Generic programming

How to write programs that don't quite know what they're doing

Overview

- The traditional view of algorithms and data structures ties them closely together
- It doesn't have to be that way
  - especially not for simple algorithms…
  - which turn out to be useful in surprisingly many contexts
- These ideas take different forms in different languages

The fundamental idea

- Instead of designing algorithms to deal with specific data structures, we can design them in terms of abstractions of data structures
- By fitting an appropriate abstraction to each of several data structures, we can make the same algorithm work with several different data structures

What is an abstraction of a data structure?

- A set of operations that are
  - common to several data structures
  - useful for several algorithms
- A way of thinking about data structures in general that does not rely on any specific data structure
- Fundamental example: the idea of a sequence

Abstracting a sequence

- What sequence operations are fundamental?
  - examine each element in turn
  - generate a sequence
- Other operations are less fundamental
  - search for a particular element
  - reverse the elements
  - sort them, etc.

What kinds of sequences are there?

- Arrays
- Lists
- Files are particularly interesting
  - elements might not all be available
  - reading is potentially destructive
  - when you write is important
- What else can you dream up?

Input and output differ

- If we are going to treat files as sequences, the distinction is essential
- Many algorithms also make the distinction
  - copying reads the input, writes the output
  - searching just reads, although whatever requested the search might write later
  - reversing a sequence reads and writes

A sample algorithm

- Search a sequence for a particular value
  - start with the first element
  - keep looking until the element is found or the sequence is exhausted
  - stop as soon as you find what you wanted
- How might we find an abstraction of sequences that will let us implement this algorithm?

What doesn't work

- We can't assume that all sequences will support the same operations
- Therefore, we cannot rely solely on operations defined along with the sequences themselves
- Our abstractions will have to be defined separately

Strategy (classical approach)

- Invent an interface that does what we want
- Declare a base class that captures that interface
- Derive a class for each data structure we care about

A concrete example

- To keep it simple, assume we are reading (not writing) sequences of integers
- What are the key operations?
  - Determine whether there are any elements left in the sequence
  - Fetch the next element in the sequence

An abstract base class

```
class InSeq {
public:
    virtual bool avail() = 0;
    virtual int next() = 0;
    virtual ~InSeq() { }
};
```

- We assume that each call to `next` will be preceded by a call to `avail`

How might we use it?

```
bool find(InSeq& s, int x)
{
    while (s.avail()) {
        if (s.next() == x)
            return true;
    }
    return false;
}
```

Using InSeq

- Suppose we have an integer array called a, with n elements
- How do we determine whether a contains a value equal to x?
  - Derive a class from InSeq that lets us use an array as a sequence
  - Call find with an appropriate object of that derived class

Deriving from InSeq

```
class IntArraySeq: public InSeq {
public:
    IntArraySeq(const int*, int);
    virtual bool avail();
    virtual int next();
private:
    int n;
    const int* p;
};
```

IntArraySeq member definitions

```
IntArraySeq::IntArraySeq
    (const int* p0, int n0):
        p(p0), n(n0) { }
bool IntArraySeq::avail()
{
    return n > 0;
}
int IntArraySeq::next()
{
    --n; return *p++;
}
```

Using IntArraySeq to search an array

```
int a[100];
IntArraySeq s(a, 100);
if (find(s, 42)) {
    // a contains the value 42
}
```

Why does it work?

- Class InSeq has defined a general interface
- Class IntArraySeq has specialized that interface for arrays
- Each time find calls s.avail() or s.next(), that is a virtual call that executes the corresponding IntArraySeq operation

Advantages of this approach

- We have to define only one abstract interface for each overall strategy for accessing sequences
- We can define another derived class from `InSeq` for each kind of sequence we care about
- Each derived class is potentially useful to many algorithms

Disadvantages of this approach

- Each call to `avail()` or `next()` is a virtual call, with associated overhead
- Using an `IntArraySeq` destroys it
- We would like to be able to copy `IntArraySeq` objects
  - We should be able to save an `IntArraySeq` before we destroy it
  - After we find a particular value, we would like to be able to remember where it was
- We don't want to deal just with integers

Overcoming the disadvantages

- The difficulty in copying is peculiar to C++
- The overhead it not, but some languages just live with it
- C++ can solve both problems by using templates, which allow compile-time polymorphism

Templates: overall idea

- Instead of having a single type `InSeq` to represent integer input sequences only, we define a family of types:

```
template <class T> class InSeq {
public:
    virtual bool avail() = 0;
    virtual T next() = 0;
    virtual ~InSeq() { }
};
```

Two kinds of templates

- Class templates: We must supply type arguments every time we use a class template
- Function templates: We generally do not supply type arguments because they are inferred from the function arguments

A simple class template

```
template<class T> class Vector {
public:
    Vector(int n0): n(n0), p(new T[n0])
           { }
    ~Vector() { delete [] p; }
    T& operator[](int k)
        { return p[k]; }
private:
    int n;
    T* p;
};
```

A simple function template

```
template<class T>
T sum(Vector<T>& v, int n)
{
    T result = 0;
    for (int i = 0; i < n; ++i)
        result += v[n];
    return result;
}
```

Using these templates

```
Vector<double> v(100);
for(int i = 0; i < 100; ++i)
    v[i] = i * i;
double s = sum(v, 100);
```

We could define `find` this way...

```
bool find(InSeq<int>& s, int x)
{
    while (s.avail()) {
        if (s.next() == x)
            return true;
    }
    return false;
}
```

But it's more useful to define it this way:

```
template<class X>
bool find(InSeq<X>& s, X x)
{
    while (s.avail()) {
        if (s.next() == x)
            return true;
    }
    return false;
}
```

We now declare ArraySeq as a generalization...

```
template<class T>
class ArraySeq: public InSeq<T> {
public:
    ArraySeq(const T*, int);
    virtual bool avail();
    virtual T next();
private:
    int n;
    const T* p;
};
```

... and define it this way:

```
template<class T> ArraySeq<T>::ArraySeq
    (const T* p0, int n0):
        p(p0), n(n0) { }
template<class T>
bool ArraySeq<T>::valid() {
    return n > 0;
}
template<class T> T ArraySeq<T>::next()
{
    --n; return *p++;
}
```

Now we can use `find` almost as before:

```
int a[100];
ArraySeq<int> s(a, 100);
if (find(s, 42)) {
    // a contains the value 42
}
```

Where are we now?

- We can define an `ArraySeq` class for an array of objects of any type
- We can derive other classes from `InSeq` for other containers
- But we still have the virtual-function overhead for each call
- Moreover, it's still unclear what the right `InSeq` interface is

What is an `InSeq`, really?

- A separate object that grants access to a data structure
- An `InSeq` captures the idea of stepping through an (unknown) data structure
- We can therefore call `InSeq` (and similar classes) *iterators*
- The next lecture will look at other forms of iterators

Homework (due Monday)

- Derive another class from `InSeq` that reads input from a file, rather than from an array
- Use that class to compute the sum of a sequence of floating-point values read from the standard input
- It's OK to use an object-oriented language other than C++ if you like