Resource Management

by Willem van Straten and Andrew Cain

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.

When using C#, the developer cares only about creation.

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

Motivation

- Resources acquired in a function scope should be released before leaving the scope unless the ownership is being transferred to another scope or object.
- Normally it means a function calls – **one to acquire a resource** and another **one to release it.**
- For example, new/delete, malloc/free, acquire/release, file-open/file-close, etc.
- Problem: forget to write the “release” part of the resource management “contract”.
- Sometimes the resource release function is never invoked: this can happen when the control flow leaves the scope because of return or an exception.

Retrieved from : *https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Resource_Acquisition_Is_Initialization*

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;
  ...
}
```

`std::ifstream` – construct object and optionally open file

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` object have the peculiarity of taking ownership of the pointers assigned to them

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;
  ...
}
```

Vectors are sequence containers representing arrays that can change in size.

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);

  [...]
}
```

File closed when function returns, and ifstream goes out of scope

Resources with object scope

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

  [...]
};
```

RandomNumberGenerator destroyed when DeckOfCards object is destroyed, and auto_ptr goes out of scope

RAII is a form of delegation

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

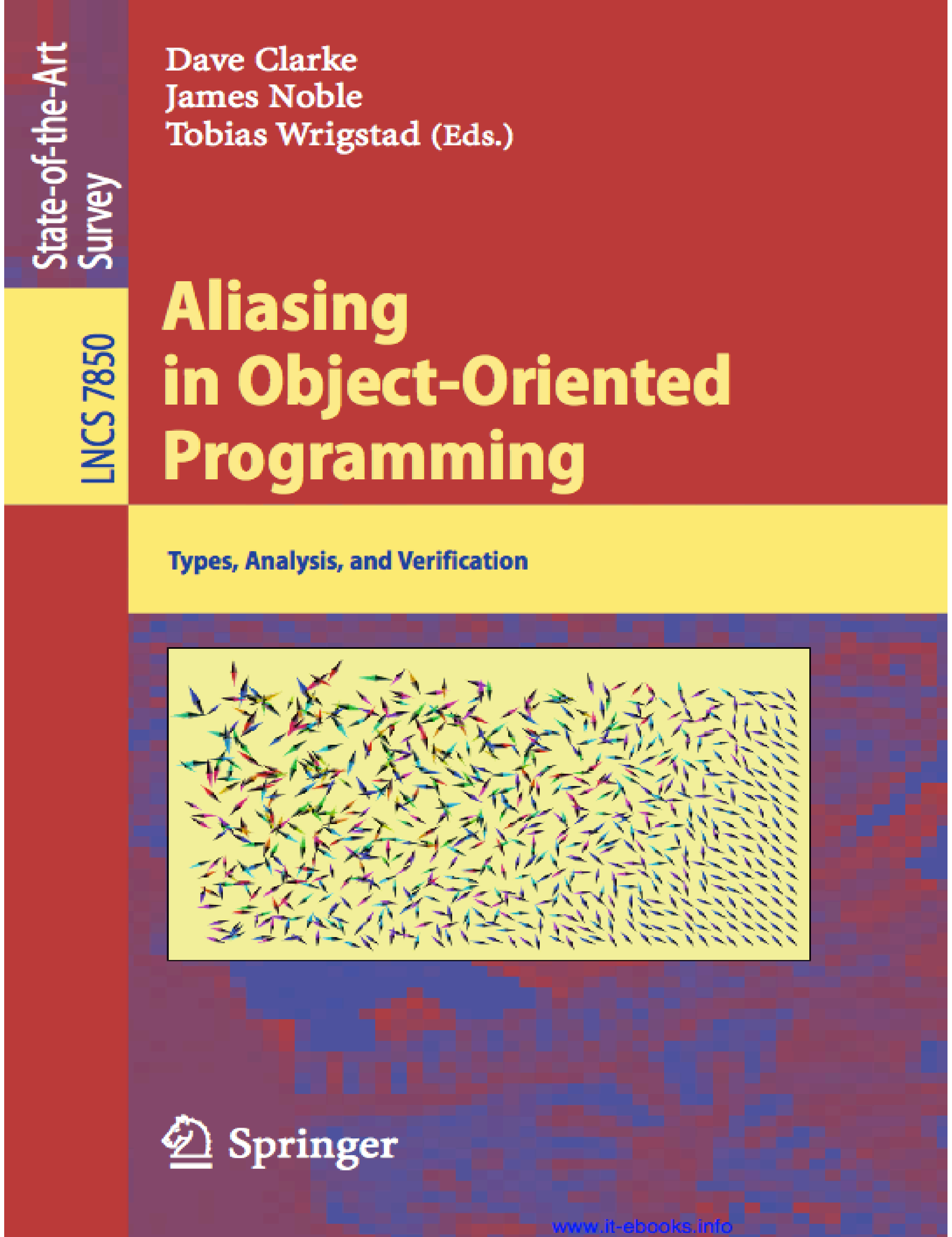
the `Polygon` class delegates responsibility for managing the array of `Point` objects to `std::vector`

RAII is key to exception safe code (Topic 10)

Static objects with function scope will be destroyed when the function is popped off the execution stack, either by returning or by throwing an exception

Objects must collaborate when managing shared resources

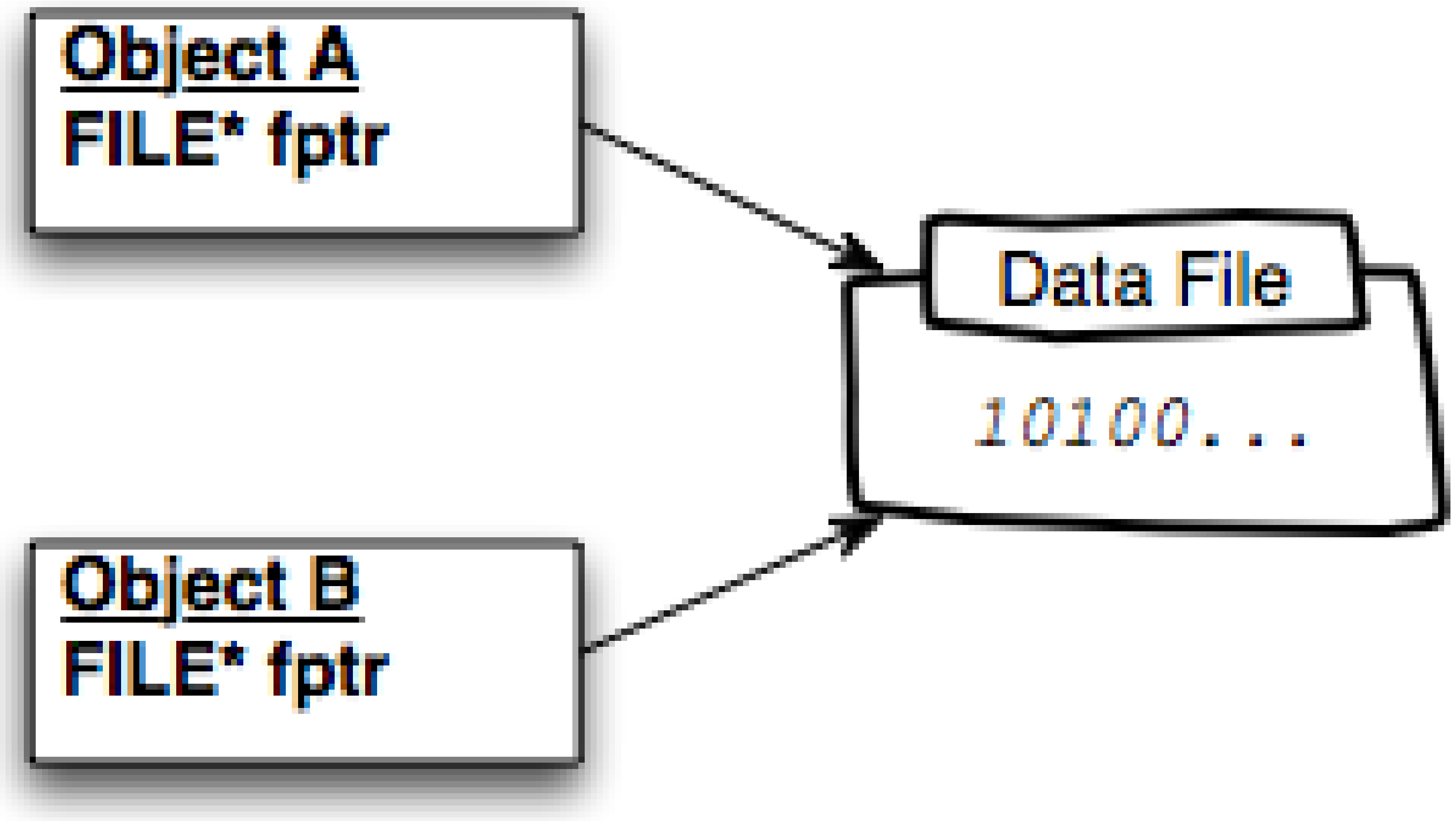
What if there are more than one objects sharing the resources?



*A resource is **aliased** when two or more objects maintain a reference to it.*

Many articles have been written on the subject.

Resource aliasing

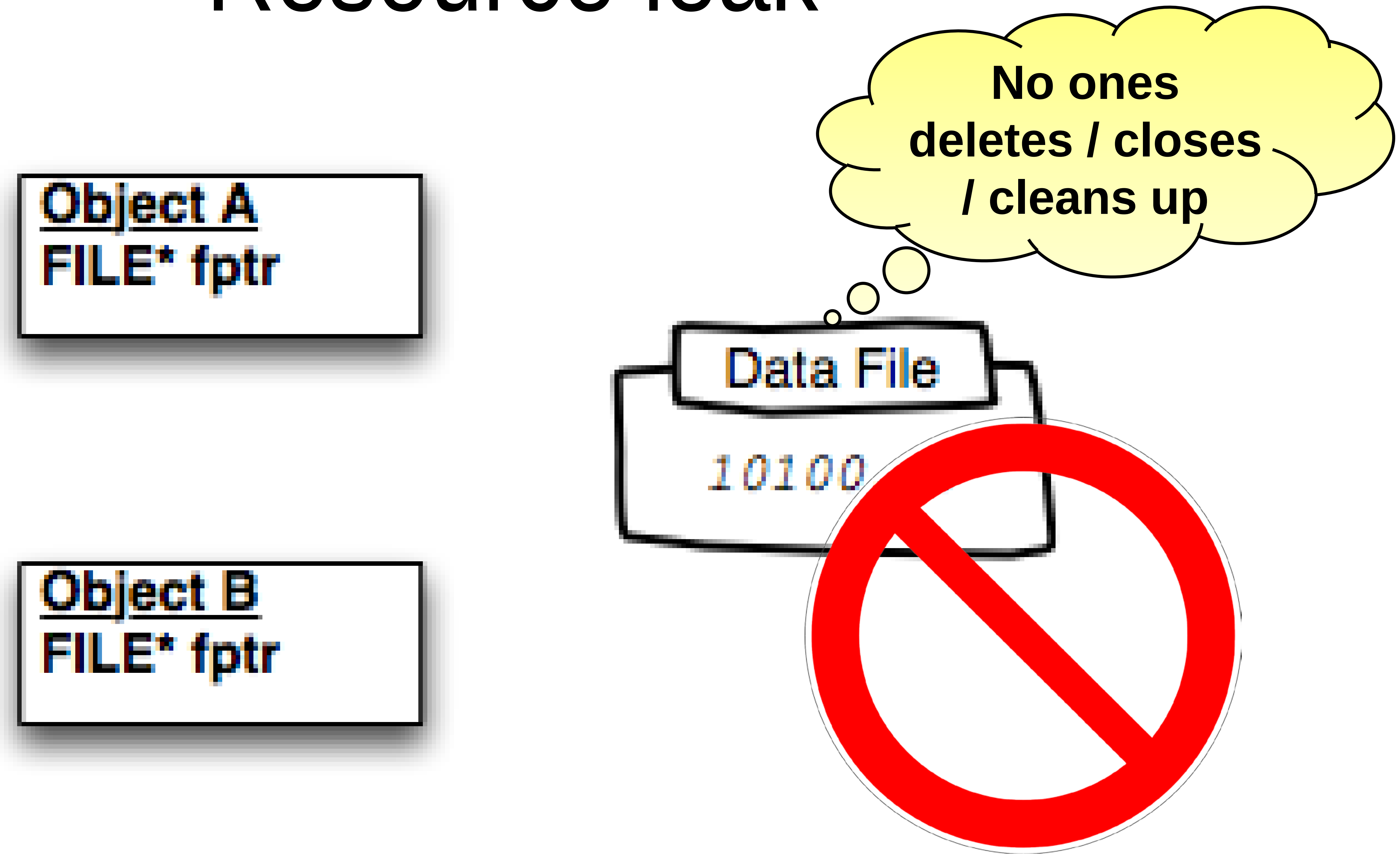


- Aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program

- Thus, modifying the data through one name implicitly modifies the values associated with all the aliased names, which may not be expected by the programmer.

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

Resource leak



Resource leak

When the programmer fails to return a dynamically allocated section of memory back to the memory manager using delete or delete[]

Causes no harmful effect

In a long-running program, or if memory allocation occurs in a section of the program that is executed repeatedly, then a memory leak can cause a program to halt because the memory manager is unable to service a request for new memory.

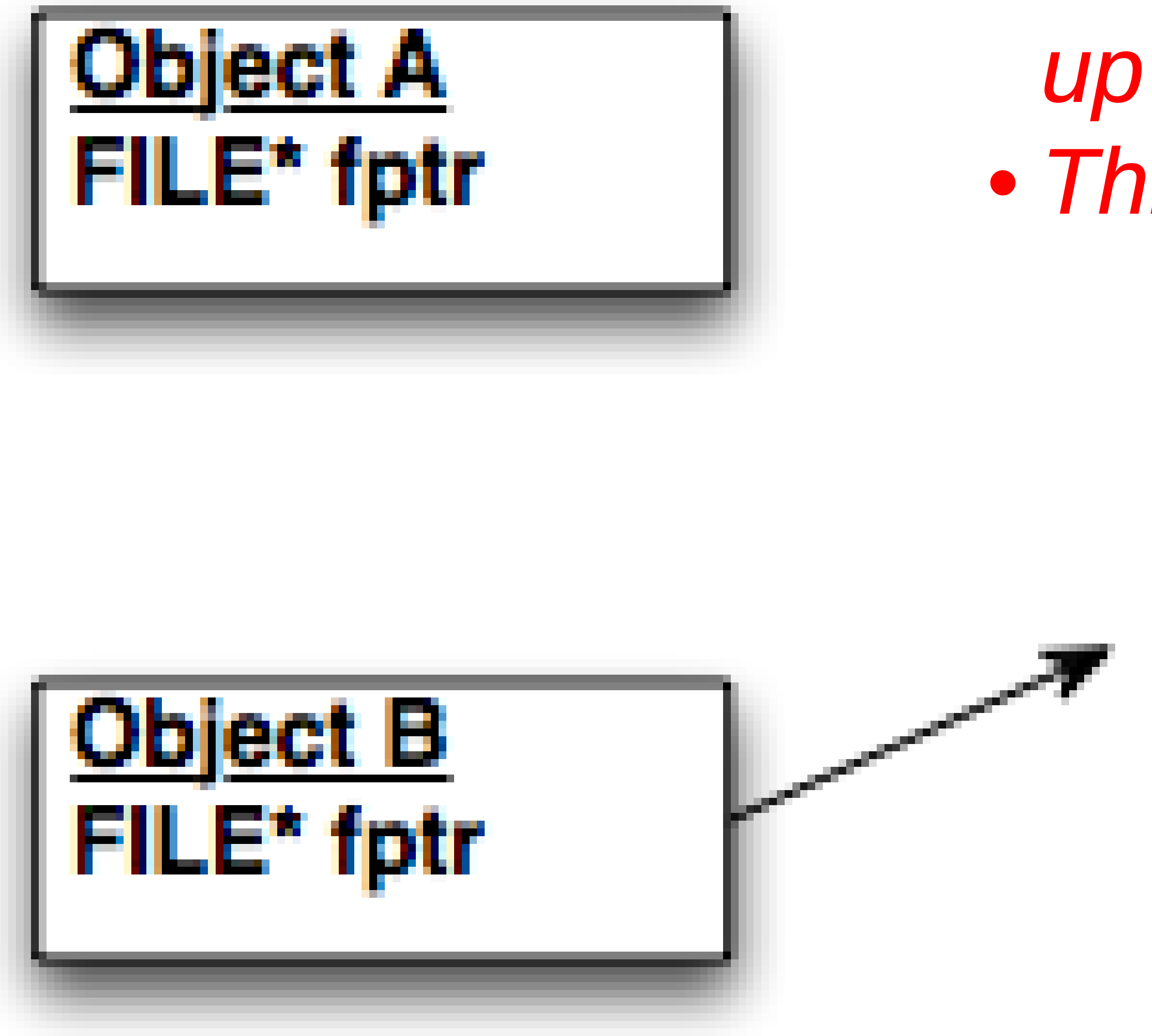
The result of successive assignments to the same pointer variables

Resource leak

```
Clock * a_clock;
…
a_clock = new TravelClock(true, “Rome”, 0);
…
a_clock = new TravelClock(true, “Tokyo”, -7); //Leak, old
memory is now lost
```

- After the second assignment, both dynamically allocated objects remain on the heap.
- However, there are no remaining pointers to the first object, so it cannot be recovered.
- The memory used by this object is lost.

Bad reference



- *only one deletes / closes / cleans up without telling others*
- *This type of error is catastrophic*

Bad reference

```
for(Node *p = ptr; p != NULL; p = p->next){
   delete p; //Error – p->next  is dereference after p  is deleted
}
```

Errors of this type are sometimes committed by a programmer forgetting that the update portion of a for loop is executed after the body of the loop

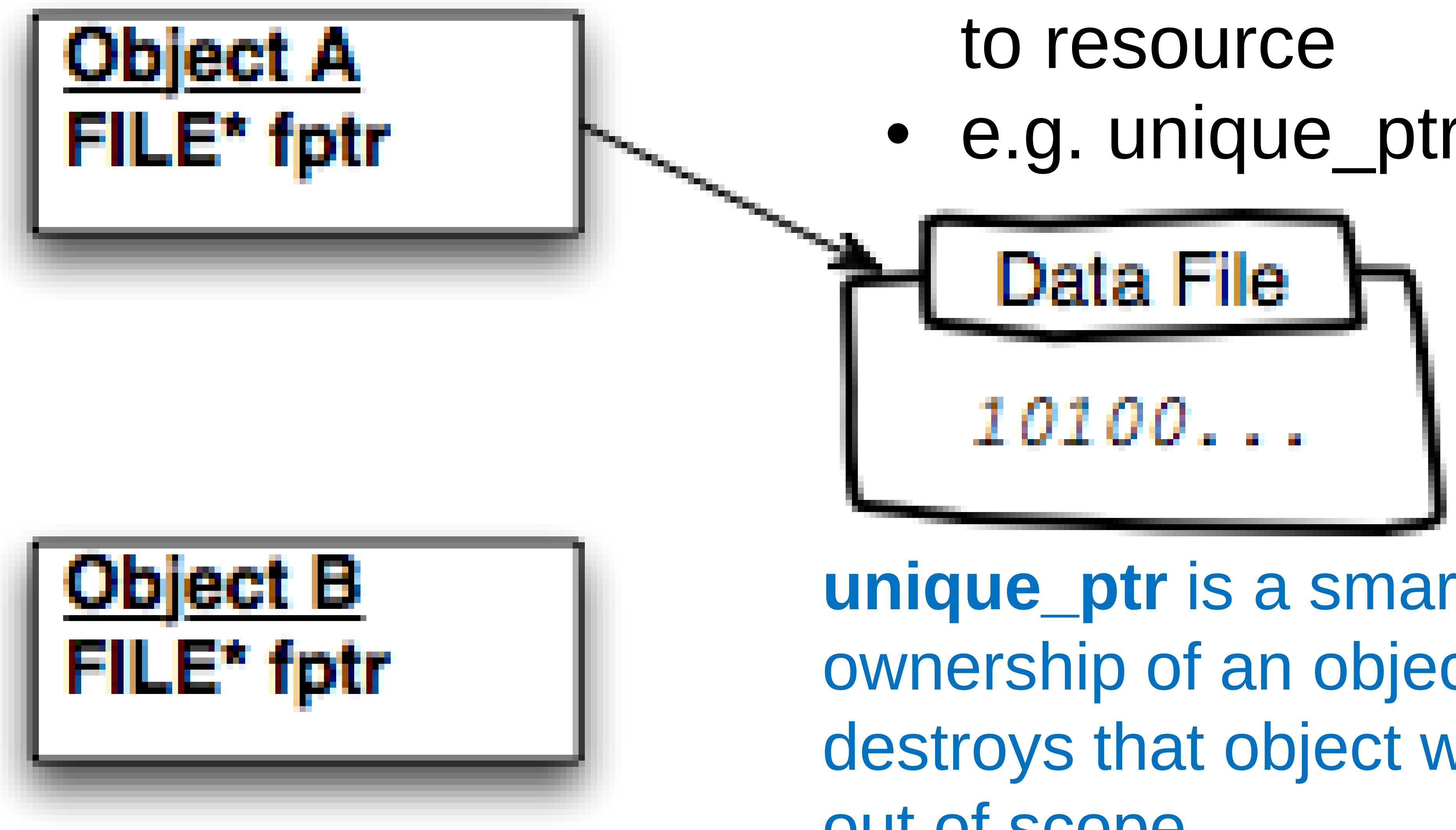
```
for(Node *p = ptr; p != NULL){
   Node *q = p->next; //OK. Read  p->next  before deleting p
   delete p;
   p = q;
}
```

Solution: *The reference to p->next can be executed after the memory that p refers to has been recovered and, potentially, overwritten. The solution is to read the value first, before performing the deletion.*

Collaborative resource management requires an ownership policy

Strict, shared, and duplicate

Strict (or exclusive or unique)

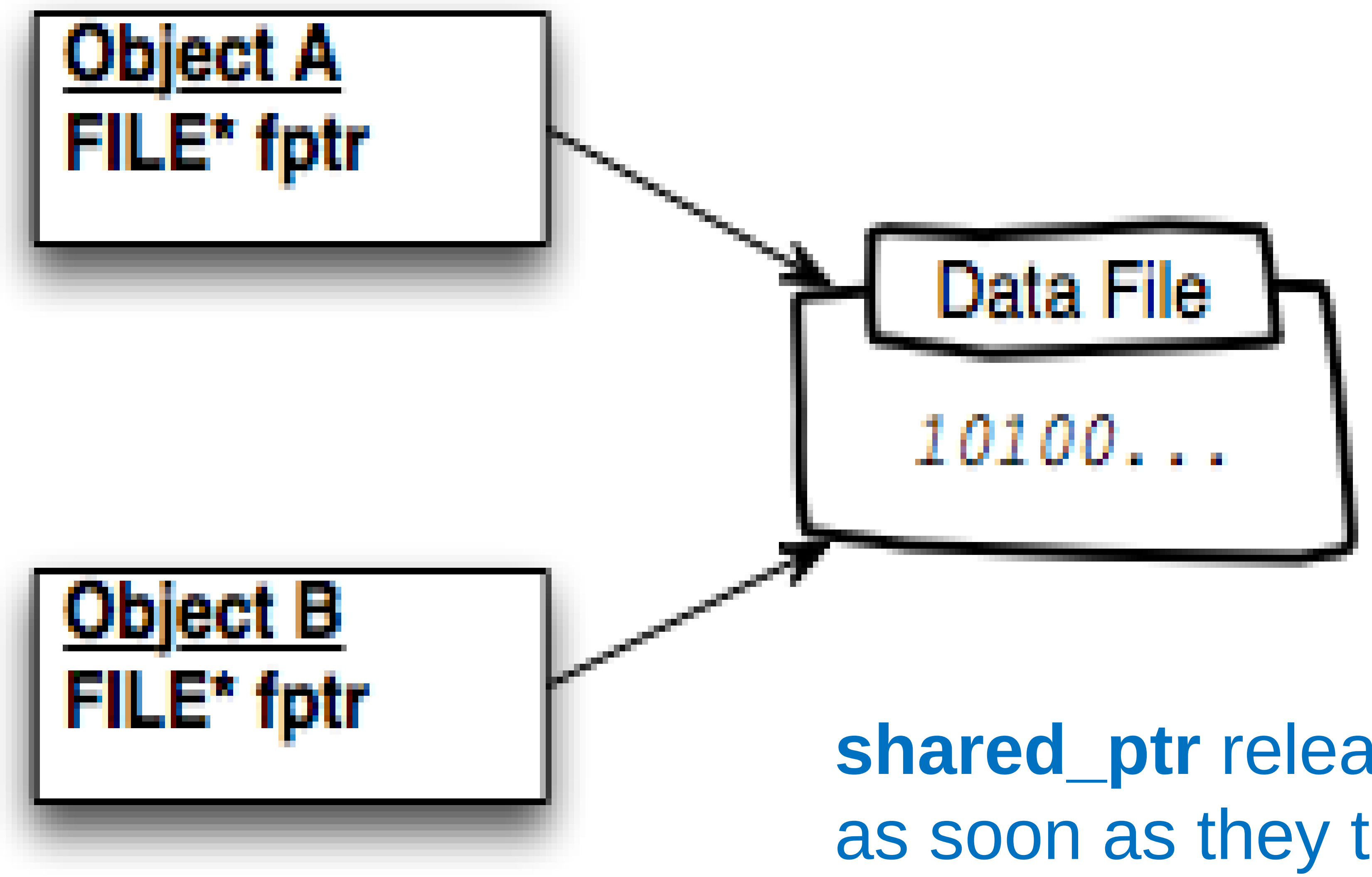


- Only one object may own / use/ refer to resource
- e.g. unique_ptr

unique_ptr is a smart pointer that retains sole ownership of an object through a pointer and destroys that object when the unique_ptr goes out of scope.

NO two **unique_ptr** instances can manage the same object.

Shared

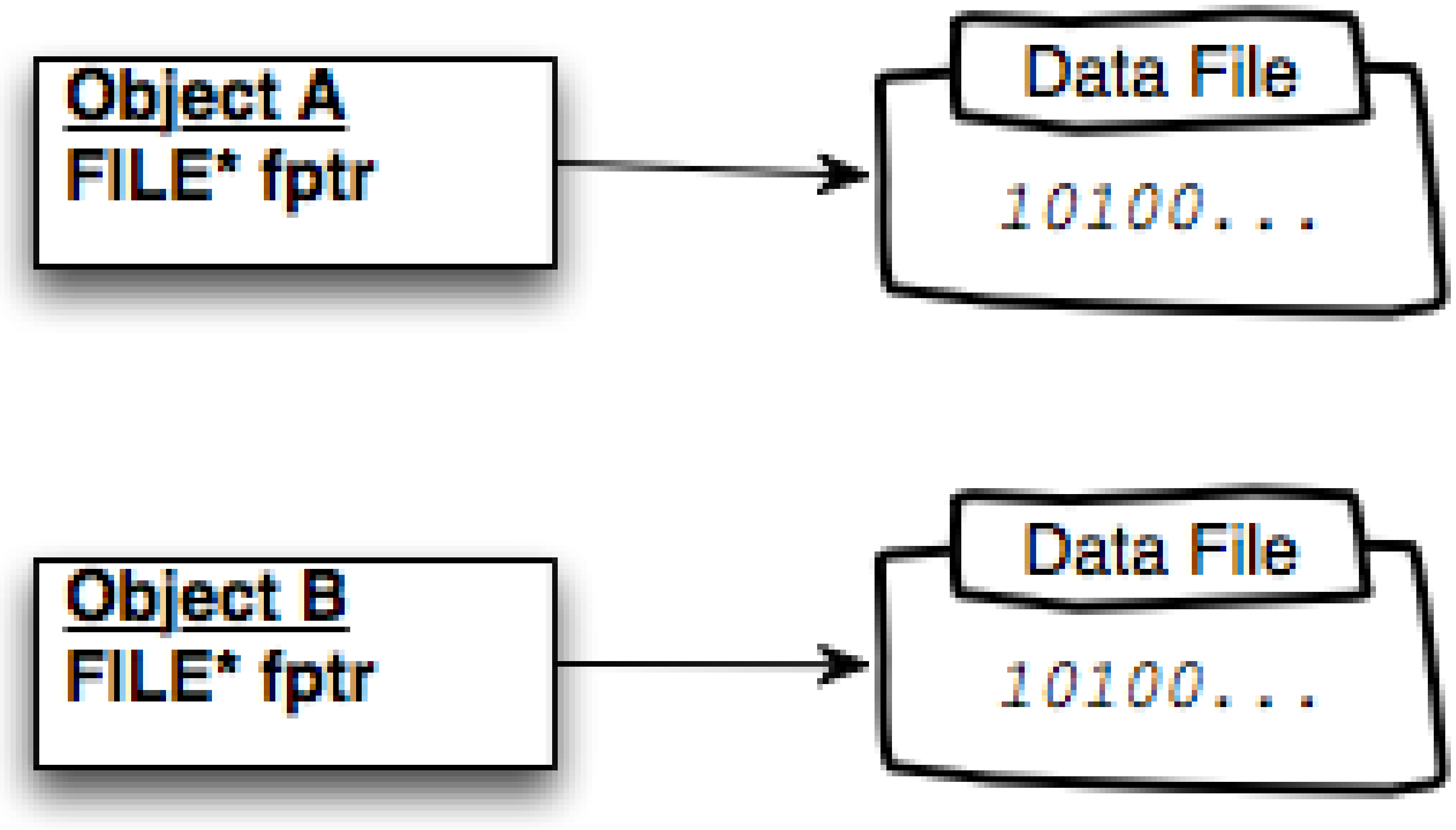


- multiple objects may own / use / refer to resource
- may include read-only, copy-on-write, and reference-counting
- reference counting implemented by shared_ptr and weak_ptr

shared_ptr release ownership on the object they co-own as soon as they themselves are destroyed.

Once all **shared_ptr** objects that share ownership over a pointer have released this ownership, the managed object is deleted.

Duplicate (or deep copy)



- each object owns / uses / refers to a copy of resource

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

Remember the Implicit Methods

If the implicit methods are not explicitly defined, then the compiler will automatically generate them

```
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

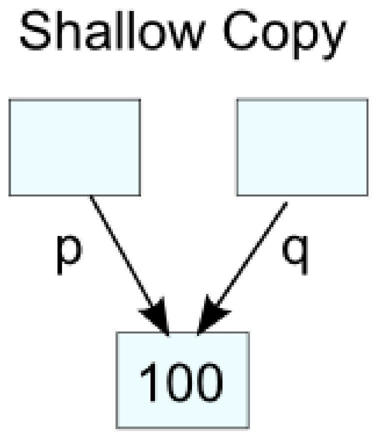
Automatic destructor

Does not free resource,
leading to resource leakage

If a class manages a resource, <u>NEVER</u> rely on the automatically generated implicit methods

Manage aliasing by implementing the implicit methods

Shallow Copy



- A *shallow copy* of an object copies all of the member field values.

- This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory.

- The pointer will be copied. but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want.

- The **default copy constructor** and **assignment operator** make shallow copies.

Example

```
class CSample {
    int x;

    public:
    //Default constructor
    CSample()
    {   x=0; }
    int GetX()
    { return x; }
};
```

```
int main() {
    //Default constructor is called.
    CSample ob1;
     //Default copy constructor called.
    CSample ob2 = new Csample(ob1);
     //Default constructor called.
    CSample ob3;
     //Default overloaded = operator
       function called.
    ob3 = ob1;

}
```

```
CSample ob2 = new CSample(ob1);
```

This line will copy the bit pattern of ob1 in ob2 so the data member x in both the object will contain same value i.e. 0.

```
ob3 = ob1;
```

This line will copy the bit pattern of ob1 in ob3 so the data member x in both the object will contain same value i.e. 0.

The above code will work as expected until the class member is not allocated any resource (file or memory).

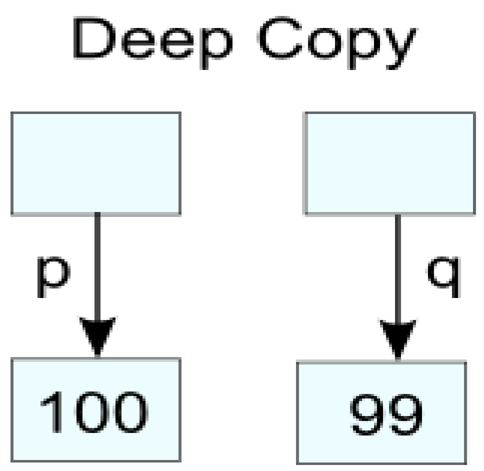
Consider a scenario where the class is updated as follow:

```
class CSample {
     int *x;
     int N;
     public:
     //Default constructor
     Csample(){
          x=NULL; }
     void AllocateX(int N){
          this->N = N;
          x = new int[this->N]; }
     int GetX() { return *x; }
     ~CSample() { delete x; } };
```

```
int main() {
     //Default constructor is called.
     CSample ob1;
     ob1.AllocateX(10);
     //Default copy constructor called.
     //problem with this line
     CSample ob2 = new CSample(ob1);
     //Default constructor called.
     CSample ob3;
     //Default overloaded = operator
       function called.
     //problem with this line
     ob3 = ob1;
 }
```

The default copy constructor and overloaded = operator will copy the pointers and value of N of one object to another object. This will lead to memory leak and dangling reference issues.

Deep Copy must introduced…



- A *deep copy* copies all fields, *and* makes copies of dynamically allocated memory pointed to by the fields.

- To make a deep copy, you must write a **copy constructor** and **overload the assignment operator**, otherwise the copy will point to the original, with disastrous consequences.

- If an object has pointers to dynamically allocated memory, and the dynamically allocated memory needs to be copied when the original object is copied, then a deep copy is required.

What do we need for Deep Copy?

- A class that requires deep copies generally needs:
  - ✓ A constructor to either make an initial allocation or set the pointer to NULL.
  - ✓ A destructor to delete the dynamically allocated memory.
  - ✓ A copy constructor to make a copy of the dynamically allocated memory.
  - ✓ An overloaded assignment operator to make a copy of the dynamically allocated memory.

The updated code by introducing copy constructor and = operator function

```
class CSample { //begin of class
    int *x;
    int N;
    public:
    //default constructor
    CSample(){
        x=NULL; }

    //copy constructor
    CSample(const CSample &ob){
        this->N = ob.N;
        this->x = new int[this->N]; }

    //=operator function with deep copy.
    void operator=(const CSample &ob){
         this->N = ob.N;
         this->x = new int[this->N];  }
```

```
    void AllocateX(int N){
        this->N = N;
        x = new int[this->N]; }

    int GetX() { return *x; }

    ~CSample() { delete x;}

}; //end of class
```

Reference for Shallow and Deep Copy:
https://www.hackerearth.com/notes/deep-copy-and-shallow-copy/

Aliasing:
understand it, anticipate it,
and manage it

The C++ compiler
does not automatically
manage aliasing

The C++ compiler does not even notice aliasing

In fact, the automatically-generated implicit methods will work against you

Understand the object ownership policies adopted by other classes

e.g. auto_ptr is exclusive

Resource management is more than dynamic memory management

Resource management skills enable you to tackle more complex projects

Let’s have a look at C++ codes for Copy Constructor and Clone()

Main Program

```
int main(){
     Temp *test = new Temp();
     X *x1 = new X();
     Y *y1 = new Y();
     test->addnum(x1);
     test->addnum(y1);
     Temp *copyTest = new Temp(*test);
     cout<<test->getNumSize(); //prints 2
     cout<<copyTest->getNumSize(); //prints 2
     return 0;
}
```

Child Class - X

```
class X:Score{
     private:
       int x1;

     public:
       X(){
          x1 = 5;
       }

       X* clone(){
          return new X();
       }
}
```

Parent Class - Score

```
class Score{
     public:
       virtual Score* clone()=0;
};
```

Child Class - Y

```
class Y:Score{
     private:
       char y1;

     public:
       Y(){
          y1 = ‘a’;
       }

       Y* clone(){
          return new Y();
       }
}
```

Class - Temp

```
class Temp{
   private:
     vector <Score*> num;

   public:
     Temp(const Temp T&){
        num = vector<Score *>(T.num.size());
        for(int i=0; i<T.num.size(); i++){
           num[i] = T.num[i]->clone();
        }
     }

     void addNum(Score *s){
          num->push_back(s);
     }

     int getNumSize(){
          return num.size();
     }
};
```

This Week’s Tasks

Supplementary Exercise: Robust Planet Rover

Supplementary Exercise : Case Study – Iterations 3

Supplementary Exercise : Case Study – Iterations 4

Distinction Task 2: Custom Program Sequence Diagram