Concept-Based Polymorphism in C++

Gladilov, Gleb

Runtime polymorphism

```
unique_ptr<Base> base = make_unique<Derived>();
base->foo(); // Derived::foo
set<unique_ptr<Base>> bases;  // What if we want to store an int in bases?
```

- Supports heterogeneous data structures
- Intrusive, less reusable
- Always heap allocated
- Calls to client classes methods are often indirect through virtual calls, including destructor

Compile-time polymorhism

```
template<class T>
class MyVector;

MyVector<int> v0;
MyVector<unique_ptr<Base>> v1;
```

- Non-intrusive, more reusable
- Does not support heterogeneous data structures

Document example

```
using Object   = int;
using Document = vector<Object>;

void print(Object const& object, size_t indentation) {
    cout << string(indentation, ' ') << object << '\n';
}

void print(Document const& document, size_t indentation) {
    cout << string(indentation, ' ') << "<document>" << '\n';
    for (auto const& object : document) {
        print(object, indentation + 2);
    }
    cout << string(indentation, ' ') << "</document>" << '\n';
}
```

```cpp
int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back(1);

    print(document, 0);

    return 0;
}
```

```cpp
// <document>
//  0
//  1
// </document>
```

```
void print(int const& object, size_t indentation) {
    cout << string(identation, ' ') << object << '\n';
}
```

```
class Object {
public:
    Object(int const& x) : self(x) {}
    friend void print(Object const& object, size_t indentation) {
        print(self, indentation);
    }
private:
    int self;
};
```

```cpp
int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back(1);

    print(document, 0);

    return 0;
}
```

```cpp
// <document>
//  0
//  1
// </document>
```

```
void print(int const& object, size_t indentation) {
    cout << string(identation, ' ') << object << '\n';
}


class Object {
public:
    Object(int const& x) : self(make_unique<IntModel>(x)) {}
    friend void print(Object const& object, size_t indentation) {
        object.self->print_(indentation);
    }
private:
    struct IntModel {
        IntModel(int const& x) : data(x) {}
        void print_(size_t indentation) const { print(data, indentation); }
        int data;
    };
    unique_ptr<IntModel> self;
};
```

```cpp
int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back(1);

    print(document, 0);

    return 0;
}
```

```cpp
// <document>
//  0
//  1
// </document>
```

```
class Object {
public:
    Object(int const& x) : self(make_unique<IntModel>(x)) {}
    Object(Object const& x) : self(make_unique<IntModel>(*x.self)) {}
    Object(Object&&) noexcept = default;
    Object& operator=(Object const& x) {
        Object copy(x); *this = move(copy); return *this;
    }
    Object& operator=(Object&&) noexcept = default;
    friend void print(Object const& object, size_t indentation) {
        object.self->print_(indentation);
    }
private:
    struct IntModel {
        IntModel(int const& x) : data(x) {}
        void print_(size_t indentation) const { print(data, indentation); }
        int data;
    };
    unique_ptr<IntModel> self;
};
```

```cpp
int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back(1);

    print(document, 0);

    return 0;
}
```

```
// <document>
//  0
//  1
// </document>
```

```
class Object {
public:
    Object(int x) : self(make_unique<IntModel>(move(x))) {}
    Object(Object const& x) : self(make_unique<IntModel>(*x.self)) {}
    Object(Object&&) noexcept = default;
    Object& operator=(Object const& x) {
        Object copy(x); *this = move(copy); return *this;
    }
    Object& operator=(Object&&) noexcept = default;
    friend void print(Object const& object, size_t indentation) {
        object.self->print_(indentation);
    }
private:
    struct IntModel {
        IntModel(int x) : data(move(x)) {}
        void print_(size_t indentation) const { print(data, indentation); }
        int data;
    };
    unique_ptr<IntModel> self;
};
```

```
void print(string const& object, size_t indentation) {
    cout << string(indentation, ' ') << object << '\n';
}

class Object {
public:
    Object(string x) : self(make_unique<StringModel>(move(x))) {}
    Object(int x) : self(make_unique<IntModel>(move(x))) {}
    Object(Object const& x) : self(make_unique<IntModel>(*x.self)) {}
    …
private:
    struct StringModel {
        StringModel(string x) : data(move(x)) {}
        void print_(size_t indentation) const { print(data, indentation); }
        string data;
    };
    struct IntModel { … };
    unique_ptr<IntModel> self;
};
```

```
class Object {
public:
    Object(string x) : self(make_unique<StringModel>(move(x))) {}
    Object(int x) : self(make_unique<IntModel>(move(x))) {}
    Object(Object const& x) : self(make_unique<IntModel>(*x.self)) {}
    …
private:
    struct Concept {
        virtual ~Concept() = default;
        virtual void print_(size_t indentation) const = 0;
    };
    struct StringModel : public Concept { … };
    struct IntModel : public Concept { … };
    unique_ptr<Concept> self;
};
```

```
class Object {
public:
    Object(string x) : self(make_unique<StringModel>(move(x))) {}
    Object(int x) : self(make_unique<IntModel>(move(x))) {}
    Object(Object const& x) : self(x.self->copy()) {}
    …
private:
    struct Concept {
        virtual ~Concept() = default;
        virtual void print_(size_t indentation) const = 0;
    };
    struct StringModel : public Concept { … };
    struct IntModel : public Concept { … };
    unique_ptr<Concept> self;
};
```

```
class Object {
public:
    …
    Object(Object const& x) : self(x.self->copy()) {}
private:
    struct Concept {
        virtual ~Concept() = default;
        virtual void print_(size_t indentation) const = 0;
        virtual unique_ptr<Concept> copy() const = 0;
    };
    struct StringModel : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<StringModel>(*this); }
    };
    struct IntModel : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<IntModel>(*this); }
    };
    unique_ptr<Concept> self;
};
```

```cpp
int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back("text");
    document.emplace_back(1);

    print(document, 0);

    return 0;
}
```

```
// <document>
//  0
//  text
//  1
// </document>
```

```
class Object {
public:
    Object(string x) : self(make_unique<StringModel>(move(x))) {}
    Object(int x) : self(make_unique<IntModel>(move(x))) {}
    …
private:
    struct Concept { … };
    struct StringModel : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<StringModel>(*this); }
        string data;
    };
    struct IntModel : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<IntModel>(*this); }
        int data
    };
    unique_ptr<Concept> self;
};
```

```
class Object {
public:
    template<class T>
    Object(T x) : self(make_unique<Model<T>>(move(x))) {}
    …
private:
    struct Concept { … };
    template<class T>
    struct Model : public Concept {
        Model(T x) : data(move(x)) {}
        void print_(size_t indentation) const override { print(data, indentation); }
        unique_ptr<Concept> copy() const override { return make_unique<Model<T>>(*this); }
        T data;
    };
    unique_ptr<Concept> self;
};
```

```cpp
struct MyClass {};
void print(MyClass const& object, size_t indentation) {
    cout << string(indentation, ' ') << "MyClass" << '\n';
}

int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back("text");
    document.emplace_back(1);
    document.emplace_back(MyClass{});

    print(document, 0);

    return 0;
}
```

```
// <document>
//  0
//  text
//  1
//  MyClass
// </document>
```

```
struct MyClass {};
void print(MyClass const& object, size_t indentation) {
    cout << string(indentation, ' ') << "MyClass" << '\n';
}

int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back("text");
    document.emplace_back(1);
    document.emplace_back(document);
    document.emplace_back(MyClass{});

    print(document, 0);

    return 0;
}
```

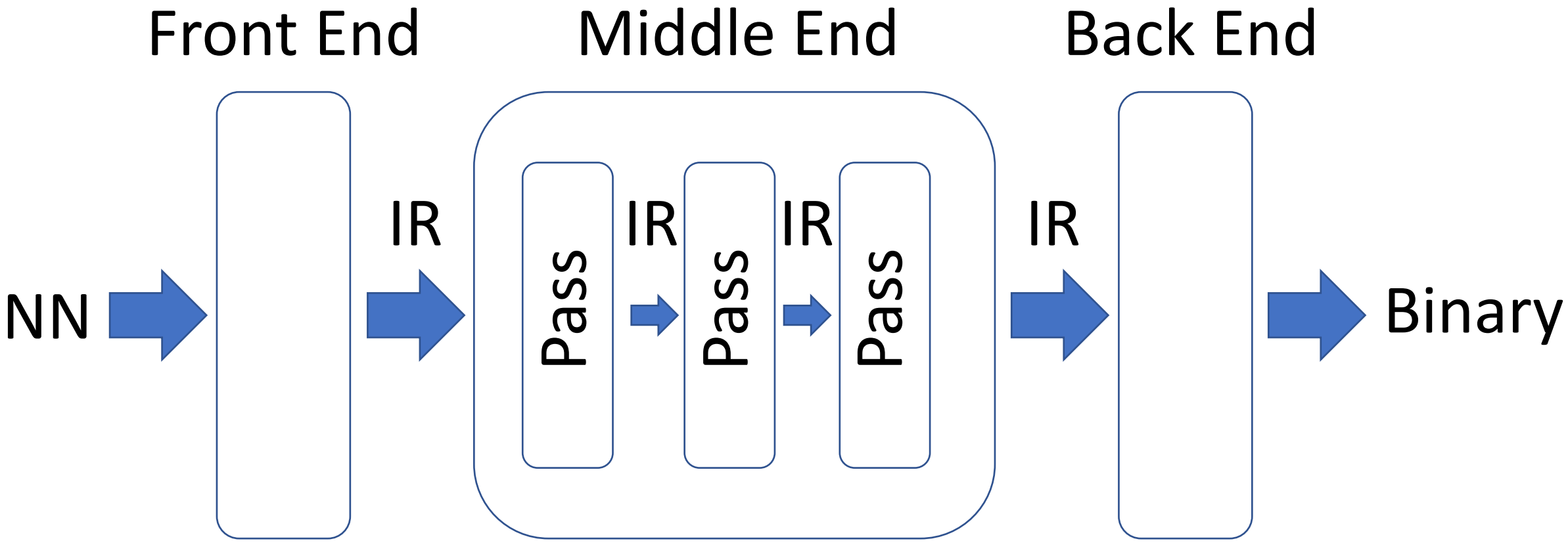
```cpp
struct MyClass {};
void print(MyClass const& object, size_t indentation) {
    cout << string(indentation, ' ') << "MyClass" << '\n';
}

int main() {
    Document document;
    document.emplace_back(0);
    document.emplace_back("text");
    document.emplace_back(1);
    document.emplace_back(document);
    document.emplace_back(MyClass{});

    print(document, 0);

    return 0;
}
```

```cpp
// <document>
//   0
//   text
//   1
//   <document>
//     0
//     text
//     1
//   </document>
//   MyClass
// </document>
```


Front End
Middle End
Back End
NN
IR
Pass
IR
Pass
IR
Pass
IR
Binary

```
template <class IR, class... Args>
struct PassConcept {
     virtual ~PassConcept() = default;
     virtual void run(IR&, Args&&...) = 0;
};


template <class IR, class Pass, class... Args>
class PassModel : public PassConcept<IR, Args...> {
     Pass pass;
public:
     explicit PassModel(Pass p) : pass(move(p)) {}
     void run(IR& ir, Args&&... args) final {
        pass.run(ir, forward<Args>(args)...);
     }
};
```

```
template<class IR, class... Args>
struct PassManager {
    vector<shared_ptr<PassConcept<IR, Args...> const>> passes;
    void run(IR& ir, Args&&... args) {
        for (auto& pass : passes) {
            pass->run(ir, forward<Args>(args)...);
        }
    }
    template<class T>
    void addPass(T pass) {
        passes.emplace_back(make_shared<PassModel<IR, T, Args...>>(move(pass)));
    }
};
```

```
struct OperationWisePass {
    void run(Operation const& operation) const {
        cout << "OperationWise: " << operation << '\n';
    }
};

struct BasicBlockWisePass {
    void run(BasicBlock const& basicBlock) const {
        cout << "BasicBlockWise: " << '[';
        for (size_t i = 0; i < basicBlock.size(); ++i) {
            if (i > 0) cout << ", ";
            cout << basicBlock[i];
        }
        cout << "]\n";
    }
};
```

```
struct FunctionWisePass {
    void run(Function const& function) const {
        cout << "FunctionWise: {\n";
        for (auto const& basicBlock : function) {
            cout << "\t[";
            for (size_t i = 0; i < basicBlock.size(); ++i) {
                if (i > 0) cout << ", ";
                cout << basicBlock[i];
            }
            cout << "]\n";
        }
        cout << "}\n";
    }
};
```

```
int main() {
    auto model = Model{
        {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9},
        },
        {
            {-1, -2, -3},
            {-4, -5, -6},
            {-7, -8, -9},
        }
    };
    …
}
```

```
int main() {
    auto model = Model{…};

    PassManager<Operation> operationWisePasses;
    operationWisePasses.addPass(OperationWisePass{});

    PassManager<BasicBlock> basicBlockWisePasses;
    basicBlockWisePasses.addPass(BasicBlockWisePass{});

    PassManager<Function> functionWisePasses;
    functionWisePasses.addPass(FunctionWisePass{});
    …
}
```

```cpp
template<class OperationWisePass, class... Args>
class OperationWiseAdaptor {
    OperationWisePass pass;
public:
    explicit OperationWiseAdaptor(OperationWisePass initializer) : pass(move(initializer)) {}
    void run(Model& model, Args&&... args) const {
        for (auto& function : model) {
            for (auto& basicBlock : function) {
                for (auto& operation : basicBlock) {
                    pass.run(operation, forward<Args>(args)...);
                }
            }
        }
    }
};

using OperationWiseManagerAdaptor = OperationWiseAdaptor<PassManager<Operation>>;
```

```
template<class BasicBlockWisePass, class... Args>
class BasicBlockWiseAdaptor {
    BasicBlockWisePass pass;
public:
    explicit BasicBlockWiseAdaptor(BasicBlockWisePass initializer) : pass(move(initializer)) {}
    void run(Model& model, Args&&... args) const {
        for (auto& function : model) {
            for (auto& basicBlock : function) {
                pass.run(basicBlock, forward<Args>(args)...);
            }
        }
    }
};

using BasicBlockWiseManagerAdaptor = BasicBlockWiseAdaptor<PassManager<BasicBlock>>;
```

```
template<class FunctionWisePass, class... Args>
class FunctionWiseAdaptor {
    FunctionWisePass pass;
public:
    explicit FunctionWiseAdaptor(FunctionWisePass initializer) : pass(move(initializer)) {}
    void run(Model& model, Args&&... args) const {
        for (auto& function : model) {
            pass.run(function, forward<Args>(args)...);
        }
    }
};

using FunctionWiseManagerAdaptor = FunctionWiseAdaptor<PassManager<Function>>;
```

```
int main() {
    auto model = Model{…};

    PassManager<Operation> operationWisePasses;
    operationWisePasses.addPass(OperationWisePass{});

    PassManager<BasicBlock> basicBlockWisePasses;
    basicBlockWisePasses.addPass(BasicBlockWisePass{});

    PassManager<Function> functionWisePasses;
    functionWisePasses.addPass(FunctionWisePass{});

    PassManager<Model> passes;
    passes.addPass(OperationWiseAdaptor{move(operationWisePasses)});
    passes.addPass(BasicBlockWiseManagerAdaptor{move(basicBlockWisePasses)});
    passes.addPass(FunctionWiseManagerAdaptor{move(functionWisePasses)});
    passes.run(model);
    return 0;
}
```

```
// OperationWise: 1
// OperationWise: 2
// …
// OperationWise: 8
// OperationWise: 9
// OperationWise: -1
// OperationWise: -2
// …
// OperationWise: -8
// OperationWise: -9
// BasicBlockWise: [1, 2, 3]
// BasicBlockWise: [4, 5, 6]
// BasicBlockWise: [7, 8, 9]
// BasicBlockWise: [-1, -2, -3]
// BasicBlockWise: [-4, -5, -6]
// BasicBlockWise: [-7, -8, -9]
// FunctionWise: {
//  [1, 2, 3]
//  [4, 5, 6]
//  [7, 8, 9]
// }
// FunctionWise: {
//  [-1, -2, -3]
//  [-4, -5, -6]
//  [-7, -8, -9]
// }
```

```cpp
using History = vector<Document>;

void commit(History& x) { assert(!x.empty()); x.push_back(x.back()); }
void undo(History& x) { assert(!x.empty()); x.pop_back(); }
Document& current(History& x) { assert(!x.empty()); return x.back(); }

int main() {
    History history(1);
    current(history).emplace_back(0);
    current(history).emplace_back("Hello");
    print(current(history), 0);
    commit(history);
    current(history)[0] = 5;
    current(history).emplace_back(current(history));
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    undo(history);
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    return 0;
}
```

```
// <document>
//   0
//   Hello
// </document>
// ==========
// <document>
//   5
//   Hello
//   <document>
//     5
//     Hello
//   </document>
// </document>
// ==========
// <document>
//   0
//   Hello
// </document>
```

```
class Object {
    struct Concept {
        …
        virtual unique_ptr<Concept> copy() const = 0;
    };
    template<class T> struct Model : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<Model<T>>(*this); }
        T data;
    };
    unique_ptr<Concept> self;
public:
    template<class T> Object(T x) : self(make_unique<Model<T>>(move(x))) {}
    Object(Object const& x) : self(x.self->copy()) { cout << "copy" << '\n'; }
    Object(Object&&) noexcept = default;
    Object& operator=(Object const& x) { Object copy(x); *this = move(copy); return *this; }
    Object& operator=(Object&&) noexcept = default;
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```cpp
using History = vector<Document>;

void commit(History& x) { assert(!x.empty()); x.push_back(x.back()); }
void undo(History& x) { assert(!x.empty()); x.pop_back(); }
Document& current(History& x) { assert(!x.empty()); return x.back(); }

int main() {
    History history(1);
    current(history).emplace_back(0);
    current(history).emplace_back("Hello");
    print(current(history), 0);
    commit(history);
    current(history)[0] = 5;
    current(history).emplace_back(current(history));
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    undo(history);
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    return 0;
}
```

```
// <document>
//   0
//   Hello
// </document>
// copy
// copy
// copy
// copy
// ==========
// <document>
//   5
//   Hello
//   <document>
//     5
//     Hello
//   </document>
// </document>
// ==========
// <document>
//   0
//   Hello
// </document>
```

```
class Object {
    struct Concept {
        …
        virtual unique_ptr<Concept> copy() const = 0;
    };
    template<class T> struct Model : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<Model<T>>(*this); }
        T data;
    };
    unique_ptr<Concept> self;
public:
    template<class T> Object(T x) : self(make_unique<Model<T>>(move(x))) {}
    Object(Object const& x) : self(x.self->copy()) { cout << "copy" << '\n'; }
    Object(Object&&) noexcept = default;
    Object& operator=(Object const& x) { Object copy(x); *this = move(copy); return *this; }
    Object& operator=(Object&&) noexcept = default;
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```
class Object {
    struct Concept {
        …
        virtual unique_ptr<Concept> copy() const = 0;
    };
    template<class T> struct Model : public Concept {
        …
        unique_ptr<Concept> copy() const override { return make_unique<Model<T>>(*this); }
        T data;
    };
    unique_ptr<Concept> self;
public:
    template<class T> Object(T x) : self(make_unique<Model<T>>(move(x))) {}
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```
class Object {
    struct Concept {
        …
    };
    template<class T> struct Model : public Concept {
        …
        T data;
    };
    unique_ptr<Concept> self;
public:
    template<class T> Object(T x) : self(make_unique<Model<T>>(move(x))) {}
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```
class Object {
    struct Concept {
        …
    };
    template<class T> struct Model : public Concept {
        …
        T data;
    };
    shared_ptr<Concept> self;
public:
    template<class T> Object(T x) : self(make_shared<Model<T>>(move(x))) {}
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```
class Object {
    struct Concept {
        …
    };
    template<class T> struct Model : public Concept {
        …
        T data;
    };
    shared_ptr<Concept const> self;
public:
    template<class T> Object(T x) : self(make_shared<Model<T>>(move(x))) {}
    friend void print(Object const& object, size_t indentation) { object.self->print_(indentation); }
};
```

```cpp
using History = vector<Document>;

void commit(History& x) { assert(!x.empty()); x.push_back(x.back()); }
void undo(History& x) { assert(!x.empty()); x.pop_back(); }
Document& current(History& x) { assert(!x.empty()); return x.back(); }

int main() {
    History history(1);
    current(history).emplace_back(0);
    current(history).emplace_back("Hello");
    print(current(history), 0);
    commit(history);
    current(history)[0] = 5;
    current(history).emplace_back(current(history));
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    undo(history);
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    return 0;
}
```

```
// <document>
//   0
//   Hello
// </document>
// ==========
// <document>
//   5
//   Hello
//   <document>
//     5
//     Hello
//   </document>
// </document>
// ==========
// <document>
//   0
//   Hello
// </document>
```

```cpp
int main() {
    History history(1);
    current(history).emplace_back(0);
    current(history).emplace_back("Hello");
    print(current(history), 0);
    commit(history);
    current(history)[0] = 5;
    auto saving = async([document = current(history)]() {
        this_thread::sleep_for(chrono::seconds(3));
        cout << "===save===" << '\n';
        print(document, 0);
    });
    current(history).emplace_back(current(history));
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    undo(history);
    cout << string(10, '=') << '\n';
    print(current(history), 0);
    return 0;
}
```

```
// <document>
//   0
//   Hello
// </document>
// ==========
// <document>
//   5
//   Hello
//   <document>
//     5
//     Hello
//   </document>
// </document>
// ==========
// <document>
//   0
//   Hello
// </document>
// ===save===
// <document>
//   5
//   Hello
// </document>
```

Concept-Based Polymorphism

- Clean client code
  - Non-intrusive design does not require class wrappers
- Polymorphism cost is paid only when it is needed
- Client does not do heap allocation, does not worry about objects lifetime
- Flexibility
  - The same object could be used in different libraries with different interface requirements

Concept-Based Polymorphism Drawbacks

- It’s hard to implement runtime-polymorphism this way if kind of multiple inheritance is needed
- Not exactly self documenting
  - If some class has method “run” to be used in a polymorphic way there is nothing in the client code that says why “run” is here

Materials

- Sean Parent: Inheritance Is The Base Class of Evil
- Sean Parent: Better Code: Runtime Polymorphism
- Habrahabr: C++ Concept-Based Polymorphism в продуктовом коде: PassManager в LLVM