debit(double amt)     { balance -= amt; }
  double get_balance() const { return balance; }
  virtual std::string type() const {
    return "Account";
  }
private:
  double balance;
};

class CheckingAccount : public Account {
public:
  CheckingAccount(double initial, double atm) :
  Account(initial), total_fees(0.0),
  atm_fee(atm) { }
  void cash_withdrawal(double amt) {
    total_fees += atm_fee;
    debit(amt + atm_fee);
  }
  double get_total_fees() const {
    return total_fees;
  }

  virtual std::string type() const {
    return "CheckingAccount";
  }

private:
  double total_fees;
  double atm_fee;
};

class SavingsAccount : public Account {
public:
  SavingsAccount(double initial, double rate) :
  Account(initial), annual_rate(rate) { }

  //Not shown here; usual compound interest calc
  double total_after_years(int years);

  virtual std::string type() const {
    return "SavingsAccount";
  }

private:
  double annual_rate;
};
```

Polymorphism: dynamic binding

```
    // account_main2.cpp

1   #include <iostream>
2   #include "account2.h"
3
4   void print_account_type(const Account& acct) {
5       std::cout << acct.type() << std::endl;
6   }
7
8   int main() {
9       Account acct(1000.0);
10      CheckingAccount checking(1000.0, 2.00);
11      SavingsAccount saving(1000.0, 0.05);
12
13      print_account_type(acct);
14      print_account_type(checking);
15      print_account_type(saving);
16
17      return 0;
18  }
```

Polymorphism: dynamic binding

```
$ g++ -o account_main2 account_main2.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./account_main2

Account
CheckingAccount
SavingsAccount
```

Quiz!

What output is printed by the following program?

```
// quiz1.cpp

#include <iostream>

class X {
public:
    void p() { std::cout << "X::p "; }
    virtual void q()
    { std::cout << "X::q "; }
};
class Y : public X {
public:
    void p() { std::cout << "Y::p "; }
    virtual void q()
    { std::cout << "Y::q "; }
};

void f(X &obj) { obj.p(); obj.q(); }

int main() { Y myObj; f(myObj); }
```

A. `X::p X::q`
B. `X::p Y::q`
C. `Y::p X::q`
D. `Y::p Y::q`
E. Some other output is printed

Quiz - answers

What output is printed by the following program?

```
// quiz1.cpp

#include <iostream>

class X {
public:
    void p() { std::cout << "X::p "; }
    virtual void q()
    { std::cout << "X::q "; }
};
class Y : public X {
public:
    void p() { std::cout << "Y::p "; }
    virtual void q()
    { std::cout << "Y::q "; }
};

void f(X &obj) { obj.p(); obj.q(); }

int main() { Y myObj; f(myObj); }
```

At line 16:

Symbols(Scope-Type)	Values
myObj(main-Y), obj(f-X)	-
(obj.p())(f)	"X::p "
(obj.q())(f)	"Y::q "

A very similar quiz!

What output is printed by the following program?

```
// quiz2.cpp

#include <iostream>

class X {
public:
    void p() { std::cout << "X::p "; }
    virtual void q()
    { std::cout << "X::q "; }
};
class Y : public X {
public:
    void p() { std::cout << "Y::p "; }
    virtual void q()
    { std::cout << "Y::q "; }
};

void f(X obj) { obj.p(); obj.q(); }

int main() { Y myObj; f(myObj); }
```

A. `X::p X::q`
B. `X::p Y::q`
C. `Y::p X::q`
D. `Y::p Y::q`
E. Some other output is printed

A very similar quiz - answers

What output is printed by the following program?

```
// quiz2.cpp

#include <iostream>

class X {
public:
    void p() { std::cout << "X::p "; }
    virtual void q()
    { std::cout << "X::q "; }
};
class Y : public X {
public:
    void p() { std::cout << "Y::p "; }
    virtual void q()
    { std::cout << "Y::q "; }
};

void f(X obj) { obj.p(); obj.q(); }

int main() { Y myObj; f(myObj); }
```

At line 16:

Symbols(Scope-Type)	Values
myObj(main-Y)	-
obj(f-X)	-
(obj.p())(f)	"X::p "
(obj.q())(f)	"X::q "