The Rule of ~~Big Three~~ ~~and a half~~ ~~Five~~ ~~Four~~ ~~and a half~~ ~~Seven~~ Zero
or
Evolution of Resource Management in C++ in the Recent Years

Adam Graliński

C++ FFFE, September 2021

Resource management

- Dynamic allocation and destruction of objects is, generally, hard.
- do it wrong and spend hours debugging:
  - the crashes, or
  - the unexpected runtime behavior
- don't do it and just watch the memory leak

The first known solution there became known as the *RAII* (or RRID) software pattern.

RAII in the Dark Ages (C++98)

en.cppreference.com/w/cpp/language/raii

- encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
  - the destructor releases the resource and never throws exceptions;
- always use the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime that is bounded by the lifetime of an automatic or temporary object

Classes with open()/close(), lock()/unlock(), or init()/copyFrom()/destroy() member functions are typical examples of non-RAII classes.

Problems with RAII

This version of RAII just transfers the problems to the manager class.

- How to pass the resource around?
- Can the manager object be copied?

The *Rule of Big Three* was coined, and it was a good rule. For a while.

Meanwhile, the copy-and-swap idiom took traction.

- How to account for the copy-and-swap idiom?

The Rule of Big Three became the Rule of Big Three*and a half*.

The Rule of Big Three[and a half]

When writing a class that manages a dynamically-allocated resource, then:

1. the resource may be passed-by-value (copied)
2. the resource may be assigned to another instance
3. the resource needs to be deallocated when falling out of scope

So, if you have implemented either:

1. a copy constructor
2. an assignment operator
3. a destructor

then you need to implement *all three of them*.

and a half: also implement own `swap()` function for copy-and-swap idiom.

The Rule of Big Three — a simple example

```
#include <string>

class Person {
private:
  std::string* name_;
  int age_;

public:
  Person(std::string const& name, int age) {
    name_ = new std::string{name};
    age_ = age;
  }

  ~Person() {
    delete name_;
  }
};

int main() {
  Person john("John Doe", 42);
  Person same_john(john); // What should happen here?
  same_john = john;       // and here?
}
```

The Rule of Big Three — a simple example

```cpp
#include <string>

class Person {
private:
  std::string* name_;
  int age_;

public:
  Person(std::string const& name, int age) {
    name_ = new std::string{name};
    age_ = age;
  }

  ~Person() {
    delete name_;
  }
};

int main() {
  Person john("John␣Doe", 42);
  Person same_john(john); // What should happen here?
  same_john = john;       // and here?
}
```

```cpp
#include <string>

class Person {
private:
  std::string* name_;
  int age_;

public:
  Person(std::string const& name, int age) {
    name_ = new std::string{name};
    age_ = age;
  }

  // (1) Copy constructor
  Person(const Person& other) {
    name_ = new std::string{*other.name_};
    age_ = other.age_;
  }

  // (2) Copy assignment operator
  Person& operator=(const Person& other) {
    name_ = new std::string{*other.name_};
    age_ = other.age_;
    return *this;
  }

  // (3) Destructor
  ~Person() {
    delete name_;
  }
};

int main() {
  Person john("John␣Doe", 42);
  Person same_john(john); // OK, well defined.
  same_john = john;       // OK, also well defined.
}
```

C++11 is a game changer. With its concept of the *r-value reference* one can finally force to **move** where C++98 would **copy**.

The resource managing classes now must also account for move semantics - the **move constructor** and **move assignment operator**.

The Rule of Big Five — a simple example

```cpp
#include <string>

class Person {
private:
  std::string* name_;
  int age_;

public:
  Person(std::string const& name, int age) {
    name_ = new std::string{name};
    age_ = age;
  }

  // (1) Copy constructor
  Person(const Person& other) {
    name_ = new std::string{*other.name_};
    age_ = other.age_;
  }

  // (2) Move constructor
  Person(Person&& from) {
    std::swap(name_, from.name_);
    age_ = from.age_;
  }

  // (3) Copy assignment operator
  Person& operator=(const Person& other) {
    name_ = new std::string{*other.name_};
    age_ = other.age_;
    return *this;
  }

  // (4) Move assignment operator
  Person& operator=(Person&& from) {
    std::swap(name_, from.name_);
    age_ = from.age_;
    return *this;
  }

  // (5) Destructor
  ~Person() {
    if (name_ != nullptr) {
      delete name_;
      name_ = nullptr;
    }
  }
};

int main() {
  Person john("John Doe", 42);
  Person same_john(john); // OK, well defined.
  same_john = john;       // OK, also well defined.

  Person lenny{"Lenny", 43};
  Person moved = std::move(lenny);
}
```

The Rule of Big Four$^{\text{and a half}}$

If you have implemented either:

1. the copy constructor
2. the assignment operator
3. the move constructor
4. the swap function

then you need to ~~implement~~ *define the usage* of all of them.

The Rule of Zero

There is an alternative to The Rule of Big Four$^{\text{and a half}}$, called *the Rule of Zero*:

You should never implement a destructor, copy constructor, move constructor or assignment operators.

With one important corollary rule:

You should never manage a resource via a raw pointer.

...but what if you must create object dynamically?

If you must create objects dynamically...

Use `std::unique_ptr` or `std::shared_ptr`

- `std::unique_ptr` for resources that can be moved, but not copied
- `std::shared_ptr` for resources that can be both moved and copied

Rule of Zero — really simple example

```
#include <string>
#include <memory>

class Person {
private:
  std::shared_ptr<std::string> name_;
  int age_;

public:
  Person(std::string const& name, int age) {
    name_ = std::make_shared<std::string>(name);
    age_ = age;
  }

  // (1) NO copy constructor
  // (Person is copy-able)

  // (2) NO move constructor

  // (3) NO copy assignment operator
  // (Person is copy-able)

  // (4) NO move assignment operator

  // (5) And finally, NO destructor
};

int main() {
  Person john("John␣Doe", 42);
  Person same_john(john); // OK, well defined.
  same_john = john;       // OK, also well defined.

  Person lenny{"Lenny", 43};
  Person moved = std::move(lenny);
}
```

Key takeaways

- Don't make your life unnecessary hard
- Let the standard library take care of your objects' lifetime
- Use smart pointers wherever you can

Thank you!