601.220 Intermediate Programming

Dynamic dispatch

Dynamic dispatch

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

The `Account` example w/o `virtual`

```cpp
// acc.cpp

#include <iostream>
#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;
};


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);
  print_account_type(acct);
  print_account_type(checking);
  return 0;
}
```

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
41   void print_account_type(const Account& acct) {
42     std::cout << acct.type() << std::endl;
43   }
44
45   int main() {
46     Account acct(1000.0);
47     CheckingAccount checking(1000.0, 2.00);
48     print_account_type(acct);
49     print_account_type(checking);
50     return 0;
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

```cpp
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
double Account::balance
$\vdots$

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

A brief memory layout of a simple derived class

```cpp
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
double Account::balance
. . .

Derived class:

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()
. . .

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()
. . .

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

Base class:

Stack segment
double Account::balance
.
.
.

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

Derived class:

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

⎱ ignored

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()
.
.

⎱ ignored

- When the compiler lays out a derived object in memory, it puts the data of the base class first
- We can cast a derived class to its base class
    - 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

```cpp
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

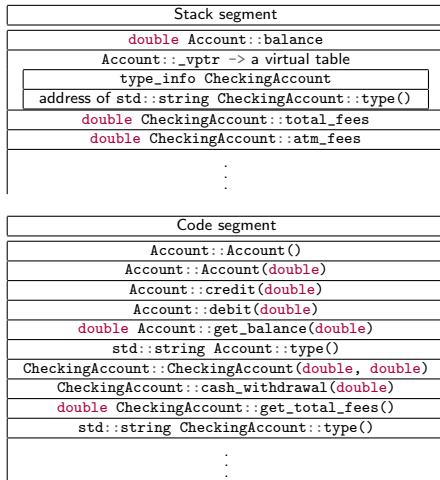| type_info Account |
| --- |
| address of std::string Account::type() |

⋮

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

```cpp
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
Account::_vptr -> a virtual table
type_info CheckingAccount
address of std::string CheckingAccount::type()
double CheckingAccount::total_fees
double 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()
.
.
.

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

Base class:

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()`
.
.

Derived class:

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

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

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

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

```cpp
// 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
31  private:
32      double total_fees;
33      double atm_fee;
34  };
35
36
37  void print_account_type(const Account& acct) {
38      std::cout << acct.type() << std::endl;
39  }
40
41  int main() {
42      Account acct(1000.0);
43      CheckingAccount checking(1000.0, 2.00);
44      print_account_type(acct);
45      print_account_type(checking);
46      return 0;
47  }
```

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
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()`

.
.
.

Derived class:

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

.
.
.

The keyword `override`

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

```cpp
// 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`

```cpp
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:

```cpp
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 ove
    std::string type() override { return "CheckingAccount"; }
                ^~~~
```

The keyword `override`

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

```cpp
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

```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?

```cpp
// acc_vt3.cpp

#include <iostream>
#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);
    }
```

```cpp
    double get_total_fees() const {
        return total_fees;
    }

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

private:
    double total_fees;
    double atm_fee;
};


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);
    print_account_type(acct);
    print_account_type(checking);
    return 0;
}
```

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 a reference of the passing object as its input (so object slicing does happen when calling the constructor), but the newly created object (inside the constructor scope) is using the base class memory layout, which means the virtual function is pointing to Account::type().