

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

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

using std::cout; using std::endl;

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) {
    cout << acct.type() << 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++ -c account1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ g++ -o account1 account1.o
```

```
$ ./account1
```

```
Account
```

```
Account
```

It doesn't work without virtual

The Account example using virtual

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

using std::cout; using std::endl;

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) {
    cout << acct.type() << 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++ -c account1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ g++ -o account1 account1.o
```

```
$ ./account1
```

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
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()
:
:

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
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
:
:

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
:
:

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()
:
:

- 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

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

Base class:

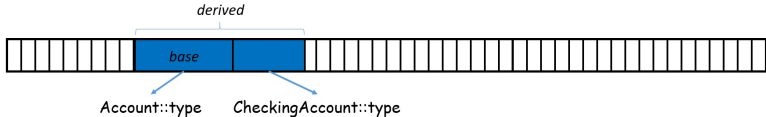
Stack segment
double Account::balance
:
:

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

Derived class:

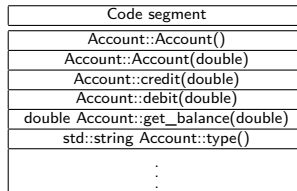
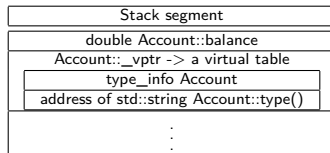
Stack segment
double Account::balance
CheckingAccount::total_fees
CheckingAccount::atm_fees
:
:

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()
:
:



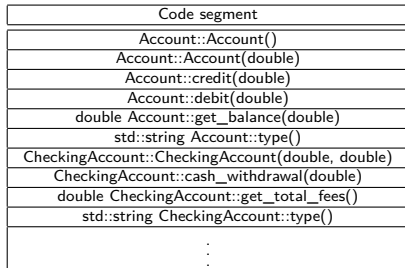
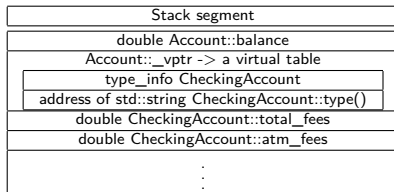
A brief memory layout of a class with virtual functions

```
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:

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 the keyword `virtual` to indicate that a method may be overridden by a derived class

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

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

```
// account3.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++ -c account3.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o account3 account3.o  
$ ./account3
```

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::CheckingAccount(double, double)
CheckingAccount::cash_withdrawal(double)
double CheckingAccount::get_total_fees()
std::string CheckingAccount::type()
:
:

The keyword override

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

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

using std::cout; using std::endl;

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

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

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

The keyword override

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

Sometimes you intend to override a function in the base class...
...but you fail

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:  
    virtual 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 parameter & return types
- The override keyword 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:
    virtual std::string type() override { return "CheckingAccount"; }
    // ~~~ use override in derived class
};
```

The keyword override

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

using std::cout; using std::endl;

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

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

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

```
$ g++ -c override3.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
override3.cpp:14:24: error: 'virtual std::string CheckingAccount::type()' marked 'override', but does not
14 |     virtual 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:  
    virtual std::string type() const override { return "CheckingAccount"; }  
};
```

The keyword override

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

using std::cout; using std::endl;

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

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

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



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


Pass by-value vs by-reference

What happens if we forget to pass by-reference?

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

using std::cout; using std::endl;

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;

void print_account_type(const Account acct) {
    cout << acct.type() << 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++ -c account2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o account2 account2.o  
$ ./account2  
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()`.