TDDE18 & 726G77

Inheritance & Polymorphism

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 `std::vector`
2 Inheritance
3 Polymorphism
4 More on Polymorphism
5 Type information
6 Exceptions
7 Command-line argument

LINKÖPING UNIVERSITY

std::vector

Storage

- Linked storage
- Sequential Storage

std::vector

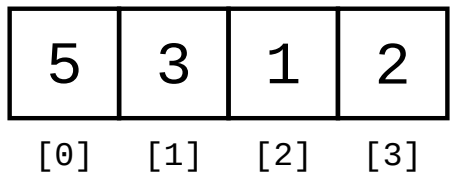
Sequential Storage

```
std::vector<int> v {5, 3, 1, 2};
```

std::vector

Sequential Storage

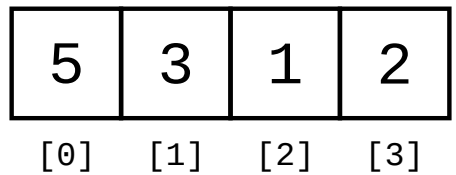
```
std::vector<int> v {5, 3, 1, 2};
```



std::vector

Sequential Storage

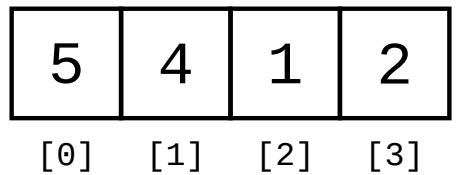
```
v.at(1) = 4;
```



std::vector

Sequential Storage

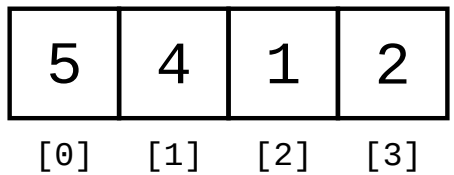
```
v.at(1) = 4;
```



std::vector

Sequential Storage

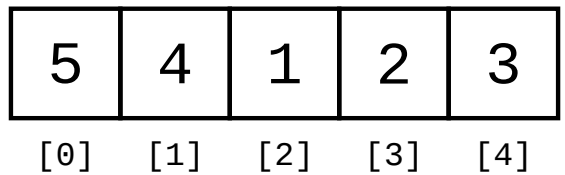
```
v.push_back(3);
```



std::vector

Sequential Storage

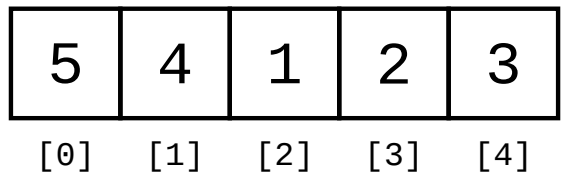
```
v.push_back(3);
```



std::vector

Sequential Storage

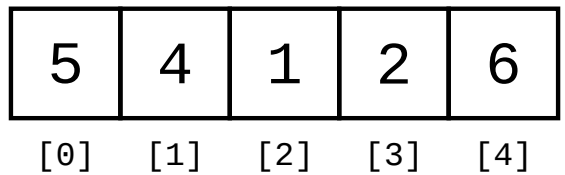
```
v.back() = 6;
```



std::vector

Sequential Storage

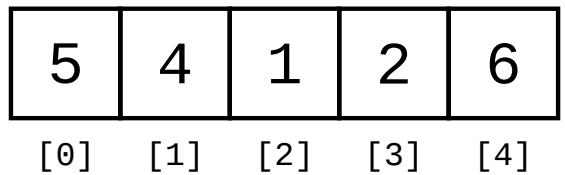
```
v.back() = 6;
```



std::vector

Sequential Storage

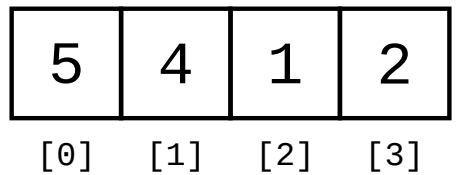
```
v.pop_back();
```



std::vector

Sequential Storage

```
v.pop_back();
```



std::vector

Looping through

```
vector<string> words {...};
for (int i{0}; i < words.size(); ++i)
{
  cout << words.at(i) << endl;
}
```

std::vector

Looping through

```
vector<string> words {...};
for (string word : words)
{
  cout << word << endl;
}
```

std::vector

Looping through

```
vector<string> words {...};
for (string const& word : words)
{
  cout << word << endl;
}
```

std::vector

Example

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
  vector<int> values{};
  int value{};

  // read values until ctrl+D
  while (cin >> value)
  {
    values.push_back(value);
  }

  // double each value
  for (int& e : values)
  {
    e = 2*e;
  }
}
```

LINKÖPING UNIVERSITY

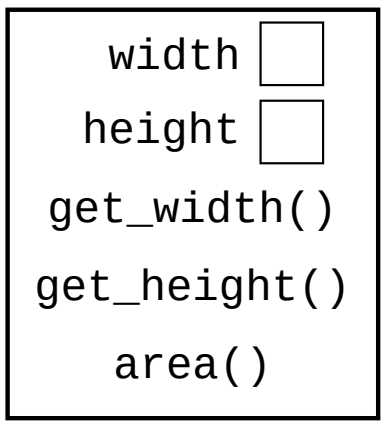
Inheritance

```
class Rectangle
{
public:
  Rectangle(double w, double h)
    : width{w}, height{h} { }

  double area() const
  {
    return height * width;
  }
  double get_height() const
  {
    return height;
  }
  double get_width() const
  {
    return width;
  }
private:
  double width;
  double height;
};
```

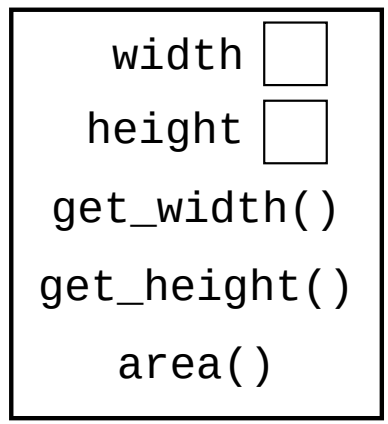
```
class Triangle
{
public:
  Triangle(double w, double h)
    : width{w}, height{h} { }

  double area() const
  {
    return height * width / 2;
  }
  double get_height() const
  {
    return height;
  }
  double get_width() const
  {
    return width;
  }
private:
  double width;
  double height;
};
```

Inheritance

What is inheritance?



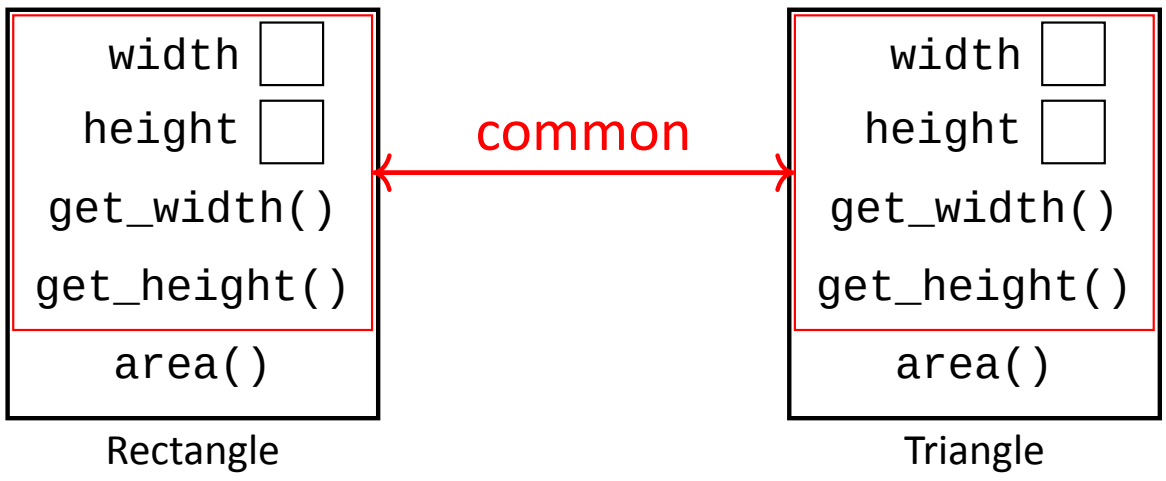
Rectangle



Triangle

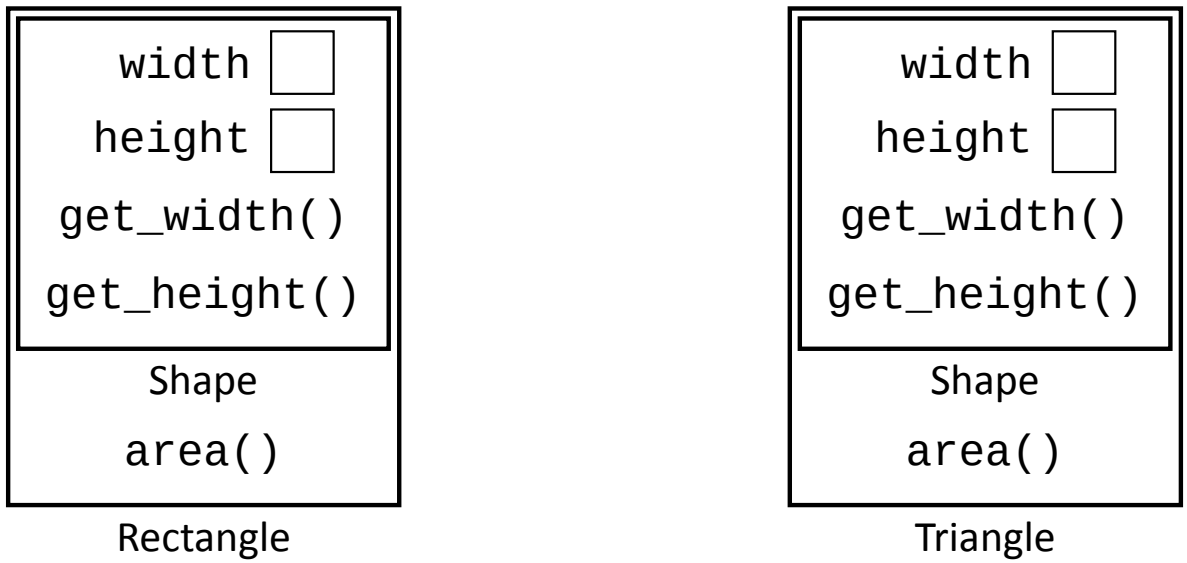
Inheritance

What is inheritance?



Inheritance

What is inheritance?



Inheritance

```
// common code
class Shape
{
public:

  Shape(double w, double h)
    : width{w}, height{h} { }

  double get_height() const
  {
    return height;
  }

  double get_width() const
  {
    return width;
  }

private:

  double width;
  double height;
};
```

Inheritance

```
// common code
class Shape
{
public:

  Shape(double w, double h)
    : width{w}, height{h} { }

  double get_height() const
  {
    return height;
  }

  double get_width() const
  {
    return width;
  }

private:

  double width;
  double height;
};
```

```
class Rectangle : public Shape
{
public:
  Rectangle(double w, double h)
    : width{w}, height{h} { }

  double area() const
  {
    return width * height;
  }
};

class Triangle : public Shape
{
public:
  Triangle(double w, double h)
    : width{w}, height{h} { }

  double area() const
  {
    return width * height / 2;
  }
};
```

Inheritance

```
Shape.cc: In constructor ‘Rectangle::Rectangle(double, double)’:
Shape.cc: error: ‘double Shape::width’ is private within this context
     : width{w}, height{h} { }
       ^~~~~
Shape.cc: note: declared private here
   double width;
          ^~~~~
Shape.cc: error: ‘double Shape::height’ is private within this context
     : width{w}, height{h} { }
                 ^~~~~~
Shape.cc: note: declared private here
   double height;
          ^~~~~~
```

Inheritance

```
// common code
class Shape
{
public:

  Shape(double w, double h)
    : width{w}, height{h} { }

  double get_height() const
  {
    return height;
  }

  double get_width() const
  {
    return width;
  }

private:

  double width;
  double height;
};
```

```
class Rectangle : public Shape
{
public:
  Rectangle(double w, double h)
    : Shape{w, h} { }

  double area() const
  {
    return width * height;
  }
};

class Triangle : public Shape
{
public:
  Triangle(double w, double h)
    : Shape{w, h} { }

  double area() const
  {
    return width * height / 2;
  }
};
```

Inheritance

```
Shape.cc: In member function ‘double Rectangle::area() const’:
Shape.cc: error: ‘double Shape::width’ is private within this context
     return width * height;
            ^~~~~
Shape.cc: note: declared private here
   double width;
          ^~~~~
Shape.cc: error: ‘double Shape::height’ is private within this context
     return width * height;
                    ^~~~~~
Shape.cc: note: declared private here
   double height;
          ^~~~~~
```

Inheritance

```
// common code
class Shape
{
public:

  Shape(double w, double h)
    : width{w}, height{h} { }

  double get_height() const
  {
    return height;
  }

  double get_width() const
  {
    return width;
  }

protected:

  double width;
  double height;
};
```

```
class Rectangle : public Shape
{
public:
  Rectangle(double w, double h)
    : Shape{w, h} { }

  double area() const
  {
    return width * height;
  }
};

class Triangle : public Shape
{
public:
  Triangle(double w, double h)
    : Shape{w, h} { }

  double area() const
  {
    return width * height / 2;
  }
};
```

Inheritance

Data members in derived class

```
class Named_Rectangle : public Rectangle
{
public:
  Named_Rectangle(int width, int height, std::string const& name)
    : Rectangle{width, height}, name{name}
  { }
private:
  std::string name{};
};
```

Inheritance

Initialization & Destruction

```
Named_Rectangle r {12, 13, "My Rectangle"};
```

Inheritance

Initialization & Destruction

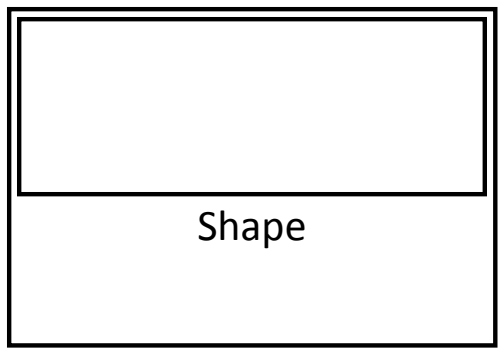
```
Named_Rectangle r {12, 13, "My Rectangle"};
```

Named_Rectangle

Inheritance

Initialization & Destruction

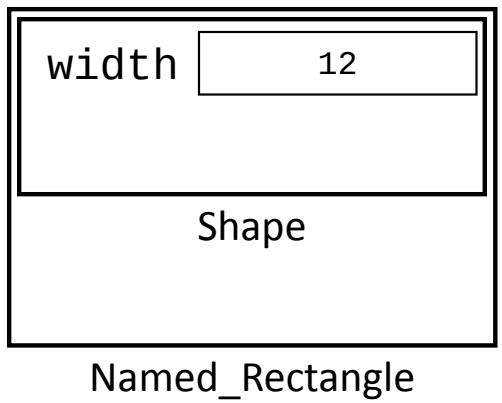
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

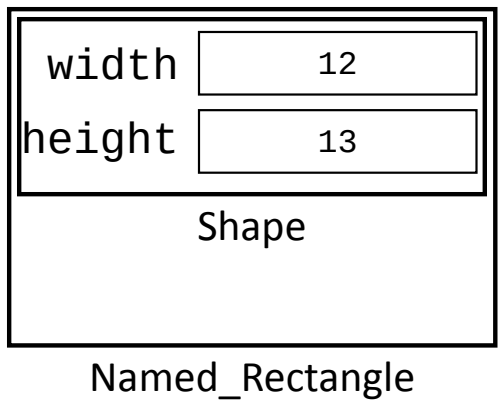
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

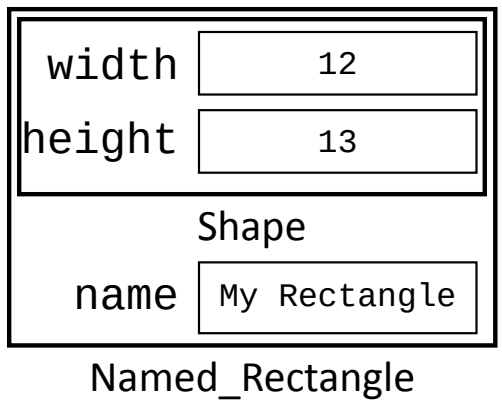
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

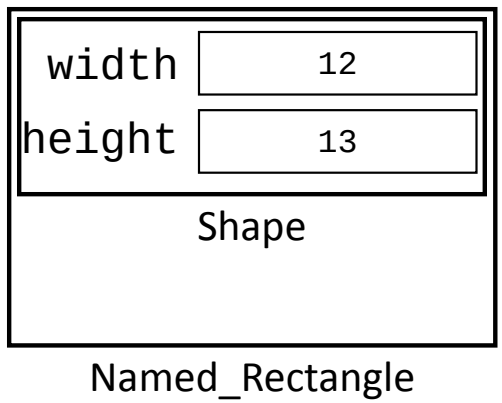
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

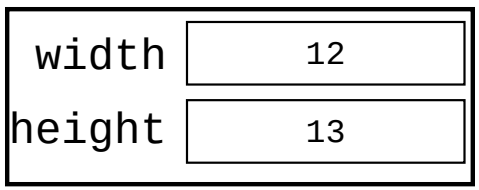
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

```
Named_Rectangle r {12, 13, "My Rectangle"};
```

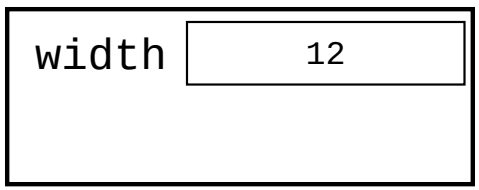


Shape

Inheritance

Initialization & Destruction

```
Named_Rectangle r {12, 13, "My Rectangle"};
```



Inheritance

Initialization & Destruction

```
Named_Rectangle r {12, 13, "My Rectangle"};
```

Shape

Inheritance

Initialization & Destruction

```
Named_Rectangle r {12, 13, "My Rectangle"};
```

Inheritance

Binding to references

```
void print_height(Triangle& triangle)
{
  cout << triangle.get_height() << endl;
}

void print_height(Rectangle& triangle)
{
  cout << triangle.get_height() << endl;
}
```

Inheritance

Binding to references

```
void print_height(Shape& shape)
{
  cout << shape.get_height() << endl;
}
```

Inheritance

`area()`

```
void print_area(Shape& shape)
{
  cout << shape.area() << endl;
}
```

Inheritance

`area()`

```
Shape.cc: In function ‘void print_area(Shape&)’:
Shape.cc: error: ‘class Shape’ has no member named ‘area’
   cout << shape.area() << endl;
                 ^~~~
```

Inheritance

Let's add `area()` to `Shape`

```
class Shape
{
public:
  // ...
  double area() const
  {
    return 0;
  }
  // ...
};
```

```
class Rectangle : public Shape
{
public:
  // ...
  double area() const
  {
    return width * height;
  }
  // ...
};
```

Inheritance

Let's add `area()` to `Shape`

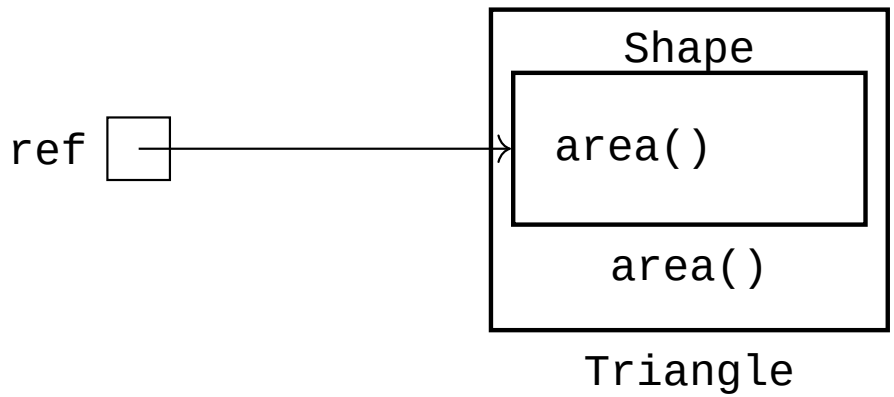
```
int main()
{
  Rectangle r {10, 15};
  cout << print_area(r) << endl; // print 0
}
```

LINKÖPING UNIVERSITY

Polymorphism

Many forms

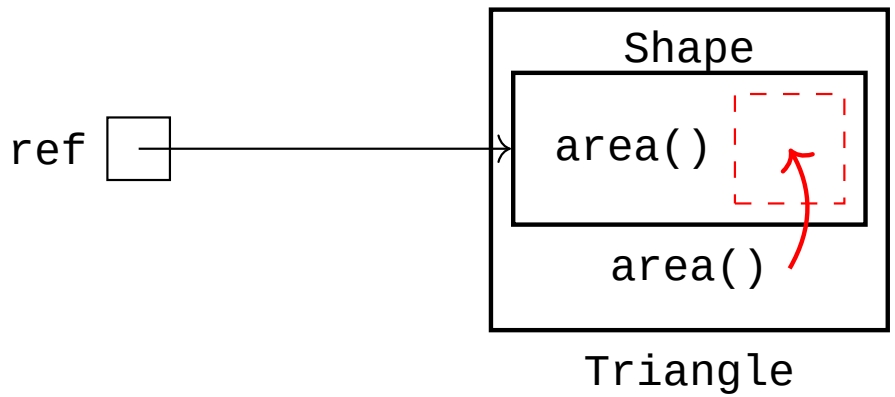
```
Triangle r{...};
Shape& ref {r};
```



Polymorphism

Many forms

```
Triangle r{...};
Shape& ref {r};
```



Polymorphism

Many forms

```
class Shape
{
public:
  // ...
  virtual double area() const
  {
    return 0;
  }
  // ...
};
```

Polymorphism

Now it works!

```
int main()
{
  Rectangle r {10, 15};
  cout << print_area(r) << endl; // prints 150
}
```

Polymorphism

Now it works!

```
int main()
{
  Rectangle r {10, 15};
  cout << print_area(r) << endl; // prints 150
}
```

It works!!

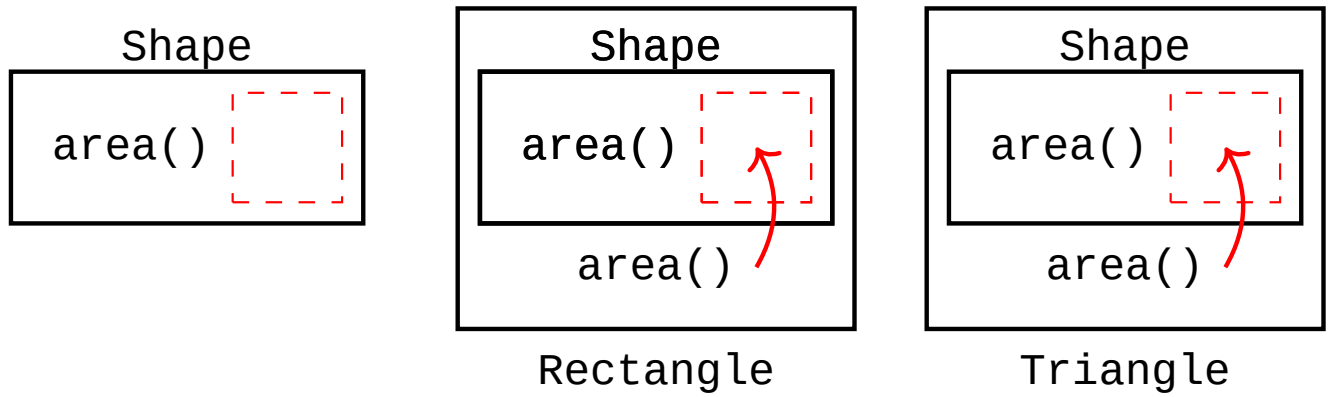
Polymorphism

When can we use polymorphism?

```
Shape s{};
Rectangle r{10, 15};
Triangle  t{3, 4};

Shape* ptr {&s};
ptr->area(); // returns 0

ptr = &r;
ptr->area(); // returns 150

ptr = &t;
ptr->area(); // returns 6
```

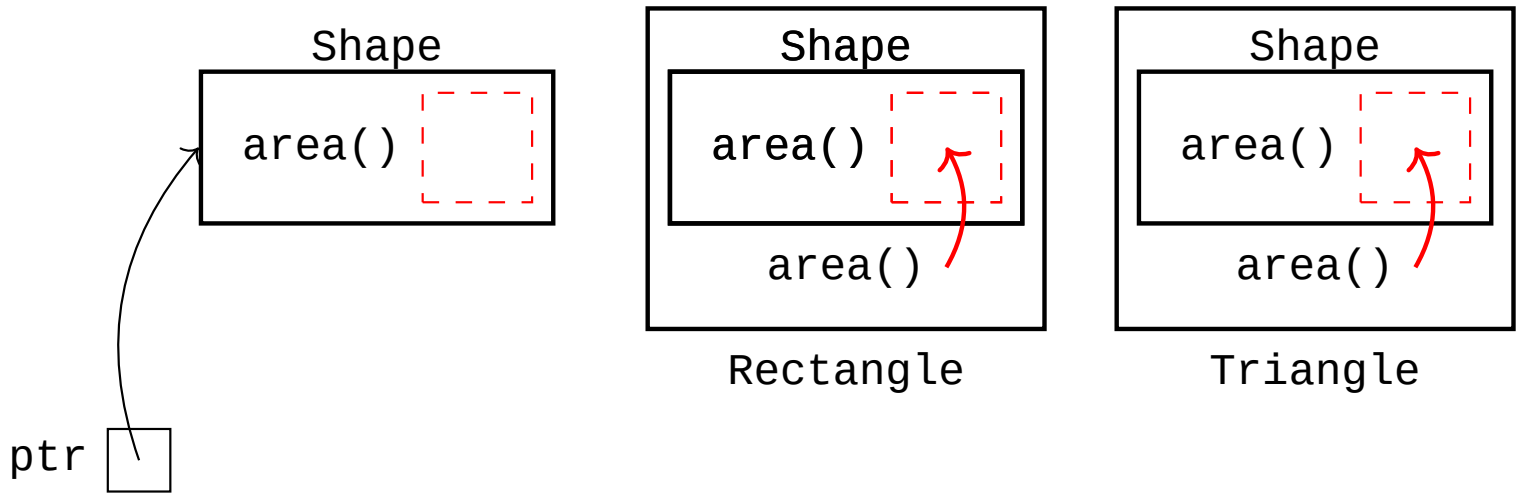
Polymorphism

Pointers & Polymorphism



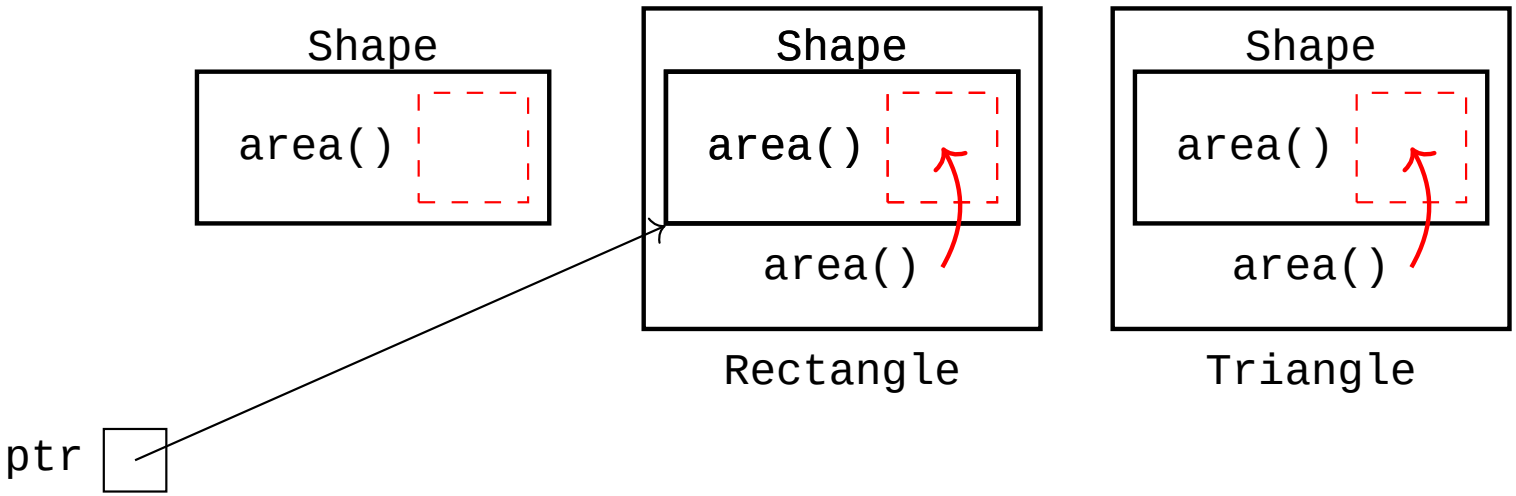
Polymorphism

Pointers & Polymorphism



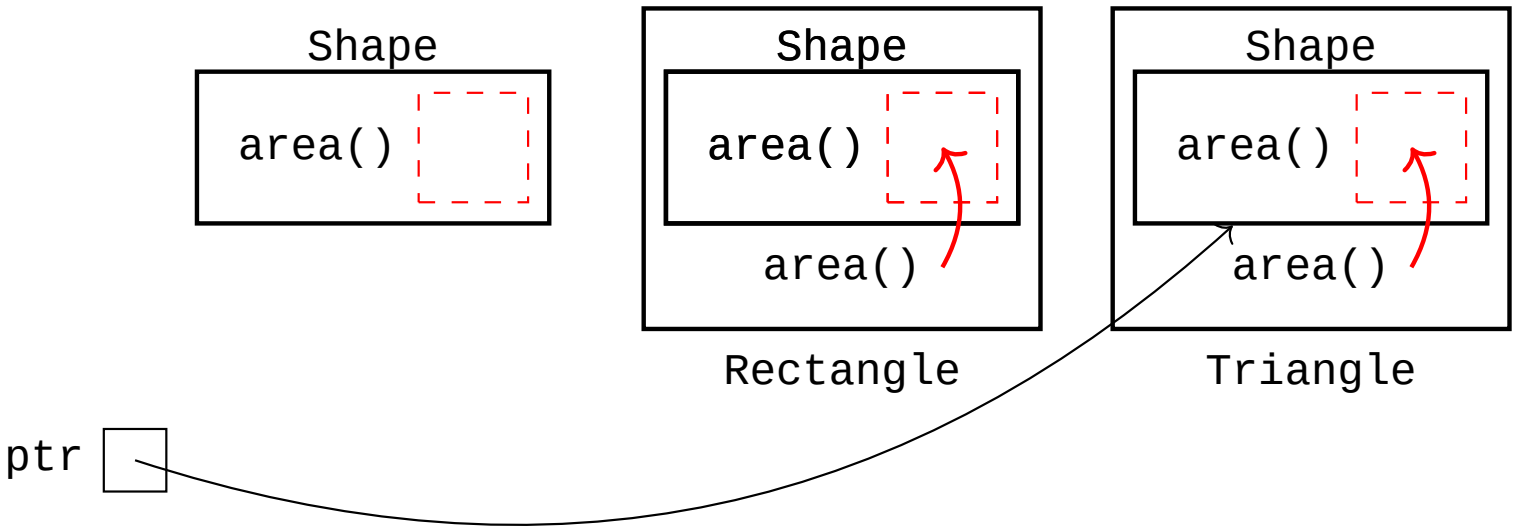
Polymorphism

Pointers & Polymorphism



Polymorphism

Pointers & Polymorphism



Polymorphism

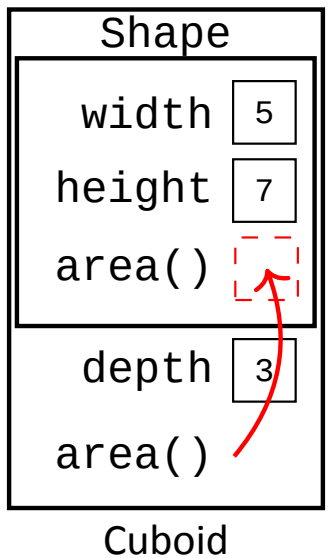
There are pitfalls...

```
class Cuboid : public Shape
{
public:
  Cuboid(double width, double height, double depth)
    : Shape{width, height}, depth{depth}
  { }

  double area() const
  {
    return 2.0 * (width * height + width * depth + height * depth);
  }

private:
  double depth;
};
```

Polymorphism

There are pitfalls...

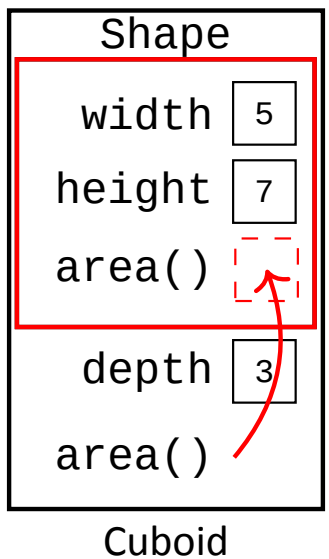
```
Cuboid c{5, 7, 3};
Shape s {c}; // slicing
```



Polymorphism

There are pitfalls...

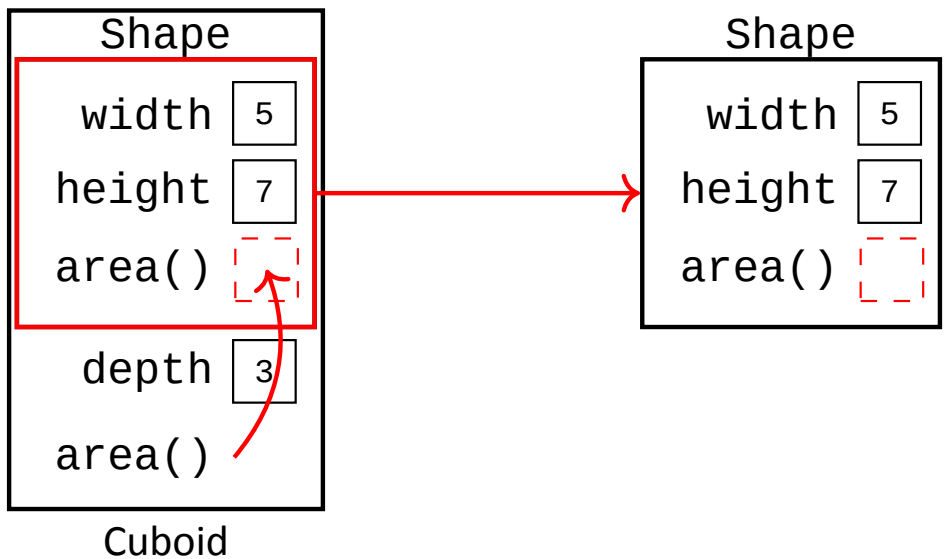
```
Cuboid c{5, 7, 3};
Shape s {c}; // slicing
```



Polymorphism

There are pitfalls...

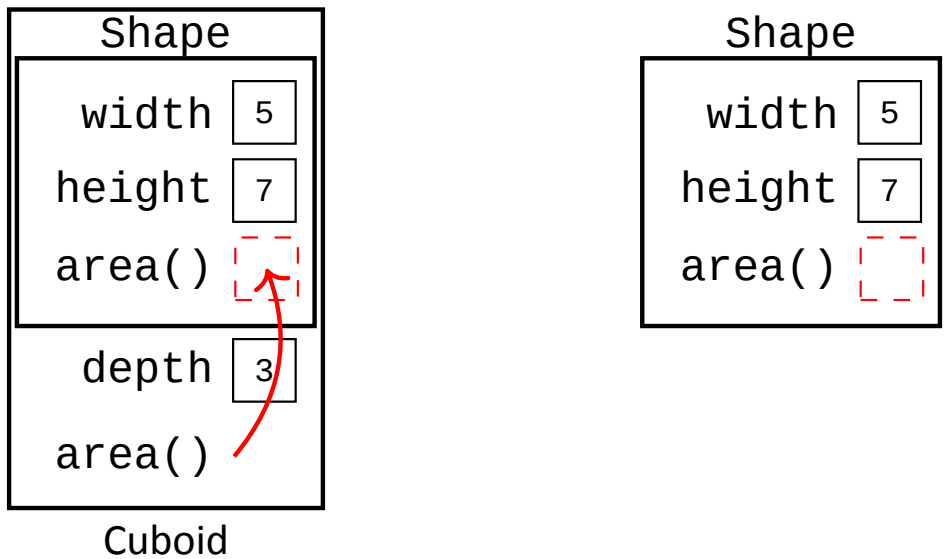
```
Cuboid c{5, 7, 3};
Shape s {c}; // slicing
```



Polymorphism

There are pitfalls...

```
Cuboid c{5, 7, 3};
Shape s {c}; // slicing
```



Polymorphism

There are pitfalls...

```
Cuboid c {2,3,4};
Shape  s {c};
cout << s.area() << endl; // prints 0
```

Polymorphism

There are pitfalls...

```
Cuboid c {2,3,4};
Shape& s {c};
cout << s.area() << endl; // prints 24
```

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer => Call the member function

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer => Call the member function
3. that is non-virtual

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer => Call the member function
3. that is non-virtual => Call the member function

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer => Call the member function
3. that is non-virtual => Call the member function
4. otherwise

Polymorphism

Rule of thumb

When calling a member function:

1. through a non-reference => Call the member function
2. through a non-pointer => Call the member function
3. that is non-virtual => Call the member function
4. otherwise => Call the overriden version

Polymorphism

Conclusion

Always use pointers or references when dealing with polymorphic objects!

Polymorphism

Another good reason for using polymorphism

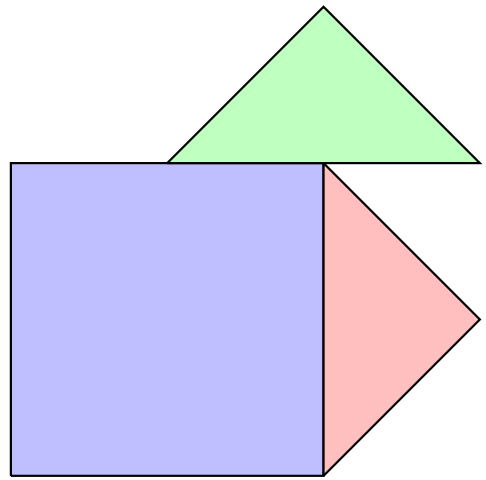
```
std::vector<Shape*> shapes {
  new Triangle{3, 4},
  new Rectangle{5, 6},
  new Cube{3, 5, 7}
};

for (Shape* shape : shapes)
{
  cout << shape->area() << endl;
}
```

LINKÖPING UNIVERSITY

More on Polymorphism

Example

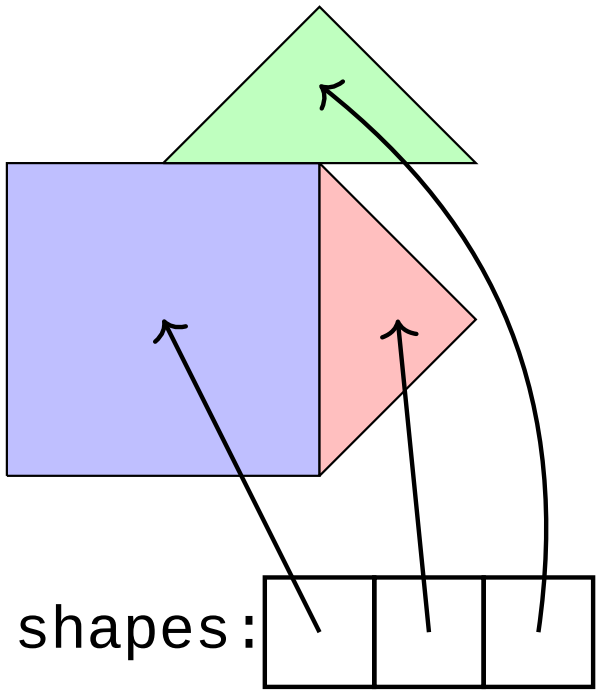
```
class Complex_Shape : public Shape
{
public:
  // ...
  double area() const
  {
    double sum{0.0};
    for (Shape* shape : shapes)
    {
      sum += shape->area();
    }
    return sum;
  }
private:
  std::vector<Shape*> shapes;
};
```



More on Polymorphism

Example

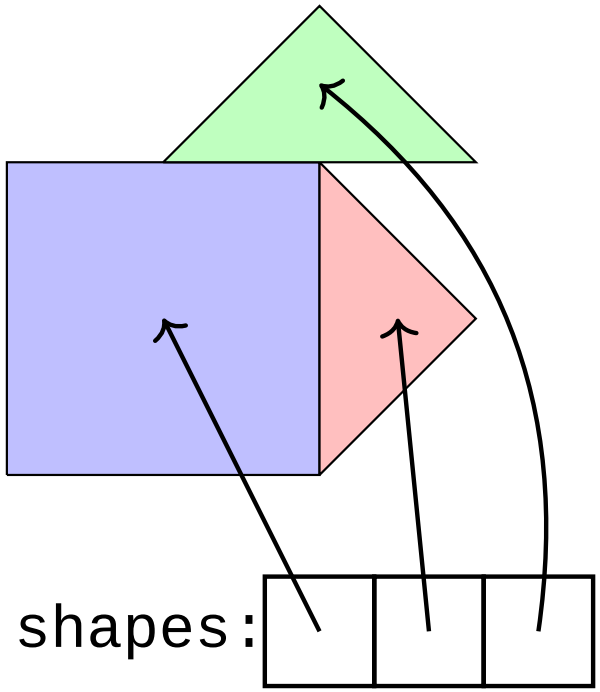
```
class Complex_Shape : public Shape
{
public:
  // ...
  double area() const
  {
    double sum{0.0};
    for (Shape* shape : shapes)
    {
      sum += shape->area();
    }
    return sum;
  }
private:
  std::vector<Shape*> shapes;
};
```



More on Polymorphism

Example

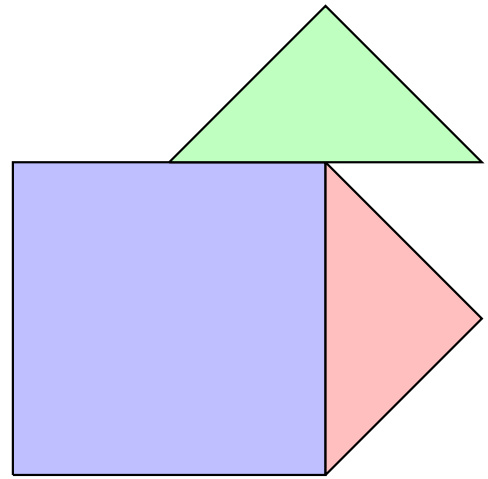
```
{
  Complex_Shape shape { ... };
  cout << shape.area() << endl;
} // what happens here?
```



More on Polymorphism

Example

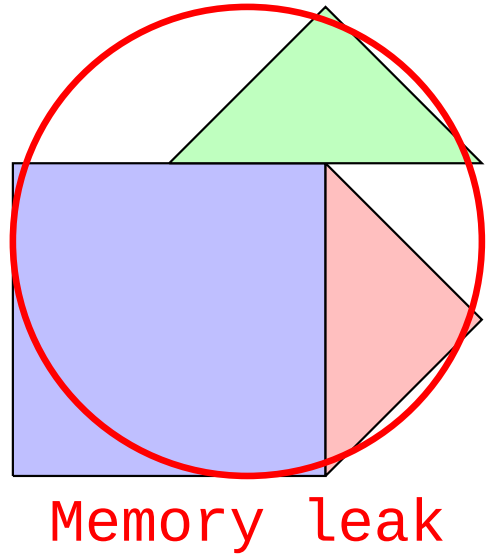
```
{
  Complex_Shape shape { ... };
  cout << shape.area() << endl;
} // what happens here?
```



More on Polymorphism

Example

```
{
  Complex_Shape shape { ... };
  cout << shape.area() << endl;
} // what happens here?
```



More on Polymorphism

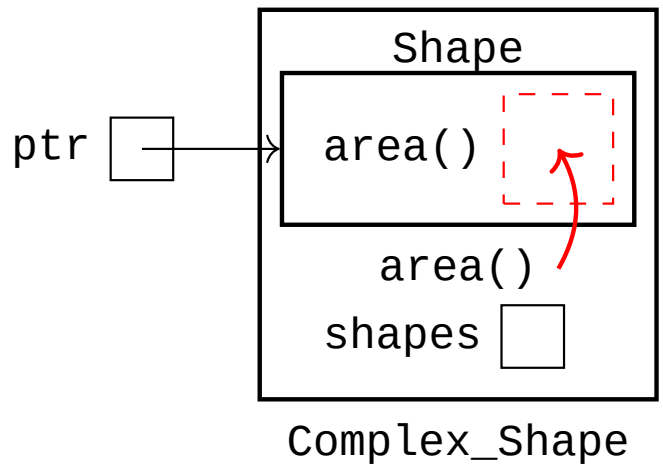
So we create a destructor!

```
class Complex_Shape : public Shape
{
public:
  // ...
  ~Complex_Shape()
  {
    for (Shape* shape : shapes)
    {
      delete shape;
    }
  }
  // ...
};
```

More on Polymorphism

What about now?

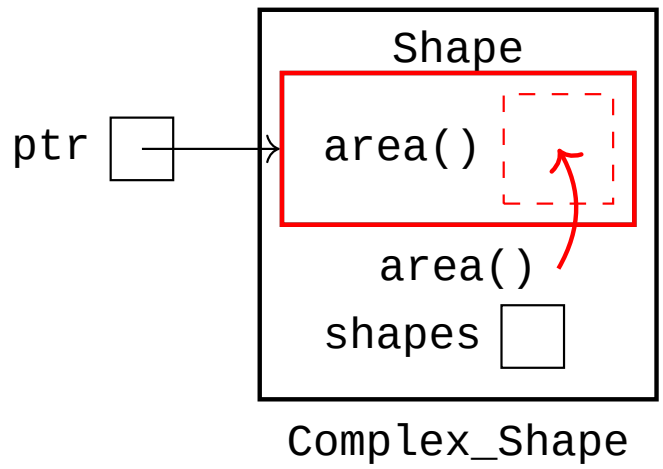
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

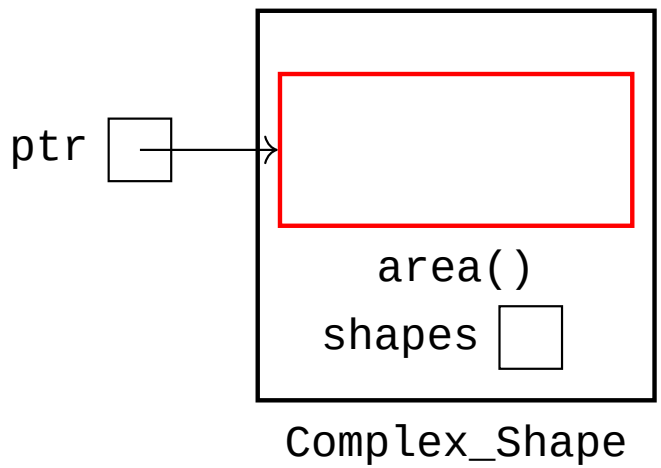
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

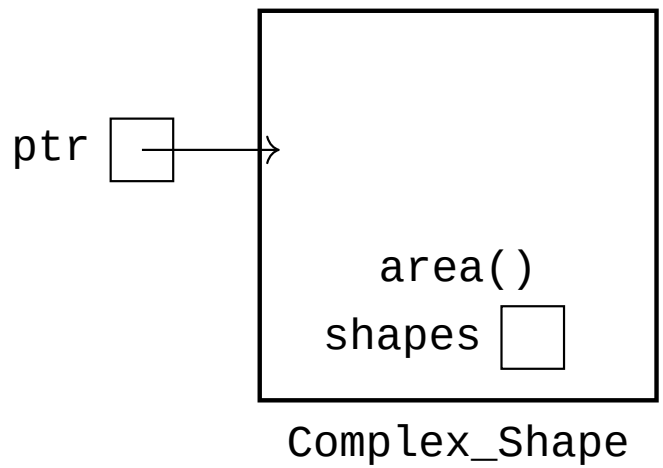
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

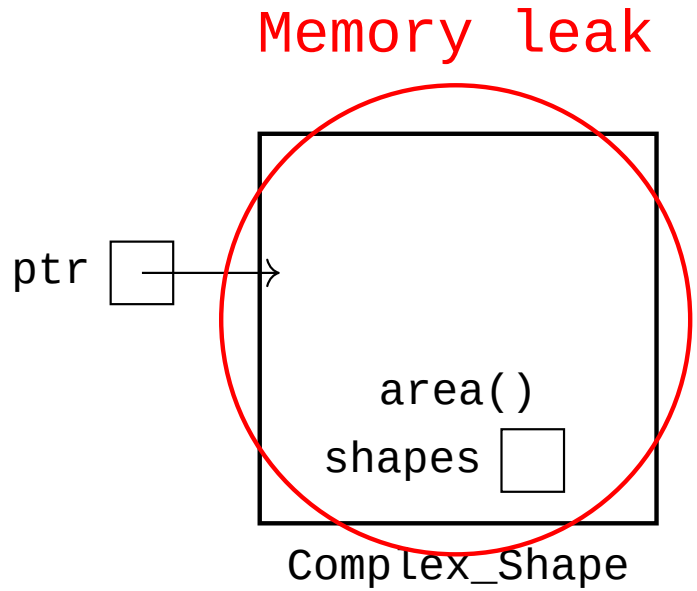
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

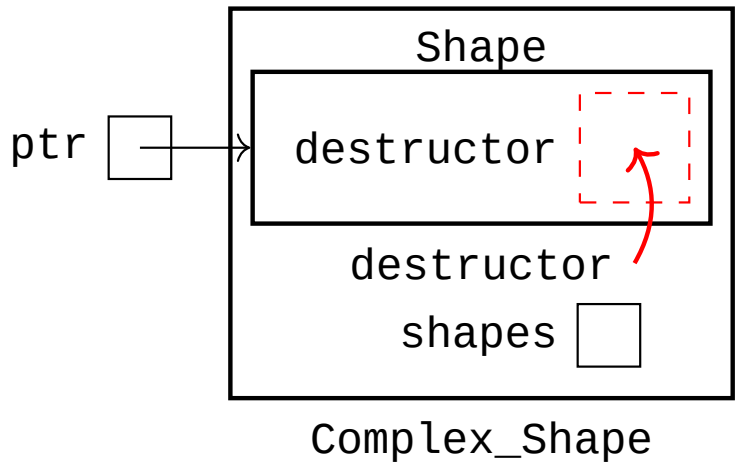
`virtual`-destructor

```
class  Shape
{
public:

  // ...
  virtual ~Shape() = default;
  // ...
};
```

More on Polymorphism

What about now?

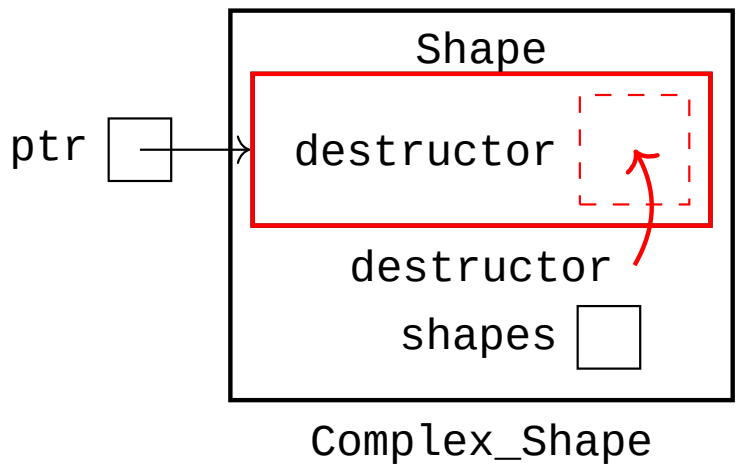
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

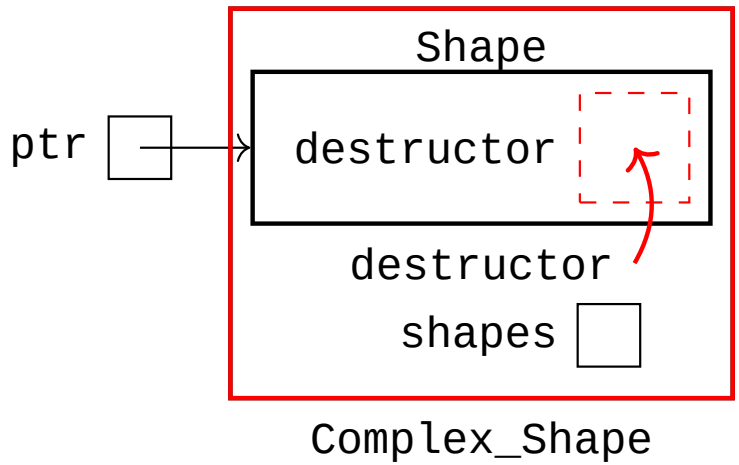
```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

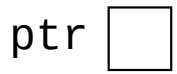
What about now?

```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



More on Polymorphism

What about now?

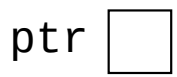



```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```

More on Polymorphism

What about now?

```
Shape* ptr {new Complex_Shape{...}};
delete ptr;
```



Nice!

More on Polymorphism

Conclusion

Always declare the destructor of a polymorphic base class as virtual!

More on Polymorphism

Sometimes humans make mistakes...

```
class My_Shape : public Shape
{
public:
  // ...
  double arae()
  {
    return 10.0;
  }
  // ...
};
```

More on Polymorphism

Sometimes humans make mistakes...

```
Shape* ptr {new My_Shape{}};
cout << ptr->area() << endl;
delete ptr;
```

More on Polymorphism

Sometimes humans make mistakes...

```
Shape* ptr {new My_Shape{}};
cout << ptr->area() << endl; // prints 0 (?!)
delete ptr;
```

More on Polymorphism

Sometimes humans make mistakes...

```
class My_Shape : public Shape
{
public:
  // ...
  double arae()
  {
    return 10.0;
  }
  // ...
};
```

Aha! A misspelling!

More on Polymorphism

Sometimes humans make mistakes...

```
class My_Shape : public Shape
{
public:
  // ...
  double area()
  {
    return 10.0;
  }
  // ...
};
```

More on Polymorphism

Sometimes humans make mistakes...

```
Shape* ptr {new My_Shape{}};
cout << ptr->area() << endl;
delete ptr;
```

More on Polymorphism

Sometimes humans make mistakes...

```
Shape* ptr {new My_Shape{}};
cout << ptr->area() << endl; // STILL 0 ?!
delete ptr;
```

More on Polymorphism

Sometimes humans make mistakes...

```
class My_Shape : public Shape
{
public:
  // ...
  double area() const
  {
    return 10.0;
  }
  // ...
};
```

We forgot const!

More on Polymorphism

Sometimes humans make mistakes...

```
Shape* ptr {new My_Shape{}};
cout << ptr->area() << endl; // prints 10
delete ptr;
```

More on Polymorphism

Can’t the compiler help us with these simple mistakes?

```
class My_Shape : public Shape
{
public:
  // ...
  double arae() override
  {
    return 10.0;
  }
  // ...
};
```

More on Polymorphism

Can’t the compiler help us with these simple mistakes?

```
shape.cc: error: ‘double My_Shape::arae()’ marked ‘override’,
 but does not override
     double arae() override
            ^~~~
```

More on Polymorphism

Rule of thumb

Always mark functions that are meant to override as override!

More on Polymorphism

Let's go back to Shape

```
class  Shape
{
public:

  // ...
  virtual ~Shape() = default;
  virtual double area() const
  {
    return 0;
  }
  // ...
};
```

More on Polymorphism

pure-virtual function

```
class  Shape
{
public:

  // ...
  virtual ~Shape() = default;
  virtual double area() const = 0;
};
```

More on Polymorphism

Abstract class

A class is *abstract* if it contains one or more pure-virtual functions

More on Polymorphism

Abstract class

```
Shape s1{1, 3};  // Error: abstract
Triangle t{1,3}; // OK: not abstract
Shape s2{t};     // Error: abstract
Shape& s3{t};    // OK: reference allowed
Shape* s4{&t};   // OK: pointer allowed
```

More on Polymorphism

Importing things from the base class

```
class Shape
{
public:

  Shape(double w, double h)
    : width{w}, height{h}
  {
  }

  // ...

protected:

  double width;
  double height;
};
```

```
class Rectangle : public Shape
{
public:

  // create an identical constructor
  // as the one in Shape
  using Shape::Shape;

  // make width public in Rectangle
  using Shape::width;

private:

  // make height private in Rectangle
  using Shape::height;
};
```

LINKÖPING UNIVERSITY

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static:

Dynamic:

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static:

Dynamic:

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static: Shape*

Dynamic:

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static: Shape*

Dynamic:

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static: Shape*

Dynamic: Triangle

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static: Shape*

Dynamic: Triangle

Type information

Static vs Dynamic type

```
Shape* ptr {new Triangle{3, 5}};
cout << ptr->area() << endl;

delete ptr;
ptr = new Rectangle{3, 5};
```

Static: Shape*

Dynamic: Rectangle

Type information

Example

```
class Cuboid : public Shape
{
public:
  // ...
  virtual double volume() const
  {
    return width * height * depth;
  }
  //...
};
```

Type information

Example

```
Shape* ptr {new Cuboid{3, 4, 5}};

// doesn't work, volume is not
// declared in Shape
cout << ptr->volume() << endl;
```

Type information

Example

```
Shape* ptr {new Cuboid{3, 4, 5}};

cout << static_cast<Cuboid*>(ptr)->volume()
     << endl;
```

Type information

When it all comes crashing down...

```
Shape* ptr {new Rectangle{3, 4}};

cout << static_cast<Cuboid*>(ptr)->volume()
     << endl;
```

Type information

When it all comes crashing down...

```
Shape* ptr {new Rectangle{3, 4}};

cout << static_cast<Cuboid*>(ptr)->volume()
     << endl;
```

Segmentation Fault

Type information

`dynamic_cast`

```
Shape* ptr1 {new Cuboid{3, 4, 5}};
Shape* ptr2 {new Rectangle{3, 4}};

Cuboid* c1 {dynamic_cast<Cuboid*>(ptr1)};
Cuboid* c2 {dynamic_cast<Cuboid*>(ptr2)};

// c1 is a pointer to a valid Cuboid object

// c2 == nullptr, since ptr2 does not
// point to a valid Cuboid object
```

Type information

Checking if dynamic type is compatible

```
Shape* ptr {...};

Cuboid* cuboid {dynamic_cast<Cuboid*>(ptr)};
if (cuboid != nullptr)
{
  // only print volume if it is a cuboid
  cout << cuboid->volume() << endl;
}
```

Type information

Also works with references!

```
Cuboid c {3,4,5};
Shape& s {c};

cout << dynamic_cast<Cuboid&>(s).volume() << endl;
```

Type information

Also works with references!

```
Rectangle r {3,4};
Shape& s {c};

cout << dynamic_cast<Cuboid&>(s).volume() << endl;
```

Type information

Also works with references!

```
$ g++ shape.cc
$ ./a.out
terminate called after throwing an instance of 'std::bad_cast'
  what():  std::bad_cast
Aborted (core dumped)
```

LINKÖPING UNIVERSITY

Exceptions

What just happend?!

Exceptions

What just happend?!

- References cannot be empty

Exceptions

What just happend?!

- References cannot be empty
- What do we do to signal error?

Exceptions

What just happend?!

- References cannot be empty
- What do we do to signal error?
- Exceptions!

Exceptions

Model

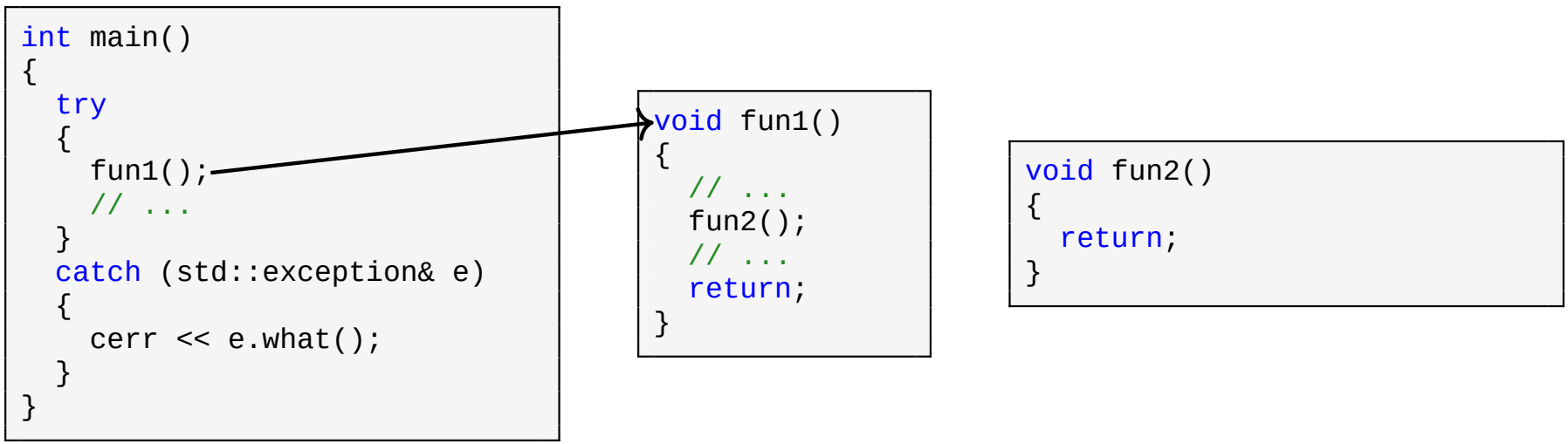
```
int main()
{
  try
  {
    fun1();
    // ...
  }
  catch (std::exception& e)
  {
    cerr << e.what();
  }
}
```

```
void fun1()
{
  // ...
  fun2();
  // ...
  return;
}
```

```
void fun2()
{
  return;
}
```

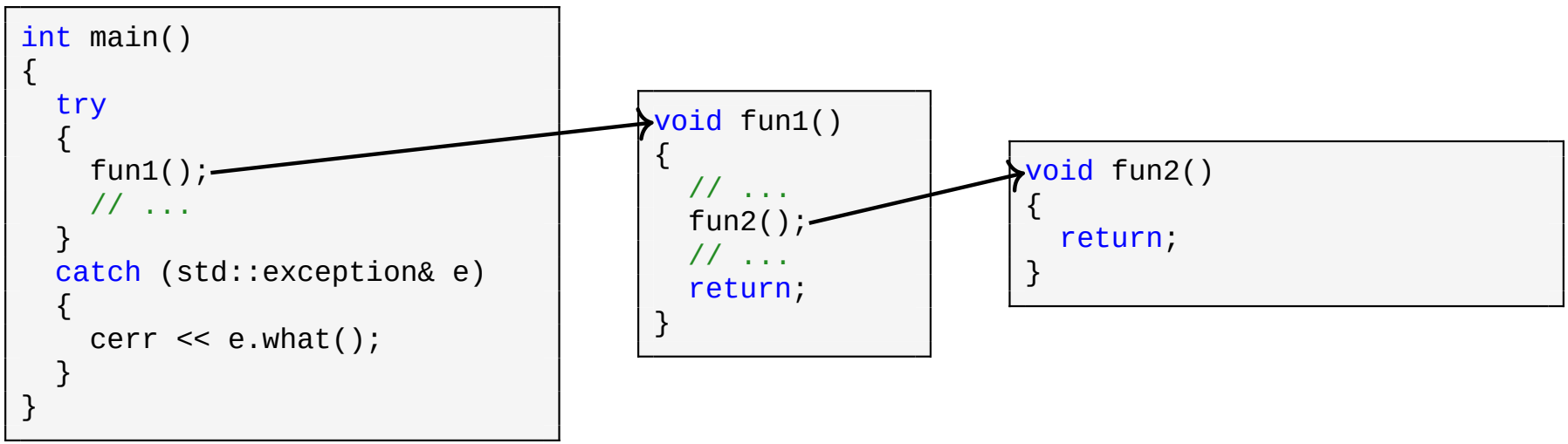
Exceptions

Model



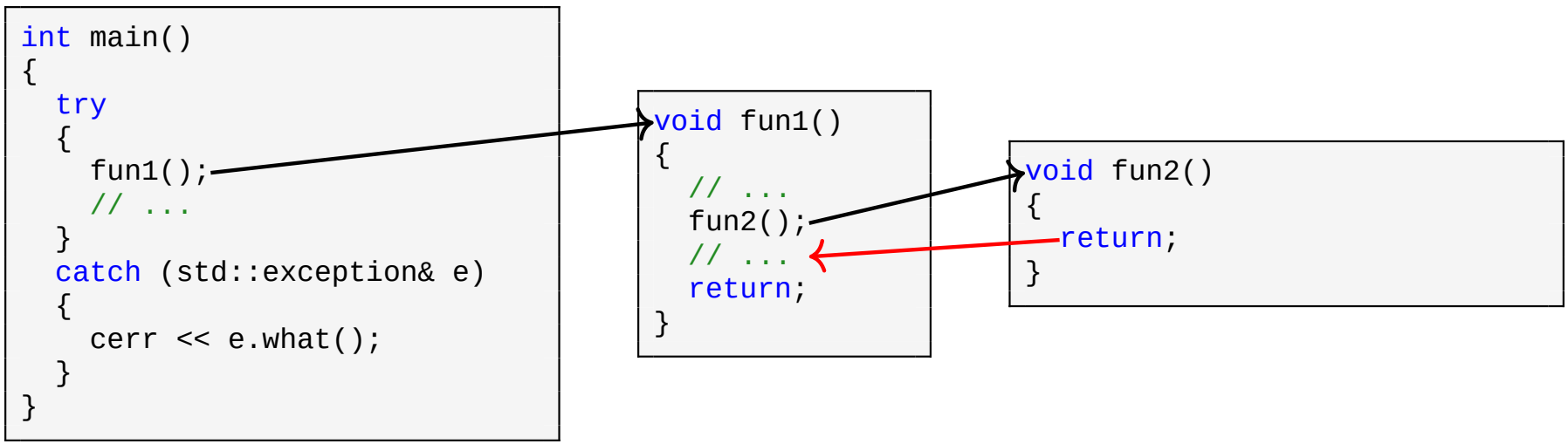
Exceptions

Model



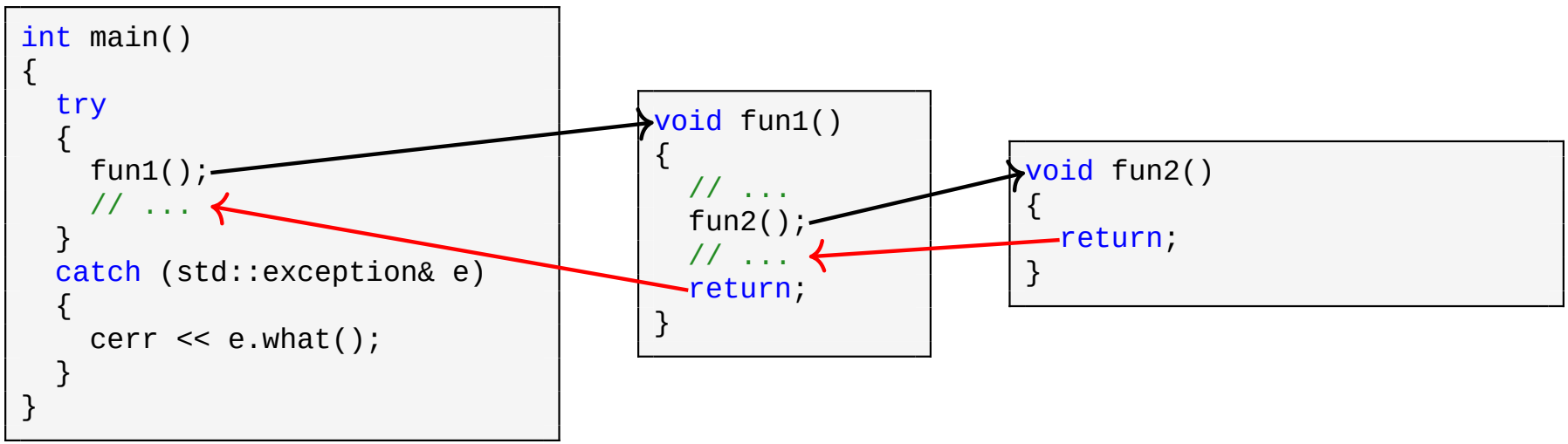
Exceptions

Model



Exceptions

Model



Exceptions

Model

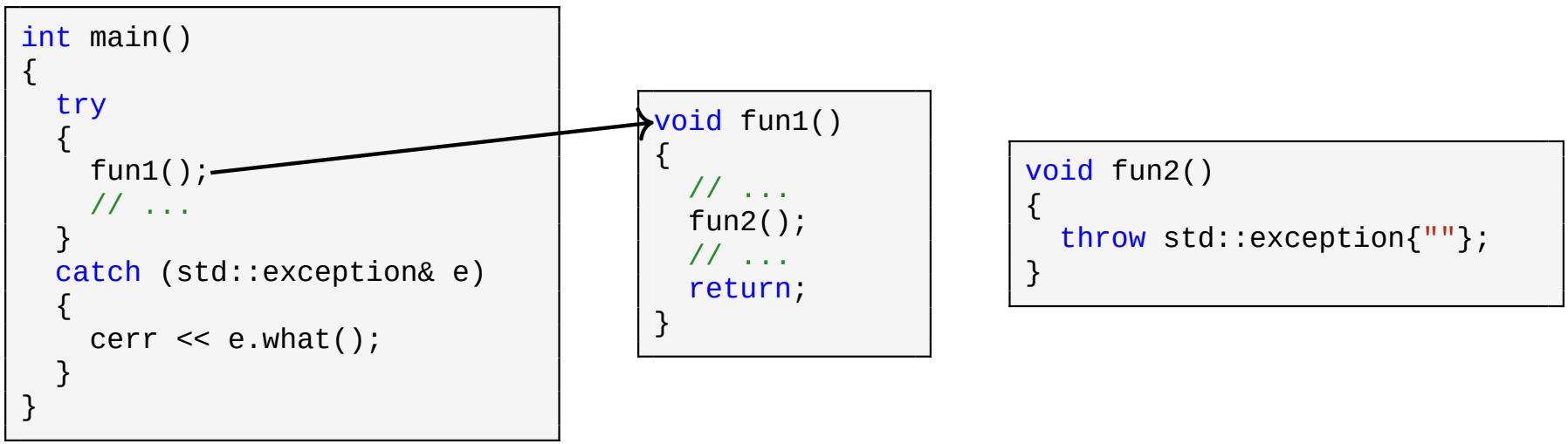
```
int main()
{
  try
  {
    fun1();
    // ...
  }
  catch (std::exception& e)
  {
    cerr << e.what();
  }
}
```

```
void fun1()
{
  // ...
  fun2();
  // ...
  return;
}
```

```
void fun2()
{
  throw std::exception{""};
}
```

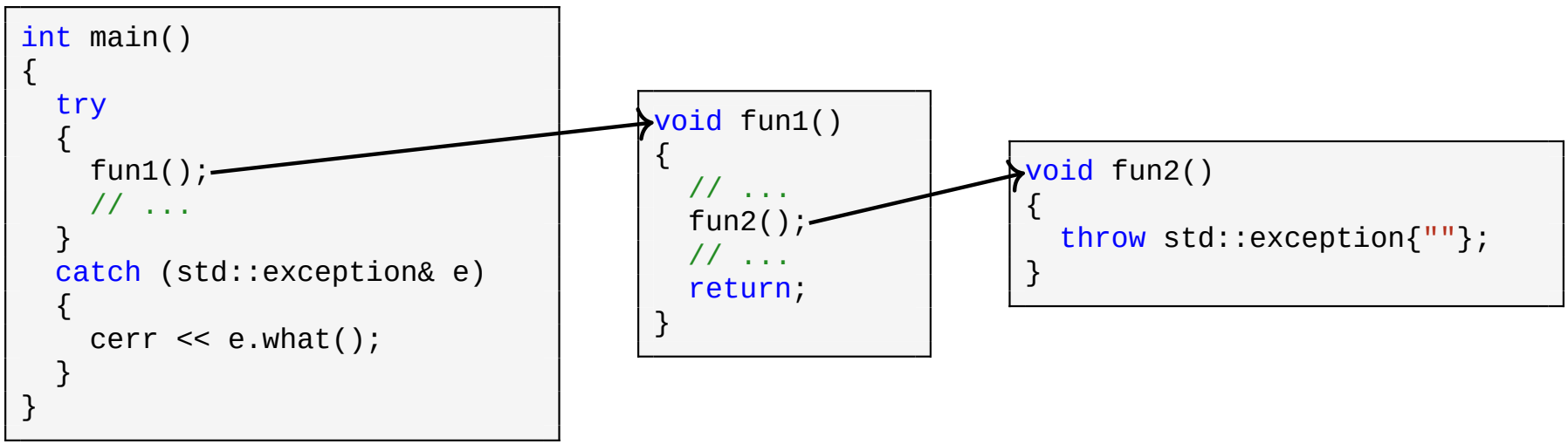
Exceptions

Model



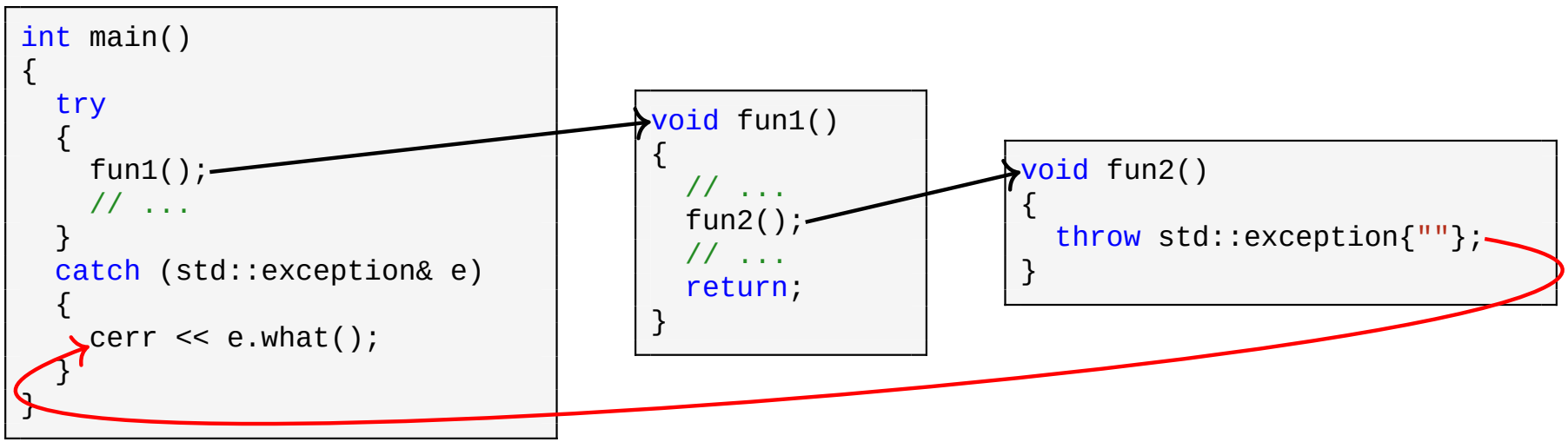
Exceptions

Model



Exceptions

Model



Exceptions

dynamic_cast

```
#include <stdexcept>

int main()
{
  Rectangle r {3,4};
  Shape& s {c};

  try
  {
    cout << dynamic_cast<Cuboid&>(s).volume()  << endl;
  }
  catch (std::bad_cast& e)
  {
    cout << "s is not a Cuboid!" << endl;
  }
  catch (std::exception& e)
  {
    cout << "Unknown error." << endl;
  }
}
```

LINKÖPING UNIVERSITY

Command-line argument

Calling a program with arguments

```
$ ./a.out a b c
```

Command-line argument

Calling a program with arguments

```
$ ./a.out a b c
```

Arguments: a, b, c

Command-line argument

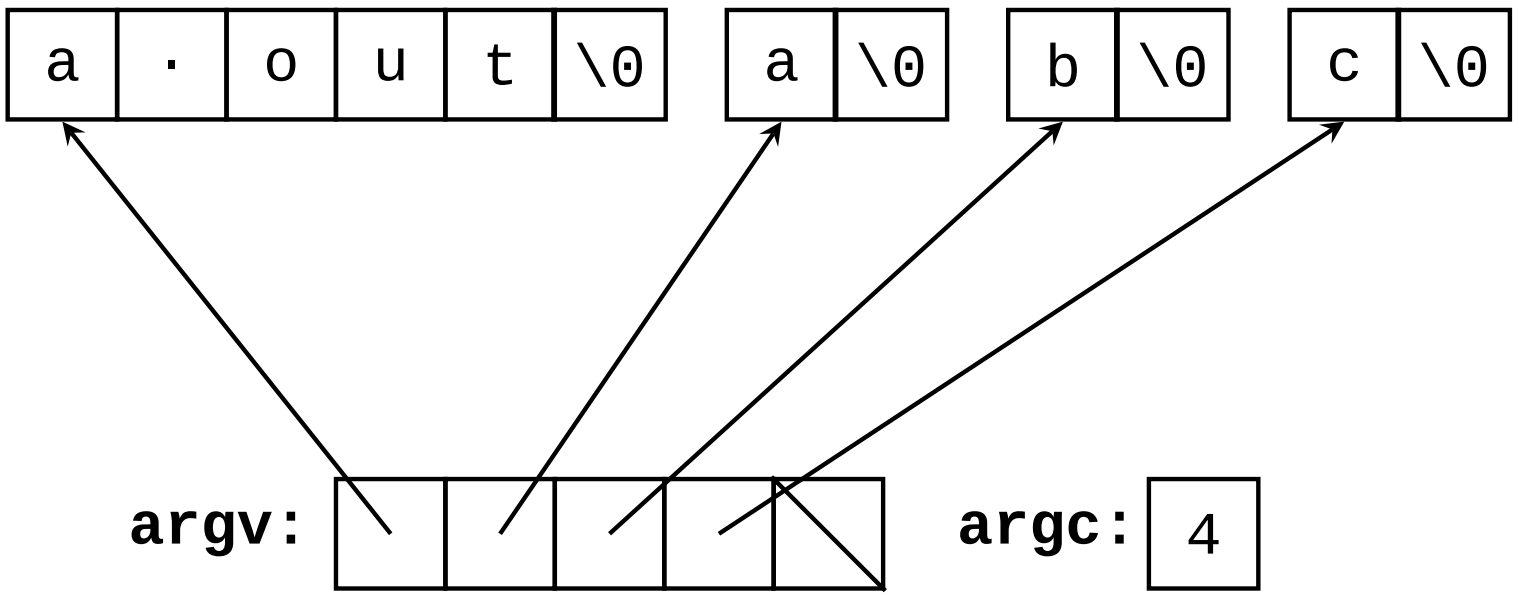
Reading arguments

```
int main(int argc, char* argv[])
{
  // argc = number of arguments passed to the program
  // argv = a pointer to an array of pointers to C-strings
}
```

Command-line argument

`argv`

```
$ a.out a b c
```



Command-line argument

Example

```
int main(int argc, char** argv)
{
  for (int i{0}; i < argc; ++i)
  {
    cout << argv[i] << endl;
  }
}
```

Command-line argument

Example

```
$ ./a.out 10 20 30
./a.out
10
20
30
```

Command-line argument

Converting arguments

- `std::stoi(argv[1])` - convert `argv[1]` to `int`
- `std::stod(argv[1])` - convert `argv[1]` to `double`
- Using `std::stringstream`:

```
std::stringstream ss{};
ss << argv[1];

int number;
ss >> number;
```

Command-line argument

Cool trick

```
vector<string> args { argv, argv + argc };
// now all arguments reside in the vector
// as std::string instead of C-strings
```

www.liu.se

LINKÖPING UNIVERSITY