327: Object-oriented programming

Lecture 15

10/27/2021

Professor Barron

const

- constant methods
  - int Loan::calcInterest() const {return loan_value * interest_rate;}
- constant object instances
  - const Loan myloan;
  - only constructor/destructor can modify it
- how can a const method be used?
  - const methods can always be called
  - const method cannot call a non-const method
  - Non-const functions can only be called by non-const objects

const int* const foo(const int* const bar)const;

const

```
1  // constructor on const object
2  #include <iostream>
3  using namespace std;
4
5  class MyClass {
6    public:
7      int x;
8      MyClass(int val) : x(val) {}
9      int get() {return x;}
10 };
11
12 int main() {
13   const MyClass foo(10);
14 // foo.x = 20;              // not valid: x cannot be modified
15   cout << foo.x << '\n';  // ok: data member x can be read
16   return 0;
17 }
```

const

```
1 int get() const {return x;}          // const member function
2 const int& get() {return x;}         // member function returning a const&
3 const int& get() const {return x;} // const member function returning a const&
```

Passing references

```
1  // passing parameters by reference
2  #include <iostream>
3  using namespace std;
4
5  void duplicate1 (int* a, int* b, int* c)
6  {
7    (*a) *= 2;
8    (*b) *= 2;
9    (*c) *= 2;
10 }
11 void duplicate2 (int& a, int& b, int& c)
12 {
13   a *= 2;
14   b *= 2;
15   c *= 2;
16 }
17
18 int main ()
19 {
20   int x=1, y=3, z=7;
21   duplicate1 (&x, &y, &z);
22   cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
23   x=1, y=3, z=7;
24   duplicate2 (x, y, z);
25   cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
26   return 0;
27 }
```

C-style

C++-style

Passing const references

- Ensure that the object passed in isn't mutated by the method

```
1  // overloading operators example
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {};
9      CVector (int a,int b) : x(a), y(b) {}
10     CVector operator + (const CVector&);
11 };
12
13 CVector CVector::operator+ (const CVector& param) {
14   CVector temp;
15   temp.x = x + param.x;
16   temp.y = y + param.y;
17   return temp;
18 }
19
20 int main () {
21   CVector foo (3,1);
22   CVector bar (1,2);
23   CVector result;
24   result = foo + bar;
25   cout << result.x << ',' << result.y << '\n';
26   return 0;
27 }
```

Templates

- Allows the type of a variable in a class to be decided during instantiation
- Same idea as generic types in Java (although done at compile time in C++)
- Unnecessary in Python

```
1  // class templates
2  #include <iostream>
3  using namespace std;
4
5  template <class T>
6  class mypair {
7      T a, b;
8    public:
9      mypair (T first, T second)
10       {a=first; b=second;}
11     T getmax ();
12 };
13
14 template <class T>
15 T mypair<T>::getmax ()
16 {
17   T retval;
18   retval = a>b? a : b;
19   return retval;
20 }
21
22 int main () {
23   mypair <int> myobject (100, 75);
24   cout << myobject.getmax();
25   return 0;
26 }
```

mypair <double> a (5.1, 3.14);

Destructor

- called when an object is deleted
- same name as constructor with ~ added to the beginning
- clean up any dynamically allocated instance variables

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Copy constructors

- used automatically when
  - setting an object equal to another of the same type
  - passing an object as an argument
  - returning (if no move constructor)
- implicit copy constructor used if you don't provide one
  - may or may not be a deep copy
  - similar functionality exists in Python's copy package

```
1  // copy constructor: deep copy
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class Example5 {
7      string* ptr;
8    public:
9      Example5 (const string& str) : ptr(new string(str)) {}
10     ~Example5 () {delete ptr;}
11     // copy constructor:
12     Example5 (const Example5& x) : ptr(new string(x.content())) {}
13     // access content:
14     const string& content() const {return *ptr;}
15 };
16
17 int main () {
18   Example5 foo ("Example");
19   Example5 bar = foo;
20
21   cout << "bar's content: " << bar.content() << '\n';
22   return 0;
23 }
```

Implicit special functions

Example foo = bar;

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

Inheritance

```
1  // derived classes
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b;}
11 };
12
13 class Rectangle: public Polygon {
14   public:
15     int area ()
16       { return width * height; }
17 };
18
19 class Triangle: public Polygon {
20   public:
21     int area ()
22       { return width * height / 2; }
23   };
24
25 int main () {
26   Rectangle rect;
27   Triangle trgl;
28   rect.set_values (4,5);
29   trgl.set_values (4,5);
30   cout << rect.area() << '\n';
31   cout << trgl.area() << '\n';
32   return 0;
33 }
```

20

10

Calling “super” constructor

```
1  // constructors and derived classes
2  #include <iostream>
3  using namespace std;
4
5  class Mother {
6    public:
7      Mother ()
8        { cout << "Mother: no parameters\n"; }
9      Mother (int a)
10       { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14   public:
15     Daughter (int a)
16       { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20   public:
21     Son (int a) : Mother (a)
22       { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26   Daughter kelly(0);
27   Son bud(0);
28
29   return 0;
30 }
```

- There is no super keyword
- For those who like super it is common to do this

```
private:
  typedef Mother super;
```

Mother: no parameters
Daughter: int parameter
Mother: int parameter
Son: int parameter

Polymorphism

- area methods can only be used on pointers of type Rectangle and Triangle

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

ppoly1->area()

Polymorphism

```
1  // virtual members
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b; }
11     virtual int area ()
12       { return 0; }
13 };
14
15 class Rectangle: public Polygon {
16   public:
17     int area ()
18       { return width * height; }
19 };
20
21 class Triangle: public Polygon {
22   public:
23     int area ()
24       { return (width * height / 2); }
25 };
```

```
26
27 int main () {
28   Rectangle rect;
29   Triangle trgl;
30   Polygon poly;
31   Polygon * ppoly1 = &rect;
32   Polygon * ppoly2 = &trgl;
33   Polygon * ppoly3 = &poly;
34   ppoly1->set_values (4,5);
35   ppoly2->set_values (4,5);
36   ppoly3->set_values (4,5);
37   cout << ppoly1->area() << '\n';
38   cout << ppoly2->area() << '\n';
39   cout << ppoly3->area() << '\n';
40   return 0;
41 }
```

20
10
0

C++ inheritance access specifier

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible

```
class Base
{
public:
    int m_public;
protected:
    int m_protected;
private:
    int m_private;
};
```

```
class D2 : private Base
{
public:
    int m_public2;
protected:
    int m_protected2;
private:
    int m_private2;
};
```

```
class D3 : public D2
{
public:
    int m_public3;
protected:
    int m_protected3;
private:
    int m_private3;
};
```

Getting back to design

- So far…
  - Classes
  - Association
  - Composition
  - Inheritance
  - Manager objects

Design patterns (Starting Chapter 9)

- General solutions to commonly occurring problems
- Not finished code, but rather a template
- Applicable to many different languages
  - implementation may look different
  - the pattern might be built into the language like iterator in python

History

- Design patterns became popular in the 1990s
- Software was rapidly becoming more complex
- Design Patterns: Elements of Reusable Object-Oriented Software
  - 1994
  - “Gang of Four”
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Why learn design patterns?

- Re-using patterns can speed up development
- Less likely you need to refactor later
- Make code more readable to others who know the pattern
- Build up a vocabulary to communicate designs with other developers/managers

Categories

- Creational
  - Ways to manage creation of objects
- Structural
  - Organizing classes and objects to provide new functionality
- Behavioral
  - Manage communication between objects
- Concurrency
  - Address challenges in multi-threading/multi-processing

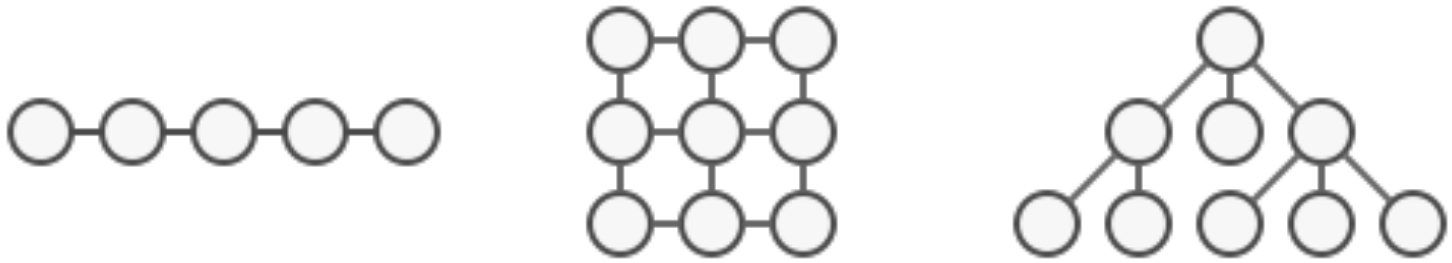
Supplemental resources

- refactoring.guru/design-patterns
- sourcemaking.com/design_patterns

- Modernize GoF book
  - Many descriptions are extremely similar
- More design patterns than our text book
- Nice diagrams
- Code examples for many languages
  - Java, C#, C++, php, Python, Ruby, Swift, Typescript, Go

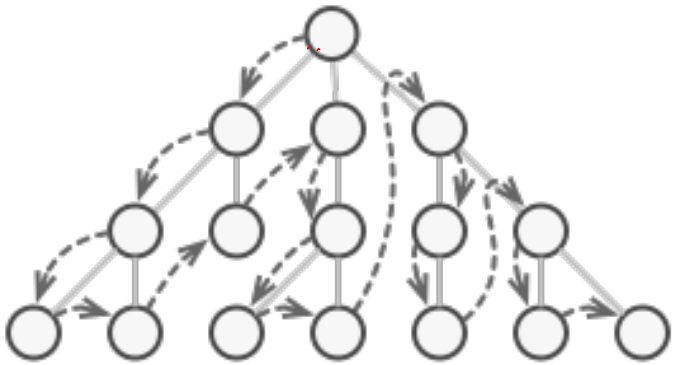
Iterator pattern (Chapter 9)

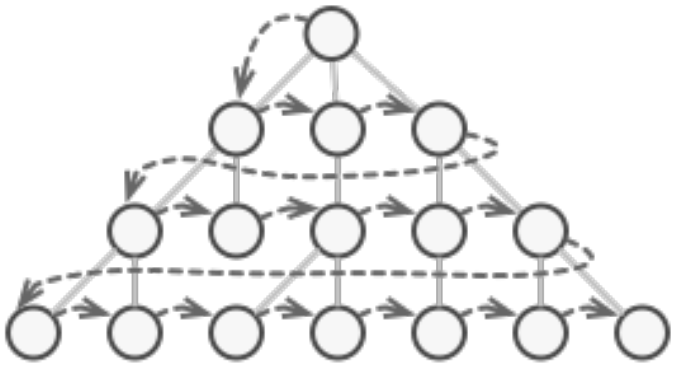
- Behavioral pattern
- An iterator object is created to traverse elements from another object
- Traverse without exposing internal structure/representation
- Add new iterator for new traversal methods without changing the original object's interface

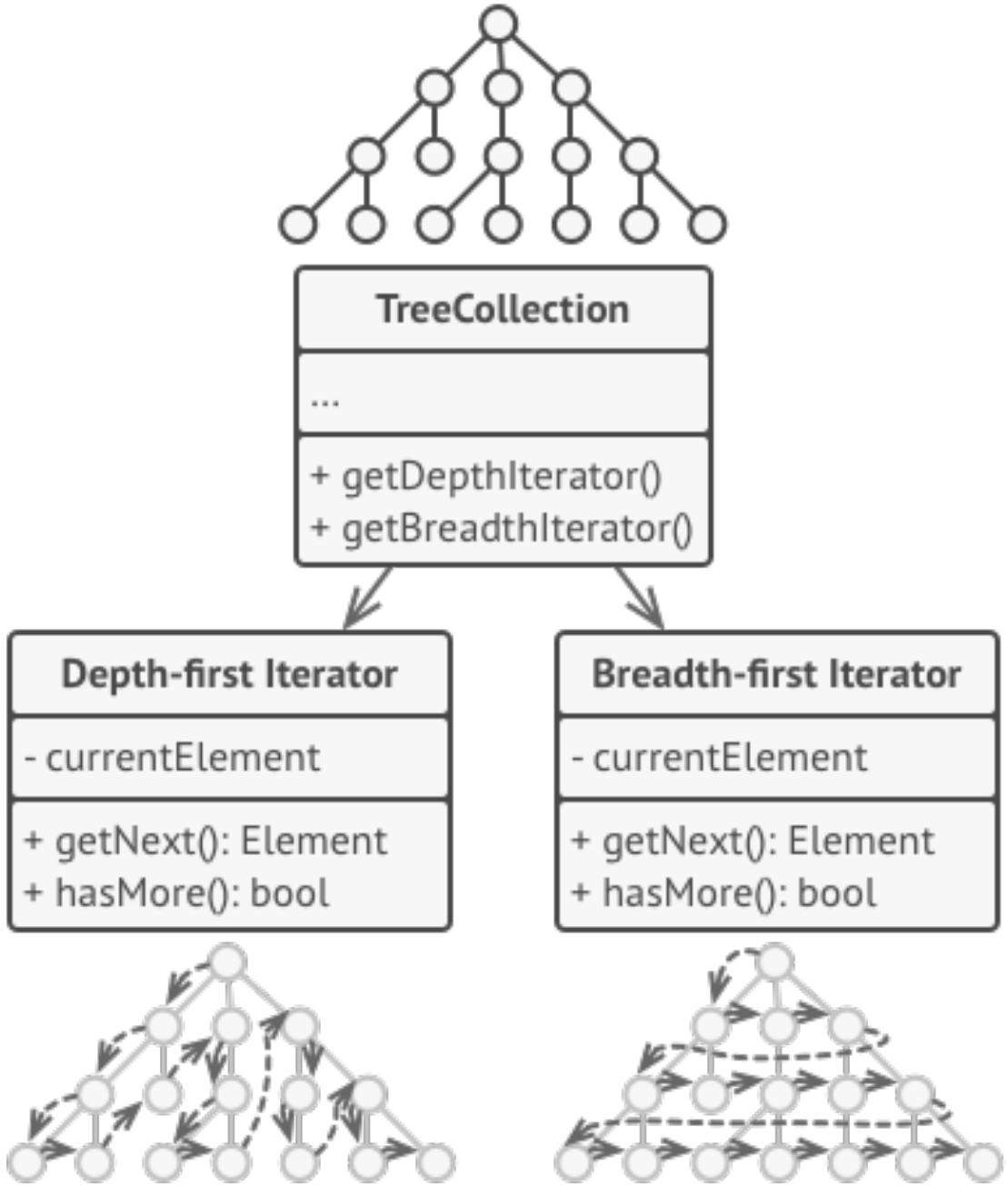


Iterator pattern (Chapter 9)

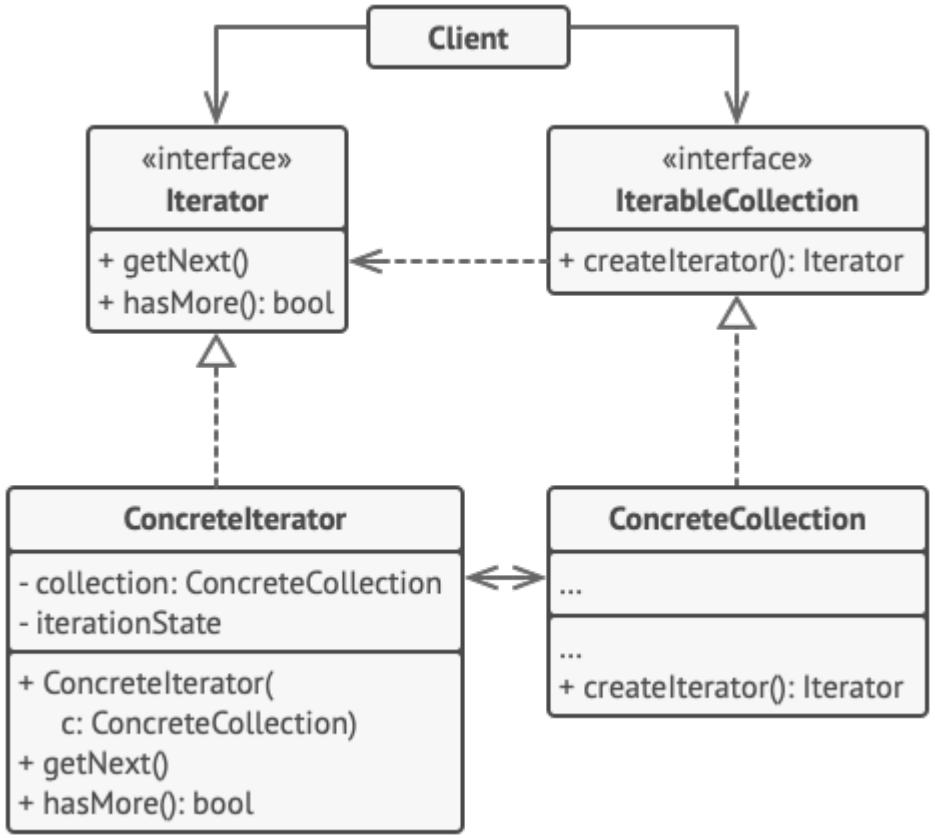
- Behavioral pattern
- An iterator object is created to traverse elements from another object
- Traverse without exposing internal structure/representation
- Add new iterator for new traversal methods without changing the original object's interface






TreeCollection
...
+ getDepthIterator()
+ getBreadthIterator()
Depth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool
Breadth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool

Iterator UML



Iterators

- Iterators are a common object-oriented pattern
- Classic…
  - for (int i=0, i< 10; i++) {
    printf("%d\n", arr[i]);}
- Iteration is controlled externally
- General iterator pattern…
  - while not iterator.done():
    print(iterator.next())
- Iteration controlled internally

for x in Iterator:

class Iterator:

def prev():

def current():

def set_current(e):