

# TDDE18 & 726G77

Inheritance & Polymorphism

Christoffer Holm

Department of Computer and information science

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

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

# std::vector

## Storage

- Linked storage
- Sequential Storage

# std::vector

## Storage

- Linked storage
  - Nodes linked together with pointers.
  - This is what we did in the List lab.
  - Very slow to access values in the middle of the collection since we have to loop from the beginning every time.
- Sequential Storage

# std::vector

## Storage

- Linked storage
- Sequential Storage
  - If we place everything next to each other in memory, then we know where each element is.
  - This is faster for accessing values in the middle.
  - However, it is now slower to insert values between two values and at the beginning.
  - This is how std::vector works!

# std::vector

Sequential Storage

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

# std::vector

## Sequential Storage

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

5	3	1	2
[0]	[1]	[2]	[3]



# std::vector

## Sequential Storage

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

5	3	1	2
[0]	[1]	[2]	[3]

# std::vector

## Sequential Storage

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

5	4	1	2
[0]	[1]	[2]	[3]

# std::vector

## Sequential Storage

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

5	4	1	2
[0]	[1]	[2]	[3]

# std::vector

## Sequential Storage

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

5	4	1	2	3
[0]	[1]	[2]	[3]	[4]

# std::vector

## Sequential Storage

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

5	4	1	2	3
[0]	[1]	[2]	[3]	[4]

# std::vector

## Sequential Storage

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

5	4	1	2	6
[0]	[1]	[2]	[3]	[4]

# std::vector

## Sequential Storage

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

5	4	1	2	6
[0]	[1]	[2]	[3]	[4]

# std::vector

## Sequential Storage

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

5	4	1	2
[0]	[1]	[2]	[3]



# std::vector

## Sequential Storage

- std::vector is defined in `#include <vector>`
- Declared like this: `std::vector<T>`.
- A `std::vector<T>` contains a sequence of values that has the data type T.
- For example: `std::vector<int>` is a vector that stores integers.

# `std::vector`

## Sequential Storage

- Each element in a `std::vector` is indexed, beginning with 0 being the first element.
- Element `i` in vector `v` can be accessed with either `v[i]` or `v.at(i)`.
- `v.at(i)` will check that element `i` exists, so it is preferred over `v[i]`.
- First element can be accessed with `v.front()` and last with `v.back()`.

# `std::vector`

## Sequential Storage

- It is possible to insert values at the end with `v.push_back(3)`.
- To remove the last element, you write `v.pop_back()`.
- To see how many elements there are, write `v.size()`.

# 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

## Looping through

- There are multiple ways to loop through a vector
- The first is to use a counter that goes through each index in order.
- The second way is what's known as a *range based for-loop*.
- A range based for-loop looks like this:  
`for (int e : v)`

# std::vector

## Looping through

- You can read it as: Loop through `v`, for each iteration the current element is stored in `e`.
- However, each element is *copied* into `e`.
- `for (int& e : v)` does not copy the element in to `e`, and it allows us to change the values inside the loop.
- Since copying is unnecessary for most cases where we want to read the elements, it is recommended that you loop through `v` like this: `for (int const& e : v)`



# 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;
    }
}
```

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

# 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;
};
```

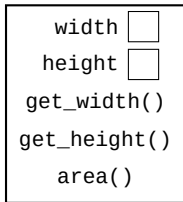
# Inheritance

Is there a problem?

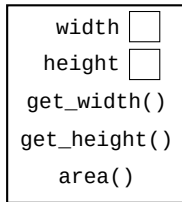
- There is a lot of code repetition here.
- We want to factor out common code, just as we did with similar functions.
- In the example above, `Rectangle` and `Triangle` share everything except the implementation of `area()`.
- How do we do this?

# Inheritance

What is inheritance?



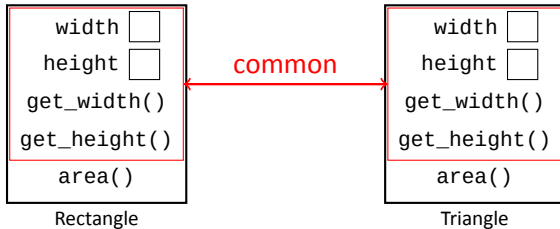
Rectangle



Triangle

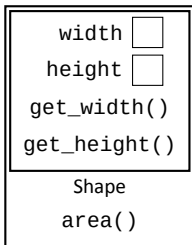
# Inheritance

What is inheritance?

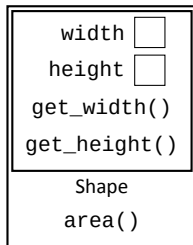


# Inheritance

What is inheritance?



Rectangle



Triangle

# Inheritance

## Terminology

- The class that contains the shared functionality is called a *Base class*.
- A class that inherits another class (a base class) is called a *Derived class*.
- A derived class inherits all the members (both functions and variables) from its base class.
- Sometimes we say that the derived class *extends* the base class, i.e. it takes everything from the base class and then add more things on top of that.



# Inheritance

## Syntax

```
class Base
{
    // ...
};

class Derived : public Base
{
    // ...
};
```

# Inheritance

## Syntax

- Derived *inherits* from Base.
- This is done by adding : `public` Base at the end of the class declaration.
- I.e. by writing: `class` Derived : `public` Base

# 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

## Delegating constructor

- width and height are `private` in Shape.
- This means that Rectangle does not have access to them.
- The constructor can therefore not initialize those members.
- But, we can call the constructor of Shape which does in fact have access to them to initialize these objects.
- You do this by adding `Shape{w, h}` to the start of the member initialization list.

# Inheritance

## Delegating constructor

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

# 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

## protected

- As mentioned before; width and height are **private** in Shape.
- This means that neither `Rectangle::area` nor `Triangle::area` have access to these variables.
- There are two ways to solve it: replace each access to width with `get_width()` and likewise for height,
- **OR** we make sure that width and height are available for Rectangle and Triangle.

# 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

## protected

- `protected` is the third and final access specifier for members in a class.
- It is the same as `private`, but with one difference: these members are also accessible by all derived classes.
- Which means: `protected` things are secrets kept within the family (inheritance hierarchy), while `private` things are secrets kept by the individual (class).

# 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"};
```

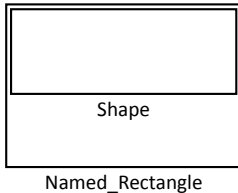


Named\_Rectangle

# Inheritance

## Initialization & Destruction

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

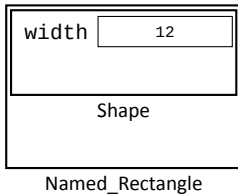




# Inheritance

## Initialization & Destruction

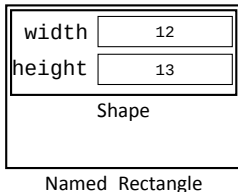
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



# Inheritance

## Initialization & Destruction

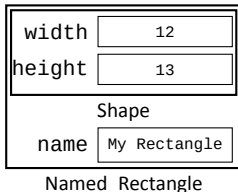
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



# Inheritance

## Initialization & Destruction

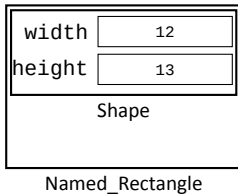
```
Named_Rectangle r {12, 13, "My Rectangle"};
```



# Inheritance

## Initialization & Destruction

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



# Inheritance

## Initialization & Destruction

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

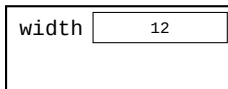
width	12
height	13

Shape

# Inheritance

## Initialization & Destruction

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



Shape

# Inheritance

## Initialization & Destruction

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



Shape

# Inheritance

## Initialization & Destruction

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



# Inheritance

## Initialization & Destruction

- The top base class of the hierarchy will be constructed first and then its derived class.
- Each data member will be construct top-to-bottom in declaration order (irregardless of the order in the data member initialization list).
- The objects will be destructed in reverse order of construction by first destroying each data member bottom-to-top and then recursively destroying the base class.

# 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

## Binding to references

- The implementation for both versions of `print_height()` are exactly the same.
- Since `get_height()` for `Rectangle` and `Triangle` is implemented in `Shape`, we can get away with just looking at the `Shape` part of the objects.
- By taking the parameter as a `Shape&` we can bind both `Rectangle` and `Triangle` in the same function.

# 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

area()

- The parameter shape is of type Shape&, meaning we can only access things that resides in Shape.
- This means that we cannot call area since it hasn't been declared in Shape.

# 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
}
```

# Inheritance

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

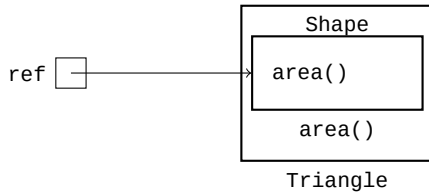
- We can solve the problem by adding `area()` to `Shape`!
- However this poses a new problem. In `print_area()` we always call `Shape::area()`.
- This is not what we want, we want to call the `area()` function of whichever type we pass in to the function...
- This problem can be solved with *Polymorphism*!

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

# Polymorphism

Many forms

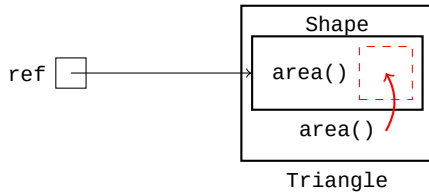
```
Triangle r{...};  
Shape& ref {r};
```



# Polymorphism

Many forms

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



# Polymorphism

## Many forms

- The way we solve the problem with `print_area()` calling the wrong version is by letting derived classes override the functionality of `Shape::area()`.
- I.e. we want the implementation of `Shape::area()` to be replaceable,
- because then the derived class could simply replace the implementation of `area()` in `Shape` with its own implementation of `area()`.
- This is done by declaring `Shape::area()` as `virtual`.

# 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};

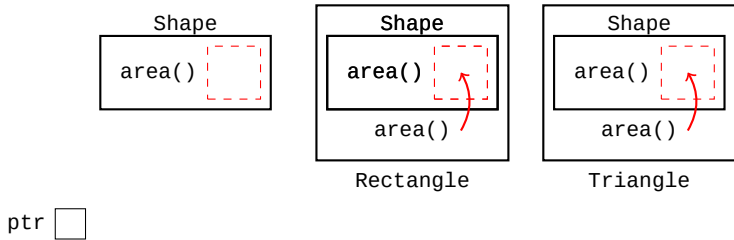
Shape* ptr {&s};
ptr->area(); // returns 0

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

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

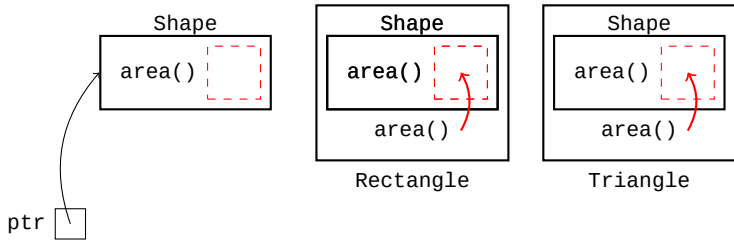
# Polymorphism

## Pointers & Polymorphism



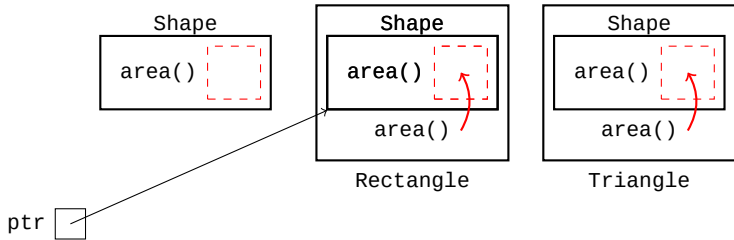
# Polymorphism

## Pointers & Polymorphism



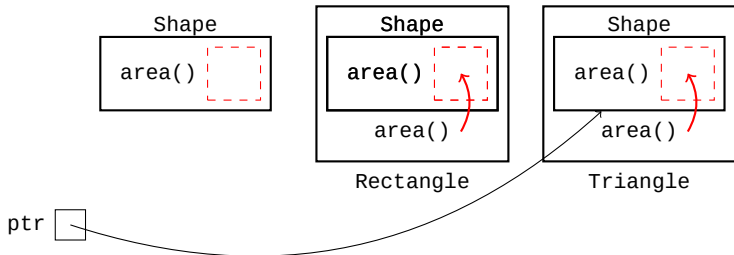
# Polymorphism

## 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}
    { }

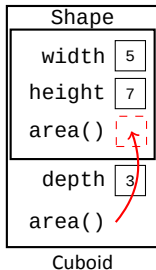
    double area() const
    {
        return 2.0 * (width * height + width * depth + height * depth);
    }

private:
    double depth;
};
```

# Polymorphism

There are pitfalls...

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

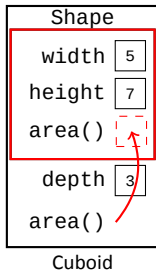




# Polymorphism

There are pitfalls...

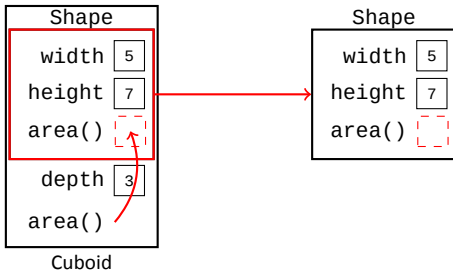
```
Cuboid c{5, 7, 3};  
Shape s {c}; // slicing
```



# Polymorphism

There are pitfalls...

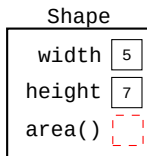
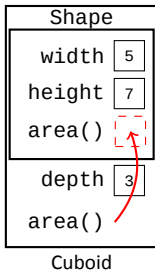
```
Cuboid c{5, 7, 3};  
Shape s {c}; // slicing
```



# Polymorphism

There are pitfalls...

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



# Polymorphism

There are pitfalls...

- It is possible to copy from a derived type into a the Base class
- However, a variable has a fixed size, so when the derived class has more members than the base class, these will be lost.
- This is called *slicing* since we slice away everything that does not fit in the Shape-object.

# 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 overridden version

# Polymorphism

## Conclusion

Always use pointers or references when dealing with polymorphic objects!

# Polymorphism

## Conclusion

- If we always use pointers of references:
- we are guaranteed to always call the correct version,
- we avoid the problems with slicing,
- we don't have to copy objects if not necessary.



# 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;  
}
```

# Polymorphism

Another good reason for using polymorphism

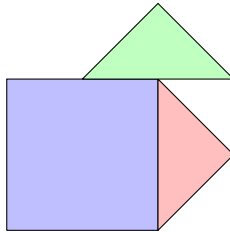
- If we have a shared base class with **virtual** functions:
- We can have base class pointer to objects of derived classes
- This means we can store different types inside an `std::vector`.
- This is useful because we can now iterate over objects of different types and get different results based on the “real” type of the objects.

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

# More on Polymorphism

## Example

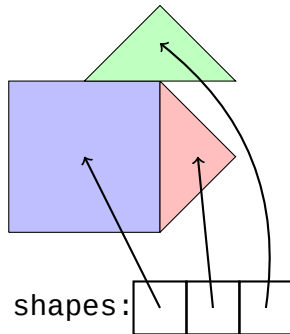
```
class Complex_Shape : public Shape
{
public:
    // ...
    double area() const
    {
        double sum{0.0};
        for (Shape* shape : shapes)
        {
            sum += shape->area();
        }
        return sum;
    }
private:
    std::vector<Shape*> shapes;
};
```



# More on Polymorphism

## Example

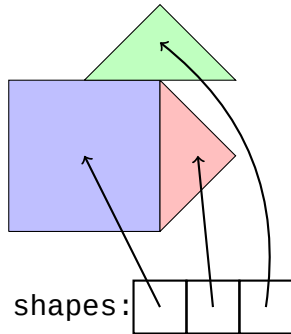
```
class Complex_Shape : public Shape
{
public:
    // ...
    double area() const
    {
        double sum{0.0};
        for (Shape* shape : shapes)
        {
            sum += shape->area();
        }
        return sum;
    }
private:
    std::vector<Shape*> shapes;
};
```



# More on Polymorphism

## Example

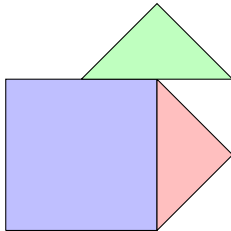
```
{  
  Complex_Shape shape { ... };  
  cout << shape.area() << endl;  
} // what happens here?
```



# More on Polymorphism

## Example

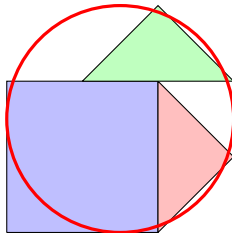
```
{  
  Complex_Shape shape { ... };  
  cout << shape.area() << endl;  
} // what happens here?
```



# More on Polymorphism

## Example

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



Memory leak



# 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

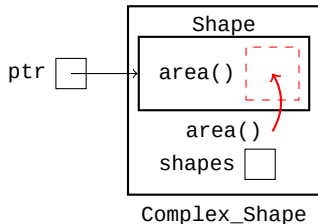
So we create a destructor!

- When having manually managed memory in a vector we have to delete it manually in the destructor.
- So of course we need one for `ComplexShape` since it keeps a record of various shapes.

## More on Polymorphism

What about now?

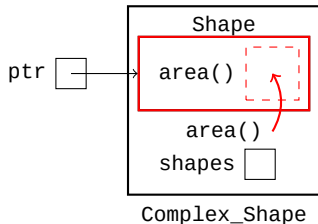
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



## More on Polymorphism

What about now?

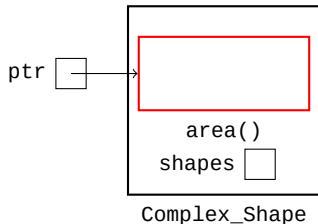
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



## More on Polymorphism

What about now?

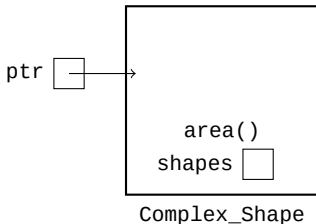
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



## More on Polymorphism

What about now?

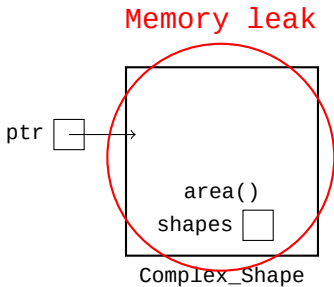
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



## More on Polymorphism

What about now?

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



## More on Polymorphism

What about now?

- When deleting `ptr` the compiler only sees the Shape-portion of the object.
- This means that it will call the destructor for Shape, even though it is really a `Complex_Shape`.
- So the problem is essentially that the compiler gets tricked into thinking you are working with a Shape object.
- We solved this problem earlier by adding `virtual` to our functions.



# More on Polymorphism

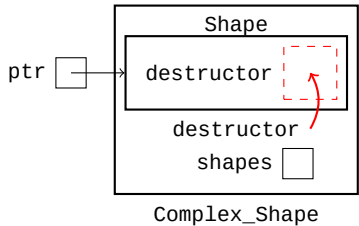
## virtual-destructor

```
class Shape
{
public:
    // ...
    virtual ~Shape() = default;
    // ...
};
```

# More on Polymorphism

What about now?

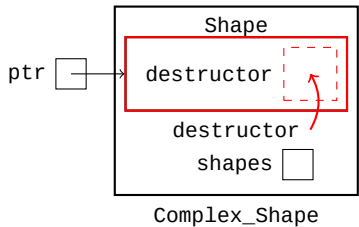
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



# More on Polymorphism

What about now?

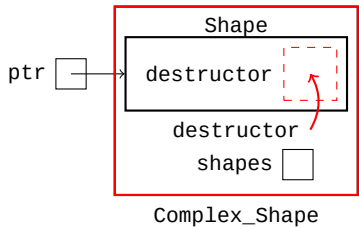
```
Shape* ptr {new Complex_Shape{...}};  
delete ptr;
```



# 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;
```

ptr 

# More on Polymorphism

What about now?

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

ptr 

Nice!

## More on Polymorphism

What about now?

- By declaring the destructor as `virtual` we are allowing derived classes to override the behaviour with their own implementation.
- This means that whenever the destructor is called through a pointer or a reference it will call the appropriate destructor.
- **Note:** The destructor of a class must also destroy the base class, but this is handled by the compiler so we don't have to think about it.

## 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 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; // prints 0 (?!)  
delete ptr;
```

# More on Polymorphism

Sometimes humans make mistakes...

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

# 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    We forgot const!
    {
        return 10.0;
    }
    // ...
};
```



# More on Polymorphism

Sometimes humans make mistakes...

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

## More on Polymorphism

Sometimes humans make mistakes...

- When overriding virtual functions the signature must match *exactly*
- The name, the parameters, specifiers etc. it all must match with the base class version of the function.
- If it doesn't, the compiler will create a normal function in the derived class with these new properties.
- This is not a syntax error, it is just a semantic error.
- We have to make sure they match otherwise the compiler gets confused...

# More on Polymorphism

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

```
class My_Shape : public Shape
{
public:
    // ...
    double area() 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

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

- If you mark a member function as `override` you tell the compiler that you intended for this member function to override a virtual function in the base class.
- This means that the compiler will check whether or not it succeeded in overriding the function.
- If something is wrong, the compiler tell us and we can fix it!
- If we don't use `override`, the code might compile with the wrong behaviour which is really bad.

## 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

Let's go back to Shape

- Does it really make sense that `Shape : : area` returns 0?
- What does it mean to take the area of a general shape?
- Wouldn't it be better to just skip the implementation?



# More on Polymorphism

pure-virtual function

```
class Shape
{
public:

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

## More on Polymorphism

### pure-virtual function

- You can add `= 0` at the end of a virtual function declaration to mark it as a *pure-virtual* function.
- This means that this function doesn't have an implementation.

# 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

## Abstract class

- No object of an abstract class is allowed to exist.
- This means that we cannot create Shape in any way possible.
- The reason is that it contains functions that would crash the program if called (because they do not have an implementation).

# More on Polymorphism

## Abstract class

- We can however have a pointer or reference of type Shape since these may refer to a derived class of Shape.
- All derived classes of an abstract class are also abstract classes until all pure-virtual functions have been overridden.
- Abstract classes are meant to represent general concept that are used as a base class to more concrete things (such as specific shapes).

# 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;
};
```

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



# 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

## Static vs Dynamic type

- The *static* type of a variable is the type it is declared as (it never changes)
- The *dynamic* type is the type of the object a pointer points to
- The dynamic type can change to any class in the hierarchy of the static type.



# 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

- Which functions you can call is directly related to the static type.
- I.e. it doesn't matter that the dynamic type of `ptr` is `Cuboid`, we can't call `volume` through a `Shape` pointer.
- Therefore we must, temporarily change the static type to match the dynamic type.

# Type information

## Example

```
Shape* ptr {new Cuboid{3, 4, 5}};  
  
cout << static_cast<Cuboid*>(ptr)->volume()  
      << endl;
```

# Type information

## Example

- We can use `static_cast` to (temporarily) change `ptr` into a `Cuboid*`, that way we can call `volume()`.
- But this is very dangerous...

# 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;
```

# Type information

When it all comes crashing down...

- We can cast `ptr` to a pointer to any derived class,
- However, this becomes a problem if the type we are casting to is not compatible with the dynamic type...
- This will, in most cases, lead to the crashing of your program...
- Would be nice if we could check first if it was possible before we cast...



## 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

### `dynamic_cast`

- `dynamic_cast` is like `static_cast`, but before it performs the conversion it will test that the dynamic type is compatible (i.e. is derived from or equal to the type we are casting to)
- if they are compatible it will return a valid pointer with the specified static type,
- if they are not compatible it will return `nullptr`.

# 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)
```

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

# 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

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



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

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

# Exceptions

## Model

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

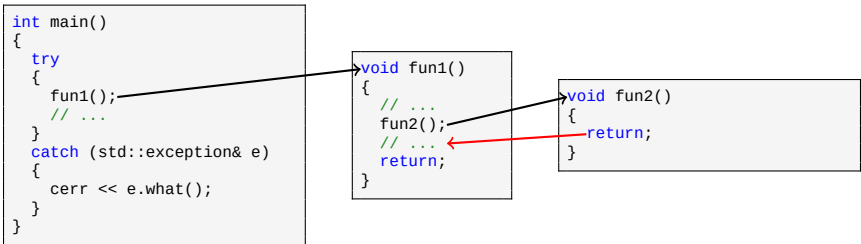
```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

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

```
graph LR
    subgraph main [main]
        direction TB
        m_try[try]
        m_fun1[fun1()]
        m_catch[catch]
    end
    subgraph fun1 [fun1]
        direction TB
        f1_fun2[fun2()]
    end
    subgraph fun2 [fun2]
        direction TB
        f2_return[return]
    end
    m_fun1 --> f1_fun2
    f1_fun2 --> f2_return
```

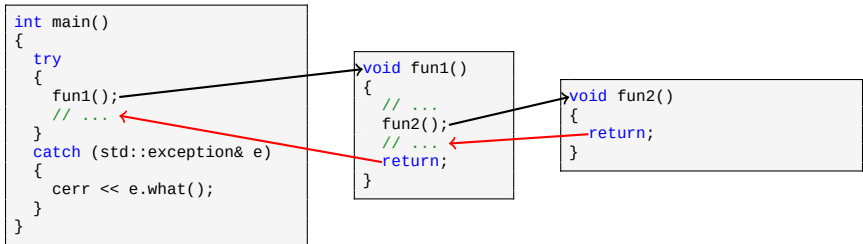
# Exceptions

## 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

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



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

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

# Exceptions

## Model

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

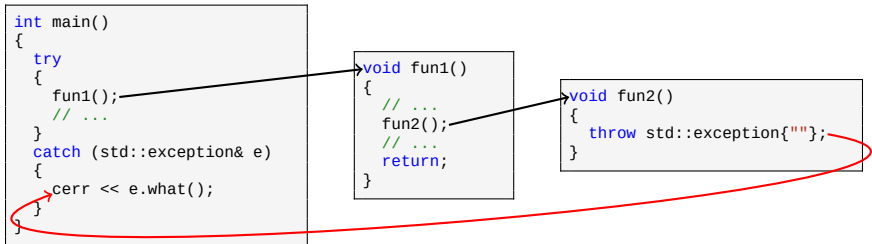
```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

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

```
graph LR
    main["main()"] --> fun1["fun1()"]
    fun1 --> fun2["fun2()"]
```

# Exceptions

## Model



# Exceptions

## Model

- An exception is an object we *throw*.
- Throwing an exception will abort the current function,
- it will move backwards in the function call chain until it hits a *try-catch* block.
- Throwing is separate from returning.
- We should only throw exceptions when something went wrong.

# 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;
    }
}
```

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

# 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

Calling a program with arguments

- Unix-systems are based on calling programs with various arguments,
- This is in fact what “commands” are in the terminal: programs that takes arguments.
- But how do we read these arguments in our own programs?

# Command-line argument

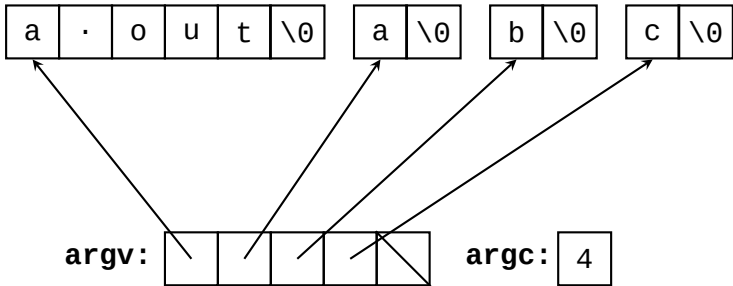
## Reading arguments

```
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

argv

- The arguments are passed into your program as C-strings.
- A C-string is an array of `char`.
- It is called a C-string because this is how strings work in C.
- The end of a C-string is indicated with the special character `'\0'`.
- **Note:** The name of the executable file is the argument at index 0.

# 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

## Example

- We can access the  $i$ :th argument with `argv[i]`.
- Notice that these are strings!
- How do we interpret them as something else?



# 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](http://www.liu.se)