

Another object-oriented program

Prime numbers the hard way
(based on a program by Ravi Sethi in
his book *Programming Languages—
Concepts and Constructs*)

The object of this program

- Compute prime numbers, by
 - generating integers
 - discarding the ones that aren't prime
- Do so in an object-oriented style
 - a chain of objects, each of which discards multiples of a particular integer
 - the chain grows dynamically for each prime number computed

The data structure

Generate integers

Discard multiples of 2

Discard multiples of 3

Discard multiples of 5

Discard multiples of 7

Prime numbers!

...

What classes do we need?

- Classes that can act as a source of integers
 - Counters generate consecutive integers
 - Filters discard multiples of an integer
- A wrapper class to do the prime-number computation

We can already declare...

```
class Source { /* ... */ };  
class Counter: public Source  
    { /* ... */ };  
class Filter: public Source  
    { /* ... */ };  
class Sieve: public Source  
    { /* ... */ };
```

... but we can do better

- A `Filter` gets numbers from a `Source`
- So does a `Sieve`
- Therefore, we can define an auxiliary class to represent the idea of “a class that gets numbers from a `Source`”
- We will call that class `Conduit`

The revised hierarchy

```
class Source { /* ... */ };
class Counter: public Source
    { /* ... */ };
class Conduit: public Source
    { /* ... */ };
class Filter: public Conduit
    { /* ... */ };
class Sieve: public Conduit
    { /* ... */ };
```

What is a Source?

- You poke at it and get a number back
- Other classes will be derived from it

```
class Source {  
public:  
    virtual int next() = 0;  
    virtual ~Source() { }  
    Source() { }  
private:  
    Source(const Source&);  
    Source& operator=(const Source&);  
};
```


Declaration of Counter

- Again, the declaration follows from the requirements

```
class Counter: public Source {  
public:  
    Counter(int);  
    virtual int next();  
  
private:  
    int n;  
};
```

Definition of Counter

```
Counter::Counter(int n0): n(n0) { }  
int Counter::next()  
{  
    return n++;  
}
```

We can already use our Counter class

```
int main()
{
    Counter c(1);
    int n;
    do {
        n = c.next();
        cout << n << endl;
    } while (n < 10);
    return 0;
}
```

What does a Conduit do?

- It takes input from a Source
- It delivers output on demand through the next function
- It lets you find its Source
- It gives you a way to change the Source to be somewhere else
- It manages memory

Memory management

- As in the last lecture, we will assume that if you give away a pointer to an object, you also delegate responsibility for deleting that object
- Class `Sieve` will hide memory details from users
- So it is sufficient for a `Conduit` to delete its `Source` when destroyed

Declaration of Conduit

```
Class Conduit: public Source {  
public:  
    Conduit(Source*);  
    virtual ~Conduit();  
protected:  
    Source* source();  
    void splice(Source*);  
private:  
    Source* src;  
};
```

Definition of Conduit

```
Conduit::Conduit(Source* s):  
    src(s) { }  
Conduit::~~Conduit() { delete src; }  
Source* source() { return src; }  
void Conduit::splice(Source* s)  
{  
    src = s;  
}
```

Class Filter

- A `Filter` accepts numbers from a `Source` and screens out multiples of a given integer
- Fundamental operations:
 - Construct a `Filter` from a given integer and `Source`
 - Fetch an integer

Filter declaration

```
class Filter: public Conduit {  
public:  
    Filter(int, Source*);  
    virtual int next();  
private:  
    int factor;  
};
```

How does a Filter work?

- Obviously, it must remember its source (Conduit does that) and what to filter
- The next function does the actual screening

Definition of Filter

```
Filter::Filter(int f, Source* s):  
    Conduit(s), factor(f) { }  
int Filter::next()  
{  
    int n;  
    do n = source()->next();  
    while (n % factor == 0);  
    return n;  
}
```

What should a Sieve do?

- Actual prime number computation
 - Start a Counter at 2
 - Each time we get back a number, create a new Filter to screen out multiples of that number
- Clean interface to the rest of the world
 - Conceal the other classes
 - Memory management

Using a Sieve

```
int main()
{
    Sieve s;
    int n;
    do {
        n = s.next();
        cout << n << endl;
    } while (n < 100);
    return 0;
}
```

Declaring class Sieve

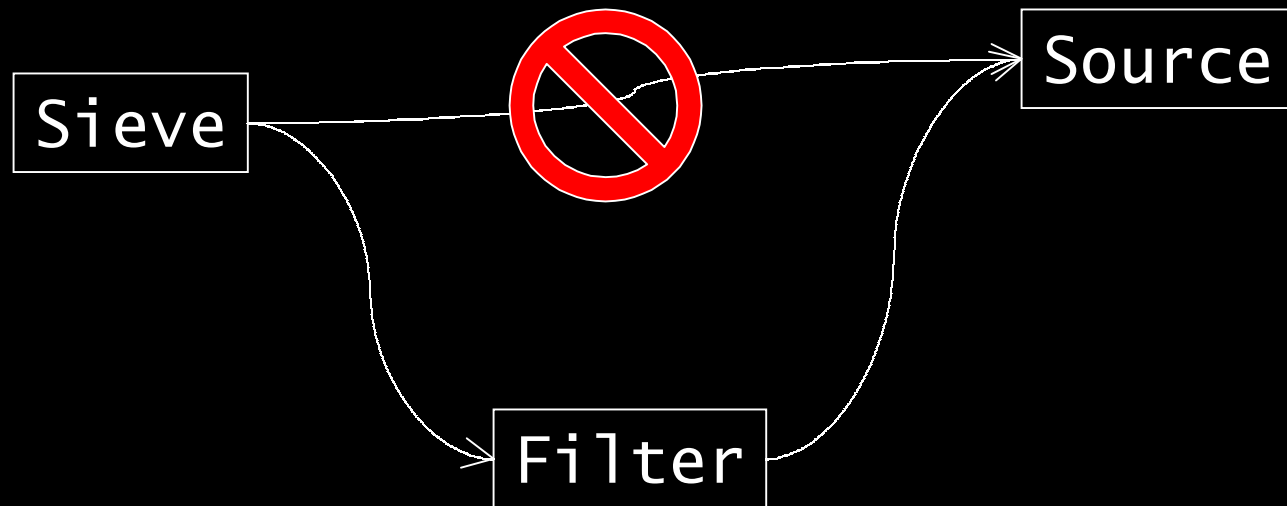
```
class Sieve: public Conduit {  
public:  
    Sieve();  
    virtual int next();  
};
```

Defining class Sieve

- The constructor is easy:

```
Sieve::Sieve():  
    Conduit(new Counter(2)) { }
```
- But what about the next function?
 - It calls `source()->next()`, which yields the next prime
 - Then it has to splice in a new `Filter`

The data structure



Definition of Sieve::next

```
int Sieve::next()
{
    int n = source()->next();
    splice(new Filter(n, source()));
    return n;
}
```

Observations

- A `Sieve` turns out to act like a `Conduit`, so it simplifies the code to use `Conduit` as a base class
- Class `Sieve` does not need an explicit destructor, because class `Conduit` takes care of it
- Class `Sieve` is the only one intended for end-user consumption

More observations

- Not an optimal algorithm
- Recursive deletes could be improved
- Nevertheless, the idea of growing a data structure to represent an increasingly complicated computation is an important one
- In effect, we've built an interpreter for a tiny, special-purpose language

Understanding object-oriented programs

- Following the whole program at once can be tricky
- One useful strategy
 - Understand the whole program approximately
 - Understand each piece and its immediate context
 - Walk through it for some test cases

Homework (due Monday)

- Add a member to class Expr:

```
class Expr {  
public:  
    int eval() const;  
    // ...  
};
```

that returns the value corresponding to an expression

- Revise your simulation to include a corresponding member