

Memory management in C++

Université Toulouse Paul Sabatier - KEAT9AA1

2024-09-10

Guilhem Saurel

Available at

**[https://homepages.laas.fr/gsaurel/teach/
2024-2025/M2_ISTR/5-memory-c++.pdf](https://homepages.laas.fr/gsaurel/teach/2024-2025/M2_ISTR/5-memory-c++.pdf)**

Under License



<https://creativecommons.org/licenses/by-sa/4.0/>

Source

**`https://gitlab.laas.fr/gsaurel/teach :
2024-2025/M2_ISTR/5-memory-c++.md`**

Contact

Matrix: @gsaurel:laas.fr
Mail: gsaurel@laas.fr

```
struct Point {  
    float x;  
    float y;  
};
```

```
struct Point {  
    float x;  
    float y;  
};
```

```
Point p = { .x = 12.2, .y = -22.7 };
```

```
class Point {  
    float x;  
    float y;  
};
```

```
class Point {  
    float x;  
    float y;  
};
```

Except everything is private by default

```
class Point {  
    float x;  
    float y;  
};
```

Except everything is private by default

- C.2: Use **class** if the class has an invariant; use **struct** if the data members can vary independently
- C.8: Use **class** rather than **struct** if any member is non-public

```
class AxisAlignedRect {  
    Point p1;  
    Point p2;  
    float area() {  
        (p2.x - p1.x) * (p2.y - p1.y);  
    }  
};
```

```
class AxisAlignedRect {  
    Point p1;  
    Point p2;  
    float area() {  
        (p2.x - p1.x) * (p2.y - p1.y);  
    }  
};
```

Can't update **Point** to:

```
class Point {  
    float rho;  
    float theta;  
};
```

```
class AxisAlignedRect {  
    Point p1;  
    Point p2;  
    float area() {  
        (p2.x - p1.x) * (p2.y - p1.y);  
    }  
};
```

Can't update **Point** to:

```
class Point {  
    float rho;  
    float theta;  
};
```

- C.9: Minimize exposure of members

```
class PolarPoint {  
    float rho;  
    float theta;  
public:  
    PolarPoint(float r, float t) : rho{r}, theta{t} {}  
    PolarPoint(float t) : rho{1}, theta{t} {}  
    PolarPoint() : rho{1}, theta{0} {}  
};
```

```
class PolarPoint {  
    float rho;  
    float theta;  
public:  
    PolarPoint(float r, float t) : rho{r}, theta{t} {}  
    PolarPoint(float t) : rho{1}, theta{t} {}  
    PolarPoint() : rho{1}, theta{0} {}  
};  
  
const PolarPoint a{};  
const PolarPoint b{1.12};  
const PolarPoint c{3, M_PI_4};
```

```
class MotorController {  
    float speed;  
public:  
    void set_speed(float s) {  
        speed = s;  
        send_command();  
    }  
    ~MotorController() { set_speed(0); }  
};
```

```
class MyVector {
    int size, index;
    int* data;
public:
    MyVector() : size{10}, index{0}, data{new int[10]}
    ~MyVector() { delete[] data; }
    void push_back(int value) {
        if (index >= size) {
            int* next = new int[size * 2];
            for (int i{0}; i<size; i++) next[i] = data[i];
            size *= 2;
            delete[] data;
            data = next;
        }
        data[index++] = value;
    };
};
```

Bad

Bad

```
void send(X* x, string_view destination) {  
    auto port = open_port(destination);  
    my_mutex.lock();  
    // ...  
    send(port, x);  
    // ...  
    my_mutex.unlock();  
    close_port(port);  
    delete x;  
}
```

Good

```
class Port {  
    PortHandle port;  
public:  
    Port(string_view destination) :  
        port{open_port(destination)} { }  
    ~Port() { close_port(port); }  
    operator PortHandle() { return port; }  
  
    // port handles can't usually be cloned  
    // so disable copying and assignment  
    Port(const Port&) = delete;  
    Port& operator=(const Port&) = delete;  
};
```

Cleaned

```
void send(unique_ptr<X> x, string_view destination) {  
    Port port{destination};  
    lock_guard<mutex> guard{my_mutex};  
    // ...  
    send(port, x);  
    // ...  
}
```

Cleaned

```
void send(unique_ptr<X> x, string_view destination) {  
    Port port{destination};  
    lock_guard<mutex> guard{my_mutex};  
    // ...  
    send(port, x);  
    // ...  
}
```

- R.1: Manage resources automatically using resource handles and RAII

- Raw pointers: $\mathbf{T}^*$

- Raw pointers: **T***
- Smart pointers: **std::shared_ptr<T>**

- Raw pointers: **T***
- Smart pointers: **std::shared_ptr<T>**
- Unique pointers: **std::unique_ptr<T>**

```
class Resource {  
public:  
    Resource() { std::cout << "Acquisition\n"; }  
    ~Resource() { std::cout << "Destruction\n"; }  
};  
auto main() -> int {  
    std::cout << "start\n";  
    auto resource = std::make_shared<Resource>();  
    std::cout << "end\n";  
    return 0;  
}
```

```
void send(unique_ptr<X> x, string_view destination) {  
    Port port{destination};  
    lock_guard<mutex> guard{my_mutex};  
    // ...  
    send(port, x);  
    // ...  
}
```

- R.20: Use **unique_ptr** or **shared_ptr** to represent ownership

- R.20: Use **unique_ptr** or **shared_ptr** to represent ownership
- R.21: Prefer **unique_ptr** over **shared_ptr** unless you need to share ownership

```
class Point {  
    float abs;  
    float ord;  
public:  
    Point(float x, float y) : abs{x}, ord{y} {}  
    friend std::ostream& operator<<(  
        std::ostream& out, const Point& p) {  
        return out << p.abs << " / " << p.ord;  
    }  
};  
  
auto main() -> int {  
    std::cout << Point{1, 2} << "\n";  
    return 0;  
}
```

- Con.1: By default, make objects immutable
- Con.2: By default, make member functions **const**
- Con.3: By default, pass pointers and references to **const**s
- Con.4: Use **const** to define objects with values that do not change after construction