Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Templated Containers and Containers of Pointers

Containers

- *Containers* like `IntSet` and `IntList` are abstract data types (ADTs) that contain other objects

- So far, our `IntList` container holds only integers

- Today, we will use the C++ `template` mechanism to reuse the same container code for any type

Changing types

- How would we change our `IntList` to hold the `char` type?

```
class IntList {
 public:
    void push_front(int v);
    int & front();
    //...
 private:
  struct Node {
    Node *next;
    int   datum;
  };
  Node *front_ptr;
};
```

```
class CharList {
 public:
    void push_front(char v);
    char & front();
    //...
 private:
  struct Node {
    Node *next;
    char datum;
  };
  Node *front_ptr;
};
```

Changing types

- All we have to do is replace each `int` with `char`
- The C++ `template` mechanism does this automatically

```
class IntList {
 public:
    void push_front(int v);
    int & front();
    //...
 private:
  struct Node {
    Node *next;
    int   datum;
  };
  Node *front_ptr;
};
```

```
class CharList {
 public:
    void push_front(char v);
    char & front();
    //...
 private:
  struct Node {
    Node *next;
    char datum;
  };
  Node *front_ptr;
};
```

Static polymorphism

- This is an example of *static polymorphism*
- *static*: at compile time
- *polymorphism*: different behavior for different types

```
class IntList {
 public:
    void push_front(int v);
    int & front();
    //...
 private:
  struct Node {
    Node *next;
    int   datum;
  };
  Node *front_ptr;
};
```

```
class CharList {
 public:
    void push_front(char v);
    char & front();
    //...
 private:
  struct Node {
    Node *next;
    char datum;
  };
  Node *front_ptr;
};
```

Using templates

- By the end of the lecture you will be able to do this:

```
int main() {
  List<int> intlist;          //use a list of int
  intlist.push_front(3);
  intlist.push_front(2);
  intlist.push_front(1);
  intlist.print();            //1 2 3

  List<char> charlist;        //use a list of char
  charlist.push_front('c');
  charlist.push_front('b');
  charlist.push_front('a');
  charlist.print();           //a b c
}
```

Homogeneous containers

- Note that these containers still *homogenous*
- One container variable can hold only one type

```
int main() {
  List<int> intlist;
  //...
  List<char> charlist;
  //...
}
```

Template notation

```
template <typename T>
class List {
  // ...
};
```

T is the type contained by this List. It's like a variable name, and some programmers like to use value_type instead of T.

```
template <class T>
class List {
  // ...
};
```

Alternate notation

Template declaration

```
template <typename T>
class List {
public:
  bool empty() const;
  T & front() const;
  void push_front(T datum);
  void pop_front();
  void print() const;
  //back(), push_back(), etc. omitted for brevity

  List();
  List(const List &other);
  ~List();
  List &operator=(const List &rhs);
private:
  // ...
};
```

Next, declare `List` using `T` instead of `int`

Template declaration

- Now, let's add the private members

```
template <typename T>
class List {
  //...
private:
  struct Node {
    Node *next;
    T datum;
  };

  void push_all(const List &other);
  void pop_all();
  Node *front_ptr;
  Node *back_ptr;
};
```

`Node` type is only available to member functions

`Node` will hold only objects of type `T`

Template member functions

- Now, we will define the member functions
- Each function begins with the template declaration:

```
template <typename T>
```

- And each method name must be in the `List<T>` scope:

```
template <typename T>
bool List<T>::empty() const {
  return front_ptr == 0;
}
```

Template member functions

Before

```
bool IntList::empty()
const {
  return front_ptr == 0;
}
```

After

```
template <typename T>
bool List<T>::empty()
const {
  return front_ptr == 0;
}
```

Template member functions

Before

```
void IntList::push_front
(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```

After

```
template <typename T>
void List<T>::push_front
(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```

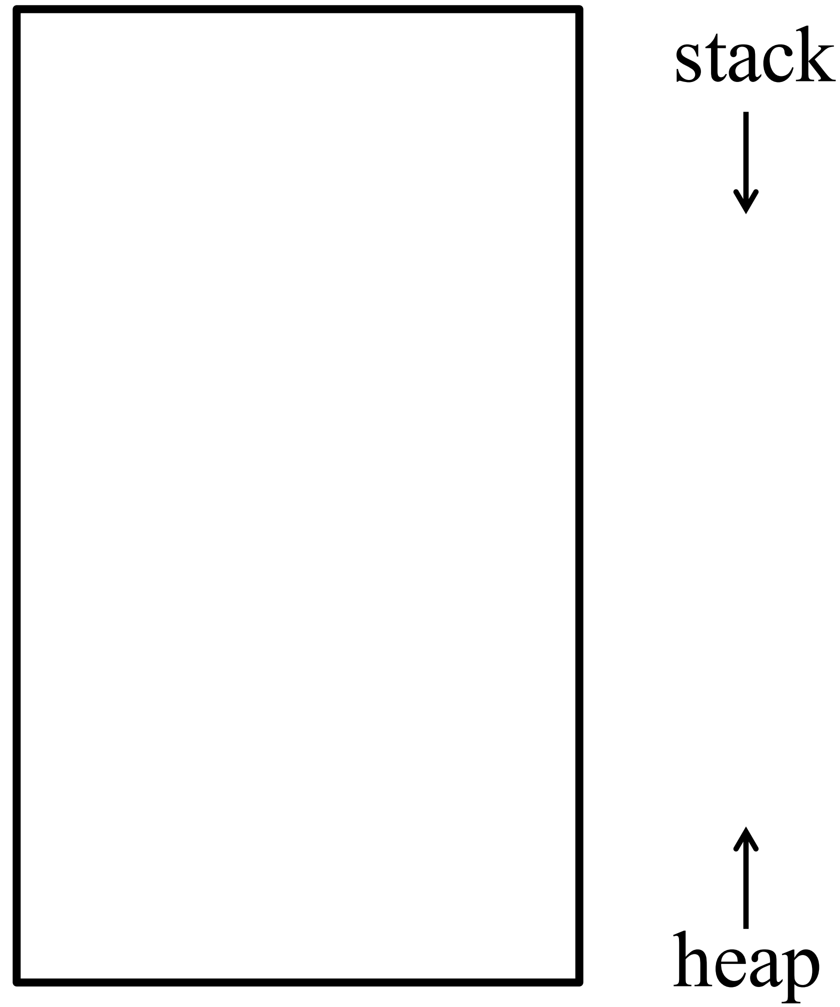
The argument, `datum`, is of type `T` which is exactly the same type as `p->datum` (convenient, huh?)

Using templates

- Draw the stack and the heap

```
int main() {
  List<int> intlist;
  intlist.push_front(3);
  intlist.push_front(2);
  intlist.push_front(1);
  intlist.print();

  List<char> charlist;
  charlist.push_front('c');
  charlist.push_front('b');
  charlist.push_front('a');
  charlist.print();
}
```



Using templates

- Each time the compiler sees a different type `T`, it copies the class, and substitutes the type (e.g., `int`) for `T`

```
int main() {
  List<int> intlist1;
  //compiler copies List, compiles with T=int

  List<char> charlist;
  //compiler copies List, compiles with T=char

  List<int> intlist2;
  //compiler reuses List compiled with T=int
}
```

Template code goes in the .h file

- To change `T` to `int`, the compiler needs to change the member function prototypes:

```
template <typename T>
class List {
 void push_front(~~T~~ int datum);
 //...
};
```

- The compiler also needs to change the function bodies:

```
template <typename T>
void List<T>::push_front(~~T~~ int datum) {
  Node *p = new Node; //"int" version of Node
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```

```
struct Node {
  Node *next;
  T datum;
};
```

Template code goes in the .h file

- Thus, the compiler needs to see both the declarations (prototypes) and definitions (implementations) of `List` to compile this:

```
#include "List.h"
int main() {
  List<int> intlist;
}
```

- This means we need to put both the class declaration (prototype) *and* definition (implementation) in `List.h` file

Include guards

- What happens if we "accidentally" `#include List.h` multiple times?

```
//Graph.h
#include "List.h"
class Graph {
  List<int> vertices;
  //...
};
```

```
//main.cpp
#include "List.h"
#include "Graph.h"
int main() {
  List<int> mylist;
  Graph mygraph;
  //...
}
```

```
$ g++ main.cpp
List.h:20: error: redefinition of 'class List<T>'
```

Include guards

- We use *include guards* to protect against redefinition
- Ensure that `List` code only appears once ☺
- Two ways to do it:

```
#ifndef LIST_H
#define LIST_H
//List.h

template <typename T>
class List {
  //...
};
//implementation ...
#endif
```

```
#pragma once
//List.h

template <typename T>
class List {
  //...
};
//implementation ...
```

Containers of values

- We can now use our container with any type, including custom types created with the class mechanism
- Let's try it with Gorillas

```
class Gorilla {
  // OVERVIEW: a big, expensive class ...
};
int main() {
  List<Gorilla> zoo;
  zoo.push_front(Gorilla());
}
```

Gorilla: a big object

- Let's make our Gorilla object more interesting
- Gorillas have names

```
class Gorilla {
  // OVERVIEW: a big, expensive class ...
  string name;
public:
  string get_name() const { return name; }
};
```

Gorilla: a big object

- Add a default constructor
- Add a construct to set the name

```
class Gorilla {
  // ...
  Gorilla() : name("noname") {
    cout << "Gorilla default ctor\n”;
  }
  Gorilla(string name_in) : name(name_in) {
    cout << "Gorilla ctor: " << name << "\n";
  }
};
```

Gorilla: a big object

- To practice the Big Three, let's add print messages

```
class Gorilla {
  //...
  ~Gorilla() {
    cout << "Gorilla dtor: " << name << "\n";
  }
  Gorilla(const Gorilla &other) {
    name = other.name + " clone";
    cout << "Gorilla copy ctor: " << name << "\n";
  }
  Gorilla & operator=(const Gorilla &rhs) {
    name = other.name + " clone";
    cout << "Gorilla operator=: " << name << "\n";
    return *this;
  }
};
```

Exercise

```
void List<T>::push_front(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```

```
struct Node {
  Node *next;
  int   datum;
};
```

```
class Gorilla {
  // OVERVIEW: a big, expensive class
  // ...
};
List<Gorilla> zoo;
zoo.push_front(Gorilla("Colo"));
```

What is the output of this code?

Two copies

```
void List<T>::push_front(T datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = front_ptr;
  front_ptr = p;
}
```

```
struct Node {
  Node *next;
  int   datum;
};
```

```
int main() {
  List<Gorilla> zoo;
  zoo.push_front(Gorilla("Colo"));
}
```

```
void List<T>::push_front(T datum) { //pass by value
  Node *p = new Node;
  p->datum = datum;                 //another copy!
  p->next = front_ptr;
  front_ptr = p;
}
```

Two copies

```
int main() {
  List<Gorilla> zoo;
  zoo.push_front(Gorilla("Colo"));
}

void List<T>::push_front(const T &datum) {//no copy ☺
  Node *p = new Node;
  p->datum = datum;                  //still copied ☹
  p->next = front_ptr;
  front_ptr = p;
}
```

- The first copy is easy to fix, just pass by reference
- The second, not so much

Containers of pointers

- We can fix this with a container of pointers

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
}
```

What is the output of this code?

Draw picture of the objects in memory

Containers of pointers: motivation

- Avoid slow copies of big objects
- Want unique objects
- If you want to manage the lifetime of the variable yourself
  - Not local, not global, but something in between
- Many parts of the code refer to an object, but only want one copy in memory. Normally, if many parts refer to the same object, those parts will be in different scopes.

Containers of pointers example

- Let's expand our zoo example
- Two Gorillas live in this zoo, Colo and Koko

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  //...
}
```

Containers of pointers example

- Francine the zoo keeper feeds the animals each day. In the morning she makes a list of all the animals.

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
}
```

Draw a picture of this in memory

Containers of pointers example

- Francine says hello to each animal as she feeds it, and then removes it from her todo list

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;

  // code this using a loop
  // Example: “Hi, Colo”


}
```

Containers of pointers example

- Francine says hello to each animal as she feeds it, and then removes it from her todo list

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  while (!todo.empty()) {
    Gorilla *g = todo.front();
    cout << "Hi, " << g->get_name() << "\n";
    todo.pop_front();
  }
}
```

Problems with containers of pointers

- What's wrong with this code? Fix it.

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  while (!todo.empty()) {
    Gorilla *g = todo.front();
    cout << "Hi, " << g->get_name() << "\n";
    todo.pop_front();
  }
} //HINT: what happens here?
```

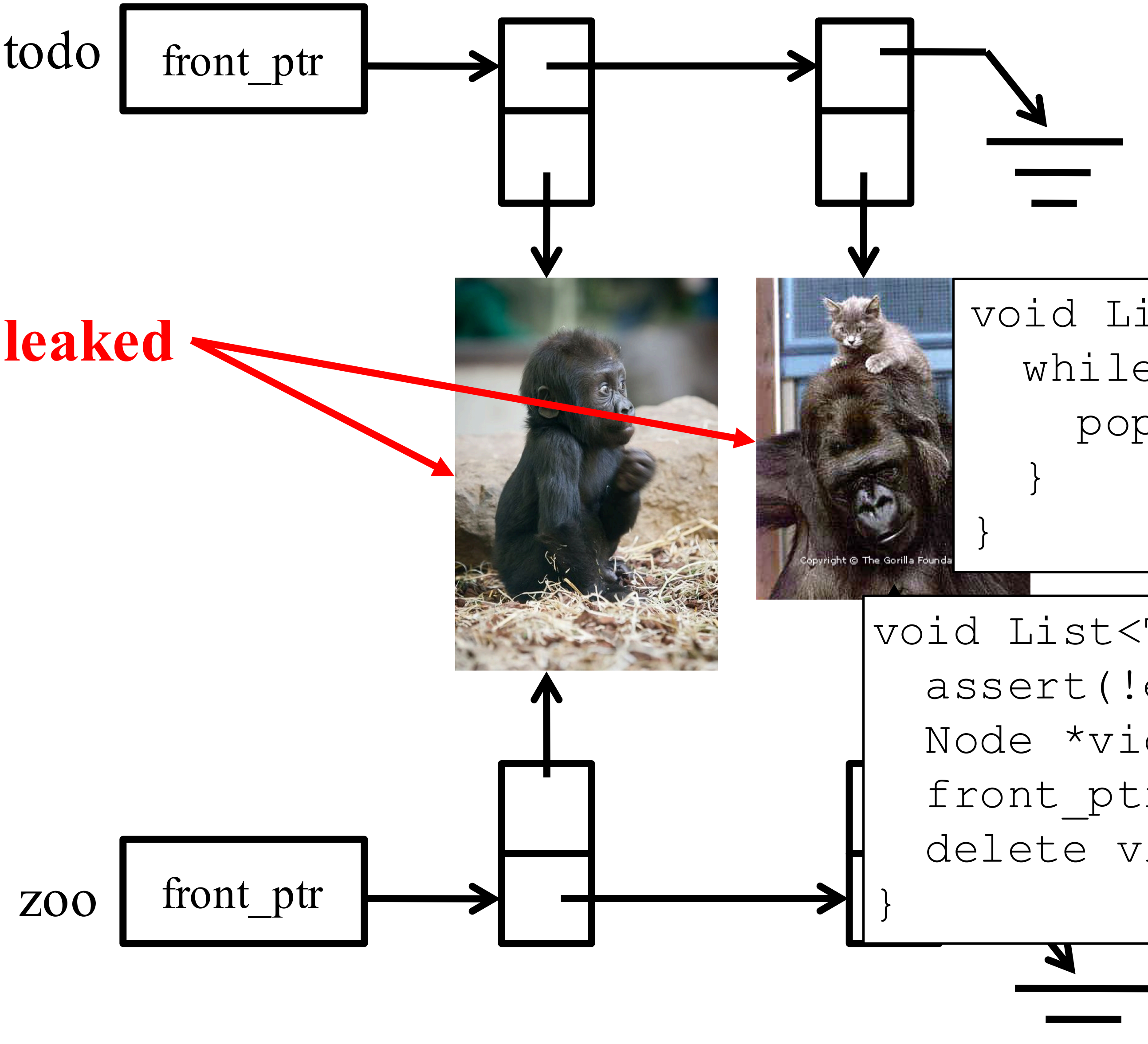
Problems with containers of pointers

- Problem: destructor removes the `Node` objects, but not the `Gorilla` objects
- You are responsible for managing dynamic memory

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  while (!todo.empty()) {
    Gorilla *g = todo.front();
    cout << "Hi, " <<              ame() << "\n";
    todo.pop_front();
  }
} ~List() is called
```

Orphan `Gorilla` on the heap!

Problems with containers of pointers




```
List<t>::~List() {
  pop_all();
}
```

```
void List<T>s::pop_all() {
  while (!empty()) {
    pop_front();
  }
}
```

```
void List<T>::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim; victim=0;
}
```

A Bad Solution

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  // ...
} ~List() is called, Gorilla objects leaked
```

- Why not have List remove Gorilla objects? Code an example.

```
List<T>::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim->datum; //New line of code
  delete victim;
}
```

A Bad Solution

```
List<T>::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim->datum; //New line of code
  delete victim;
}
```

- Look what breaks!

```
int main() {
  List<int> li;
  li.push_front(7); // won't compile
   // because pop_front() tries to delete an object
   // that is not a pointer
}
```

A Bad Solution

```
List<T>::pop_front() {
  assert(!empty());
  Node *victim = front_ptr;
  front_ptr = front_ptr->next;
  delete victim->datum; //New line of code
  delete victim;
}
```

- Also, this code would break in a very confusing way!

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  Gorilla *g = l.front();
  l.pop_front(); //Gorilla object deleted
  g->get_name(); //THIS CODE IS NOW UNDEFINED!!!
}
```

Problems with containers of pointers

- Problem: destructor removes the `Node` objects, but not the `Gorilla` objects
- You are responsible for managing dynamic memory

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  while (!todo.empty()) {
    Gorilla *g = todo.front();
    cout << "Hi, " << g->get_name() << "\n";
    todo.pop_front();
  }
  while (!zoo.empty()) {
    delete zoo.front();  //fixed ☺
    zoo.pop_front();
  }
}
```

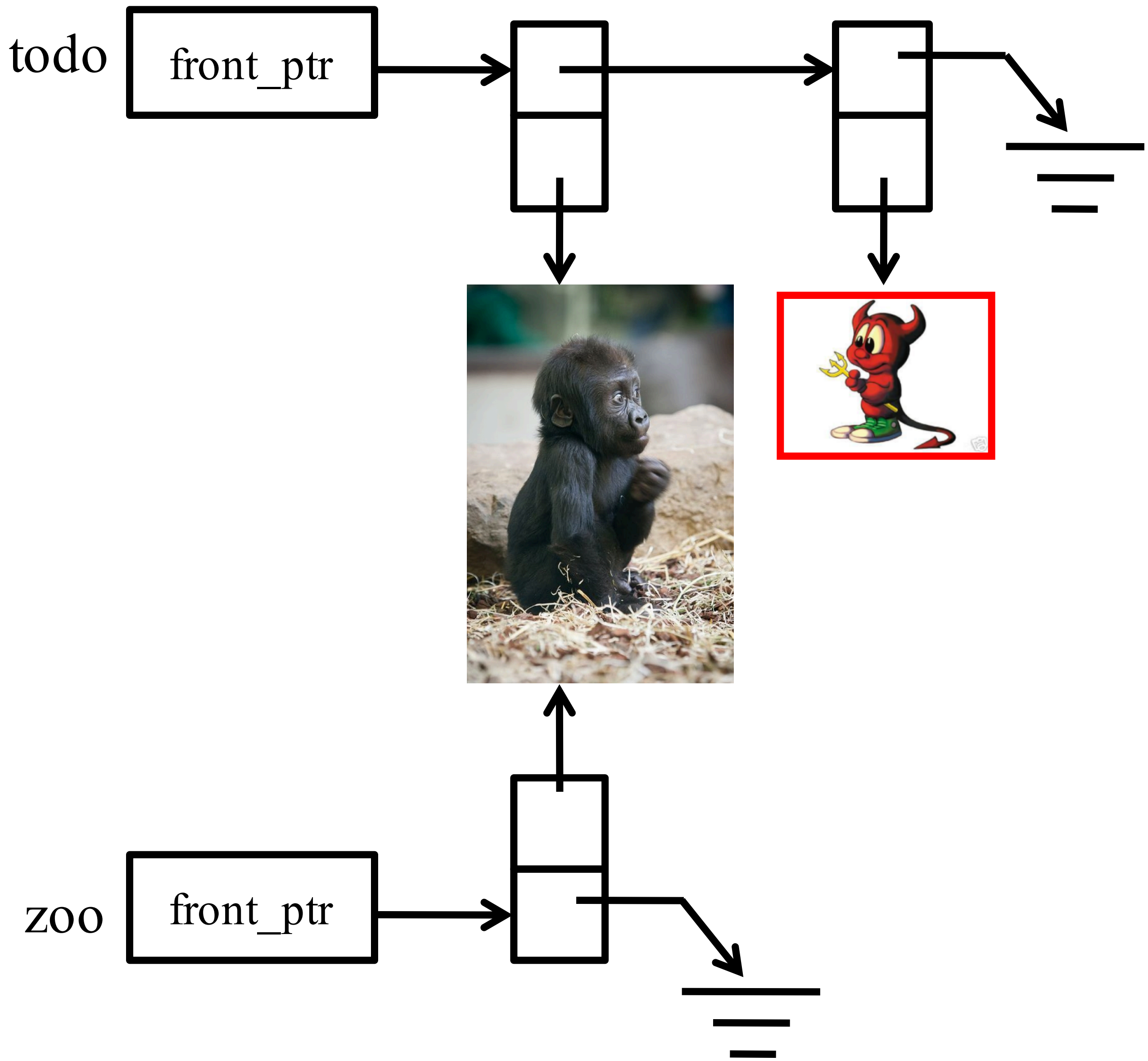
Problems with containers

- Let's see another common problem

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  // Koko moves to another zoo
  delete zoo.front(); zoo.pop_front();
  // Francine starts feeding Gorillas ...
```

- What's the problem?

Problems with containers



Problems with containers

- Let’s see another common problem

```
int main() {
  List<Gorilla*> zoo;
  zoo.push_front(new Gorilla("Colo"));
  zoo.push_front(new Gorilla("Koko"));
  List <Gorilla*> todo = zoo;
  // Koko moves to another zoo
  delete zoo.front(); zoo.pop_front();
  // Francine starts feeding Gorillas ...
```

- Francine tries to feed Koko, not knowing that he’s not here any more!

Pattern of use for container-of-ptr

Containers-of-pointers are subject to two broad kinds of bugs

1. Using an object after it has been deleted
2. Leaving an object orphaned by never deleting it

- Containers do not manage memory outside of themselves
  - Container doesn't know what you want to do!

Pattern of use for avoiding these bugs

- Whoever creates memory is responsible for deleting it
- Every time you create memory with `new`, you must remove it with `delete`

Disabling copy

- Sometimes, it’s helpful to disable copying if you truly don’t *ever* want the objects copied
- Use private to cause a compiler error whenever a Gorilla is copied

```
class Gorilla {
  //...
 private:
  Gorilla(const Gorilla &other);
  Gorilla & operator=(const Gorilla &rhs);
};
```

Containers of pointers: reminder

A reminder about when to use containers of pointers

- Avoid slow copies of big objects
- Want unique objects
- If you want to manage the lifetime of the variable yourself
  - Not local, not global, but something in between
- Many parts of the code refer to an object, but only want one copy in memory. Normally, if many parts refer to the same object, those parts will be in different scopes.