Special Member functions: Rule of ~~Three~~ Five

- Special member functions are those that compiler can automatically generate for you.
- Generated copy functions copy each member; move function move each member.
- Classes that require custom logic in any of these normally require implementing all of these
- [C++11] If either copy member function is declared, no move operations can be auto-generated.

Rule of Five

Rule of Three

0.	Default constructor	`Foo::Foo()`
1.	Destructor	`Foo::~Foo()`
2.	Copy constructor	`Foo::Foo(Foo &)`
3.	Copy assignment operator	`Foo& Foo::operator=(const Foo &)`
4.	Move constructor	`Foo::Foo(Foo &&)`
5.	Move assignment operator	`Foo& Foo::operator=(Foo &&)`

ioda::ObsVector

```
32  class ObsVector : public util::Printable,
33                    private util::ObjectCounter<ObsVector> {
34   public:
35    static const std::string classname() {return "ioda::ObsVector";}
36
37    ObsVector(ObsSpace &,
38              const std::string & name = "", const bool fail = true);
39    ObsVector(const ObsVector &);
40    ObsVector(ObsSpace &, const ObsVector &);
41    ~ObsVector();
42
43    ObsVector & operator = (const ObsVector &);
```

ufo::GeoVaLs

```
38  class GeoVaLs : public util::Printable,
39                  private util::ObjectCounter<GeoVaLs> {
40   public:
41    static const std::string classname() {return "ufo::GeoVaLs";}
42
43    explicit GeoVaLs(const eckit::mpi::Comm &);
44    GeoVaLs(const Locations &, const oops::Variables &);
45    GeoVaLs(const eckit::Configuration &, const ioda::ObsSpace &,
46            const oops::Variables &);
47    GeoVaLs(const GeoVaLs &);
48
49    ~GeoVaLs();
50
51    GeoVaLs & operator = (const GeoVaLs &);
```

oops::Variables

```
23  class Variables : public util::Printable {
24   public:
25    static const std::string classname() {return "oops::Variables";}
26
27    Variables();
28    explicit Variables(const eckit::Configuration &);
29    explicit Variables(const std::vector<std::string> &, const std::string & conv = "");
30    Variables(const std::vector<std::string> &, const std::vector<int>);
31
32    ~Variables();
33
34    Variables(const Variables &);
```

Polymorphism in C++

Dynamic polymorphism

- Class hierarchy with virtual functions
- Type is only known at run-time
- Store objects of derived classes as (smart) pointers to base class
  - Loss of spatial locality (-)
- Virtual function calls are implemented with a vtable
  - Double indirection (-)
  - No inlining! (- -)
- Costs paid at run-time

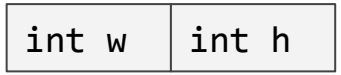
Static polymorphism

- Template functions and Template classes.
- True types are known at compile time
- Objects can be stored directly on stack or inside composite objects (+)
- Templated function calls can be inlined (++)
- Must instantiate all required classes and functions at compile time. (-)
- Template implementation must be in headers. (-)
- Costs paid at compile-time / code-time

Example: Polymorphic Shapes

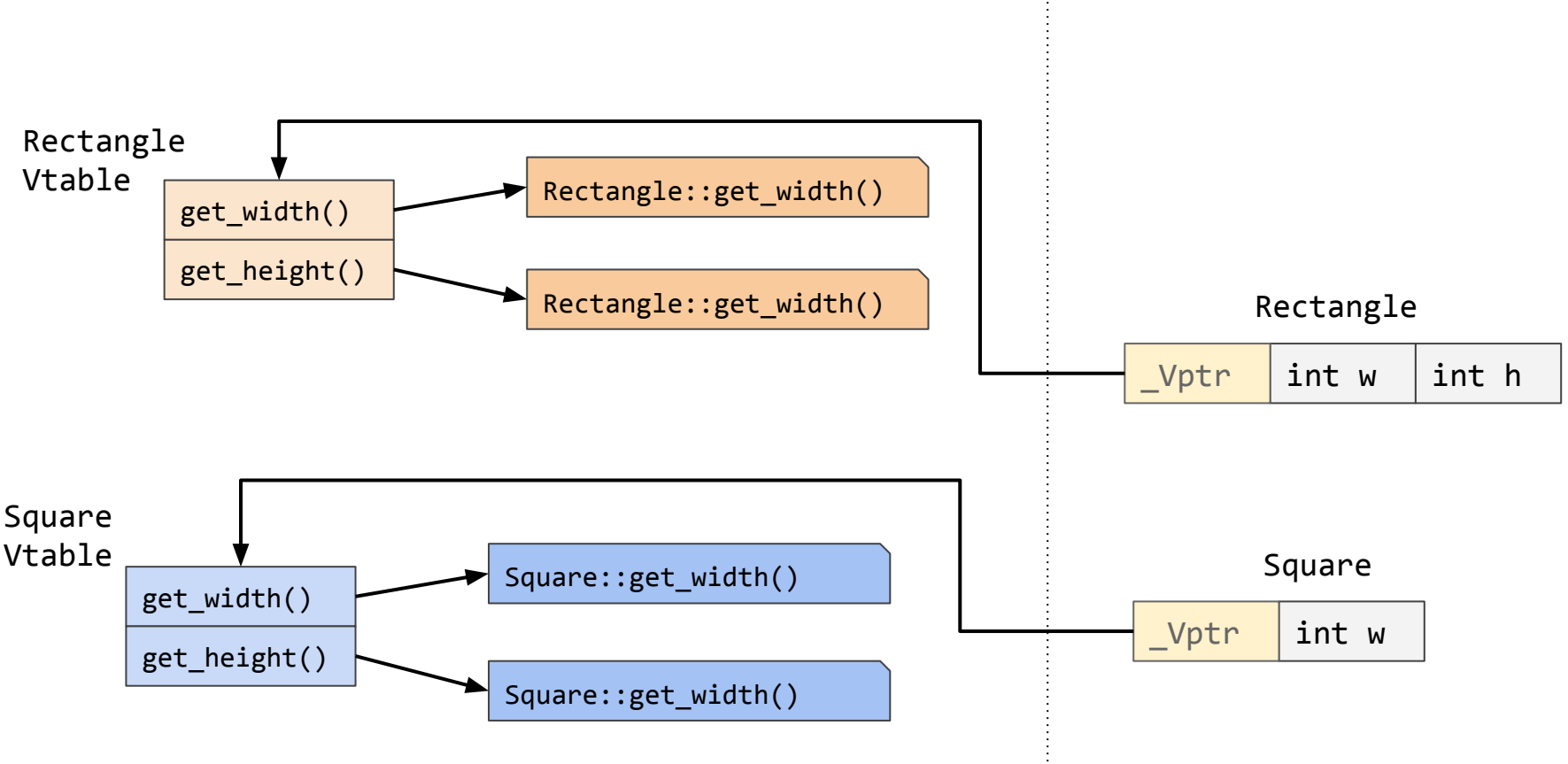
```
1  struct Shape {
2      virtual ~Shape() {};
3      virtual int get_width() const=0;
4      virtual int get_height() const=0;
5  };
6
7  struct Rectangle : Shape {
8      int w,h;
9      Rectangle(int w, int h): w(w), h(h) {}
10     int get_width() const override { return w; }
11     int get_height() const override { return h; }
12 };
13
14 struct Square : Shape {
15     int w;
16     Square(int w): w(w) {}
17     int get_width() const override { return w; }
18     int get_height() const override { return w; }
19 };
```

Rectangle



Square

Example: Polymorphic Shapes

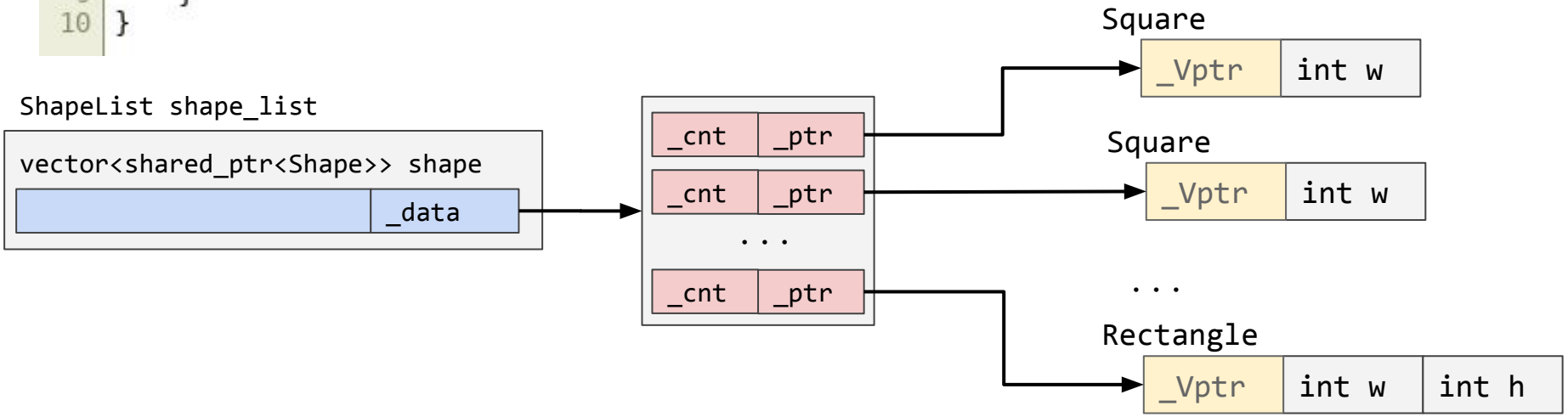


Example: ShapeList

```
1  class ShapeList {
2  public:
3      void add_rectangle(int w, int h)
4      { shapes.push_back(std::shared_ptr<Rectangle>(new Rectangle(w, h))); }
5
6      void add_square(int w)
7      { shapes.push_back(std::shared_ptr<Square>(new Square(w))); }
8
9      const Shape& get_shape(int idx) const
10     { return *shapes[idx]; }
11
12     int get_width(int idx) const
13     { return shapes[idx]->get_width(); }
14
15     int get_height(int idx) const
16     { return shapes[idx]->get_height(); }
17
18 private:
19     std::vector<std::shared_ptr<Shape>> shapes;
20 };
```

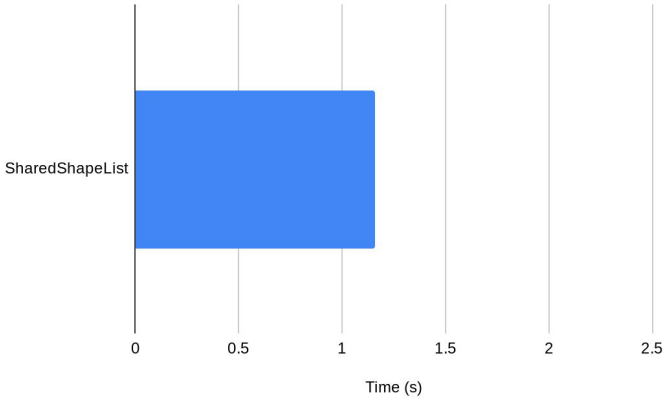
Performance: ShapeList

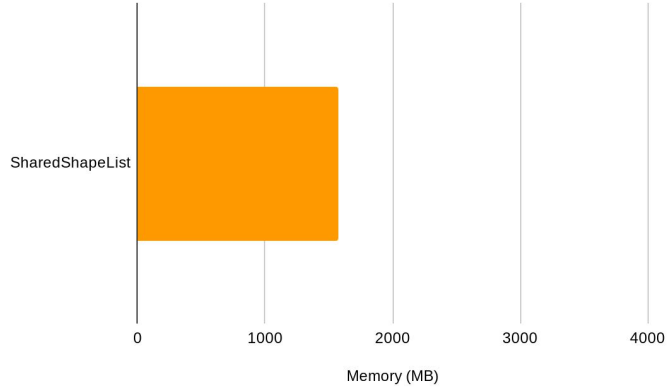
```
1  void main() {
2      ShapeList shape_list;
3      for(int i=0; i<N; i++) shape_list.add_square(i);
4      for(int i=0; i<N; i++) shape_list.add_rectangle(i,2*i);
5      long total_w=0, total_h=0;
6      for(int i=0; i<2*N; i++) {
7          total_w += shape_list.get_width(i);
8          total_h += shape_list.get_height(i);
9      }
10 }
```



Performance: ShapeList

```
1  void main() {
2      ShapeList shape_list;
3      for(int i=0; i<N; i++) shape_list.add_square(i);
4      for(int i=0; i<N; i++) shape_list.add_rectangle(i,2*i);
5      long total_w=0, total_h=0;
6      for(int i=0; i<2*N; i++) {
7          total_w += shape_list.get_width(i);
8          total_h += shape_list.get_height(i);
9      }
10 }
```





Example: ShapeList2 - `std::make_shared<>()`

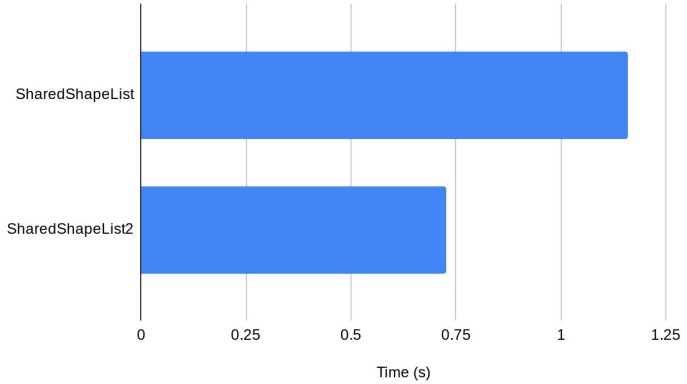
```
1  class ShapeList2 {
2  public:
3      void add_rectangle(int w, int h)
4      { shapes.push_back(std::make_shared<Rectangle>(w, h)); }
5
6      void add_square(int w)
7      { shapes.push_back(std::make_shared<Square>(w)); }
8
9      const Shape& get_shape(int idx) const
10     { return *shapes[idx]; }
11
12     int get_width(int idx) const
13     { return shapes[idx]->get_width(); }
14
15     int get_height(int idx) const
16     { return shapes[idx]->get_height(); }
17
18 private:
19     std::vector<std::shared_ptr<Shape>> shapes;
20 };
```

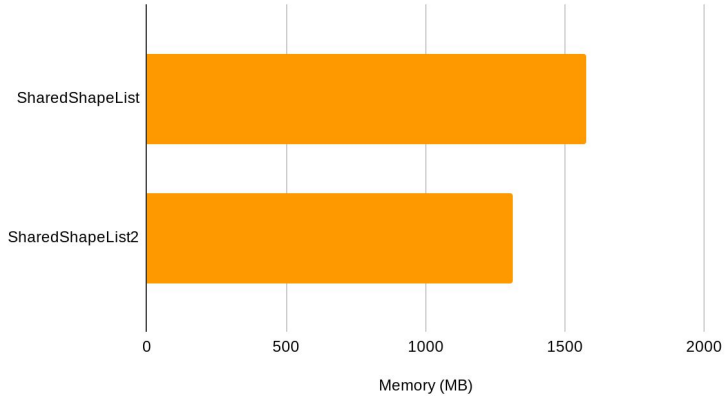
make_shared<>()

- Avoids explicit call to new
- Possible to allocate the storage for the object along with the reference count.
- Exception safety
- Uses perfect forwarding for constructor arguments
- Always preferred to direct call of shared_ptr constructor

Performance: ShapeList2 -- `std::make_shared<>()`

```
1  void main() {
2      ShapeList2 shapes;
3      for(int i=0; i<N; i++) shapes.add_square(i);
4      for(int i=0; i<N; i++) shapes.add_rectangle(i,2*i);
5      long total_w=0, total_h=0;
6      for(int i=0; i<2*N; i++) {
7          total_w += shapes.get_width(i);
8          total_h += shapes.get_height(i);
9      }
10 }
```





Example: ShapeListUnique - `std::make_unique<>()`

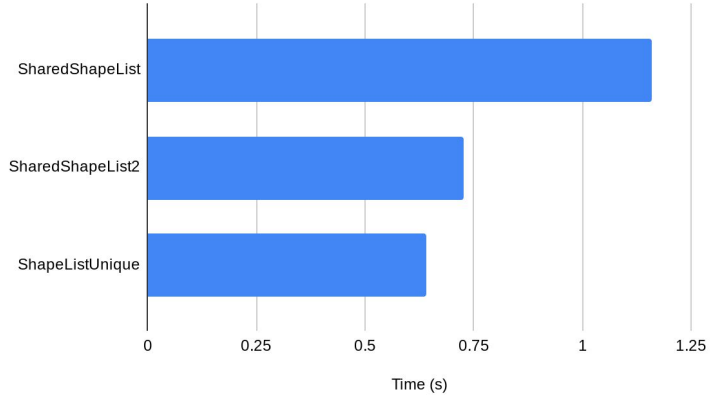
```
1  class ShapeListUnique {
2  public:
3      void add_rectangle(int w, int h)
4      { shapes.push_back(std::make_unique<Rectangle>(w, h)); }
5
6      void add_square(int w)
7      { shapes.push_back(std::make_unique<Square>(w)); }
8
9      const Shape& get_shape(int idx) const
10     { return *shapes[idx]; }
11
12     int get_width(int idx) const
13     { return shapes[idx]->get_width(); }
14
15     int get_height(int idx) const
16     { return shapes[idx]->get_height(); }
17
18 private:
19     std::vector<std::unique_ptr<Shape>> shapes;
20 };
```

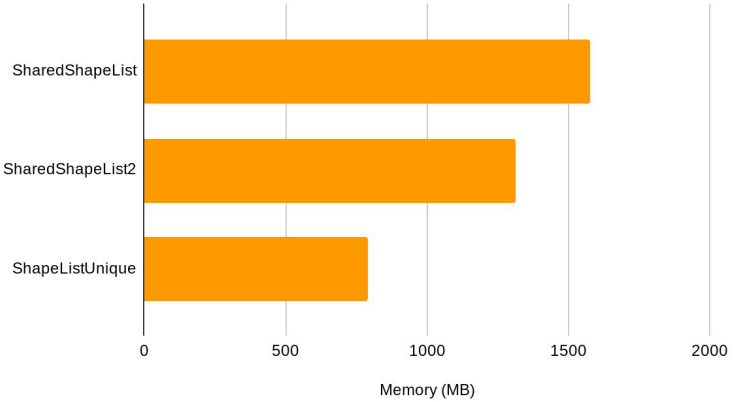
make_unique<>()

- Avoids explicit call to new
- Exception safety
- Uses perfect forwarding for constructor arguments
- Always preferred to direct call of std::unique_ptr constructor
- Unique pointers are move-only
- Unique pointers don't have a reference count

Performance: ShapeListUnique -- `std::make_unique<>()`

```
1  void main() {
2      ShapeListUnique shapes;
3      for(int i=0; i<N; i++) shapes.add_square(i);
4      for(int i=0; i<N; i++) shapes.add_rectangle(i,2*i);
5      long total_w=0, total_h=0;
6      for(int i=0; i<2*N; i++) {
7          total_w += shapes.get_width(i);
8          total_h += shapes.get_height(i);
9      }
10 }
```





Example: StaticShapeList

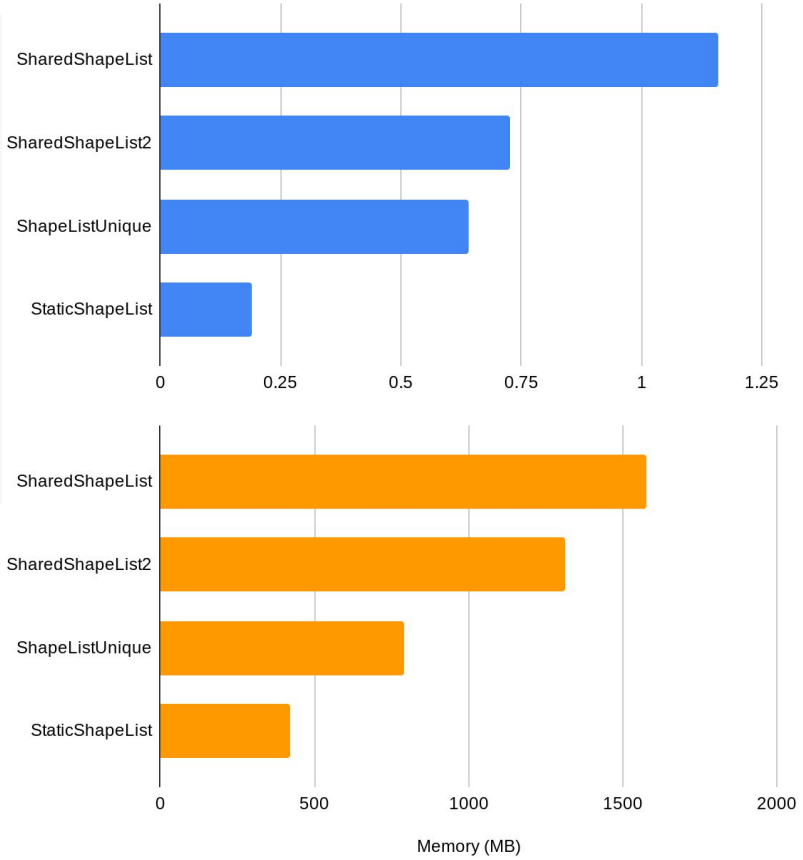
```
template<class SHAPE>
class StaticShapeList {
public:
    void add(const SHAPE& sq) { shapes.emplace_back(sq); }
    void add(SHAPE&& sq) { shapes.emplace_back(std::move(sq)); }

    const SHAPE& get_shape(int idx) const
    { return shapes[idx]; }

    int get_width(int idx) const
    { return shapes[idx].get_width(); }

    int get_height(int idx) const
    { return shapes[idx].get_height(); }

private:
    std::vector<SHAPE> shapes;
};
```

Static polymorphism

- One list type for each shape type
- Can store objects directly in vector
- Can access member functions directly
- Can inline member function calls
- Handle l-value and r-value references

Performance: StaticShapeList

```
1  void main() {
2      StaticShapeList<Square> squares;
3      StaticShapeList<Rectangle> rectangles;
4      for(int i=0; i<N; i++) squares.add(Square(i));
5      for(int i=0; i<N; i++) rectangles.add(Rectangle(i,2*i));
6      long total_w=0, total_h=0;
7      for(int i=0; i<N; i++) {
8          total_w += squares.get_width(i);
9          total_h += squares.get_height(i);
10     }
11     for(int i=0; i<N; i++) {
12         total_w += rectangles.get_width(i);
13         total_h += rectangles.get_height(i);
14     }
15 }
```



Example: StaticShapeList -- Perfect Forwarding

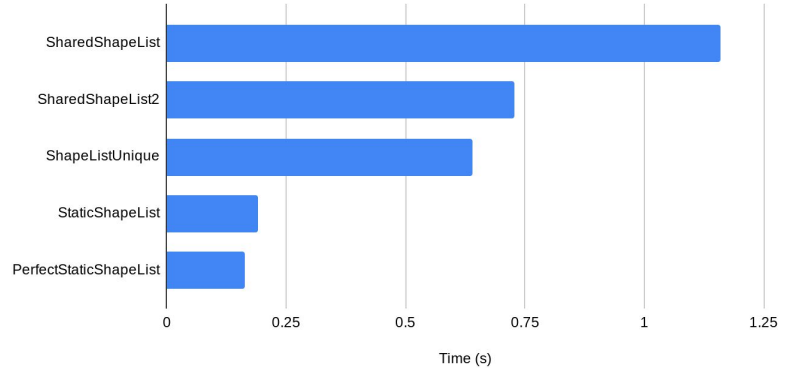
```
1  template<class SHAPE>
2  class PerfectStaticShapeList {
3  public:
4      template<class... Ts>
5      void add(Ts&&... args)
6      { shapes.emplace_back(std::forward<Ts>(args)...); }
7
8      const SHAPE& get_shape(int idx) const
9      { return shapes[idx]; }
10
11     int get_width(int idx) const
12     { return shapes[idx].get_width(); }
13
14     int get_height(int idx) const
15     { return shapes[idx].get_height(); }
16
17 private:
18     std::vector<SHAPE> shapes;
19 };
```

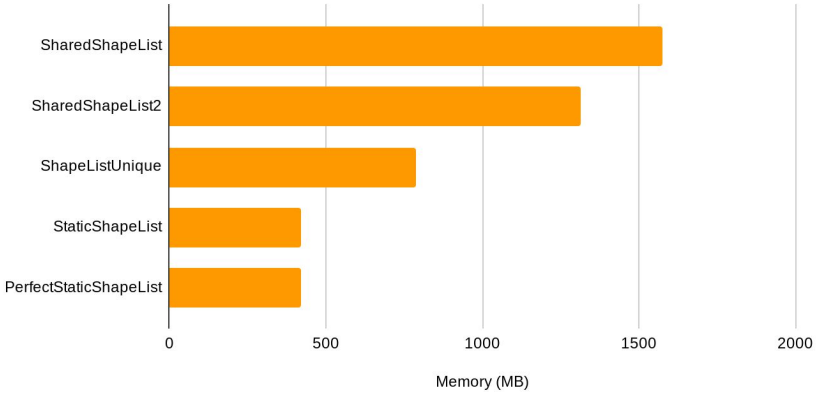
Perfect forwarding

- Use variadic templates to accept arbitrary arguments
- Use universal references to handle lvalue and rvalue references
- Use std::forward<>() to forward arguments as lvalue or rvalue references
- emplace_back calls constructor directly
- Constructor can create object inplace with no static allocation

Performance: StaticShapeList -- Perfect Forwarding

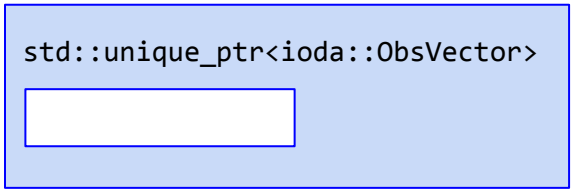
```
1  void main() {
2      PerfectStaticShapeList<Square> squares;
3      PerfectStaticShapeList<Rectangle> rectangles;
4      for(int i=0; i<N; i++) squares.add(i);
5      for(int i=0; i<N; i++) rectangles.add(i,2*i);
6      long total_w=0, total_h=0;
7      for(int i=0; i<N; i++) {
8          total_w += squares.get_width(i);
9          total_h += squares.get_height(i);
10     }
11     for(int i=0; i<N; i++) {
12         total_w += rectangles.get_width(i);
13         total_h += rectangles.get_height(i);
14     }
15 }
```





ObsVector Memory Layout

oops::ObsVector<fv3jedi::Traits>



oops::ObsVector<MODEL>

```
38  template <typename MODEL>
39  class ObsVector : public util::Printable,
40                    private util::ObjectCounter<ObsVector<MODEL> > {
41    typedef typename MODEL::ObsVector             ObsVector_;
42
43   public:
.....
53    ObsVector_ & obsvector() {return *data_;}
54    const ObsVector_ & obsvector() const {return *data_;}
.....
76   private:
77    void print(std::ostream &) const;
78    std::unique_ptr<ObsVector_> data_;
79  };
```

fv3jedi::Traits

```
38  namespace fv3jedi {
39
40  struct Traits {
.....
66    typedef ioda::ObsVector                       ObsVector;
67    template <typename DATA> using ObsDataVector = ioda::ObsDataVector<DATA>;
68  };
```