

Dynamic dispatch

- What is **object slicing**?
- How does **virtual** work? (Dynamic dispatch)
- The keyword/modifier **override**.

The Account example w/o virtual

```
// acc.cpp
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      Account() : balance(0.0) { }
7      Account(double initial) : balance(initial) { }
8
9      void credit(double amt)    { balance += amt; }
10     void debit(double amt)     { balance -= amt; }
11     double get_balance() const { return balance; }
12     std::string type() const { return "Account"; }
13 private:
14     double balance;
15 };
16
17 class CheckingAccount : public Account {
18 public:
19     CheckingAccount(double initial, double atm) :
20         Account(initial), total_fees(0.0),
21         atm_fee(atm) { }
22     void cash_withdrawal(double amt) {
23         total_fees += atm_fee;
24         debit(amt + atm_fee);
25     }
26     double get_total_fees() const {
27         return total_fees;
28     }
29
30     std::string type() const {
31         return "CheckingAccount";
32     }
33 private:
34     double total_fees;
35     double atm_fee;
36 };
37
38 void print_account_type(const Account& acct) {
39     std::cout << acct.type() << std::endl;
40 }
41
42 int main() {
43     Account acct(1000.0);
44     CheckingAccount checking(1000.0, 2.00);
45     print_account_type(acct);
46     print_account_type(checking);
47     return 0;
48 }
```

The Account example w/o `virtual`

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

```
$ ./acc
```

```
Account
```

```
Account
```

It doesn't work without `virtual`.

The Account example using `virtual`

```
// acc_vt.cpp
```

```
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      Account() : balance(0.0) { }
7      Account(double initial) : balance(initial) { }
8
9      void credit(double amt)    { balance += amt; }
10     void debit(double amt)     { balance -= amt; }
11     double get_balance() const { return balance; }
12     virtual std::string type() const
13     { return "Account"; }
14 private:
15     double balance;
16 };
17
18 class CheckingAccount : public Account {
19 public:
20     CheckingAccount(double initial, double atm) :
21     Account(initial), total_fees(0.0),
22     atm_fee(atm) { }
23     void cash_withdrawal(double amt) {
24         total_fees += atm_fee;
25         debit(amt + atm_fee);
26     }
27     double get_total_fees() const {
28         return total_fees;
29     }
30
31     std::string type() const {
32         return "CheckingAccount";
33     }
34
35 private:
36     double total_fees;
37     double atm_fee;
38 };
39
40 void print_account_type(const Account& acct) {
41     std::cout << acct.type() << std::endl;
42 }
43
44 int main() {
45     Account acct(1000.0);
46     CheckingAccount checking(1000.0, 2.00);
47     print_account_type(acct);
48     print_account_type(checking);
49     return 0;
50 }
51 }
```

The Account example using `virtual`

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

```
$ ./acc_vt
```

```
Account
```

```
CheckingAccount
```

But how does it work internally in C++?

A brief memory layout of a simple class

```
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;  
};
```

Stack segment
<code>double Account::balance</code>
<code>:</code>

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>:</code>

A brief memory layout of a simple derived class

```
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;
};
```

Stack segment
double Account::balance
CheckingAccount::total_fees
CheckingAccount::atm_fees
⋮
⋮

Code segment
Account::Account()
Account::Account(double)
Account::credit(double)
Account::debit(double)
double Account::get_balance(double)
std::string Account::type()
CheckingAccount::CheckingAccount(double, double)
CheckingAccount::cash_withdrawal(double)
double CheckingAccount::get_total_fees()
std::string CheckingAccount::type()
⋮
⋮

Brief memory layouts of base and derived class

Base class:

Stack segment
<code>double Account::balance</code>
<code>:</code>
<code>:</code>

Derived class:

Stack segment
<code>double Account::balance</code>
<code>CheckingAccount::total_fees</code>
<code>CheckingAccount::atm_fees</code>
<code>:</code>
<code>:</code>

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>:</code>
<code>:</code>

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>CheckingAccount::CheckingAccount(double, double)</code>
<code>CheckingAccount::cash_withdrawal(double)</code>
<code>double CheckingAccount::get_total_fees()</code>
<code>std::string CheckingAccount::type()</code>
<code>:</code>
<code>:</code>

C++ classes: Inheritance - casting (object slicing)

Base class:

Stack segment
<code>double Account::balance</code>
<code>:</code>

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>:</code>

Derived class:

Stack segment
<code>double Account::balance</code>
<code>CheckingAccount::total_fees</code>
<code>CheckingAccount::atm_fees</code>
<code>:</code>

} ignored

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>CheckingAccount::CheckingAccount(double, double)</code>
<code>CheckingAccount::cash_withdrawal(double)</code>
<code>double CheckingAccount::get_total_fees()</code>
<code>std::string CheckingAccount::type()</code>
<code>:</code>

} ignored

- When the compiler lays out a derived object in memory, it puts the data of the base class first
- We can convert from a derived class back to its base
 - The compiler **slices out** the derived class, i.e. ignores the contents of memory past the base data;

A brief memory layout of a class with **virtual** functions

```
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;  
};
```

Stack segment	
double Account::balance	
Account::_vptr -> a virtual table	
type_info Account	
address of std::string Account::type()	
	:

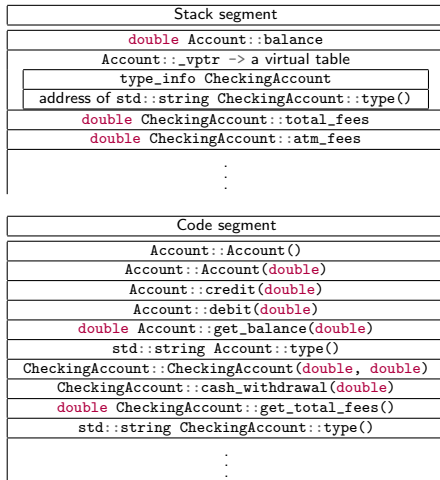
Code segment	
Account::Account()	
Account::Account(double)	
Account::credit(double)	
Account::debit(double)	
double Account::get_balance(double)	
std::string Account::type()	
	:

A brief memory layout of a simple derived class

```
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;
};
```



C++ classes: Inheritance - **virtual** (dynamic dispatch)

Base class:

Stack segment
<code>double Account::balance</code>
<code>Account::_vptr -> a virtual table</code>
<code>type_info Account</code>
<code>address of std::string Account::type()</code>
<code>:</code>
<code>:</code>

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>:</code>
<code>:</code>

Derived class:

Stack segment
<code>double Account::balance</code>
<code>Account::_vptr -> a virtual table</code>
<code>type_info CheckingAccount</code>
<code>address of std::string CheckingAccount::type()</code>
<code>double CheckingAccount::total_fees</code>
<code>double CheckingAccount::atm_fees</code>
<code>:</code>
<code>:</code>
Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>CheckingAccount::CheckingAccount(double, double)</code>
<code>CheckingAccount::cash_withdrawal(double)</code>
<code>double CheckingAccount::get_total_fees()</code>
<code>std::string CheckingAccount::type()</code>
<code>:</code>
<code>:</code>

- Use **virtual** keyword to indicate a method to be **overridden** by the derived class

C++ classes: Inheritance - **virtual** (dynamic dispatch)

What if the derive class has not overridden the virtual function?

// acc_vt2.cpp

```
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      Account() : balance(0.0) { }
7      Account(double initial) : balance(initial) { }
8
9      void credit(double amt)    { balance += amt; }
10     void debit(double amt)     { balance -= amt; }
11     double get_balance() const { return balance; }
12     virtual std::string type() const
13     { return "Account"; }
14 private:
15     double balance;
16 };
17
18 class CheckingAccount : public Account {
19 public:
20     CheckingAccount(double initial, double atm) :
21     Account(initial), total_fees(0.0),
22     atm_fee(atm) { }
23     void cash_withdrawal(double amt) {
24         total_fees += atm_fee;
25         debit(amt + atm_fee);
26
27         double get_total_fees() const {
28             return total_fees;
29         }
30     private:
31         double total_fees;
32         double atm_fee;
33     };
34
35     void print_account_type(const Account& acct) {
36         std::cout << acct.type() << std::endl;
37     }
38
39     int main() {
40         Account acct(1000.0);
41         CheckingAccount checking(1000.0, 2.00);
42         print_account_type(acct);
43         print_account_type(checking);
44         return 0;
45     }
```

C++ classes: Inheritance - **virtual** (dynamic dispatch)

```
$ g++ -o acc_vt2 acc_vt2.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./acc_vt2
```

```
Account
```

```
Account
```

No compilation error!

The virtual table (dynamic dispatch) uses the base class's implementation by default (if the derived class doesn't have one).

C++ classes: Inheritance - **virtual** (dynamic dispatch)

In this case, the memory layouts look like:

Base class:

Stack segment		
<code>double Account::balance</code>		
<code>Account::_vptr -> a virtual table</code>		
<table><tr><td><code>type_info Account</code></td></tr><tr><td><code>address of std::string Account::type()</code></td></tr></table>	<code>type_info Account</code>	<code>address of std::string Account::type()</code>
<code>type_info Account</code>		
<code>address of std::string Account::type()</code>		
<code>:</code>		
<code>:</code>		

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>:</code>
<code>:</code>

Derived class:

Stack segment		
<code>double Account::balance</code>		
<code>Account::_vptr -> a virtual table</code>		
<table><tr><td><code>type_info CheckingAccount</code></td></tr><tr><td><code>address of std::string Account::type()</code></td></tr></table>	<code>type_info CheckingAccount</code>	<code>address of std::string Account::type()</code>
<code>type_info CheckingAccount</code>		
<code>address of std::string Account::type()</code>		
<code>double CheckingAccount::total_fees</code>		
<code>double CheckingAccount::atm_fees</code>		
<code>:</code>		
<code>:</code>		

Code segment
<code>Account::Account()</code>
<code>Account::Account(double)</code>
<code>Account::credit(double)</code>
<code>Account::debit(double)</code>
<code>double Account::get_balance(double)</code>
<code>std::string Account::type()</code>
<code>CheckingAccount::cash_withdrawal(double)</code>
<code>double CheckingAccount::get_total_fees()</code>
<code>:</code>
<code>:</code>

The keyword `override`

Even worse is we believe we have overridden the virtual function:

```
// override.cpp
```

```
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      virtual std::string type() const { return "Account"; }
7  };
8
9  class CheckingAccount : public Account {
10 public:
11     std::string type() { return "CheckingAccount"; }
12 };
13
14 int main() {
15     CheckingAccount checking;
16     Account& acct = checking;
17     std::cout << acct.type() << std::endl; // dynamic dispatch?
18     return 0;
19 }

```

```
$ g++ -o override override.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./override
Account
```


The keyword `override`

In this case, it was just a matter of missing a `const`

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

class CheckingAccount : public Account {
public:
    std::string type() { return "CheckingAccount"; }
    //             ^^ missed const
};
```

The keyword `override`

- This is a typical mistake, often because we:
 - fail to match `const` status
 - fail to exactly match parameters & return types
- The keyword `override` helps
- When you intend to override a function, add the `override` modifier:

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

class CheckingAccount : public Account {
public:
    std::string type() override { return "CheckingAccount"; }
    //          ^^ use override in derived class
};
```

The keyword `override`

```
// override2.cpp
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      virtual std::string type() const { return "Account"; }
7  };
8  class CheckingAccount : public Account {
9  public:
10     std::string type() override { return "CheckingAccount"; }
11 };
12 int main() {
13     CheckingAccount checking;
14     Account& acct = checking;
15     std::cout << acct.type() << std::endl; // dynamic dispatch?
16     return 0;
17 }
```

```
$ g++ -o override2 override2.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
override2.cpp:10:17: error: std::__cxx11::string CheckingAccount::type() marked override, but does not override
    std::string type() override { return "CheckingAccount"; }
                    ~~~~~
```

The keyword `override`

Now we combine it with `const` to fix the problem:

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

The keyword `override`

```
// override3.cpp
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      virtual std::string type() const { return "Account"; }
7  };
8  class CheckingAccount : public Account {
9  public:
10     std::string type() const override { return "CheckingAccount"; }
11 };
12 int main() {
13     CheckingAccount checking;
14     Account& acct = checking;
15     std::cout << acct.type() << std::endl; // dynamic dispatch?
16     return 0;
17 }
$ g++ -o override3 override3.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./override3
CheckingAccount
```

Pass by-value vs by-reference

What happens if we forget to pass by-reference?

// acc_vt3.cpp

```
1  #include <iostream>
2  #include <string>
3
4  class Account {
5  public:
6      Account() : balance(0.0) { }
7      Account(double initial) : balance(initial) { }
8
9      void credit(double amt)    { balance += amt; }
10     void debit(double amt)     { balance -= amt; }
11     double get_balance() const { return balance; }
12     virtual std::string type() const
13     { return "Account"; }
14 private:
15     double balance;
16 };
17
18 class CheckingAccount : public Account {
19 public:
20     CheckingAccount(double initial, double atm) :
21     Account(initial), total_fees(0.0),
22     atm_fee(atm) { }
23     void cash_withdrawal(double amt) {
24         total_fees += atm_fee;
25         debit(amt + atm_fee);
26
27     double get_total_fees() const {
28         return total_fees;
29     }
30
31     std::string type() const override {
32         return "CheckingAccount";
33     }
34
35 private:
36     double total_fees;
37     double atm_fee;
38 };
39
40 void print_account_type(const Account acct) {
41     std::cout << acct.type() << std::endl;
42 }
43
44 int main() {
45     Account acct(1000.0);
46     CheckingAccount checking(1000.0, 2.00);
47     print_account_type(acct);
48     print_account_type(checking);
49     return 0;
50 }
51 }
```

Pass by-value vs by-reference

It won't work! Why?

```
$ g++ -o acc_vt3 acc_vt3.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./acc_vt3
```

```
Account
```

```
Account
```

Recall: when passing by-value, **copy constructor** is called to create a copy of the passing object. The copy constructor takes the passing object as a reference (so object slicing does happen inside the constructor), but the new created object is using the base class memory layout, which means the virtual function is pointing to `Account::type()`.