CRTP C++ Examples

Email · Print · WhatsApp · LinkedIn · Reddit · Twitter · Facebook · Messenger

Curious Recurring Template Pattern (CRTP) is a technique in C++ programming where a class inherits from a base class that takes itself as a template parameter. This allows compile-time polymorphism, enabling static dispatch instead of runtime dispatch. While this article might help you to provide a thought process on where CRTP fits in day-to-day coding, And, if you are an Embedded Programmer, you may run into CRTP more often. Although, static_cast<Derived *>(this).method() will also help but 90% of the complex for embedded processors or low-end CPUs with standard or dumb.

There is various material effectively accessible for "How" and "Where" on CRTP. So, I won't centre there rather address "Where" part i.e. CRTP Applicability.

Contents [hide]

1 CRTP and Static Polymorphism in C++
2 Limiting Object Count with CRTP
3 CRTP to Avoid Code Duplication
4 Modern C++ Composition Pattern Leveraging CRTP
5 C++20 Spaceship Operator With the Help of CRTP
5.1 Problem
5.2 Solution 01-C++17 with CRTP
5.3 C++20 Solution - Spaceship Operator
6 Enabling Polymorphic Method Chaining
7 Enabling Polymorphic Copy Construction in C++ with CRTP
7.1 Problem
7.2 Solution 1 - Dynamic Polymorphism
7.3 Solution 2 - Static Polymorphism
8 Wrap-Up
9 References

CRTP and Static Polymorphism in C++

```
template <typename T>
class CountedObjectBase {
protected:
    static std::atomic<unsigned_64_int> count;

    CountedObjectBase() noexcept {
        ++count;
    }
    CountedObjectBase(const CountedObjectBase&) noexcept {
        ++count;
    }
    CountedObjectBase(CountedObjectBase&&) noexcept {
        ++count;
    }
    ~CountedObjectBase() noexcept {
        --count;
    }
    static int64_t getCount() noexcept {
        return count.load();
    }
};

template <typename T>
std::atomic<unsigned_64_int> CountedObjectBase<T>::count{0};

class Foo : public CountedObjectBase<Foo> {
    int64_t* p{nullptr};
    int32_t* q{nullptr};
public:
    Foo() noexcept = default;
    int64_t* getP() const noexcept {
        return p;
    }
    int32_t* getQ() const noexcept {
        return q;
    }
    void setP(int64_t* pArg) noexcept {
        p = pArg;
    }
    void setQ(int32_t* qArg) noexcept {
        q = qArg;
    }
    int64_t* allocateP(int64_t pCount) {
        return new int64_t[pCount];
    }
    int32_t* allocateQ(int32_t qCount) {
        return new int32_t[qCount];
    }
    void deleteP(int64_t* pArg) {
        delete[] pArg;
    }
    void deleteQ(int32_t* qArg) {
        delete[] qArg;
    }
};

int64_t Foo::getCount() {
    return CountedObjectBase<Foo>::getCount();
}

int main() {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};

    Foo* pFoo = new Foo();
    pFoo->setP(pFoo->allocateP(10));
    pFoo->setQ(pFoo->allocateQ(20));
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    pFoo->deleteP(pFoo->getP());
    pFoo->deleteQ(pFoo->getQ());
    delete pFoo;
    pFoo = nullptr;
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    return EXIT_SUCCESS;
}
```

- *Curiously Recurring Template Pattern widely employed for static polymorphism* without bearing the cost of runtime polymorphism. Here, the base class is templated on the derived type and a static inherited the functionality of counting.
- How it works is not the topic of this article. So, I am leaving it to you to figure out.

Limiting Object Count with CRTP

```
template <typename T>
class LimitedObjectBase {
    static std::atomic<unsigned_64_int> count;
    static std::atomic<unsigned_64_int> maxObjectCount{std::numeric_limits<unsigned_64_int>::max()};

    LimitedObjectBase() {
        if (++count > maxObjectCount) {
            throw std::runtime_error("Too many instances");
        }
    }
    LimitedObjectBase(const LimitedObjectBase&) {
        if (++count > maxObjectCount) {
            throw std::runtime_error("Too many instances");
        }
    }
    LimitedObjectBase(LimitedObjectBase&&) {
        if (++count > maxObjectCount) {
            throw std::runtime_error("Too many instances");
        }
    }
    ~LimitedObjectBase() {
        --count;
    }
    static int64_t getCount() {
        return count.load();
    }
    static int64_t setMaxCount(int64_t maxCount) {
        int64_t oldMax = maxObjectCount.load();
        while (!maxObjectCount.compare_exchange_weak(oldMax, maxCount));
        return oldMax;
    }
};

template <typename T>
std::atomic<unsigned_64_int> LimitedObjectBase<T>::count{0};

class Foo : public LimitedObjectBase<Foo> {
    int64_t* p{nullptr};
    int32_t* q{nullptr};
public:
    Foo() = default;
    int64_t* getP() const {
        return p;
    }
    int32_t* getQ() const {
        return q;
    }
    void setP(int64_t* pArg) {
        p = pArg;
    }
    void setQ(int32_t* qArg) {
        q = qArg;
    }
    int64_t* allocateP(int64_t pCount) {
        return new int64_t[pCount];
    }
    int32_t* allocateQ(int32_t qCount) {
        return new int32_t[qCount];
    }
    void deleteP(int64_t* pArg) {
        delete[] pArg;
    }
    void deleteQ(int32_t* qArg) {
        delete[] qArg;
    }
};

int64_t Foo::getCount() {
    return LimitedObjectBase<Foo>::getCount();
}

int64_t Foo::setMaxCount(int64_t maxCount) {
    return LimitedObjectBase<Foo>::setMaxCount(maxCount);
}

int main() {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};

    Foo::setMaxCount(1);
    Foo* pFoo = new Foo();
    pFoo->setP(pFoo->allocateP(10));
    pFoo->setQ(pFoo->allocateQ(20));
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    pFoo->deleteP(pFoo->getP());
    pFoo->deleteQ(pFoo->getQ());
    delete pFoo;
    pFoo = nullptr;
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    return EXIT_SUCCESS;
}
```

- You might be wondering that what is the point of the template parameter: TodayJobs(). If it isn't used. In that case, you should review locally and you will realize the above example can be rewritten with static_cast<T>(this).

CRTP to Avoid Code Duplication

- Let's say you have a set of classes that support the functions: begin() & end(). But, the standard library's requirements for comparators require functions like: friend bool operator<(const T&, const T&);. Now, you can change the function signature to:

```
class Foo {
    int64_t* p{nullptr};
    int32_t* q{nullptr};
public:
    Foo() noexcept = default;
    int64_t* getP() const noexcept {
        return p;
    }
    int32_t* getQ() const noexcept {
        return q;
    }
    void setP(int64_t* pArg) noexcept {
        p = pArg;
    }
    void setQ(int32_t* qArg) noexcept {
        q = qArg;
    }
    int64_t* allocateP(int64_t pCount) {
        return new int64_t[pCount];
    }
    int32_t* allocateQ(int32_t qCount) {
        return new int32_t[qCount];
    }
    void deleteP(int64_t* pArg) {
        delete[] pArg;
    }
    void deleteQ(int32_t* qArg) {
        delete[] qArg;
    }
    friend bool operator<(const Foo&, const Foo&);
};

int64_t Foo::getCount() {
    return LimitedObjectBase<Foo>::getCount();
}

int64_t Foo::setMaxCount(int64_t maxCount) {
    return LimitedObjectBase<Foo>::setMaxCount(maxCount);
}

int64_t* Foo::allocateP(int64_t pCount) {
    return new int64_t[pCount];
}

int32_t* Foo::allocateQ(int32_t qCount) {
    return new int32_t[qCount];
}

void Foo::deleteP(int64_t* pArg) {
    delete[] pArg;
}

void Foo::deleteQ(int32_t* qArg) {
    delete[] qArg;
}

bool operator<(const Foo&, const Foo&) {
    return true;
}

int main() {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};

    Foo::setMaxCount(1);
    Foo* pFoo = new Foo();
    pFoo->setP(pFoo->allocateP(10));
    pFoo->setQ(pFoo->allocateQ(20));
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    pFoo->deleteP(pFoo->getP());
    pFoo->deleteQ(pFoo->getQ());
    delete pFoo;
    pFoo = nullptr;
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    return EXIT_SUCCESS;
}
```

Modern C++ Composition Pattern Leveraging CRTP

- Composition Design Pattern states that we should *treat the group of objects in the same manner as a single object*. And for implementation, we can leverage the CRTP.
- For example, as a part of machine learning, we have to deal with Neuron which for simplicity defined as:

```
class Neuron {
    int64_t* pOutputs{nullptr};
    int64_t* pInputs{nullptr};
    int64_t* pWeights{nullptr};
    int64_t* pDelta{nullptr};
    int64_t* pGradient{nullptr};
    int64_t outputCount{0};
    int64_t inputCount{0};
    int64_t weightCount{0};
    int64_t deltaCount{0};
    int64_t gradientCount{0};
    int64_t* allocateArray(int64_t pCount) {
        return new int64_t[pCount];
    }
    void deleteArray(int64_t* pArg) {
        delete[] pArg;
    }
    int64_t* getOutputs() {
        return pOutputs;
    }
    int64_t* getInputs() {
        return pInputs;
    }
    int64_t* getWeights() {
        return pWeights;
    }
    int64_t* getDelta() {
        return pDelta;
    }
    int64_t* getGradient() {
        return pGradient;
    }
    int64_t getOutputCount() {
        return outputCount;
    }
    int64_t getInputCount() {
        return inputCount;
    }
    int64_t getWeightCount() {
        return weightCount;
    }
    int64_t getDeltaCount() {
        return deltaCount;
    }
    int64_t getGradientCount() {
        return gradientCount;
    }
    void setOutputs(int64_t* pArg, int64_t pCount) {
        pOutputs = pArg;
        outputCount = pCount;
    }
    void setInputs(int64_t* pArg, int64_t pCount) {
        pInputs = pArg;
        inputCount = pCount;
    }
    void setWeights(int64_t* pArg, int64_t pCount) {
        pWeights = pArg;
        weightCount = pCount;
    }
    void setDelta(int64_t* pArg, int64_t pCount) {
        pDelta = pArg;
        deltaCount = pCount;
    }
    void setGradient(int64_t* pArg, int64_t pCount) {
        pGradient = pArg;
        gradientCount = pCount;
    }
    void deleteOutputs(int64_t* pArg) {
        deleteArray(pArg);
    }
    void deleteInputs(int64_t* pArg) {
        deleteArray(pArg);
    }
    void deleteWeights(int64_t* pArg) {
        deleteArray(pArg);
    }
    void deleteDelta(int64_t* pArg) {
        deleteArray(pArg);
    }
    void deleteGradient(int64_t* pArg) {
        deleteArray(pArg);
    }
};

int64_t* Neuron::allocateArray(int64_t pCount);
void Neuron::deleteArray(int64_t* pArg);
int64_t* Neuron::getOutputs();
int64_t* Neuron::getInputs();
int64_t* Neuron::getWeights();
int64_t* Neuron::getDelta();
int64_t* Neuron::getGradient();
int64_t Neuron::getOutputCount();
int64_t Neuron::getInputCount();
int64_t Neuron::getWeightCount();
int64_t Neuron::getDeltaCount();
int64_t Neuron::getGradientCount();
void Neuron::setOutputs(int64_t* pArg, int64_t pCount);
void Neuron::setInputs(int64_t* pArg, int64_t pCount);
void Neuron::setWeights(int64_t* pArg, int64_t pCount);
void Neuron::setDelta(int64_t* pArg, int64_t pCount);
void Neuron::setGradient(int64_t* pArg, int64_t pCount);
void Neuron::deleteOutputs(int64_t* pArg);
void Neuron::deleteInputs(int64_t* pArg);
void Neuron::deleteWeights(int64_t* pArg);
void Neuron::deleteDelta(int64_t* pArg);
void Neuron::deleteGradient(int64_t* pArg);
```

- And there is also a NeuronLayer i.e. collection of Neuron which for simplicity defined as:

```
class NeuronLayer {
    std::vector<Neuron> neurons{};
    int64_t* pOutputs{nullptr};
    int64_t* pInputs{nullptr};
    int64_t* pWeights{nullptr};
    int64_t* pDelta{nullptr};
    int64_t* pGradient{nullptr};
    int64_t outputCount{0};
    int64_t inputCount{0};
    int64_t weightCount{0};
    int64_t deltaCount{0};
    int64_t gradientCount{0};
    int64_t* allocateArray(int64_t pCount);
    void deleteArray(int64_t* pArg);
    int64_t* getOutputs();
    int64_t* getInputs();
    int64_t* getWeights();
    int64_t* getDelta();
    int64_t* getGradient();
    int64_t getOutputCount();
    int64_t getInputCount();
    int64_t getWeightCount();
    int64_t getDeltaCount();
    int64_t getGradientCount();
    void setOutputs(int64_t* pArg, int64_t pCount);
    void setInputs(int64_t* pArg, int64_t pCount);
    void setWeights(int64_t* pArg, int64_t pCount);
    void setDelta(int64_t* pArg, int64_t pCount);
    void setGradient(int64_t* pArg, int64_t pCount);
    void deleteOutputs(int64_t* pArg);
    void deleteInputs(int64_t* pArg);
    void deleteWeights(int64_t* pArg);
    void deleteDelta(int64_t* pArg);
    void deleteGradient(int64_t* pArg);
};

int64_t* NeuronLayer::allocateArray(int64_t pCount);
void NeuronLayer::deleteArray(int64_t* pArg);
int64_t* NeuronLayer::getOutputs();
int64_t* NeuronLayer::getInputs();
int64_t* NeuronLayer::getWeights();
int64_t* NeuronLayer::getDelta();
int64_t* NeuronLayer::getGradient();
int64_t NeuronLayer::getOutputCount();
int64_t NeuronLayer::getInputCount();
int64_t NeuronLayer::getWeightCount();
int64_t NeuronLayer::getDeltaCount();
int64_t NeuronLayer::getGradientCount();
void NeuronLayer::setOutputs(int64_t* pArg, int64_t pCount);
void NeuronLayer::setInputs(int64_t* pArg, int64_t pCount);
void NeuronLayer::setWeights(int64_t* pArg, int64_t pCount);
void NeuronLayer::setDelta(int64_t* pArg, int64_t pCount);
void NeuronLayer::setGradient(int64_t* pArg, int64_t pCount);
void NeuronLayer::deleteOutputs(int64_t* pArg);
void NeuronLayer::deleteInputs(int64_t* pArg);
void NeuronLayer::deleteWeights(int64_t* pArg);
void NeuronLayer::deleteDelta(int64_t* pArg);
void NeuronLayer::deleteGradient(int64_t* pArg);
```

- You see this is state-space explosion/implosion in layman terms problem and it's not good. Because we want a single function that enumerate both the layer as well as individual neurons. CRTP comes handy here as:

```
template <typename T>
class Iterable {
    int64_t* pOutputs{nullptr};
    int64_t* pInputs{nullptr};
    int64_t* pWeights{nullptr};
    int64_t* pDelta{nullptr};
    int64_t* pGradient{nullptr};
    int64_t outputCount{0};
    int64_t inputCount{0};
    int64_t weightCount{0};
    int64_t deltaCount{0};
    int64_t gradientCount{0};
    int64_t* allocateArray(int64_t pCount);
    void deleteArray(int64_t* pArg);
    int64_t* getOutputs();
    int64_t* getInputs();
    int64_t* getWeights();
    int64_t* getDelta();
    int64_t* getGradient();
    int64_t getOutputCount();
    int64_t getInputCount();
    int64_t getWeightCount();
    int64_t getDeltaCount();
    int64_t getGradientCount();
    void setOutputs(int64_t* pArg, int64_t pCount);
    void setInputs(int64_t* pArg, int64_t pCount);
    void setWeights(int64_t* pArg, int64_t pCount);
    void setDelta(int64_t* pArg, int64_t pCount);
    void setGradient(int64_t* pArg, int64_t pCount);
    void deleteOutputs(int64_t* pArg);
    void deleteInputs(int64_t* pArg);
    void deleteWeights(int64_t* pArg);
    void deleteDelta(int64_t* pArg);
    void deleteGradient(int64_t* pArg);
    friend bool operator<(const Iterable&, const Iterable&);
};

int64_t* Iterable::allocateArray(int64_t pCount);
void Iterable::deleteArray(int64_t* pArg);
int64_t* Iterable::getOutputs();
int64_t* Iterable::getInputs();
int64_t* Iterable::getWeights();
int64_t* Iterable::getDelta();
int64_t* Iterable::getGradient();
int64_t Iterable::getOutputCount();
int64_t Iterable::getInputCount();
int64_t Iterable::getWeightCount();
int64_t Iterable::getDeltaCount();
int64_t Iterable::getGradientCount();
void Iterable::setOutputs(int64_t* pArg, int64_t pCount);
void Iterable::setInputs(int64_t* pArg, int64_t pCount);
void Iterable::setWeights(int64_t* pArg, int64_t pCount);
void Iterable::setDelta(int64_t* pArg, int64_t pCount);
void Iterable::setGradient(int64_t* pArg, int64_t pCount);
void Iterable::deleteOutputs(int64_t* pArg);
void Iterable::deleteInputs(int64_t* pArg);
void Iterable::deleteWeights(int64_t* pArg);
void Iterable::deleteDelta(int64_t* pArg);
void Iterable::deleteGradient(int64_t* pArg);
bool operator<(const Iterable&, const Iterable&);

class Neuron : public Iterable {
    int64_t* pOutputs{nullptr};
    int64_t* pInputs{nullptr};
    int64_t* pWeights{nullptr};
    int64_t* pDelta{nullptr};
    int64_t* pGradient{nullptr};
    int64_t outputCount{0};
    int64_t inputCount{0};
    int64_t weightCount{0};
    int64_t deltaCount{0};
    int64_t gradientCount{0};
    int64_t* allocateArray(int64_t pCount);
    void deleteArray(int64_t* pArg);
    int64_t* getOutputs();
    int64_t* getInputs();
    int64_t* getWeights();
    int64_t* getDelta();
    int64_t* getGradient();
    int64_t getOutputCount();
    int64_t getInputCount();
    int64_t getWeightCount();
    int64_t getDeltaCount();
    int64_t getGradientCount();
    void setOutputs(int64_t* pArg, int64_t pCount);
    void setInputs(int64_t* pArg, int64_t pCount);
    void setWeights(int64_t* pArg, int64_t pCount);
    void setDelta(int64_t* pArg, int64_t pCount);
    void setGradient(int64_t* pArg, int64_t pCount);
    void deleteOutputs(int64_t* pArg);
    void deleteInputs(int64_t* pArg);
    void deleteWeights(int64_t* pArg);
    void deleteDelta(int64_t* pArg);
    void deleteGradient(int64_t* pArg);
};

int64_t* Neuron::allocateArray(int64_t pCount);
void Neuron::deleteArray(int64_t* pArg);
int64_t* Neuron::getOutputs();
int64_t* Neuron::getInputs();
int64_t* Neuron::getWeights();
int64_t* Neuron::getDelta();
int64_t* Neuron::getGradient();
int64_t Neuron::getOutputCount();
int64_t Neuron::getInputCount();
int64_t Neuron::getWeightCount();
int64_t Neuron::getDeltaCount();
int64_t Neuron::getGradientCount();
void Neuron::setOutputs(int64_t* pArg, int64_t pCount);
void Neuron::setInputs(int64_t* pArg, int64_t pCount);
void Neuron::setWeights(int64_t* pArg, int64_t pCount);
void Neuron::setDelta(int64_t* pArg, int64_t pCount);
void Neuron::setGradient(int64_t* pArg, int64_t pCount);
void Neuron::deleteOutputs(int64_t* pArg);
void Neuron::deleteInputs(int64_t* pArg);
void Neuron::deleteWeights(int64_t* pArg);
void Neuron::deleteDelta(int64_t* pArg);
void Neuron::deleteGradient(int64_t* pArg);
bool operator<(const Neuron&, const Neuron&);

class NeuronLayer : public Iterable {
    std::vector<Neuron> neurons{};
    int64_t* pOutputs{nullptr};
    int64_t* pInputs{nullptr};
    int64_t* pWeights{nullptr};
    int64_t* pDelta{nullptr};
    int64_t* pGradient{nullptr};
    int64_t outputCount{0};
    int64_t inputCount{0};
    int64_t weightCount{0};
    int64_t deltaCount{0};
    int64_t gradientCount{0};
    int64_t* allocateArray(int64_t pCount);
    void deleteArray(int64_t* pArg);
    int64_t* getOutputs();
    int64_t* getInputs();
    int64_t* getWeights();
    int64_t* getDelta();
    int64_t* getGradient();
    int64_t getOutputCount();
    int64_t getInputCount();
    int64_t getWeightCount();
    int64_t getDeltaCount();
    int64_t getGradientCount();
    void setOutputs(int64_t* pArg, int64_t pCount);
    void setInputs(int64_t* pArg, int64_t pCount);
    void setWeights(int64_t* pArg, int64_t pCount);
    void setDelta(int64_t* pArg, int64_t pCount);
    void setGradient(int64_t* pArg, int64_t pCount);
    void deleteOutputs(int64_t* pArg);
    void deleteInputs(int64_t* pArg);
    void deleteWeights(int64_t* pArg);
    void deleteDelta(int64_t* pArg);
    void deleteGradient(int64_t* pArg);
};

int64_t* NeuronLayer::allocateArray(int64_t pCount);
void NeuronLayer::deleteArray(int64_t* pArg);
int64_t* NeuronLayer::getOutputs();
int64_t* NeuronLayer::getInputs();
int64_t* NeuronLayer::getWeights();
int64_t* NeuronLayer::getDelta();
int64_t* NeuronLayer::getGradient();
int64_t NeuronLayer::getOutputCount();
int64_t NeuronLayer::getInputCount();
int64_t NeuronLayer::getWeightCount();
int64_t NeuronLayer::getDeltaCount();
int64_t NeuronLayer::getGradientCount();
void NeuronLayer::setOutputs(int64_t* pArg, int64_t pCount);
void NeuronLayer::setInputs(int64_t* pArg, int64_t pCount);
void NeuronLayer::setWeights(int64_t* pArg, int64_t pCount);
void NeuronLayer::setDelta(int64_t* pArg, int64_t pCount);
void NeuronLayer::setGradient(int64_t* pArg, int64_t pCount);
void NeuronLayer::deleteOutputs(int64_t* pArg);
void NeuronLayer::deleteInputs(int64_t* pArg);
void NeuronLayer::deleteWeights(int64_t* pArg);
void NeuronLayer::deleteDelta(int64_t* pArg);
void NeuronLayer::deleteGradient(int64_t* pArg);
```

- As you can see we have covered all four different permutation scenarios using a single IterableName::commonJob method. And both Neuron & NeuronLayer conforms to this interface via self-implementation.

C++20 Spaceship Operator With the Help of CRTP

Problem

```
class Foo {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};
    int64_t* allocateP(int64_t pCount);
    int32_t* allocateQ(int32_t qCount);
    void deleteP(int64_t* pArg);
    void deleteQ(int32_t* qArg);
    int64_t* getP();
    int32_t* getQ();
    int64_t getFooCount();
    void setP(int64_t* pArg);
    void setQ(int32_t* qArg);
    void setFooCount(int64_t pCount);
    friend bool operator<(const Foo&, const Foo&);
};

int64_t* Foo::allocateP(int64_t pCount);
int32_t* Foo::allocateQ(int32_t qCount);
void Foo::deleteP(int64_t* pArg);
void Foo::deleteQ(int32_t* qArg);
int64_t* Foo::getP();
int32_t* Foo::getQ();
int64_t Foo::getFooCount();
void Foo::setP(int64_t* pArg);
void Foo::setQ(int32_t* qArg);
void Foo::setFooCount(int64_t pCount);
bool operator<(const Foo&, const Foo&);

int64_t* Foo::allocateP(int64_t pCount) {
    return new int64_t[pCount];
}

int32_t* Foo::allocateQ(int32_t qCount) {
    return new int32_t[qCount];
}

void Foo::deleteP(int64_t* pArg) {
    delete[] pArg;
}

void Foo::deleteQ(int32_t* qArg) {
    delete[] qArg;
}

int64_t* Foo::getP() {
    return pFoo;
}

int32_t* Foo::getQ() {
    return qFoo;
}

int64_t Foo::getFooCount() {
    return fooCount;
}

void Foo::setP(int64_t* pArg) {
    pFoo = pArg;
}

void Foo::setQ(int32_t* qArg) {
    qFoo = qArg;
}

void Foo::setFooCount(int64_t pCount) {
    fooCount = pCount;
}

bool operator<(const Foo&, const Foo&) {
    return true;
}

int main() {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};

    Foo::setMaxCount(1);
    Foo* pFoo = new Foo();
    pFoo->setP(pFoo->allocateP(10));
    pFoo->setQ(pFoo->allocateQ(20));
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    pFoo->deleteP(pFoo->getP());
    pFoo->deleteQ(pFoo->getQ());
    delete pFoo;
    pFoo = nullptr;
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    return EXIT_SUCCESS;
}
```

C++20 Solution - Spaceship Operator

```
class Foo {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};
    auto operator<=>(const Foo&, const Foo&) = default;
};
```

Enabling Polymorphic Method Chaining

- *Method Chaining* is a common syntax for invoking multiple methods on a single object back to back. That, too, in a single statement without requiring variables to store the intermediate results. For example:

```
class Foo {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};
    int64_t* allocateP(int64_t pCount);
    int32_t* allocateQ(int32_t qCount);
    void deleteP(int64_t* pArg);
    void deleteQ(int32_t* qArg);
    int64_t* getP();
    int32_t* getQ();
    int64_t getFooCount();
    void setP(int64_t* pArg);
    void setQ(int32_t* qArg);
    void setFooCount(int64_t pCount);
    friend bool operator<(const Foo&, const Foo&);
};

int64_t* Foo::allocateP(int64_t pCount);
int32_t* Foo::allocateQ(int32_t qCount);
void Foo::deleteP(int64_t* pArg);
void Foo::deleteQ(int32_t* qArg);
int64_t* Foo::getP();
int32_t* Foo::getQ();
int64_t Foo::getFooCount();
void Foo::setP(int64_t* pArg);
void Foo::setQ(int32_t* qArg);
void Foo::setFooCount(int64_t pCount);
bool operator<(const Foo&, const Foo&);

int64_t* Foo::allocateP(int64_t pCount) {
    return new int64_t[pCount];
}

int32_t* Foo::allocateQ(int32_t qCount) {
    return new int32_t[qCount];
}

void Foo::deleteP(int64_t* pArg) {
    delete[] pArg;
}

void Foo::deleteQ(int32_t* qArg) {
    delete[] qArg;
}

int64_t* Foo::getP() {
    return pFoo;
}

int32_t* Foo::getQ() {
    return qFoo;
}

int64_t Foo::getFooCount() {
    return fooCount;
}

void Foo::setP(int64_t* pArg) {
    pFoo = pArg;
}

void Foo::setQ(int32_t* qArg) {
    qFoo = qArg;
}

void Foo::setFooCount(int64_t pCount) {
    fooCount = pCount;
}

bool operator<(const Foo&, const Foo&) {
    return true;
}

int main() {
    int64_t* pFoo{nullptr};
    int32_t* qFoo{nullptr};
    int64_t fooCount{0};

    Foo::setMaxCount(1);
    Foo* pFoo = new Foo();
    pFoo->setP(pFoo->allocateP(10));
    pFoo->setQ(pFoo->allocateQ(20));
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    pFoo->deleteP(pFoo->getP());
    pFoo->deleteQ(pFoo->getQ());
    delete pFoo;
    pFoo = nullptr;
    fooCount = Foo::getCount();
    std::cout << "Foo count: " << fooCount << std::endl;
    return EXIT_SUCCESS;
}
```

- Compiling above code prompt you with the following error:

```
/home/shubham/Desktop/Projects/CodeFormatter/cmake-build-debug/CMakeFiles/CodeFormatter.dir/main.o: In function `Foo::setP(long*)':
main.cpp:(.text._ZN3Foo3setPEPl[_ZN3Foo3setPEPl]+0x0): multiple definition of `Foo::setP(long*)'; /home/shubham/Desktop/Projects/CodeFormatter/cmake-build-debug/CMakeFiles/CodeFormatter.dir/main.o:main.cpp:(.text._ZN3Foo3setPEPl[_ZN3Foo3setPEPl]+0x0)
```

- This happens because we "lose" the concrete class as soon as we invoke a function of the base class.
- The CRTP can be useful to avoid such problem and to enable Polymorphic Method Chaining.

```
template <typename T>
class Base {
    int64_t* pBaseObj{nullptr};
public:
    int64_t* getBaseObj() const noexcept {
        return pBaseObj;
    }
    void setBaseObj(int64_t* pArg) noexcept {
        pBaseObj = pArg;
    }
    T& setBaseObjAndReturnSelf(int64_t* pArg) {
        pBaseObj = pArg;
        return static_cast<T&>(*this);
    }
    int64_t* allocateBaseObj(int64_t pCount) {
        return new int64_t[pCount];
    }
    void deleteBaseObj(int64_t* pArg) {
        delete[] pArg;
    }
    T& setMyFieldAndReturnSelf(int64_t pArg) {
        static_cast<T&>(*this).myField = pArg;
        return static_cast<T&>(*this);
    }
    int64_t getMyFieldFromBase() {
        return static_cast<T&>(*this).myField;
    }
    int64_t* getBaseObjFromChild() {
        return static_cast<T&>(*this).getBaseObj();
    }
    void setBaseObjFromChild(int64_t* pArg) {
        static_cast<T&>(*this).setBaseObj(pArg);
    }
    T& setBaseObjAndReturnSelfFromChild(int64_t* pArg) {
        static_cast<T&>(*this).setBaseObj(pArg);
        return static_cast<T&>(*this);
    }
    int64_t* allocateBaseObjFromChild(int64_t pCount) {
        return static_cast<T&>(*this).allocateBaseObj(pCount);
    }
    void deleteBaseObjFromChild(int64_t* pArg) {
        static_cast<T&>(*this).deleteBaseObj(pArg);
    }
    T& setMyFieldAndReturnSelfFromChild(int64_t pArg) {
        static_cast<T&>(*this).setMyFieldAndReturnSelf(pArg);
        return static_cast<T&>(*this);
    }
    int64_t getMyFieldFromBaseFromChild() {
        return static_cast<T&>(*this).getMyFieldFromBase();
    }
};

class Child : public Base<Child> {
    int64_t myField{0};
    int64_t* pChildObj{nullptr};
    int64_t* allocateChildObj(int64_t pCount);
    void deleteChildObj(int64_t* pArg);
    int64_t* getChildObj();
    void setChildObj(int64_t* pArg);
    int64_t* allocateChildObjFromBase(int64_t pCount);
    void deleteChildObjFromBase(int64_t* pArg);
    int64_t* getChildObjFromBase();
    void setChildObjFromBase(int64_t* pArg);
    friend bool operator<(const Child&, const Child&);
};

int64_t* Child::allocateChildObj(int64_t pCount);
void Child::deleteChildObj(int64_t* pArg);
int64_t* Child::getChildObj();
void Child::setChildObj(int64_t* pArg);
int64_t* Child::allocateChildObjFromBase(int64_t pCount);
void Child::deleteChildObjFromBase(int64_t* pArg);
int64_t* Child::getChildObjFromBase();
void Child::setChildObjFromBase(int64_t* pArg);
bool operator<(const Child&, const Child&);

int64_t* Child::allocateChildObj(int64_t pCount) {
    return new int64_t[pCount];
}

void Child::deleteChildObj(int64_t* pArg) {
    delete[] pArg;
}

int64_t* Child::getChildObj() {
    return pChildObj;
}

void Child::setChildObj(int64_t* pArg) {
    pChildObj = pArg;
}

int64_t* Child::allocateChildObjFromBase(int64_t pCount) {
    return allocateChildObj(pCount);
}

void Child::deleteChildObjFromBase(int64_t* pArg) {
    deleteChildObj(pArg);
}

int64_t* Child::getChildObjFromBase() {
    return getChildObj();
}

void Child::setChildObjFromBase(int64_t* pArg) {
    setChildObj(pArg);
}

bool operator<(const Child&, const Child&) {
    return true;
}

int main() {
    int64_t* pBaseObj{nullptr};
    int64_t* pChildObj{nullptr};
    int64_t myFieldFromBase{0};

    Child* pChild = new Child();
    pBaseObj = pChild->allocateBaseObj(10);
    pChild->setBaseObj(pBaseObj);
    pChild->setMyFieldAndReturnSelf(20);
    myFieldFromBase = pChild->getMyFieldFromBase();
    std::cout << "myFieldFromBase: " << myFieldFromBase << std::endl;
    pChildObj = pChild->allocateChildObjFromBase(30);
    pChild->setChildObjFromBase(pChildObj);
    pChild->setMyFieldAndReturnSelfFromChild(40);
    myFieldFromBase = pChild->getMyFieldFromBaseFromChild();
    std::cout << "myFieldFromBase: " << myFieldFromBase << std::endl;
    pBaseObj = pChild->getBaseObjFromChild();
    pChild->deleteBaseObjFromChild(pBaseObj);
    pChildObj = pChild->getChildObjFromBase();
    pChild->deleteChildObjFromBase(pChildObj);
    delete pChild;
    pChild = nullptr;
    return EXIT_SUCCESS;
}
```

Enabling Polymorphic Copy Construction in C++ with CRTP

Problem

- C++ has the support of polymorphic object destruction using it's base class's virtual destructor, but, equivalent support for creation and copying of objects is missing as C++ doesn't support virtual constructor/copy-constructor. Although, there are workarounds, but, they are not pretty. Also, the workarounds require dynamic memory allocation of space it needs to allocate. For the same reason, copy of an object may require its type to known at compile-time.

```
class Base {
    int64_t* pBaseObj{nullptr};
    int64_t* allocateBaseObj(int64_t pCount);
    void deleteBaseObj(int64_t* pArg);
    int64_t* getBaseObj();
    void setBaseObj(int64_t* pArg);
    friend bool operator<(const Base&, const Base&);
};

int64_t* Base::allocateBaseObj(int64_t pCount);
void Base::deleteBaseObj(int64_t* pArg);
int64_t* Base::getBaseObj();
void Base::setBaseObj(int64_t* pArg);
bool operator<(const Base&, const Base&);

int64_t* Base::allocateBaseObj(int64_t pCount) {
    return new int64_t[pCount];
}

void Base::deleteBaseObj(int64_t* pArg) {
    delete[] pArg;
}

int64_t* Base::getBaseObj() {
    return pBaseObj;
}

void Base::setBaseObj(int64_t* pArg) {
    pBaseObj = pArg;
}

bool operator<(const Base&, const Base&) {
    return true;
}

int64_t* pBaseObj{nullptr};

Base* pBase = new Base();
pBaseObj = pBase->allocateBaseObj(10);
pBase->setBaseObj(pBaseObj);
Base* pBaseCopy = new Base(*pBase);
std::cout << "Base copy created" << std::endl;
pBaseObj = pBase->getBaseObj();
pBase->deleteBaseObj(pBaseObj);
delete pBase;
pBase = nullptr;
pBaseObj = pBaseCopy->getBaseObj();
pBaseCopy->deleteBaseObj(pBaseObj);
delete pBaseCopy;
pBaseCopy = nullptr;
```

Solution 1 - Dynamic Polymorphism

- As the name suggests, we will use virtual methods to delegate the act of copying/another creation of the object as below:

```
class Base {
    int64_t* pBaseObj{nullptr};
    int64_t* allocateBaseObj(int64_t pCount);
    void deleteBaseObj(int64_t* pArg);
    int64_t* getBaseObj();
    void setBaseObj(int64_t* pArg);
    friend bool operator<(const Base&, const Base&);
    Base* getNewObjPtr();
    Base* getCopyObjPtr(Base* pArg);
};

int64_t* Base::allocateBaseObj(int64_t pCount);
void Base::deleteBaseObj(int64_t* pArg);
int64_t* Base::getBaseObj();
void Base::setBaseObj(int64_t* pArg);
bool operator<(const Base&, const Base&);
Base* Base::getNewObjPtr();
Base* Base::getCopyObjPtr(Base* pArg);

int64_t* Base::allocateBaseObj(int64_t pCount) {
    return new int64_t[pCount];
}

void Base::deleteBaseObj(int64_t* pArg) {
    delete[] pArg;
}

int64_t* Base::getBaseObj() {
    return pBaseObj;
}

void Base::setBaseObj(int64_t* pArg) {
    pBaseObj = pArg;
}

bool operator<(const Base&, const Base&) {
    return true;
}

Base* Base::getNewObjPtr() {
    return new Base();
}

Base* Base::getCopyOb
```