Polymorphism

Ben Langmead
ben.langmead@gmail.com
www.langmead-lab.org

Source markdown available at github.com/BenLangmead/c-cpp-notes
Polymorphism

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"; }
    ...
};
#include <iostream>
#include "account2.h"

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

void print_account_type(const Account& acct) {
    cout << acct.type() << 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;
}
Note the types:

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

```cpp
int main() {
    ... 
    CheckingAccount checking(1000.0, 2.00);
    ... 
    print_account_type(checking);
    ... 
}
```
In main, `checking_acct` has type `CheckingAccount`.

Passed to `print_account_type` as `const Account&`; you may use a variable of a derived type as though it has the base type.

- Makes sense: `CheckingAccount` is a `Account`.
int main() {
    vector<Account> my_accounts;

    // this is OK; CheckingAccount is
    // derived from Account
    my_accounts.push_back(CheckingAccount(2000.0));

    cout << my_accounts.back().type() << endl;
    return 0;
}
Polymorphism

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

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

Does `acct.type()` call:

- `Account::type()` – the parameter’s type?
- `CheckingAccount::type()` – checking’s declared type?
Polymorphism

$ g++ -c account_main3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main3 account_main3.o
$ ./account_main3
Account
Account
Account

It calls Account::type()

What if we want the call the member corresponding to the original declared type of the variable (CheckingAccount) rather than the base type we happen to be using for it right now (Account)?

This requires *dynamic binding*

- Declare relevant member functions as *virtual:*
Polymorphism

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"; }
  ...
};
```cpp
#include <iostream>
#include "account3.h" // now with *virtual* `type` member functions

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

void print_account_type(const Account& acct) {
    cout << acct.type() << 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

$ g++ -c account_main4.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main4 account_main4.o
$ ./account_main4

Account
CheckingAccount
SavingsAccount

Dynamic binding / virtual functions enable C++ polymorphism

- In Java & Python all methods are virtual; you have no choice

A class object can look “on the surface” like the base class while behaving like the derived class
# Polymorphism

```cpp
#include <iostream>
#include "account3.h" // still with *virtual* `type` member functions

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

// !!! *** Pass by value this time *** !!!
void print_account_type(Account acct) {
    cout << acct.type() << 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

$ g++ -c account_main5.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main5 account_main5.o
$ ./account_main5
Account
Account
Account
Account

Fields only in the derived class are simply not copied when passed by value using the base type

- This is object slicing

Passing by reference doesn’t cause slicing and preserves our ability to call virtual functions in the derived class; another reason to prefer passing class variables by reference
Polymorphism

https://www.completelydelicious.com/cut-cake-even-layers/
class Base {
public:
    void normal();
    virtual void virt();
};

class Derived : public Base {
public:
    void normal();
    virtual void virt();
};

Say we declare Derived o and later pass o to a function. The member functions called with .normal() and .virt() depend on the parameter type.

<table>
<thead>
<tr>
<th>Param type</th>
<th>a.normal()</th>
<th>a.virt()</th>
<th>What happens?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived&amp; a</td>
<td>Derived::normal()</td>
<td>Derived::virt()</td>
<td>Passed by ref</td>
</tr>
<tr>
<td>Derived a</td>
<td>Derived::normal()</td>
<td>Derived::virt()</td>
<td>Copied, not sliced</td>
</tr>
<tr>
<td>Base&amp; a</td>
<td>Base::normal()</td>
<td>Derived::virt()</td>
<td>Passed by ref</td>
</tr>
<tr>
<td>Base a</td>
<td>Base::normal()</td>
<td>Base::virt()</td>
<td>Sliced &amp; copied</td>
</tr>
</tbody>
</table>
Polymorphism

Same reasoning applies when converting between related types implicitly or using casting:

```cpp
int main() {
    Derived original;

    // sliced; b.virt() -> Base::virt()
    Base b = (Base)original;

    // reference; bref.virt() -> Derived::virt()
    Base& bref = original;

    Derived d = original;  // simple copy
    Derived& dref = original;  // simple reference

    return 0;
}
```
Virtual functions allow a class variable to remember and act like its declared type even when temporarily taking the base type

- ...but how do they remember?

When a class has a virtual member function, it also gets a hidden virtual function table or vtable

- Points to declared-type implementation

Table adds 1 pointer (8 bytes) to size of class, regardless of number of virtual functions
Blue-highlighted items take memory in class variables. Memory is also needed for function code and virtual-function tables (vtbls), but those aren’t per-object, and sizeof won’t include them.
Polymorphism

Sometimes the base class is too generic to be useful as much more than a “guide” for derived classes

class Shape {
public:
    virtual double area() const { } // NO SENSIBLE IMPLEMENTATION FOR THIS
};

class Rectangle : public Shape {
public:
    ...
    virtual double area() const { return width * height; }
    ...
private:
    double width, height;
};

class Circle : public Shape {
public:
    ...
    virtual double area() const { return PI * radius * radius; }
    ...
private:
    double radius;
};
We use pure virtual functions to flag member functions that cannot be usefully implemented in the base class and must be overridden and implemented in a derived class.

```cpp
class Shape {
public:
    virtual double area() const = 0; // pure virtual
};
```

A class with any pure virtual functions is an abstract base class.

- You may not instantiate an abstract base class.
- E.g. Shape s; is not allowed; use derived type(s) instead.
```cpp
#include <iostream>

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

class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) { }
    virtual double area() const { return width * height; }
private:
    double width, height;
};

int main() {
    Shape s;
    Rectangle r = {10.0, 5.0};
    cout << "r area = " << r.area() << endl;
    return 0;
}
```
$ g++ -c shapes.cpp -std=c++11 -pedantic -Wall -Wextra

shapes.cpp: In function 'int main()':
shapes.cpp:19:11: error: cannot declare variable 's' to be of abstract type 'Shape'
    Shape s;
    ^

shapes.cpp:5:7: note: because the following virtual functions are pure within 'Shape':
class Shape {
    ^~~~~~

shapes.cpp:7:20: note: virtual double Shape::area() const
    virtual double area() const = 0;
    ^~~~
#include <iostream>

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

class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) { }
    virtual double area() const { return width * height; }
private:
    double width, height;
};

int main() {
    // Shape s; // **** got rid of this *****
    Rectangle r = {10.0, 5.0};
    cout << "r area = " << r.area() << endl;
    return 0;
}
Polymorphism

$ g++ -c shapes.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o shapes shapes.o
$ ./shapes
r area = 50
Polymorphism

virtual syntax details:

- Omit virtual keyword when defining a virtual member function outside a class definition
- When declaring a virtual member function that overrides one in the parent class, you may omit the virtual keyword
- When you declare a virtual member function that should override one in the base class, use the override keyword
  - Helps catch mistakes where you intended to override but failed to due to a minor difference in function signature