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

```
// 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

#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);
  }
  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;
}
```

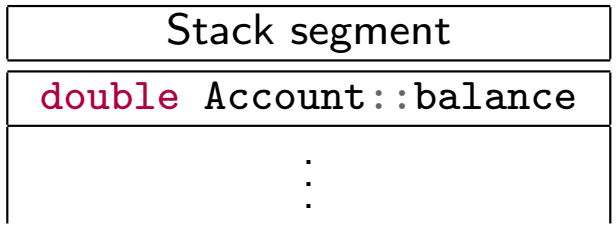
The `Account` example using `virtual`

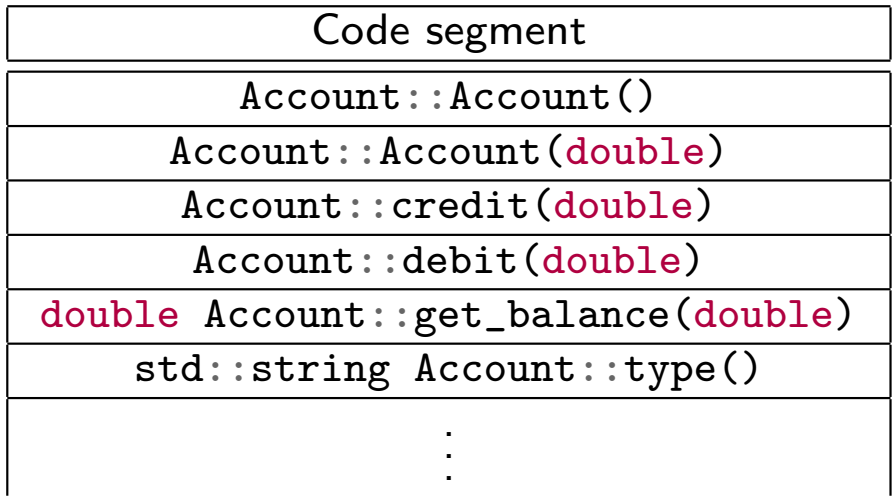
```
$ g++ -o acc_vt acc_vt.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./acc_vt

Account
CheckingAccount
```

But how does it work internally in C++?

A brief memory layout of a simple class

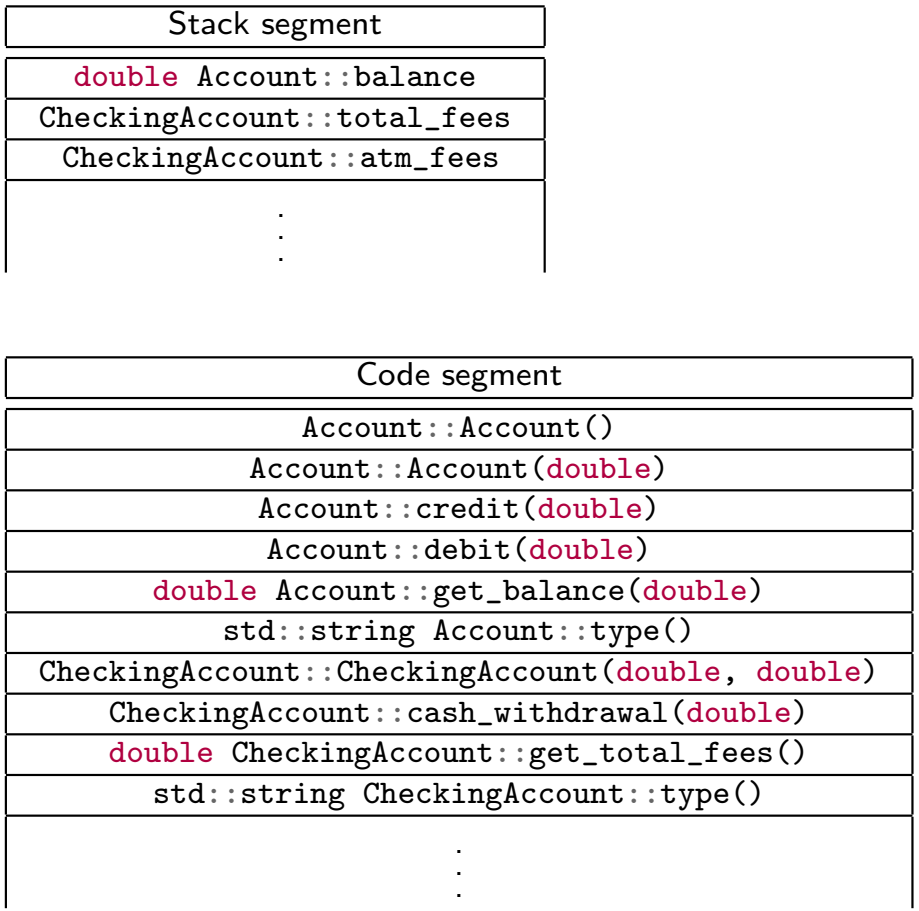
```
class Account {
public:
    Account() : balance(0.0) { }
    Account(double initial)
      : balance(initial) { }

    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;
};
```





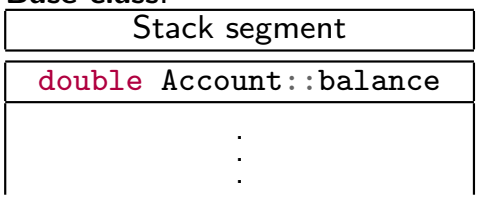
A brief memory layout of a simple derived class

```
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;
};
```

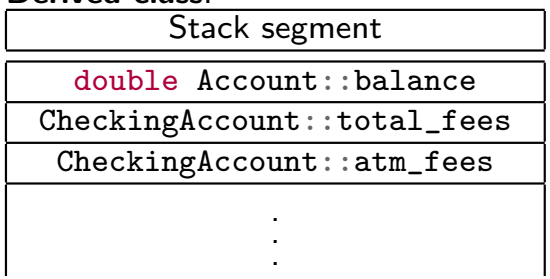


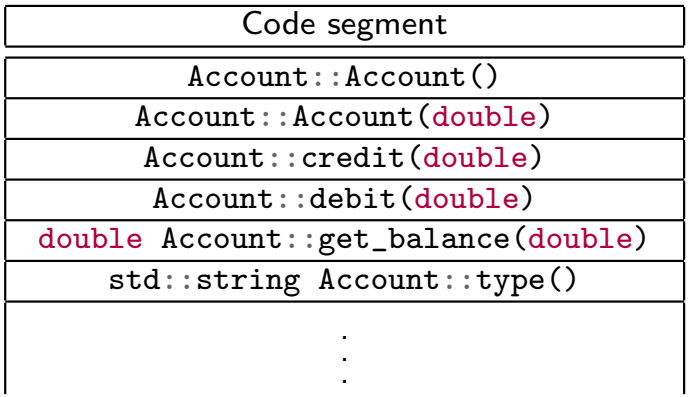
Brief memory layouts of base and derived class

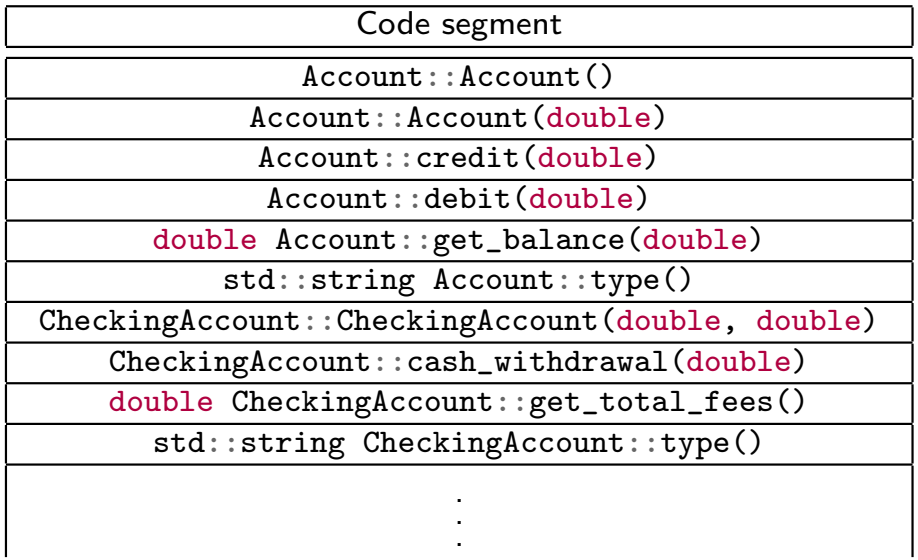
Base class:



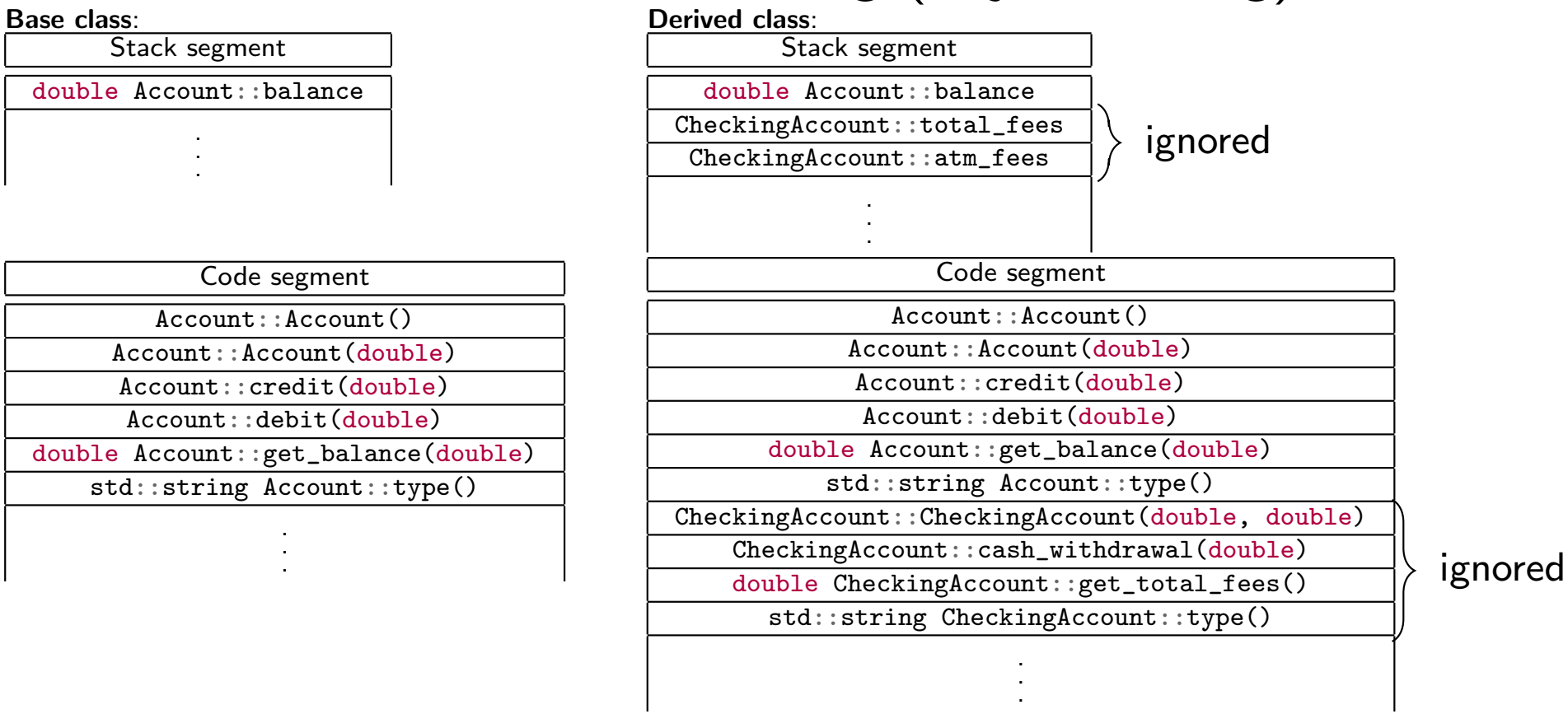
Derived class:







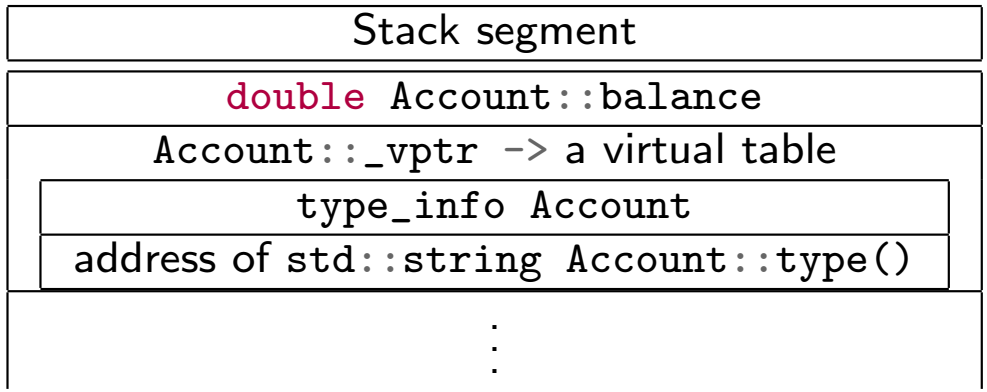
C++ classes: Inheritance - casting (object slicing)

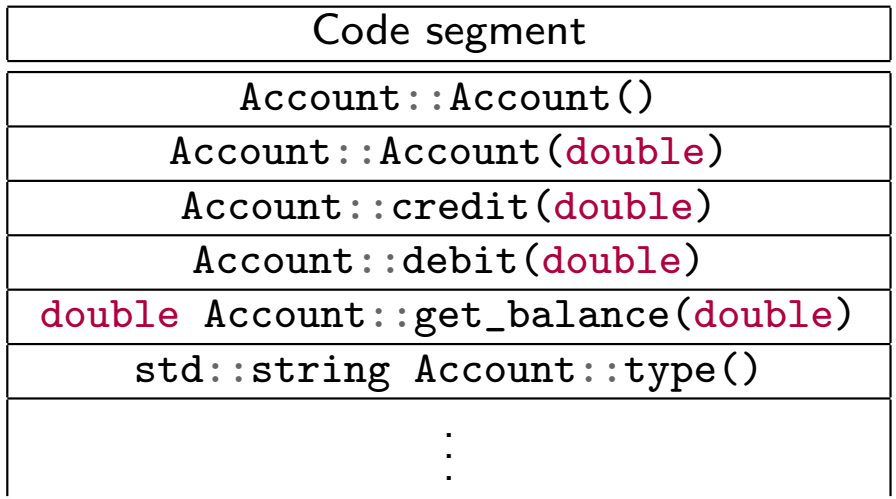


- When the compiler lays out a derived object in memory, it puts the data of the base class first
- We can cast a derived class to its base class
  - The compiler **slices out** the derived class, i.e. ignores the contents of memory past the base data;

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

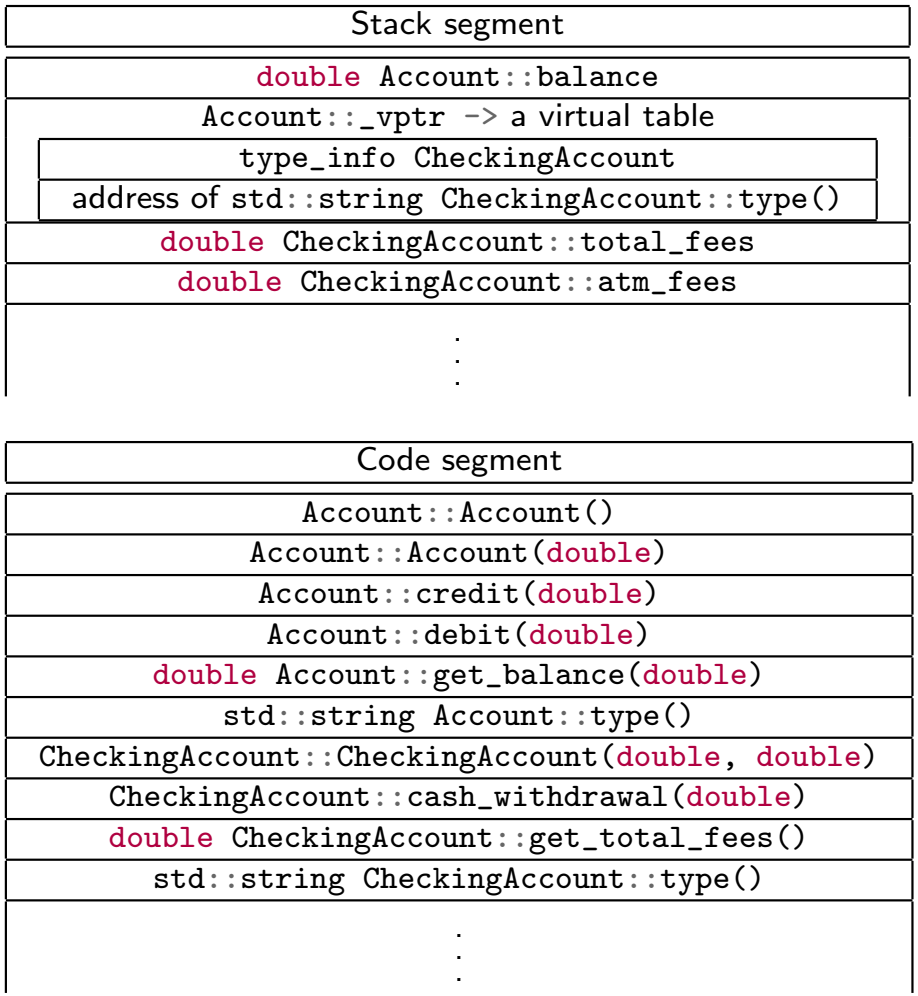
```
class Account {
public:
    Account() : balance(0.0) { }
    Account(double initial)
      : balance(initial) { }

    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;
};
```





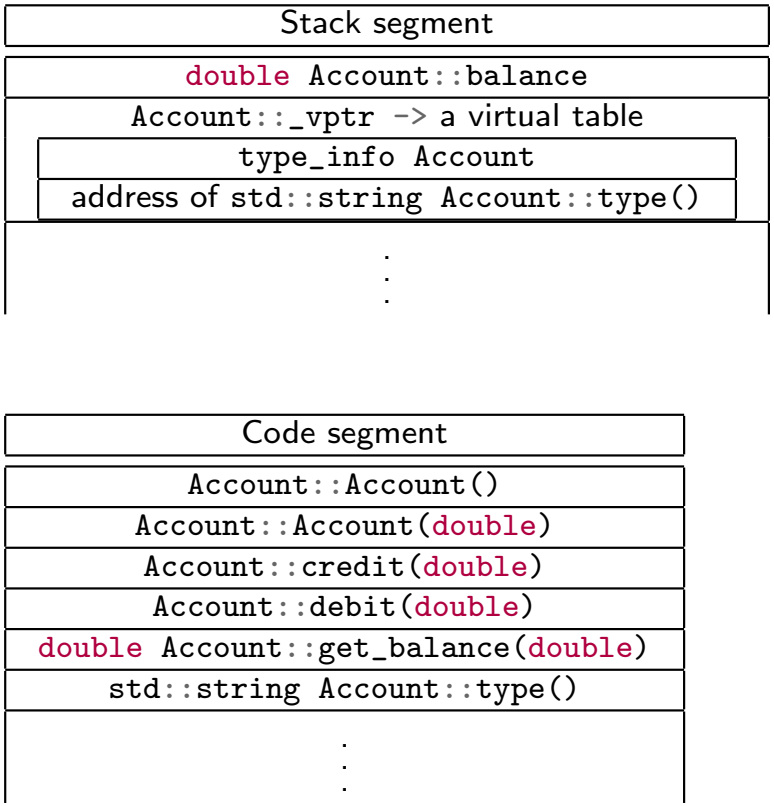
A brief memory layout of a simple derived class

```
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;
};
```

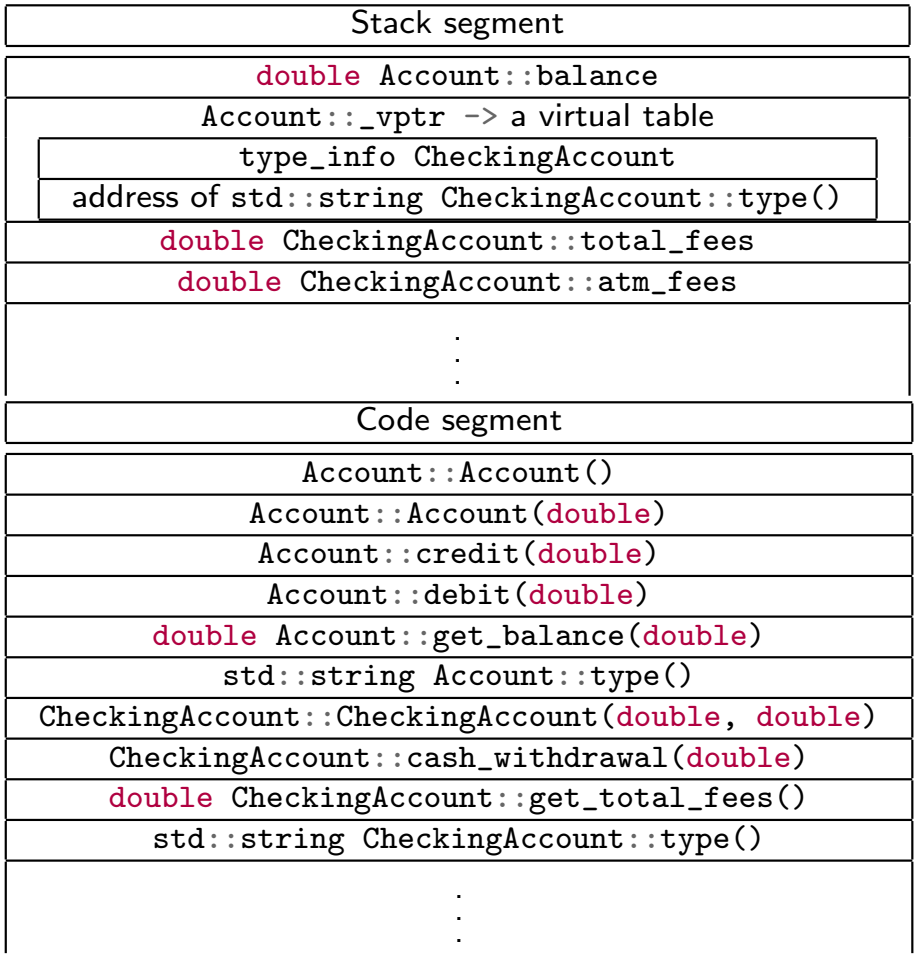


C++ classes: Inheritance - `virtual` (dynamic dispatch)

Base class:



Derived class:



- 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?

```
// acc_vt2.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);
    }
    double get_total_fees() const {
        return total_fees;
    }

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;
}
```

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

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

$ ./acc_vt2

Account
Account
```

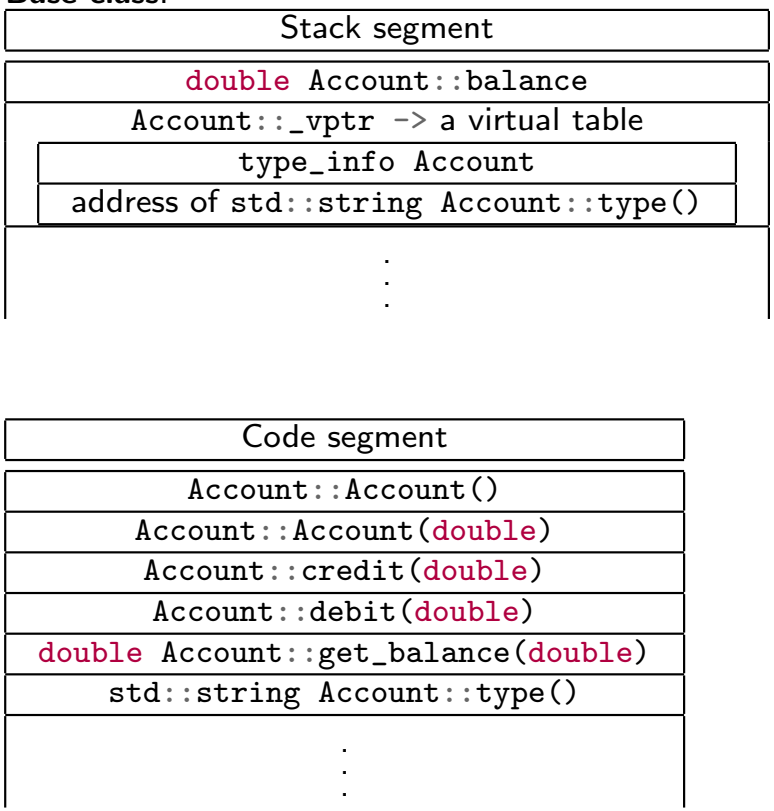
No compilation error!

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

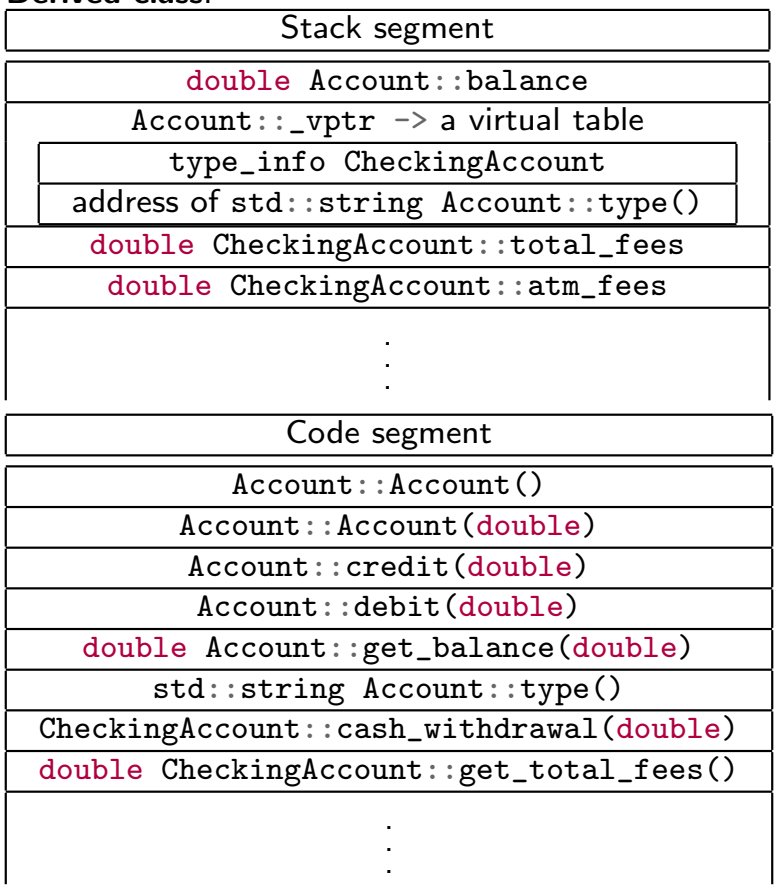
C++ classes: Inheritance - `virtual` (dynamic dispatch)

In this case, the memory layouts look like:

Base class:



Derived class:



The keyword `override`

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

```
// override.cpp

#include <iostream>
#include <string>

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

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

int main() {
    CheckingAccount checking;
    Account& acct = checking;
    std::cout << acct.type() << std::endl; // dynamic dispatch?
    return 0;
}
```

```
$ 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
#include <iostream>
#include <string>

class Account {
public:
    virtual std::string type() const { return "Account"; }
};
class CheckingAccount : public Account {
public:
    std::string type() override { return "CheckingAccount"; }
};
int main() {
    CheckingAccount checking;
    Account& acct = checking;
    std::cout << acct.type() << std::endl; // dynamic dispatch?
    return 0;
}
```

```
$ 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:

```
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
#include <iostream>
#include <string>

class Account {
public:
    virtual std::string type() const { return "Account"; }
};
class CheckingAccount : public Account {
public:
    std::string type() const override { return "CheckingAccount"; }
};
int main() {
    CheckingAccount checking;
    Account& acct = checking;
    std::cout << acct.type() << std::endl; // dynamic dispatch?
    return 0;
}
```

```
$ 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

#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);
    }
    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()`.