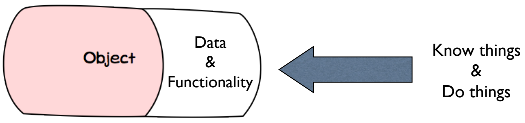
Resource Management

by Willem van Straten and Andrew Cain

SWINBURNE UNIVERSITY OF TECHNOLOGY

Object Oriented Programming

Object oriented programming involves objects that know and do things

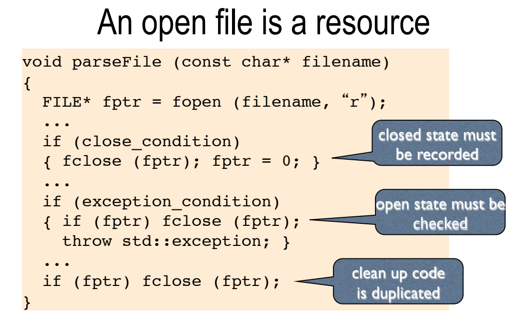


In a dynamic application, objects must acquire and relinquish resources

Developers must manage resource acquisition and relinquishment

In C++, developers must manage object creation **and** destruction.

Objects are responsible for managing the resources that they acquire



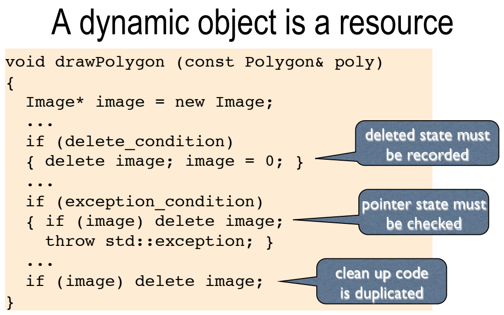
An open file is a resource

```
void parseFile (const char* filename)
{
  FILE* fptr = fopen (filename, “r”);
  ...
  if (close_condition)
  { fclose (fptr); fptr = 0; }
  ...
  if (exception_condition)
  { if (fptr) fclose (fptr);
    throw std::exception; }
  ...
  if (fptr) fclose (fptr);
}
```

closed state must be recorded

open state must be checked

clean up code is duplicated



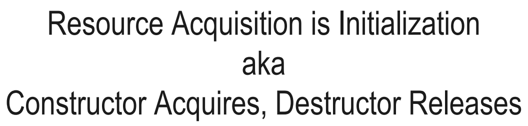
A dynamic object is a resource

```
void drawPolygon (const Polygon& poly)
{
  Image* image = new Image;
  ...
  if (delete_condition)
  { delete image; image = 0; }
  ...
  if (exception_condition)
  { if (image) delete image;
    throw std::exception; }
  ...
  if (image) delete image;
}
```

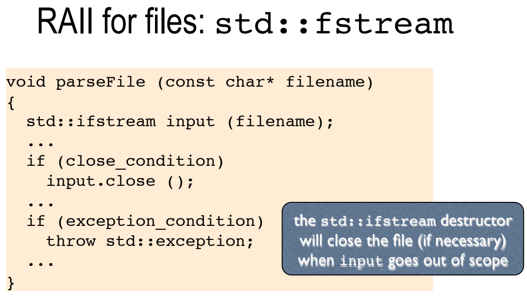
deleted state must be recorded

pointer state must be checked

clean up code is duplicated



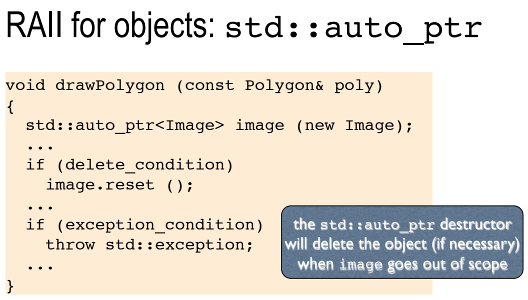
Resource Acquisition is Initialization
aka
Constructor Acquires, Destructor Releases



RAII for files: `std::fstream`

```
void parseFile (const char* filename)
{
  std::ifstream input (filename);
  ...
  if (close_condition)
    input.close ();
  ...
  if (exception_condition)
    throw std::exception;
  ...
}
```

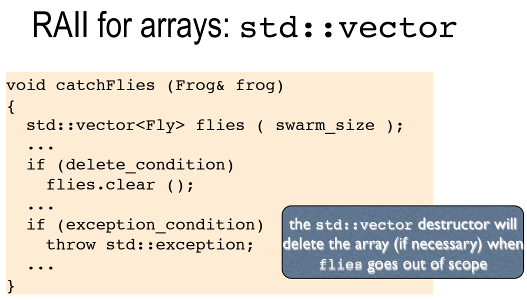
the `std::ifstream` destructor will close the file (if necessary) when `input` goes out of scope



RAII for objects: `std::auto_ptr`

```
void drawPolygon (const Polygon& poly)
{
  std::auto_ptr<Image> image (new Image);
  ...
  if (delete_condition)
    image.reset ();
  ...
  if (exception_condition)
    throw std::exception;
  ...
}
```

the `std::auto_ptr` destructor will delete the object (if necessary) when `image` goes out of scope



RAII for arrays: `std::vector`

```
void catchFlies (Frog& frog)
{
  std::vector<Fly> flies ( swarm_size );
  ...
  if (delete_condition)
    flies.clear ();
  ...
  if (exception_condition)
    throw std::exception;
  ...
}
```

the `std::vector` destructor will delete the array (if necessary) when `flies` goes out of scope

RAII uses static object scope to constrain resource lifetime

Resources with function scope

```
void parseFile (const char* filename)
{
  std::ifstream input (filename);

  [...]
}
```

Resources with object scope

```
class DeckOfCards
{
private:
  std::auto_ptr<RandomNumberGenerator> shuffler;

  [...]
};
```

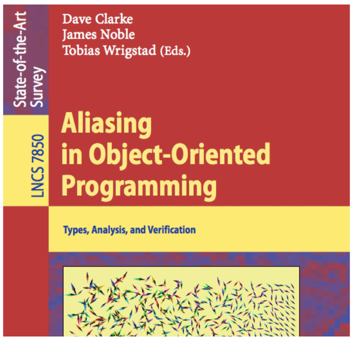
RAII is a form of delegation

the Polygon class delegates responsibility for managing the array of Point objects to std::vector

```
class Polygon
{
  std::vector<Point> vertices;
public:
  Polygon (int vertices);
}
```

RAII is key to exception safe code (Topic 10)

Objects must collaborate when managing shared resources



State-of-the-Art Survey

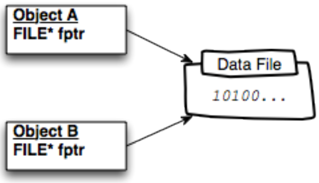
LNCS 7850

Dave Clarke
James Noble
Tobias Wrigstad (Eds.)

Aliasing in Object-Oriented Programming

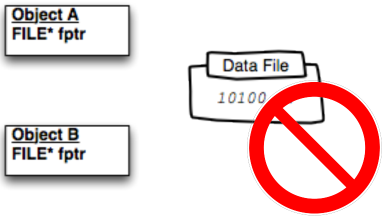
Types, Analysis, and Verification

Resource aliasing

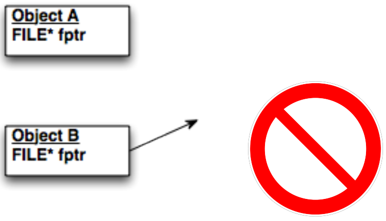


Unintentional or poorly managed aliasing leads to
resource leaks and bad references

Resource leak



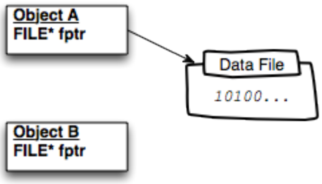
Bad reference



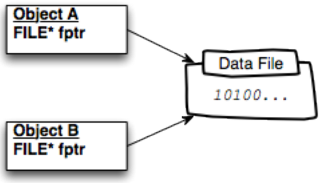
Collaborative resource management
requires an ownership policy

Strict, shared, and duplicate

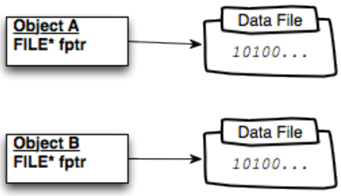
Strict (or exclusive or unique)



Shared



Duplicate (or deep copy)



C++ implicit methods are of central importance to ownership policy implementation

Remember the Implicit Methods

```
void function ()
{
  MyClass A;

  MyClass B (A);

  A = B;

}
```

Default Constructor

Copy Constructor

Assignment Operator

Destructor (x 2)

Consider the Implicit Methods

```
void function ()
{
  MyClass A;

  MyClass B (A);

  A = B;

}
```

What happens if `MyClass` owns a resource?
Does B take from A?
Does B share with A?
Does B duplicate A' s resources?

Understand the Implicit Methods

Automatically generated versions
do not implement any ownership policy

Automatic default constructor

Does not initialize resource,
leading to undefined behaviour

Automatic copy constructor and assignment operator

Perform a shallow copy,
leading to unintended aliasing

Automatic destructor

Does not free resource,
leading to resource leakage

If a class manages a resource,
never rely on the automatically
generated implicit methods

Manage aliasing by
implementing the implicit methods

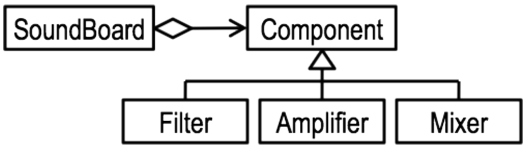
Simple Example: Deep Copy

```
class SoundBoard
{
  std::vector<Component*> components;
  [...]
};
```

see `SoundBoard.cpp`

Use the clone pattern to
handle polymorphism in deep copy

Polymorphism and Deep Copy



```
class SoundBoard
{
  std::vector<Component*> components;
[...]
};
```

```
SoundBoard::SoundBoard (const SoundBoard& that)
{
  for (int i=0; i<that.components.size(); i++)
    components.push_back( new ???? );
}
```

which derived type should be constructed here?

The clone pattern

```
class Base
{
public:
  virtual ~Base ();
  virtual Base* clone () const = 0;
};
```

why virtual?

why const?

```
class Derived
{
public:
  Derived* clone () const
  { return new Derived (*this); }
};
```

Derived class copy constructor is called

Polymorphic Deep Copy

```
SoundBoard::SoundBoard (const SoundBoard& that)
{
  for (int i=0; i<that.components.size(); i++)
  {
    Component* component = that.components[i];
    components.push_back( component->clone() );
}
```

Aliasing:
understand it, anticipate it,
and manage it

The C++ compiler
does not automatically
manage aliasing

The C++ compiler
does not even notice aliasing

Understand the object ownership
policies adopted by other classes

Resource management is
more than dynamic memory
management

Resource management skills
enable you to tackle
more complex projects

This Week's Tasks

Pass Task 18: Robust Planet Rover

Credit Task 2: Case Study – Iterations 4 & 5

Distinction Task 2: Custom Program Sequence Diagram