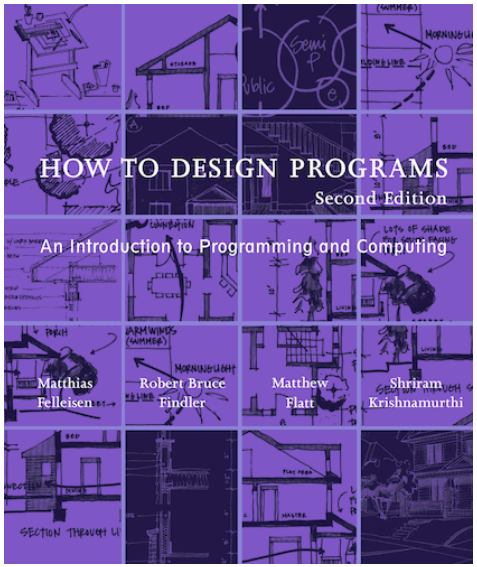
How to Design Programs

using C++



http://www.htdp.org

How to Design Programs

Key idea: **data drives design**

- Return the number of cookies left after eating one

  ⇒

- Keep track of the number of cookies in a cookie jar

How to Design Programs

Key idea: **data drives design**

- Calculate the value of an arithmetic expression

  ⇒

- Represent and manipulate expressions

The Design Recipe

- Determine the **representation**
  - classes, if needed
- Write a **purpose** statement
- Write **examples**
  - as tests
- Create a **template** for the implementation
  - method, if variants
  - extract field values, if any
  - cross- and self-calls, if data references
- Finish **body** implementation case-by-case
- Run **tests**

Representation

- Keep track of the number of cookies in a cookie jar

Functional view: take and return cookie count

Question is **not**

Where do I keep the cookie count?

Question is

What datatype lets me represent a cookie count?

Representation

- Keep track of the number of cookies in a cookie jar

```
int
```

```
int eat_cookie(int n);
```

Purpose

```
// To return the number of cookies
// left after eating one
int eat_cookie(int n);
```

Examples

```
// To return the number of cookies
// left after eating one
int eat_cookie(int n);
```

Examples are tests

```
CHECK( eat_cookie(10) == 9 );
CHECK( eat_cookie(1) == 0 );
CHECK( eat_cookie(0) == 0 );
CHECK( eat_cookie(-1) ==   );
```

Examples

```
// To return the number of cookies
// left after eating one; given number
// of cookies must be non-negative
int eat_cookie(int n);
```

Examples are tests

```
CHECK( eat_cookie(10) == 9 );
CHECK( eat_cookie(1) == 0 );
CHECK( eat_cookie(0) == 0 );
```

Examples may refine purpose and suggest **contracts**

Template

```
// To return the number of cookies
// left after eating one; given number
// of cookies must be non-negative
int eat_cookie(int n) {
   .... n ....
}
```

Template step lists data available in body

Body

```
// To return the number of cookies
// left after eating one; given number
// of cookies must be non-negative
int eat_cookie(int n) {
  .... n ....
}
```

```
CHECK( eat_cookie(10) == 9 );
CHECK( eat_cookie(1) == 0 );
CHECK( eat_cookie(0) == 0 );
```

Body

```
// To return the number of cookies
// left after eating one; given number
// of cookies must be non-negative
int eat_cookie(int n) {
  if (n > 0)
    return n - 1;
  else
    return 0;
}
```

```
CHECK( eat_cookie(10) == 9 );
CHECK( eat_cookie(1) == 0 );
CHECK( eat_cookie(0) == 0 );
```

Test

```
// To return the number of cookies
// left after eating one; given number
// of cookies must be non-negative
int eat_cookie(int n) {
  if (n > 0)
    return n - 1;
  else
    return 0;
}
```

```
CHECK( eat_cookie(10) == 9 );
CHECK( eat_cookie(1) == 0 );
CHECK( eat_cookie(0) == 0 );
```

Representation

- Track a position on the screen

Multiple components ⇒ new class

```
class Posn {
  int x;
  int y;
};
```

Representation

- Track a position on the screen

Multiple components ⇒ new class

```
class Posn {
  int x;
  int y;

  Posn *flip() {
    ....
  }
};
```

Purpose

```
class Posn {
  int x;
  int y;

  // To return the opposite
  // posn over the diagonal
  Posn *flip() {
    ....
  }
};
```

Examples

```
class Posn {
  int x;
  int y;

  // To return the opposite
  // posn over the diagonal
  Posn *flip() {
    ....
  }
};
```

```
CHECK( (new Posn(1, 17))->flip()->equals(new Posn(17, 1)) );
CHECK( (new Posn(-3, 4))->flip()->equals(new Posn(4, -3)) );
```

Template

```
class Posn {
  int x;
  int y;

  // To return the opposite
  // posn over the diagonal
  Posn *flip() {
    .... x .... y ....
  }
};
```

Body

```
class Posn {
  int x;
  int y;

  // To return the opposite
  // posn over the diagonal
  Posn *flip() {
    return new Posn(y, x);
  }
};
```

```
CHECK( (new Posn(1, 17))->flip()->equals(new Posn(17, 1)) );
CHECK( (new Posn(-3, 4))->flip()->equals(new Posn(4, -3)) );
```

Representation

- Track an ant, which has a location and a weight

Complex component ⇒ use other class

```
class Ant {
  Posn *loc;
  double weight;

  bool is_at_home() {
     ....
  }
};
```

Purpose

```
class Ant {
  Posn *loc;
  double weight;

  bool is_at_home() {
     ....
  }
};
```

Purpose

```
class Ant {
  Posn *loc;
  double weight;

  // To determine whether the
  // ant is home (at origin)
  bool is_at_home() {
     ....
  }
};
```

Examples

```
class Ant {
  Posn *loc;
  double weight;

  // To determine whether the
  // ant is home (at origin)
  bool is_at_home() {
    ....
  }
};
```

```
CHECK( (new Ant(new Posn(0, 0), 0.0001))->is_at_home() == true );
CHECK( (new Ant(new Posn(5, 10), 0.0001))->is_at_home() == false );
```

Template

```
class Ant {
  Posn *loc;
  double weight;

  // To determine whether the
  // ant is home (at origin)
  bool is_at_home() {
    .... loc->is_home() .... weight ....
  }
};
```

```
class Posn {
  int x;
  int y;

  // Represents origin?
  bool is_home() {
    .... x .... y ....
  }
};
```

Body

```
class Ant {
  Posn *loc;
  double weight;

  // To determine whether the
  // ant is home (at origin)
  bool is_at_home() {
    return loc->is_home();
  }
};
```

```
class Posn {
  int x;
  int y;

  // Represents origin?
  bool is_home() {
    return (x == 0) && (y == 0);
  }
};
```

```
CHECK( (new Ant(new Posn(0, 0), 0.0001))->is_at_home() == true );
CHECK( (new Ant(new Posn(5, 10), 0.0001))->is_at_home() == false );
CHECK( (new Posn(0, 0))->at_home() == true );
CHECK( (new Posn(-1, 3))->at_home() == false );
```

Representation

- Track an animal, which is a tiger or a snake

When data has variants, need a base class and subclasses

```
class Animal {
  bool is_heavy() = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool is_heavy() {
    ....
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool is_heavy() {
    ....
  }
};
```

Purpose

```
class Animal {
  // Move using more than one person?
  bool is_heavy() = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool is_heavy() {
    ....
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool is_heavy() {
    ....
  }
};
```

Purpose is the same for all implementations of the method

Examples

```
class Animal {
  // Move using more than one person?
  bool is_heavy() = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool is_heavy() {
    ....
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool is_heavy() {
    ....
  }
};
```

```
CHECK( (new Tiger("orange", 14))->is_heavy() == true );
CHECK( (new Snake("green", 10, "rats"))->is_heavy() == true );
CHECK( (new Snake("yellow", 8, "cake"))->is_heavy() == false );
```

Template

```
class Animal {
  // Move using more than one person?
  bool is_heavy() = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool is_heavy() {
    .... color .... stripe_count ....
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool is_heavy() {
    .... color .... weight .... food ....
  }
};
```

Body

```
class Animal {
  // Move using more than one person?
  bool is_heavy() = 0;
};
```

```
class Tiger : public Animal {
  std::string color;
  int stripe_count;

  bool is_heavy() {
    return true;
  }
};
```

```
class Snake : public Animal {
  std::string color;
  int weight;
  std::string food;

  bool is_heavy() {
    return weight >= 10;
  }
};
```

```
CHECK( (new Tiger("orange", 14))->is_heavy() == true );
CHECK( (new Snake("green", 10, "rats"))->is_heavy() == true );
CHECK( (new Snake("yellow", 8, "cake"))->is_heavy() == false );
```

Representation

- Track an aquarium, which has any number of fish, each with a weight

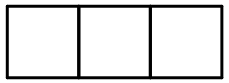
How many components is that?

Objects as Boxes

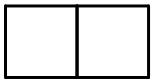
`class` lets us define a new kind of box

The box can have as many compartments as we want, but we have to pick how many, once and for all

```
class Snake {
  std::string color;
  int weight;
  std::string food;
}
```

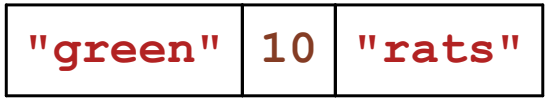
⇒ 

```
class Ant {
  Posn *loc;
  double weight;
}
```

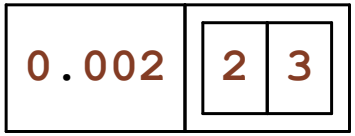
⇒ 

Boxes Stretch

The boxes stretch to fit any one thing in each slot:



Even other boxes:



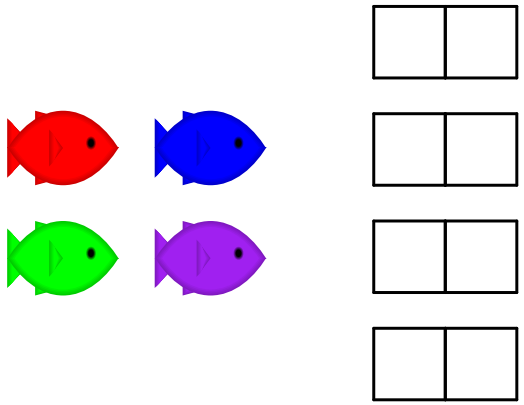
Still, the number of slots is fixed

Packing Boxes

Suppose that

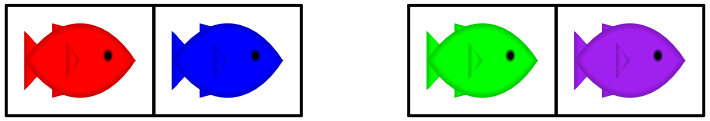
- You have four things to pack as one
- You only have 2-slot boxes
- Every slot must contain exactly one thing

How can you create a single package?



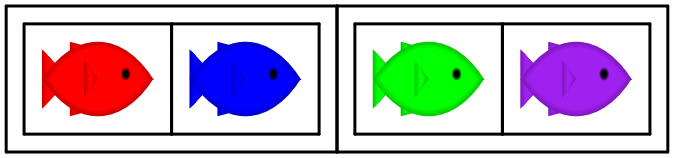
Packing Boxes

This isn't good enough



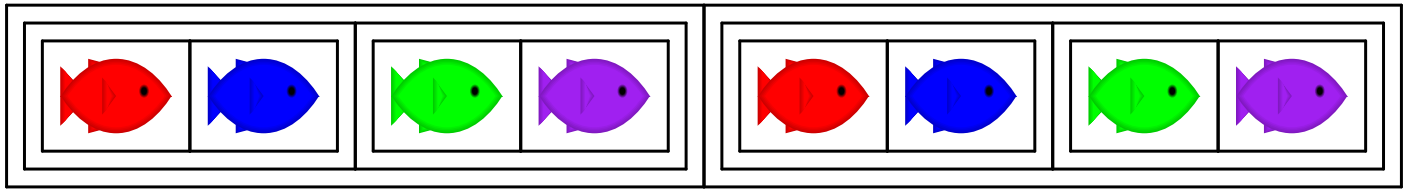
because it's still two boxes...

But this works!

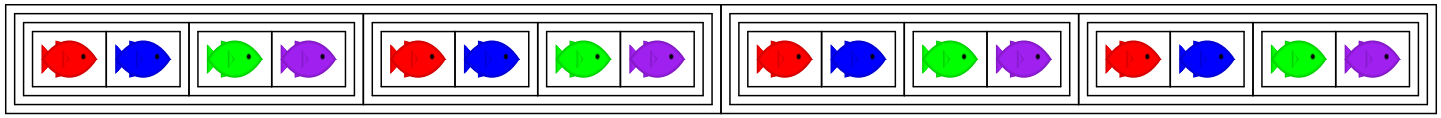


Packing Boxes

And here's 8 fish:



And here's 16 fish!



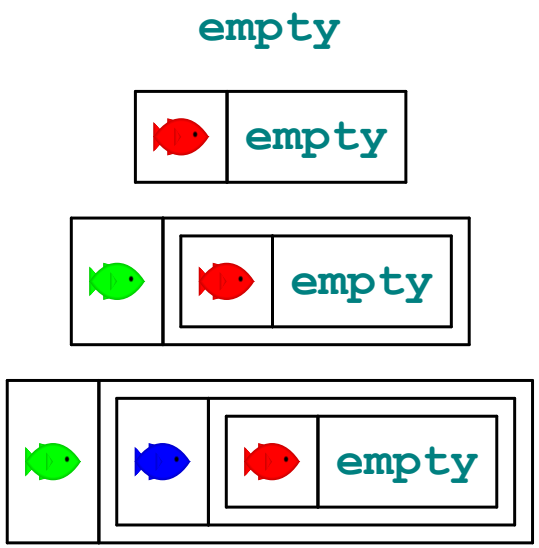
But what if we just add 1 fish, instead of doubling the fish?

But what if we have 0 fish?

General Strategy for Packing Boxes

- For 0 fish, use `empty`
- If you have a package and a new fish, put them together

So, to combine many fish, start with `empty` and add fish one at a time



General Strategy for a List of Numbers

In code:

- `empty = new EmptyAq()`
- `new BiggerAq(`*fish*`,` *aq*`)`

```
                    empty

              new BiggerAq(10, empty)

        new BiggerAq(5, new BiggerAq(10, empty))

new BiggerAq(7, new BiggerAq(5, new BiggerAq(10, empty)))
```

Representation

- Track an aquarium, which has any number of fish, each with a weight

```
class Aq {
  virtual Aq *feed_fish() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual Aq *feed_fish() {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual Aq *feed_fish() {
    ....
  }
};
```

Purpose

```
class Aq {
  // Return aquarium after each fish
  // is fed 1 pound of food
  virtual Aq *feed_fish() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual Aq *feed_fish() {
     ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual Aq *feed_fish() {
     ....
  }
};
```

Examples

```
class Aq {
  // Return aquarium after each fish
  // is fed 1 pound of food
  virtual Aq *feed_fish() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual Aq *feed_fish() {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual Aq *feed_fish() {
    ....
  }
};
```

```
CHECK( (new EmptyAq())->feed_fish()->equals(new EmptyAq()) );
CHECK( (new BiggerAq(2, new EmptyAq()))->feed_fish()
       ->equals(new BiggerAq(3, new EmptyAq()) ));
```

Template

```
class Aq {
  // Return aquarium after each fish
  // is fed 1 pound of food
  virtual Aq *feed_fish() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual Aq *feed_fish() {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual Aq *feed_fish() {
    .... fish
    .... rest->feed_fish() ....
  }
};
```

cruical part of the template!

Body

```
class Aq {
  // Return aquarium after each fish
  // is fed 1 pound of food
  virtual Aq *feed_fish() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual Aq *feed_fish() {
    return new EmptyAq();
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual Aq *feed_fish() {
    return new BiggerAq(fish+1,
                        rest->feed_fish());
  }
};
```

```
CHECK( (new EmptyAq())->feed_fish()->equals(new EmptyAq()) );
CHECK( (new BiggerAq(2, new EmptyAq()))->feed_fish()
       ->equals(new BiggerAq(3, new EmptyAq()) ));
```

Total Aquarium Weight

```
class Aq {
  // Total weight of fish in aquarium
  virtual int total_weight() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual int total_weight() {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual int total_weight() {
    ....
  }
};
```

```
CHECK( (new EmptyAq())->total_weight() == 0 );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->total_weight() == 5 );
```

Total Aquarium Weight

```
class Aq {
  // Total weight of fish in aquarium
  virtual int total_weight() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual int total_weight() {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual int total_weight() {
    .... fish
    .... rest->total_weight() ....
  }
};
```

```
CHECK( (new EmptyAq())->total_weight() == 0 );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->total_weight() == 5 );
```

Total Aquarium Weight

```
class Aq {
  // Total weight of fish in aquarium
  virtual int total_weight() = 0;
};
```

```
class EmptyAq : public Aq {
  virtual int total_weight() {
    return 0;
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual int total_weight() {
    return fish
           + rest->total_weight();
  }
};
```

```
CHECK( (new EmptyAq())->total_weight() == 0 );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->total_weight() == 5 );
```

Finding Fish

```
class Aq {
  // Determine whether any fish has given size
  virtual bool has_size(int sz) = 0;
};
```

```
class EmptyAq : public Aq {
  virtual bool has_size(int sz) {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual bool has_size(int sz) {
    ....
  }
};
```

```
CHECK( (new EmptyAq())->has_size(3) == false );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(3) == true );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(4) == false );
```

Finding Fish

```
class Aq {
  // Determine whether any fish has given size
  virtual bool has_size(int sz) = 0;
};
```

```
class EmptyAq : public Aq {
  virtual bool has_size(int sz) {
    ....
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual bool has_size(int sz) {
    .... fish
    .... rest->has_size(sz) ....
  }
};
```

```
CHECK( (new EmptyAq())->has_size(3) == false );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(3) == true );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(4) == false );
```

Finding Fish

```
class Aq {
  // Determine whether any fish has given size
  virtual bool has_size(int sz) = 0;
};
```

```
class EmptyAq : public Aq {
  virtual bool has_size(int sz) {
    return false;
  }
};
```

```
class BiggerAq : public Aq {
  int fish;
  Aq *rest;

  virtual bool has_size(int sz) {
    return (fish == sz)
           || rest->has_size(sz);
  }
};
```

```
CHECK( (new EmptyAq())->has_size(3) == false );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(3) == true );
CHECK( (new BiggerAq(2, new BiggerAq(3, new EmptyAq())))->has_size(4) == false );
```

Expressions

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ + ⟨expr⟩
| ⟨expr⟩ * ⟨expr⟩

```
class Expr {
  virtual bool equals(Expr *e) = 0;
};
```

```
class Num : public Expr {
  int val;

  bool equals(Expr *e) {
    .... val ....
  }
};
```

```
class Add : public Expr {
  Expr *lhs;
  Expr *rhs;

  bool equals(Expr *e) {
    .... lhs->equals(e) ...
    .... rhs->equals(e) ....
  }
};
```

```
class Mult : public Expr {
  Expr *lhs;
  Expr *rhs;

  bool equals(Expr *e) {
    .... lhs->equals(e) ...
    .... rhs->equals(e) ....
  }
};
```