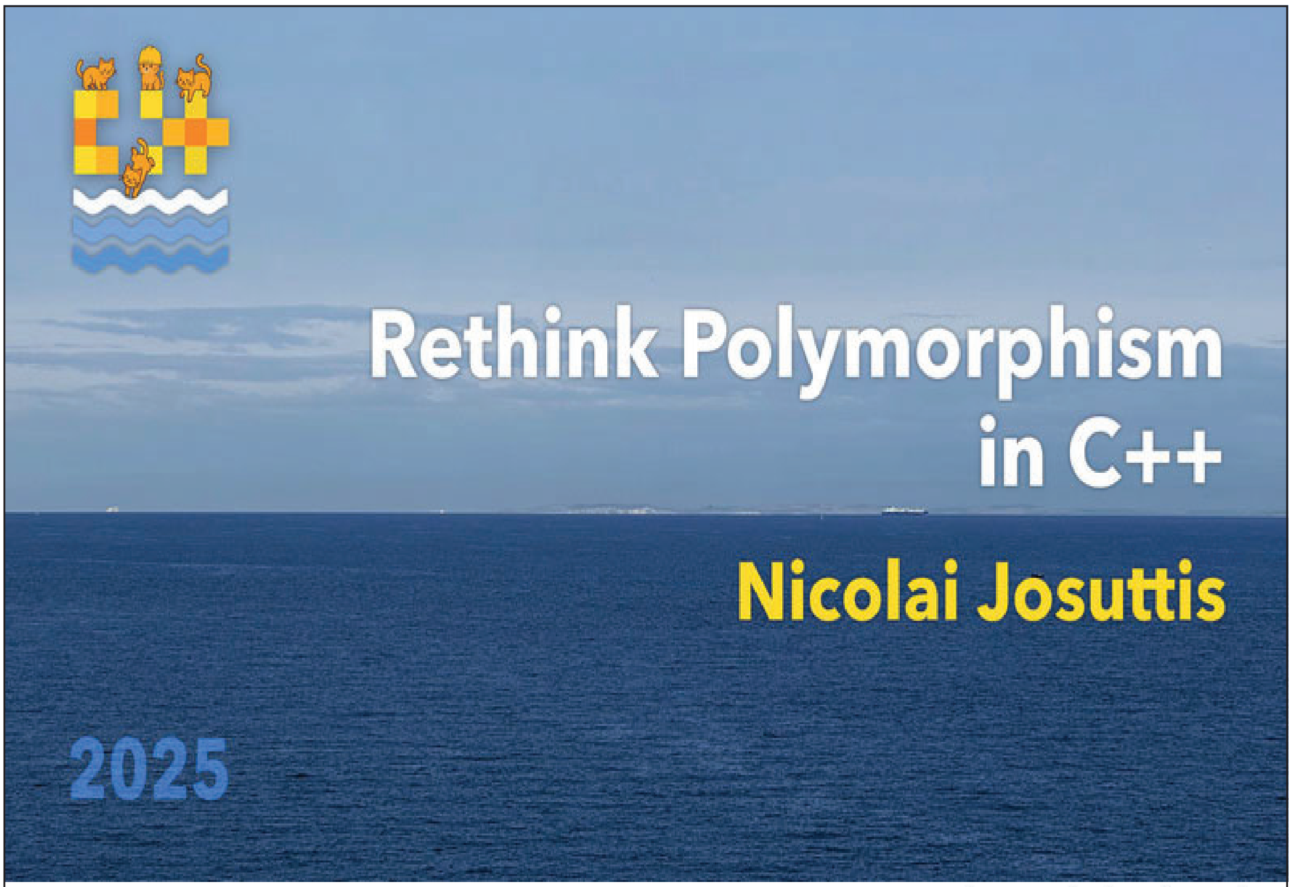


Rethink Polymorphism in C++

Nicolai Josuttis

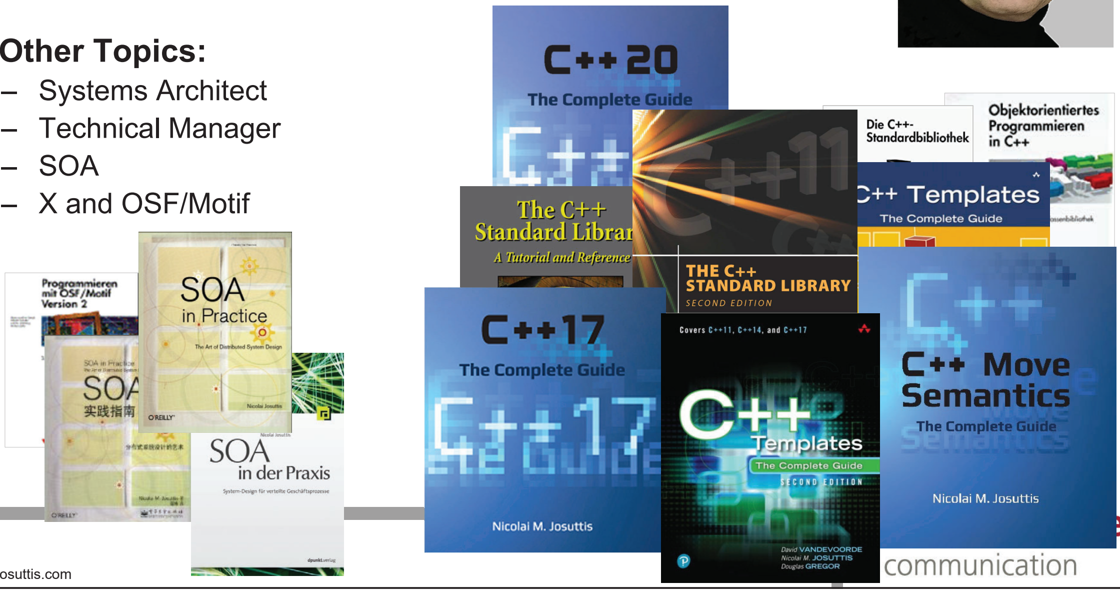
2025

C++

josuttis | eckstein
IT communication

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif


C++20 The Complete Guide
C++11
The C++ Standard Library A Tutorial and Reference
THE C++ STANDARD LIBRARY SECOND EDITION
Die C++-Standardbibliothek
Objektorientiertes Programmieren in C++
C++ Templates The Complete Guide
Programmieren mit OSF/Motif Version 2
SOA in Practice
SOA 实践指南
SOA in der Praxis
C++17 The Complete Guide
C++ Templates The Complete Guide SECOND EDITION
C++ Move Semantics The Complete Guide
Nicolai M. Josuttis

C++

josuttis | eckstein
IT communication

Modern C++

Polymorphism

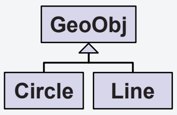
Polymorphism with Inheritance

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```

Abstract **B**ase **C**lass (interface)

`virtual` for **references/pointers** means: "*look what it really is*"

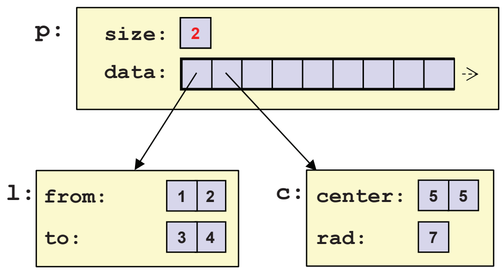


```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

Concrete classes (can create objects)

https://www.godbolt.org/z/vfhc1T5KT

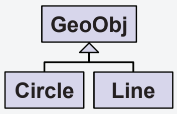
```
std::vector<GeoObj*> p;   // heterogenous collection
Line l{Coord{1,2}, Coord{3,4}};
Circle c{Coord{5,5}, 7};
p.push_back(&l);
p.push_back(&c);

for (GeoObj* gp : p) {
  gp->draw();             // polymorphic call
}
```



Polymorphism with Inheritance

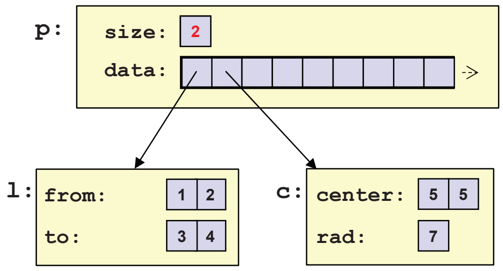
C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

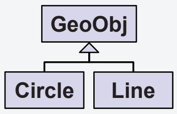
```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line l{Coord{1,2}, Coord{3,4}};
  Circle c{Coord{5,5}, 7};
  p.push_back(&l);
  p.push_back(&c);
  return p;
}
```



Polymorphism with Inheritance

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```

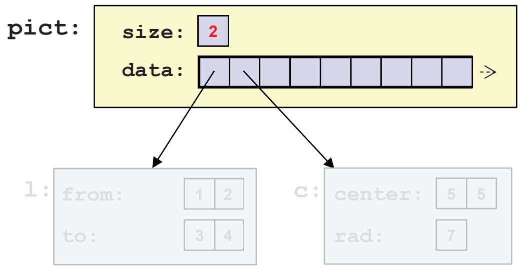


```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line l{Coord{1,2}, Coord{3,4}};
  Circle c{Coord{5,5}, 7};
  p.push_back(&l);
  p.push_back(&c);
  return p;
}
```

Fatal runtime problem: Returns vector with pointers to destroyed local objects

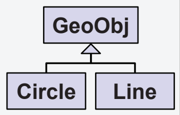
```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();                 // ERROR: undefined behavior
}
```



Polymorphism with Heap Memory

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```

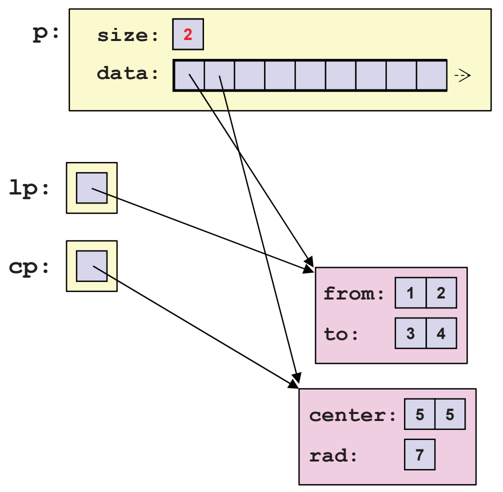


```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

Polymorphism with Heap Memory

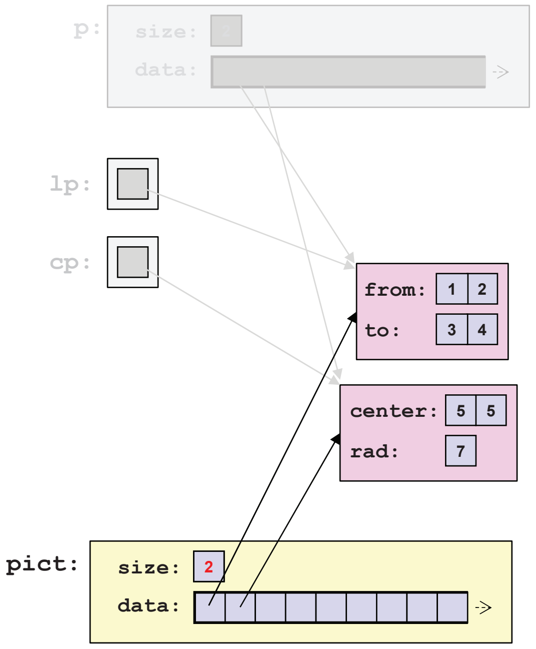
C++11



```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

Polymorphism with Heap Memory

C++11

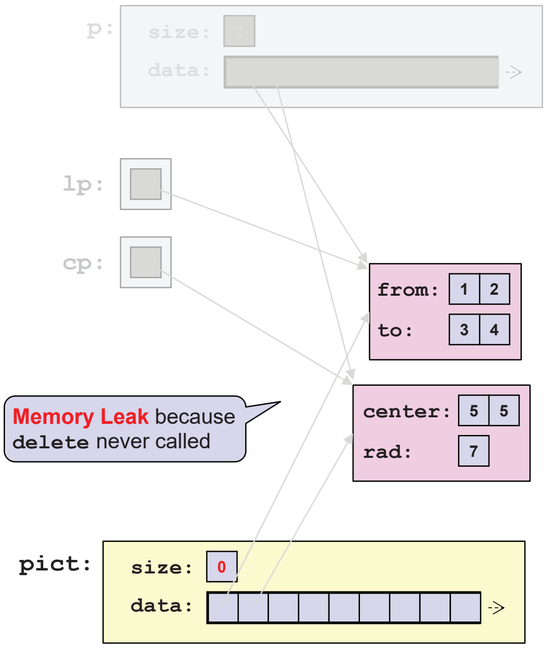


```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();            // polymorphic call
}
```

Polymorphism with Heap Memory

C++11

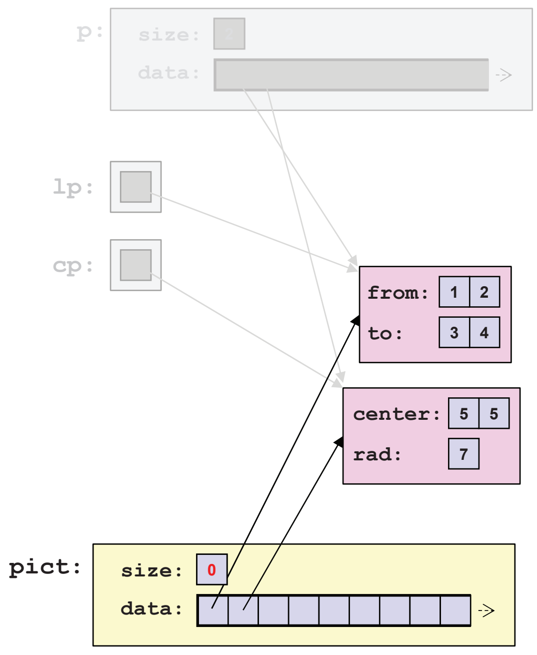


```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();            // polymorphic call
}
...

// remove all elements:
pict.clear();            // remove all elements in the vector
```

Polymorphism with Heap Memory

C++11



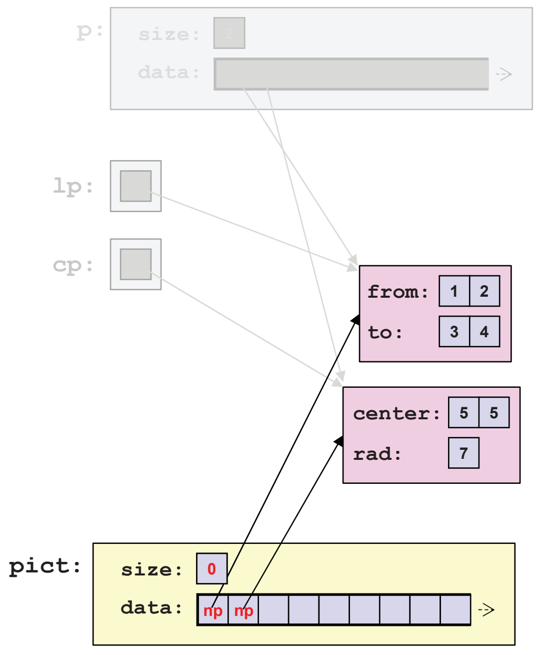
```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();             // polymorphic call
}
...
```

```
// remove all elements without memory leak:
for (GeoObj* geoPtr : pict) {
  delete geoPtr;          // call delete for each element
}
pict.clear();             // remove all elements in the vector
```

Polymorphism with Heap Memory

C++11



```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

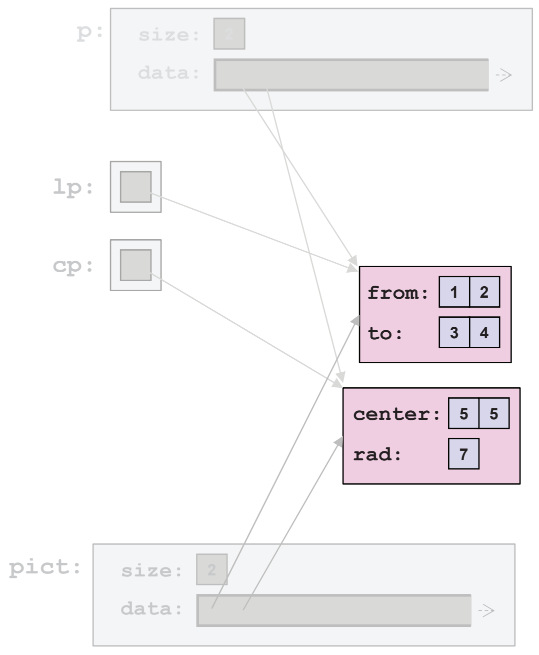
```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();             // polymorphic call
}
...
```

reference to pointer to ensure we modify original `pict` element

```
// remove all elements without memory leak:
for (GeoObj*& geoPtr : pict) {
  delete geoPtr;          // call delete for each element
  geoPtr = nullptr;       // disable element pointer in the vector
}
pict.clear();             // remove all elements in the vector
```

Polymorphism with Heap Memory

C++11



```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;  // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

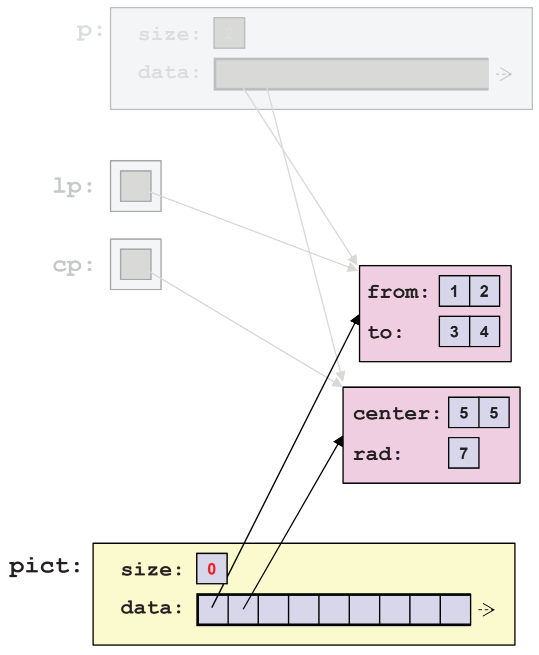
```
std::vector<GeoObj*> pict = createPicture();

for (GeoObj* gp : pict) {
  gp->draw();              // polymorphic call
}
...
```

Memory Leak on exception because `delete` not reached

```
// remove all elements without memory leak:
for (GeoObj*& geoPtr : pict) {
  delete geoPtr;           // call delete for each element
  geoPtr = nullptr;        // disable element pointer in the vector
}
pict.clear();              // remove all elements in the vector
```

Fixing Memory Leaks on Exceptions

C++11



```
std::vector<GeoObj*> createPicture()
{
  std::vector<GeoObj*> p;  // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

```
void cleanupPicture(std::vector<GeoObj*>& p)
{
  // remove all elements without memory leak:
  for (GeoObj* gp : p) {
    delete gp ;            // call delete for each element
  }
  p.clear();               // remove all elements in the vector
}
```

```
void foo()
{
  std::vector<GeoObj*> pict = createPicture();
  try {
    ...
  }
  catch (...) {
    cleanupPicture(pict);     // clean-up memory
    throw;                    // and rethrow exception
  }
  cleanupPicture(pict);
}
```

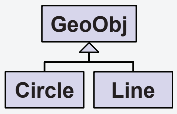
No memory leak on exception (`delete` always called)

necessary on each **return**

Fixing Memory Leaks on Exceptions

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
std::vector<GeoObj*> createPicture()
{
   std::vector<GeoObj*> p;   // heterogenous collection
   Line* lp = new Line{Coord{1,2}, Coord{3,4}};
   Circle* cp = new Circle{Coord{5,5}, 7};
   p.push_back(lp);
   p.push_back(cp);
   return p;
}
```

```
void cleanupPicture(std::vector<GeoObj*>& p)
{
   // remove all elements without memory leak:
   for (GeoObj* gp : p) {
     delete gp ;          // call delete for each element
   }
   p.clear();             // remove all elements in the vector
}
```

```
void foo()
{
   std::vector<GeoObj*> pict = createPicture();
   try {
     …
   }
   catch (...) {
     cleanupPicture(pict);   // clean-up memory
     throw;                  // and rethrow exception
   }
   cleanupPicture(pict);
}
```

Modern C++

The RAII Pattern

RAII Pattern

C++98

- **Resource Acquisition Is Initialization**
  - To clean-up a resource, initialize an object
    - Destructor automatically cleans-up (releases or frees the resource)
    - Copying and assignment implemented accordingly

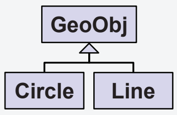
```
{
  // C API:
  FILE* fp = fopen(name, "r");
  ...
                        What happens if we return
                        or leave scope here?

  fclose(fp);
}
```

```
{
  // C++ API using RAII:
  std::ifstream f{fname};   // open file
  ...


} // end of lifetime of f closes the file immediately
...   File is guaranteed to be closed here
      (right after leaving the scope)
```

Fixing Memory Leaks on Exceptions with RAII

C++98/C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  ...
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  ...
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  ...
};
```

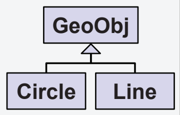
```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  ...
  void push_back(GeoObj* gp) {
    elems.push_back(gp);
  }
  ~Picture() {                          Should disable copying
    for (GeoObj* gp : elems) {          (to avoid multiple owners)
      delete gp;
    }
  }
};
```

```
Picture createPicture()
{
  Picture p;   // heterogenous collection
  Line* lp = new Line{Coord{1,2}, Coord{3,4}};
  Circle* cp = new Circle{Coord{5,5}, 7};
  p.push_back(lp);
  p.push_back(cp);
  return p;
}                                  No memory leak on exception
                                   (delete always called)
void foo()
{
  Picture pict = createPicture();
  ...
} // destructor cleans-up automatically (even on exceptions)
```

RAII Type `Picture`

C++98/C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  void insertLine(Coord c1, Coord c2) {
    elems.push_back(new Line{c1, c2});
  }
  void insertCircle(Coord c, int r) {
    elems.push_back(new Circle{c, r});
  }
  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
  }
};
```

Should disable copying
(to avoid multiple owners)

new and delete
completely encapsulated

```
Picture createPicture()
{
  Picture p;   // heterogenous collection
  p.insertLine(Coord{1,2}, Coord{3,4});
  p.insertCircle(Coord{5,5}, 7);
  return p;
}

void foo()
{
  Picture pict = createPicture();
  …
} // destructor cleans-up automatically (even on exceptions)
```

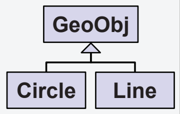
C++

josuttis | eckstein
IT communication

RAII Type `Picture` with Generic Insertion

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types... args) {
    elems.push_back(new T{args...});
  }
  …

  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
  }
};
```

Should disable copying
(to avoid multiple owners)

new and delete
completely encapsulated

```
Picture createPicture()
{
  Picture p;   // heterogenous collection
  p.insert<Line>(Coord{1,2}, Coord{3,4});
  p.insert<Circle>(Coord{5,5}, 7);
  return p;
}

void foo()
{
  Picture pict = createPicture();
  …
} // destructor cleans-up automatically (even on exceptions)
```

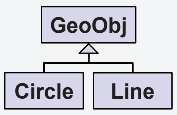
C++

josuttis | eckstein
IT communication

Type Picture with Generic Insertion and Move Semantics

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

https://www.godbolt.org/z/aEYxs8Mvr

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      new T{std::forward<Types>(args)...});
  }
  …
  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
  }
};
```

with perfect forwarding

Should disable copying (to avoid multiple owners)

new and delete completely encapsulated

```
Picture createPicture()
{
  Picture p;   // heterogenous collection
  p.insert<Line>(Coord{1,2}, Coord{3,4});
  p.insert<Circle>(Coord{5,5}, 7);
  return p;
}

void foo()
{
  Picture pict = createPicture();
  …
} // destructor cleans-up automatically (even on exceptions)
```

Modern C++

Smart Pointers

Smart Pointers

C++11

Use RAII types to clean up

- **Smart pointers**
  - Objects can be used like pointers, but are smarter
  - Act as "owners" of the objects
    - **Call `delete`** for the objects they "own" **when the last "owner" gives up ownership**
  - **Shared pointers**
    - **Shared ownership**
    - Some overhead
  - **Unique pointers**
    - **Exclusive ownership**
    - No overhead

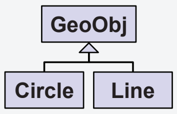
Modern C++

Shared Pointers

Polymorphism Example with Shared Pointers

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
// define type alias:
using GeoPtr = std::shared_ptr<GeoObj>;
```

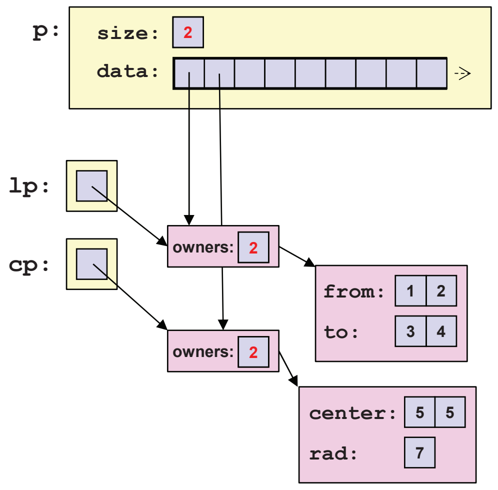
same as:
```
typedef std::shared_ptr<GeoObj> GeoPtr;
```

```
std::vector<GeoPtr> pict;
```

same as:
```
std::vector<std::shared_ptr<GeoObj>> pict;
```

Polymorphism Example with Shared Pointers

C++11

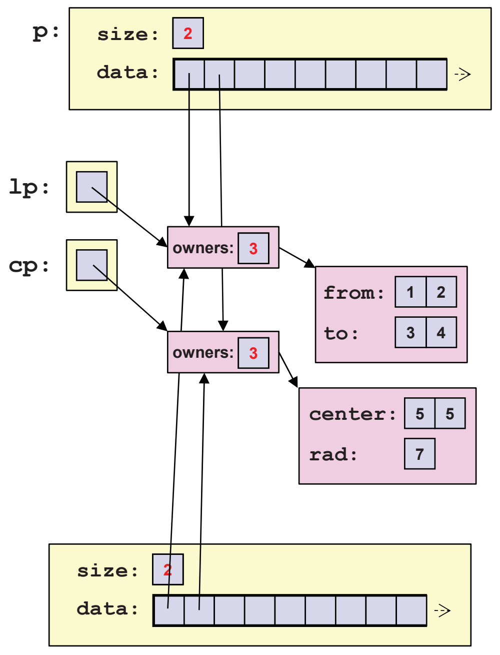


```
// define type alias:
using GeoPtr = std::shared_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_shared<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_shared<Circle>(Coord{5,5}, 7);
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

calls new for object and counter
returns shared_ptr<Line>

Polymorphism Example with Shared Pointers

C++11



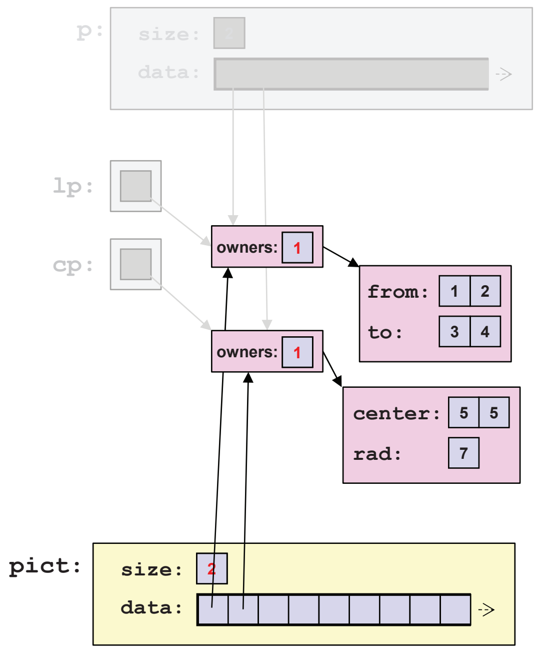
```
// define type alias:
using GeoPtr = std::shared_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_shared<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_shared<Circle>(Coord{5,5}, 7);
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

calls new for object and counter
returns shared_ptr<Line>

```
std::vector<GeoPtr> pict = createPicture();
```

Polymorphism Example with Shared Pointers

C++11



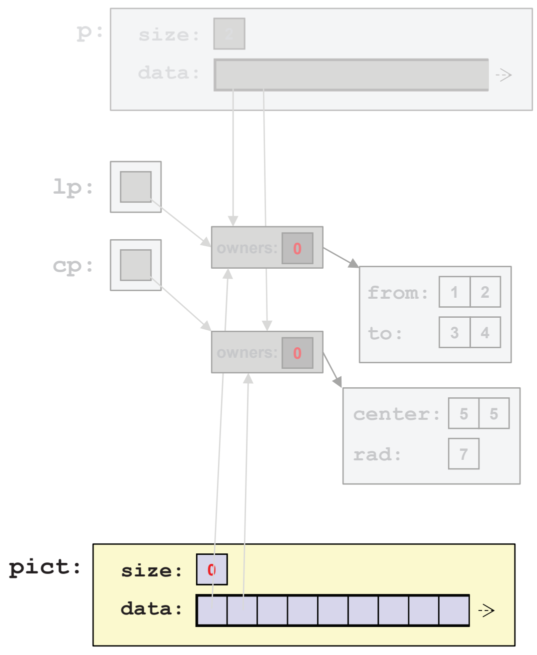
```
// define type alias:
using GeoPtr = std::shared_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_shared<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_shared<Circle>(Coord{5,5}, 7);
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

calls new for object and counter
returns shared_ptr<Line>

```
std::vector<GeoPtr> pict = createPicture();

for (const GeoPtr& gp : pict) {
  gp->draw();     // polymorphic call
}
...
```

Polymorphism Example with Shared Pointers

C++11



```
// define type alias:
using GeoPtr = std::shared_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_shared<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_shared<Circle>(Coord{5,5}, 7);
  p.push_back(lp);
  p.push_back(cp);
  return p;
}
```

calls new for object and counter returns shared_ptr<Line>

```
std::vector<GeoPtr> pict = createPicture();

for (const GeoPtr& gp : pict) {
  gp->draw();     // polymorphic call
}
...

pict.clear();     // remove all elements in the vector
```

calls delete for each object because last owner destroyed

Shared Pointers used by Multiple Threads

C++11

- **Is copying shared pointers in different threads OK?**
  - Issue during standardization of C+11
    - http://wg21.link/lwg896

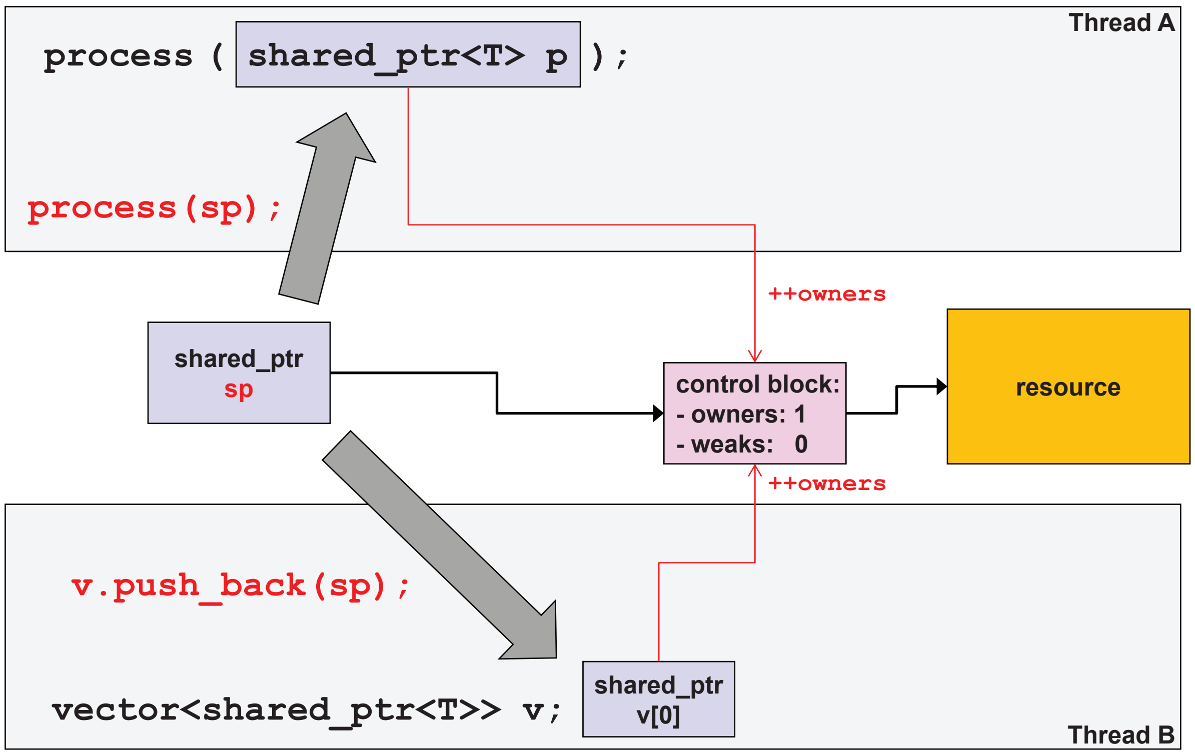
```
#include <memory>
...
std::shared_ptr<T> sp;
```

```
...
vector<shared_ptr<T>> v;
...
v.push_back(sp);   ?
...
```

```
void process (shared_ptr<T> p);

void foo()
{
   ...
   process(sp);   ?
   ...
}
```

Shared Pointers used by Multiple Threads

C++11



Shared Pointers used by Multiple Threads

C++11

- **Is copying shared pointers in different threads OK?**
  - Issue during standardization of C+11
    - http://wg21.link/lwg896

Every other read/write access (using the shared pointer or the object it refers to) **it not thread safe**

- **Yes:**

"*For purposes of determining the presence of a data race, member functions shall access and modify only the shared_ptr and weak_ptr objects themselves and not objects they refer to. Changes in* `use_count()` *do not reflect modifications that can introduce data races.*"

```
#include <memory>
...
std::shared_ptr<T> sp;
```

```
...
vector<shared_ptr<T>> v;
...
v.push_back(sp);   ✓
...
```

```
void process (shared_ptr<T> p);

void foo()
{
   ...
   process(sp);   ✓
   ...
}
```

How Expensive is Copying Shared Pointers?

C++11

sharedptrloop.cpp:

```
// initialize vector with 1000 shared pointers:
std::vector<std::shared_ptr<Type>> coll;
for (int i = 0; i < 1000; ++i) {
  coll.push_back(std::make_shared<Type>());
}

int numIterations = 1'000'000;    // 1 million times

void threadLoop (int numThreads)
{
  // loop 1 million times (partitioned over all threads) over all shared pointers:
  for (int i = 0; i < numIterations/numThreads; ++i) {
    for (auto& sp : coll) {
      sp->incrementLocalInt();
    }
  }
}
```

Pass shared and weak pointers by reference

& optional
=> optional copying
the shared pointers

By reference can be faster by a factor of 2 to 1000

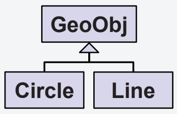
Modern C++

Unique Pointers

Polymorphism Example with Unique Pointers

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
// define type alias:
using GeoPtr = std::unique_ptr<GeoObj>;
```

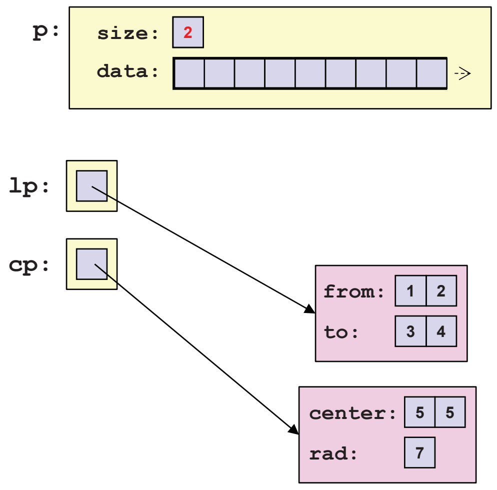
same as:
```
typedef std::unique_ptr<GeoObj> GeoPtr;
```

```
std::vector<GeoPtr> pict;
```

same as:
```
std::vector<std::unique_ptr<GeoObj>> pict;
```

Polymorphism Example with Unique Pointers

C++14

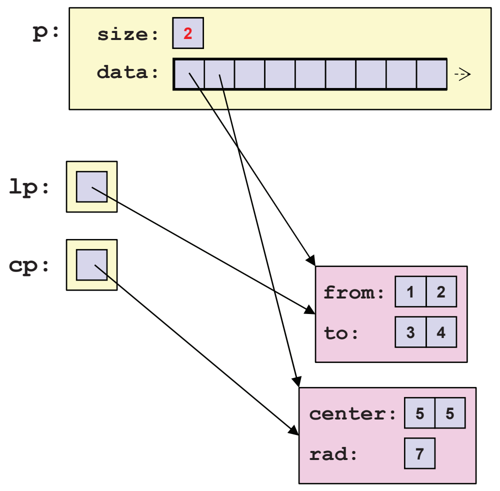


```
// define type alias:
using GeoPtr = std::unique_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
    std::make_unique<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
    std::make_unique<Circle>(Coord{5,5}, 7);
  p.push_back(lp);   // ERROR: copying disabled
  p.push_back(cp);   // ERROR: copying disabled
  return p;
}
```

calls `new` for object and returns `unique_ptr<Line>`

Polymorphism Example with Unique Pointers

C++14

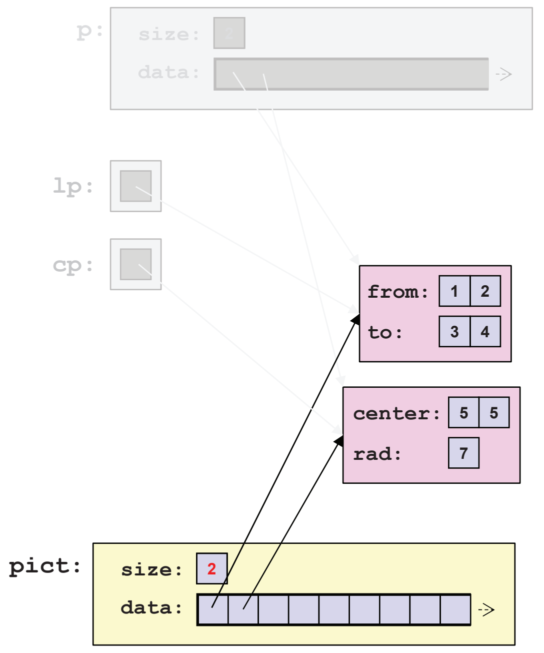


```
// define type alias:
using GeoPtr = std::unique_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_unique<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_unique<Circle>(Coord{5,5}, 7);
  p.push_back(std::move(lp));  // moves ownership
  p.push_back(std::move(cp));  // moves ownership
  return p;
}
```

calls new for object and returns unique_ptr<Line>

Polymorphism Example with Unique Pointers

C++14



```
// define type alias:
using GeoPtr = std::unique_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_unique<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_unique<Circle>(Coord{5,5}, 7);
  p.push_back(std::move(lp));  // moves ownership
  p.push_back(std::move(cp));  // moves ownership
  return p;
}
```

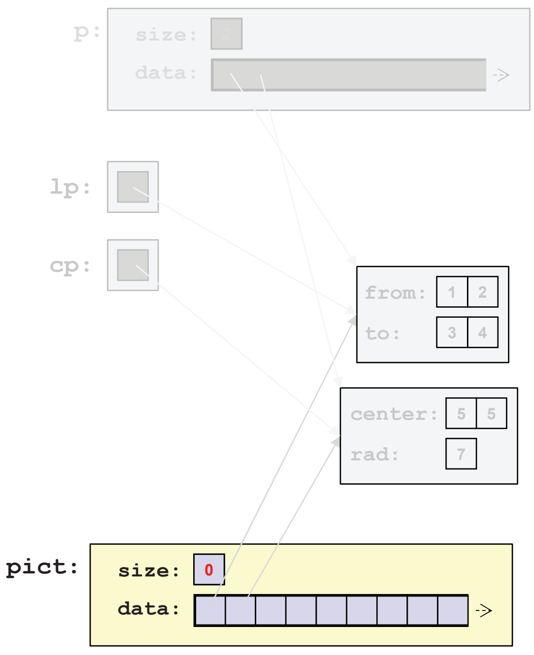
calls new for object and returns unique_ptr<Line>

```
std::vector<GeoPtr> pict = createPicture();

for (const GeoPtr& gp : pict) {
  gp->draw();    // polymorphic call
}
...
```

has to iterate by reference

Polymorphism Example with Unique Pointers

C++14



```
// define type alias:
using GeoPtr = std::unique_ptr<GeoObj>;

std::vector<GeoPtr> createPicture()
{
  std::vector<GeoPtr> p;
  auto lp =
   std::make_unique<Line>(Coord{1,2}, Coord{3,4});
  auto cp =
   std::make_unique<Circle>(Coord{5,5}, 7);
  p.push_back(std::move(lp));  // moves ownership
  p.push_back(std::move(cp));  // moves ownership
  return p;
}
```

calls `new` for object and returns `unique_ptr<Line>`

```
std::vector<GeoPtr> pict = createPicture();

for (const GeoPtr& gp : pict) {
  gp->draw();    // polymorphic call
}
...

pict.clear();    // remove all elements in the vector
```

has to iterate by reference

calls `delete` for objects because current owners

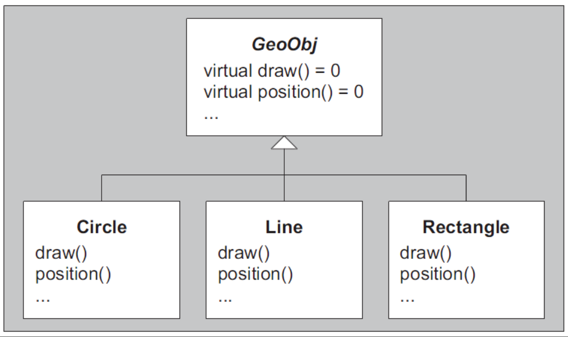
Modern C++

Polymorphism with Templates

Runtime Polymorphism with Inheritance

C++98/C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual Coord position() const = 0;
  …
  virtual ~GeoObj() = default;
};

class Circle : public GeoObj {
 public:
  virtual void draw() const override;
  virtual Coord position() const overric
  …
};

class Line : public GeoObj {
 public:
  virtual void draw() const override;
  virtual Coord position() const overric
  …
};
…
```



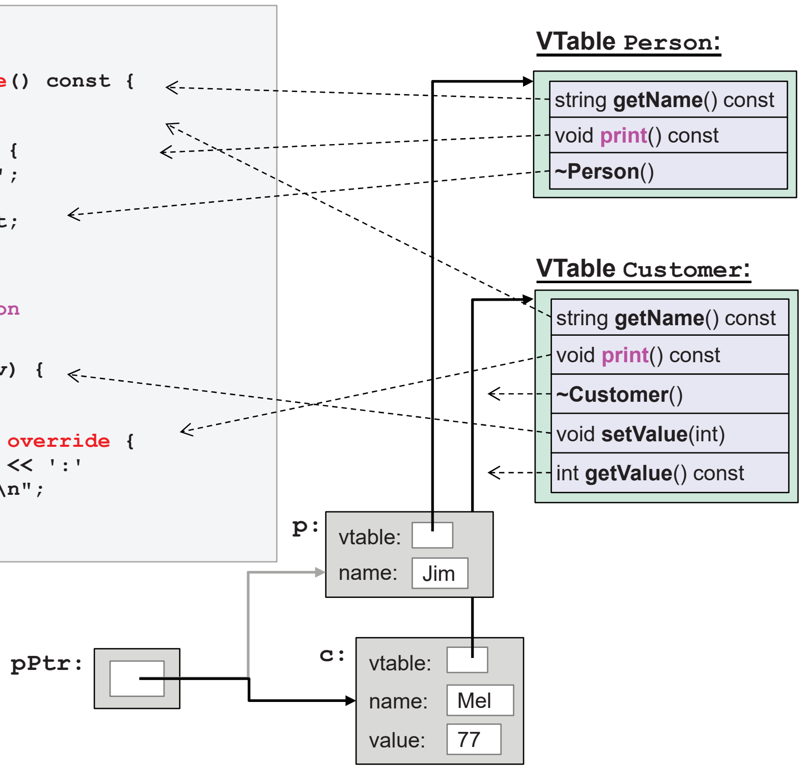
```
void myDraw(const GeoObj& obj) {
  obj.draw();
}

Coord distance(const GeoObj& x1, const GeoObj& x2) {
  Coord a = x1.position() - x2.position();
  return a.abs();
}

void drawElems(const std::vector<GeoObj*>& coll) {
  for (GeoObj* geoobjPtr : coll) {
    geoobjPtr->draw();
  }
}
```

support for **heterogeneous** collection:

Virtual Function Table

C++98/C++11

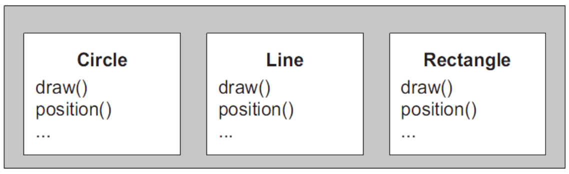
```
class Person
{
  …
  virtual std::string getName() const {
    return name;
  }
  virtual void print() const {
    std::cout << name << '\n';
  }
  virtual ~Person() = default;
  …
};

class Customer : public Person
{
  …
  virtual void setValue(int v) {
    name = n;
  }
  virtual void print() const override {
    std::cout << '[' << name << ':'
              << value << "]\n";
  }
  …
};

Person p{"Jim"};
Customer c{"Mel", 77};

Person* pPtr;
pPtr = &p;
pPtr->print();
pPtr = &c;
pPtr->print();
```



Compile-time Polymorphism with Templates

C++98/C++11

```
class Circle {
  public:
    void draw() const;
    Coord position() const;
    ...
};

class Line {
  public:
    void draw() const;
    Coord position() const;
    ...
};
...
```



```
template <typename GeoObj>
void myDraw (const GeoObj& obj) {
    obj.draw();
}

template <typename GeoObj1, typename GeoObj2>
Coord distance(const GeoObj1& x1, const GeoObj2& x2) {
    Coord a = x1.position() - x2.position();
    return a.abs();
}

template <typename GeoObj>
void drawElems(const std::vector<GeoObj>& coll) {
    for (const auto& geoobj : coll) {
        geoobj.draw();
    }
}
```

support for **homogeneous** collection only:

Runtime versus Compile-time Polymorphism

C++

- **Runtime polymorphism with inheritance:**
  - + enables **open** inhomogeneous collections
  - + less code size
  - + can add new concrete types without source code
  - + explicitly defined requirements for all types
- **Compile-time polymorphism with templates:**
  - + faster (direct function calls, better optimizations)
  - + more type safety (no inhomogeneous collections possible)
  - + nonintrusive
    - don't have to inherit
    - any class that provides the required interface is fine
    - thus, fundamental types can be used
  - + concrete types do not have to support the whole interface
    - enough if used operations are provided

C++

- **Significant better runtime performance**
  - Numbers to be taken as a hint only

System	Optimization	Inheritance	Templates
codesize (bytes):			
Linux, g++ 2.95.2		79k	105k
Linux, g++ 2.95.2	-O	72k	101k
NT, Visual C++ 6.0	debug/std.	545k	569k
NT, Visual C++ 6.0	release/speed	102k	106k
Cygwin, g++5.4.0	**-O2**	**75.5k**	**66.8k**
runtime (sec.):			
Linux, g++ 2.95.2		24.0	11.0
Linux, g++ 2.95.2	-O	9.9	1.9
NT, Visual C++ 6.0	debug/std.	300.4	161.1
NT, Visual C++ 6.0	release/speed	36.0	7.1
Cygwin, g++5.4.0	**-O2**	**4.6s**	**1.7s**
Win7, VS15	**/Ox**	**4.0s**	**1.7s**

(different hardware used)

different hardware used

1 billion calls
(1000 times calling virtual function for 1 Million lines in a vector)

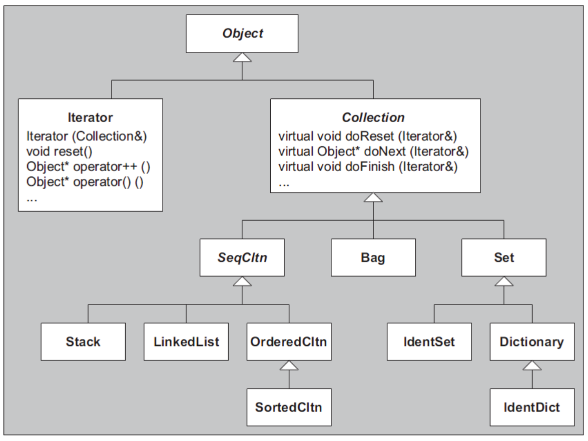
C++

Runtime versus Compile-time Polymorphism

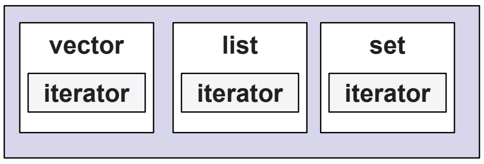
C++98

- **The NIHCL[1] implemented iterators with runtime polymorphism (the Smalltalk way):**
  - One iterator type
  - to iterate over any container

[1]: National Institute of Health Class Library:
The first "famous" C++ library



- **The STL implements iterators with compile-time polymorphism:**
  - Each container has its own iterator type
  - Common interface for all iterators



C++

C++17

`std::variant<>`

C++17: `std::variant<>`

- `std::variant<>`
  - Closed discriminated union
  - Structure to hold a value of one of the specified "*alternatives*"
  - Value type
    - Values are stored in the variant without a pointer
    - Size fits for every possible alternative type
    - No heap allocation necessary
  - Supports runtime polymorphism without inheritance

C++17: Example of `std::variant<>`

C++17

variant with 3 "alternatives"

```
#include <variant>

std::variant<int, long, std::string> var;            // initializes 1st alternative (index()==0)

std::cout << var.index() << '\n';                     // 0
std::cout << std::get<0>(var) << '\n';                // 0
std::cout << std::get<int>(var) << '\n';              // 0

var = "hello";                                        // sets string, index()==2
std::cout << var.index() << '\n';                     // 2
std::cout << std::get<2>(var) << '\n';                // "hello"
std::cout << std::get<std::string>(var) << '\n';      // "hello"

var = 42;                                             // sets int, index()==0
std::cout << var.index() << '\n';                     // 0

var = 77L;                                            // sets long, index()==1
std::cout << var.index() << '\n';                     // 1

std::cout << std::get<0>(var) << '\n';                // std::bad_variant_access exception

std::cout << std::get<3>(var) << '\n';                // compile-time error: no 4th alternative
std::cout << std::get<long long>(var) << '\n';        // compile-time error: no long long alt.
```

C++

josuttis | eckstein
IT communication

C++17: `std::variant<>` Visitors

C++17

```
std::variant<int, std::string> var;
```

```
switch (var.index()) {
 case 0:
  {
    int i = std::get<0>(var);
    std::cout << i << '\n';
  }
  break;
 case 1:
  {
    auto s = std::get<1>(var);
    std::cout << s << '\n';
  }
  break;
}
```

OK if all types supported without ambiguity

```
struct Printer {
  void operator()(int i) const {
    std::cout << "int: " << i << '\n';
  }
  void operator()(const std::string& s) const {
    std::cout << s << '\n';
  }
};

std::visit(Printer{}, var);  // calls matching operator()
```

for each alternative the lambda is compiled

```
auto printer = [] (const auto& x) {
                  std::cout << x << '\n';
               };

std::visit(printer, var);  // calls lambda for type
```

C++

josuttis | eckstein
IT communication

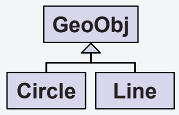
C++17

Polymorphism with `std::variant<>`

Polymorphism with `std::variant<>`

C++17

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createPicture()
{
  std::vector<GeoObjVar> p;   // heterogenous collection
  p.push_back(Line{Coord{1,2}, Coord{3,4}});
  p.push_back(Circle{Coord{5,5}, 7});
  return p;
}
```

value semantics

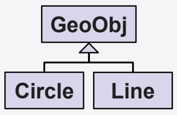
```
std::vector<GeoObjVar> pict = createPicture();

for (const GeoObjVar& geoobj : pict) {
  switch (geoobj.index()) {
   case 0:
     std::get<0>(geoobj).draw();
     break;
   case 1:
     std::get<1>(geoobj).draw();
     break;
  }
}
```

Polymorphism with `std::variant<>` Visitors

C++17

- no pointers
- no **new**/**delete**
- objects located together

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createPicture()
{
  std::vector<GeoObjVar> p;   // heterogenous collection
  p.push_back(Line{Coord{1,2}, Coord{3,4}});
  p.push_back(Circle{Coord{5,5}, 7});
  return p;
}
```

value semantics

```
std::vector<GeoObjVar> pict = createPicture();

for (const GeoObjVar& geoobj : pict) {
  std::visit([] (const auto& obj) {
               obj.draw();   // polymorphic call
             },
             geoobj);
}
```

for each alternative the lambda is compiled (local vtable)

```
// remove all elements in the vector:
pict.clear();
```

no dangling pointers or memory leak possible

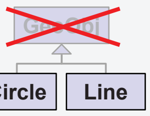
C++

josuttis | eckstein
IT communication

Polymorphism with `std::variant<>` Visitors

C++17

+ no pointers
+ no **new**/**delete**
+ objects located together
+ no common base class required
+ no virtual functions
– all elems have maximum size
– copying by value takes time
– closed set of alternatives



```
class Circle {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  void draw() const;
  …
};

class Line {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  void draw() const;
  …
};
```

```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createPicture()
{
  std::vector<GeoObjVar> p;   // heterogenous collection
  p.push_back(Line{Coord{1,2}, Coord{3,4}});
  p.push_back(Circle{Coord{5,5}, 7});
  return p;
}
```

value semantics

```
std::vector<GeoObjVar> pict = createPicture();

for (const GeoObjVar& geoobj : pict) {
  std::visit([] (const auto& obj) {
               obj.draw();   // polymorphic call
             },
             geoobj);
}
```

```
// remove all elements in the vector:
pict.clear();
```

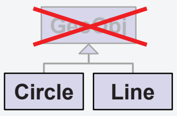
no dangling pointers or memory leak possible

C++

josuttis | eckstein
IT communication

"Downcast" with `std::variant<>`

C++17



```
class Circle {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  void draw() const;
  Coord getCenter() const;
  …
};

class Line {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  void draw() const;
  …
};
```

```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createPicture()
{
  std::vector<GeoObjVar> p;   // heterogenous collection
  p.push_back(Line{Coord{1,2}, Coord{3,4}});
  p.push_back(Circle{Coord{5,5}, 7});
  return p;
}
```

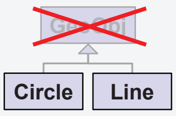
```
std::vector<GeoObjVar> pict = createPicture();

for (const GeoObjVar& geoobj : pict) {
  std::visit([] (const auto& obj) {
                 obj.draw();   // polymorphic call
               },
               geoobj);
  // downcast for variant:
  if (Circle* cp = std::get_if<Circle>(&geoobj)) {
    std::cout << cp->getCenter() << '\n';
  }
}

// remove all elements in the vector:
pict.clear();
```

C++

josuttis | eckstein
IT communication

"Downcast" with `std::variant<>`

C++17



```
class Circle {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  void draw() const;
  Coord getCenter() const;
  …
};

class Line {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  void draw() const;
  …
};
```

```
using GeoObjVar = std::variant<Circle, Line>;
```

```
void drawElems(const std::vector<GeoObjVar>& v)
{
  for (const GeoObjVar& geoobj : v) {
    std::visit([] (const auto& obj) {
                   obj.draw();   // polymorphic call
                   // downcast inside visitor:
                   if constexpr(std::is_same_v<decltype(obj),
                                               const Circle&>) {
                     std::cout << obj.getCenter() << '\n';
                   }
                 },
                 geoobj);
  }
}
```

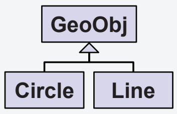
C++

josuttis | eckstein
IT communication

RAII Type Picture with Inheritance

C++11

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      new T{std::forward<Types>(args)...});
  }

  Picture(const Picture&) = delete;
  Picture& operator=(const Picture&) = delete;

  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
  }

  void draw() const {
    for (GeoObj* gp : elems) {
      gp->draw();   // calls virtual function
    }
  }
  …
};
```

https://www.godbolt.org/z/qYdz8jbes

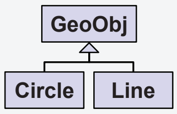
C++

josuttis | eckstein
IT communication

RAII Type Picture with `std::variant<>`

C++17

```
class GeoObj {
 public:
  GeoObj() = default;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  …
};
```



```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle(Coord c, int r);
  virtual void draw() const override;
  …
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line(Coord f, Coord t);
  virtual void draw() const override;
  …
};
```

```
class Picture {
 private:
  std::vector<std::variant<Circle,Line>> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      T{std::forward<Types>(args)...});
  }

  ~~Picture(const Picture&) = delete;~~
  ~~Picture& operator=(const Picture&) = delete;~~

  ~~~Picture() {~~
    ~~for (GeoObj* gp : elems) {~~
      ~~delete gp;~~
    ~~}~~
  ~~}~~

  void draw() const {
    for (std::variant<Circle,Line> gv : elems) {
      std::visit([](auto g) { g.draw(); }, gv);
    }
  }
  …
};
```

https://www.godbolt.org/z/7ebh7e165

C++

josuttis | eckstein
IT communication

Inheritance and Pointers versus `std::variant<>`

C++11/C++17

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      new T{std::forward<Types>(args)...});
  }

  Picture(const Picture&) = delete;
  Picture& operator=(const Picture&) =delete;

  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
  }

  void draw() const {
    for (GeoObj* gp : elems) {
      gp->draw();   // calls virtual function
    }
  }
  …
};
```

```
class Picture {
 private:
  std::vector<std::variant<Circle,Line>> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      T{std::forward<Types>(args)...});
  }

  ~~Picture(const Picture&) = delete;~~
  ~~Picture& operator=(const Picture&) = delete;~~

  ~~~Picture() {~~
    ~~for (GeoObj* gp : elems) {~~
      ~~delete gp;~~
    ~~}~~
  ~~}~~

  void draw() const {
    for (std::variant<Circle,Line> gv : elems) {
      std::visit([](auto g) { g.draw(); }, gv);
    }
  }
  …
};
```

https://www.godbolt.org/z/qYdz8jbes

https://www.godbolt.org/z/7ebh7e165

Inheritance and Pointers versus `std::variant<>`

C++11/C++17

```
class Picture {
 private:
  std::vector<GeoObj*> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      new T{std::forward<Types>(args)...});
  }

  Picture(const Picture&) = delete;
  Picture& operator=(const Picture&) =delete;

  ~Picture() {
    for (GeoObj* gp : elems) {
      delete gp;
    }
```

```
class Picture {
 private:
  std::vector<std::variant<Circle,Line>> elems;
 public:
  template<typename T, typename... Types>
  void insert(Types&&... args) {
    elems.push_back(
      T{std::forward<Types>(args)...});
  }

  ~~Picture(const Picture&) = delete;~~
  ~~Picture& operator=(const Picture&) = delete;~~

  ~~~Picture() {~~
    ~~for (GeoObj* gp : elems) {~~
      ~~delete gp;~~
    ~~}~~
```

	Platform A: GeoObj*	Platform A: std::variant<>	Platform B: GeoObj*	Platform B: std::variant<>	Platform C: GeoObj*	Platform C: std::variant<>
create	1000	700	1060	1106	1974	166
call member	56	30	72	148	186	74
destruct	800	150	315	318	760	57
downcast	230	35	482	125	430	94

https://www.godbolt.org/z/qYdz8jbes

https://www.godbolt.org/z/7ebh7e165

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis

C++20 The Complete Guide Nicolai M. Josuttis
C++17 The Complete Guide Nicolai M. Josuttis
The C++ Standard Library A Tutorial and Reference
C++ Templates The Complete Guide
C++ Move Semantics The Complete Guide Nicolai M. Josuttis
THE C++ STANDARD LIBRARY SECOND EDITION A Tutorial and Reference NICOLAI M. JOSUTTIS
Covers C++11, C++14, and C++17 C++ Templates The Complete Guide SECOND EDITION David VANDEVOORDE Nicolai M. JOSUTTIS Douglas GREGOR