Object-oriented scientific programming with C++

Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 3

Overview

During last several lectures, we learned polymorphism, that is, inheritance of one class from another class

- Implementation of common functionality in **base class** (possibly realised as **abstract class** that cannot even be instantiated)
- Derivation of specialised class(es) from the base class that
  - implement the missing functionality (**pure virtual functions**)
  - override generic functionality by specialised variants (**virtual functions**)
  - reuse all other functionality from the base class

Overview

Today, a more careful view on polymorphism

- **Static polymorphism**: static binding/method overloading
- **Dynamic polymorphism**: dynamic binding/method overriding Auto functionality of C++11/14/17.

Template meta programming

- A powerful variant of static polymorphism

Quiz: Guess what happens

In [2]:

```
struct Base {
  virtual void hello() const = 0;
  virtual ~Base() {} // virtual destructor for proper cleanup
};

struct Derived : public Base {
  void hello() const override // Correct function name to override the base class method
  {
    std::cout << "Hallo\n";
  }
};
```

Quiz: Guess what happens

Run the main function to verify your guess

In [3]:

```
Derived d; // Create an instance of Derived
Base* ptr = &d; // Pointer to Base type, pointing to Derived instance

ptr->hello(); // Virtual dispatch ensures Derived::hello() is called
```

```
Hallo
```

Discussion

Is there another way to call the function? Think about what we have learned in the past two weeks.
HINT: the above example called the function by using a pointer, what we have learned at the same time with pointer?

Discussion

Is there another way to call the function? Think about what we have learned in the past two weeks.

In [4]:

```
Derived d; // Create an instance of Derived
Base& ref = d; // Reference to Base type, referring to Derived instance

ref.hello();
```

```
Hallo
```

Quiz: Guess what happens

```
struct Base {
  virtual void hello() const = 0;
};
struct Derived : public Base {
  void hello() const
  {
    std::cout << “Hallo\n”;
  }
};
```

Quiz: Guess what happens

```
struct Base {
  virtual void hello() const = 0;
};
struct Derived : public Base {
  void hello() const
  {
    std::cout << “Hallo\n”;
  }
};
```

- Structurally correct in terms of C++ syntax.
- `Derived` properly overrides the pure virtual function `hello()` from `Base`, allowing instances of `Derived` to be created and used polymorphically through pointers or references to `Base`.

Quiz: Guess what happens

```
struct Base {
  virtual void hello() const {
    std::cout &lt< "Hello\n";
  }
};
struct Derived : public Base {
  void hallo() const //Good luck with debugging
  {
    std::cout &lt< "Hallo\n";
  }
};
```

Quiz: Guess what happens

```cpp
struct Base {
  virtual void hello() const {
    std::cout &lt< "Hello\n";
  }
};
struct Derived : public Base {
  void hallo() const //Good luck with debugging
  {
    std::cout &lt< "Hallo\n";
  }
};
```

- The Derived struct inherits from Base and introduces a new function hallo() that prints " Hallo\n" to the standard output.
- However, Derived does not override the hello() function from Base; it simply provides an additional function.

The override keyword

C++11 introduces the override keyword to explicitly state that a function from the base class shall be overridden

Modify the program below yourself to see the different with Quiz var2

In [5]:

```
struct Base {
  virtual void hello() const = 0;
  virtual ~Base() {} // virtual destructor for proper cleanup
};
struct Derived : public Base {
  void hello() const override {std::cout << "Hallo\n";}
};

Derived d; // Create an instance of Derived
d.hello(); // Calls the overridden hello() method in Derived
Base* b = &d; // Since hello() is a virtual function, you can also call it through a Base pointer
b->hello(); // This will also call Derived::hello()
            //NOTE: If the function to be overridden does not exist in the base class an error is thrown by
 the compiler
```

```
Hallo
Hallo
```

The override keyword

The overriding function must exactly match the signature of the function from the base class

```
struct Base { virtual void hello() const { std::cout &lt&lt “Hello\n”; } };
struct Derived : public Base { void hello() const override { std::cout &lt&lt “Hallo\n”; } };
```

~~struct Derived2 : public Base { void hello() override { std::cout &lt&lt “Hallo\n”; } };~~

The final keyword

C++11 introduces the final keyword to explicitly state that a function from the base class must not be overridden

```
struct Base { virtual void hello() const final { std::cout &lt&lt “Hello\n”; } };
```

If a derived class tries to override the function hello an error is thrown by the compiler

The override and final keywords

The override and final keywords are often used together
struct Base { // Pure virtual function must be overridden in a derived class virtual void hello() const = 0; }; struct Derived : public Base { // Override pure virtual function and make it final void hello() const override final { std::cout &lt&lt “Hello\n”; } }; struct Derived2 : public Derived { // Must not try to override function hello };

Overloading vs. overriding

Implementation of method or operator with identical name but a different interface, e.g.

```
double operator*(const
Vector & other) const {...}
```

overloads

```
double operator*(double
  other) const {...}
```

Implementation of method or operator with identical name and the same interface in a derived class or structure, e.g.

```
struct Derived : public Base {
    void hello(...) const override
{...}
};
```

overrides

```
struct Base {
    virtual void hello(...) const
{...};
};
```

Overloading vs. overriding

- **Static polymorphism** is
  - *more efficient* at run time but
  - *less flexible* because all decisions must be made at compile time
- **Dynamic polymorphism** is
  - *less efficient* at run time but
  - *more flexible* because decisions are made at run time

```
Derived d; d.welcome();
```

```
struct Base {
  virtual void hello() const = 0;
  virtual void welcome() const {
    cout << "Say\n";
    hello();
  }
};

struct Derived : public Base {
  void hello() const override {
    cout << "Hello\n";
  }
};
```

Implementation time! Task: Calculator

Write a class (or for demonstrating purposes a hierarchy of classes) that provide(s) a member function to calculate the **sum of two and three integer values,** respectively

- Use *static* polymorphism: method overloading
- Use *dynamic* polymorphism: method overriding

Static polymorphism

Method overloading (at compile time), the `StaticCalc` has two member functions with identical names but different interface; compiler can decide at compile time which of the two functions should be used

In [6]:

```
struct StaticCalc {
    int sum(int a, int b) const {
        return a + b;
    }
    // Overloaded function to sum three integers
    int sum(int a, int b, int c) const { return sum(sum(a,b),c);}
};

StaticCalc statC;
std::cout << "Static sum of two integers: " << statC.sum(1, 2) << std::endl;
std::cout << "Static sum of three integers: " << statC.sum(1, 2, 3) << std::endl;
```

```
Static sum of two integers: 3
Static sum of three integers: 6
```

Static polymorphism

Method overloading (not working!)

```
struct StaticCalcBug { int sum(int a, int b) { return a+b; } void sum(int a, int b) { std::cout&lt&lt a+b &lt<std::endl;} };
```

- Difference must be in the interface of the arguments passed. Same name with different parameter lists is allowed (either the number of parameters or their types must differ).
- You cannot have two functions that only differ by their return type. Because the return type could be cast into another type

Static polymorphism

Method overloading (also not working!)

```
struct StaticCalcBug2 { int sum(int a, int b) { return a+b; } int sum(int c, int d) { return c+d; } };
```

- The interface only refers to the types and not to the names

Static polymorphism - Laaaaast! counter example

Method overloading (still not working!)

`struct StaticCalcBugLast { int sum(int a, int b) { return a+b; } int sum(int a, int b, int c=1) { return a+b+c; } };`

- Be careful with default values. Here, the compiler cannot decide unambiguously which of the two functions should be used in the case `sum(arg0, arg1)`

Rule of thumb:

If it is not crystal clear to you which function should be used then the compiler will fail, too.

Static polymorphism (master version1)

Method overloading: decision about which method to call is made at compile time; hence the compiler can decide to inline code to improve performance (no overhead due to function calls and copying of data to and from the stack!)

```
std::cout << statC.sum(1,2)   << std::endl;
std::cout << statC.sum(1,2,3) << std::endl;
```

becomes

```
std::cout << (1+2)        << std::endl;
std::cout << ((1+2)+3)    << std::endl;
```

Static polymorphism (master version2)

Method overloading: since static polymorphism takes place at compile time, the inline specifier can be used to explicitly 'suggest' to the compiler to inline the function

```
struct StaticCalcMaster { inline int sum(int a, int b) { return a+b; } inline int sum(int a, int b, int c) { return sum(sum(a,b),c); } };
```

Dynamic polymorphism

Method overloading: reimplementation of a function inherited from a base class with new function body and same interface!

In [7]:

```
struct BaseCalc {
    virtual int sum2(int a, int b) { return a+b; }
    int sum3(int a, int b, int c)
        { return sum2(sum2(a,b),c); }
  };
struct DerivedCalc : public BaseCalc {
    int sum2(int a, int b) override final { return b+a; }
};
```

In [8]:

```
BaseCalc baseCalc;
DerivedCalc derivedCalc;
// Using the base class to sum two numbers
std::cout << "BaseCalc sum2: " << baseCalc.sum2(1, 2) << std::endl;
// Using the base class to sum three numbers
std::cout << "BaseCalc sum3: " << baseCalc.sum3(1, 2, 3) << std::endl;
// Using the derived class to sum two numbers, this will use the overridden method
std::cout << "DerivedCalc sum2: " << derivedCalc.sum2(1, 2) << std::endl;
// Using the derived class to sum three numbers, this will use the base
// class method sum3, which in turn will use the overridden sum2 method
// from DerivedCalc
std::cout << "DerivedCalc sum3: " << derivedCalc.sum3(1, 2, 3) << std::endl;
```

```
BaseCalc sum2: 3
BaseCalc sum3: 6
DerivedCalc sum2: 3
DerivedCalc sum3: 6
```

Dynamic polymorphism(where things can go wrong)

Method overloading: a common pitfall is to forget the virtual specifier in the base class to indicate that the sum2 function from the base class can be overridden in a derived class

```
struct BaseCalc { int sum2(int a, int b) { return a+b; } int sum3(int a, int b, int c){ return sum2(sum2(a,b),c); } }; struct DerivedCalc : public BaseCalc { int sum2(int a, int b) { return b+a; } };
```

Dynamic polymorphism(where things can go wrong)

Take home notes, read after the lecture. Here's what happens if you forget `vitual` specifier

- Method Hiding: In `DerivedCalc`, the `sum2` method hides the `sum2` method from `BaseCalc`. This means that if you have an object of `DerivedCalc` and call `sum2`, it will use `DerivedCalc's` `sum2`. However, if you have a pointer or reference to `BaseCalc` that actually points to a `DerivedCalc` object and call `sum2`, it will still use `BaseCalc's` `sum2`. This is because the method is not virtual and hence does not support polymorphic behavior.

- No Polymorphism: Without the virtual keyword in `BaseCalc`, `sum2` in `DerivedCalc` doesn't exhibit polymorphic behavior, crucial for method overriding in C++.

Rule of thumb: These bugs are hard to find (they often remain unrecognised) and can be prevented by using the `override` keyword in C++11. It explicitly indicates that a function is intended to override a virtual function in a base class.

Dynamic polymorphism - general suggestions

Method overriding: decision about which virtual method to call is made at run time; hence inlining is not possible

Common design pattern

- Specify expected minimal functionality of a group of classes in abstract base class via pure virtual member functions
- Implement generic common functionality of a group of classes in abstract base class via virtual member functions
- Implement specific functionality of a particular derived class by overriding the pure virtual member function

Example: inner product space

In linear algebra, an inner product space is a vector space V that is equipped with a special mapping (inner product)

$$\langle \cdot\,,\ \cdot \rangle : V \times V \to \mathrm{R} \text{ or } \mathrm{C}$$

Inner product spaces have a naturally induced norm

$$\|x\| = \sqrt{\langle x, x \rangle}$$

Example: inner product space

Class InnerProductSpaceBase declares inner product as pure virtual and implements the naturally induced norm

struct InnerProductBase { virtual double inner_product(... x,... y) = 0; double norm(... x) { return inner_product(x,x); } };

Derived InnerProductSpace class implements inner product

struct InnerProductSpace : public InnerProductSpaceBase { double inner_product(... x, ... y) = { return x*y; }};

Implementation time! Task: Calculator2

Extend the calculator class so that it can handle numbers of integer, float and double type at the same time

- Prevent manual code duplication
- Prevent explicit type casting
- Make use of auto-functionality (C++11/14/17)
- Make use of template meta programming

Vanilla implementation in C++

In [9]:

```cpp
#include <iostream>
struct Calc2 {
    int sum(int a, int b) { return a + b; }
    int sum(int a, int b, int c) { return sum(sum(a, b), c); }
};
Calc2 calculator;
// Using the sum method with two arguments
std::cout << "Sum of 1 and 2: " << calculator.sum(1, 2) << std::endl;
// Using the sum method with three arguments
std::cout << "Sum of 1, 2, and 3: " << calculator.sum(1, 2, 3) << std::endl;
```

```
Sum of 1 and 2: 3
Sum of 1, 2, and 3: 6
```

Automatic return type deduction (C++11)

- Explicit definition of the function return type

```
int sum(int a, int b){ return a+b; }
```

- Automatic function return type (since C++11)

```
auto sum(int a, int b) -> decltype(a+b) { return a+b; }
```

- By using decltype, the return type of the sum function is determined automatically as the type of operator+(a,b)

Automatic return type deduction

decltype specifier (C++11) queries the type of an expression

In [10]:

```
#include <iostream>
struct Calc2 {
    auto sum(int a, int b) -> decltype(a + b) {return a + b;}
    auto sum(int a, int b, int c) -> decltype(sum(sum(a, b), c)) {
        return sum(sum(a, b), c);
    }
};

Calc2 C;
std::cout << C.sum(1, 2) << std::endl;   // Outputs: 3
std::cout << C.sum(1, 2, 3) << std::endl; // Outputs: 6
```

```
3
6
```

Out[10]:

```
@0x7ffb425f6de0
```

Automatic type deduction (C++14)

C++14 deduces the type of parameters automatically

```
auto sum(int a, int b) // no -> decltype(...) needed { return a+b; } auto sum(int a, int b, int c) // no -> decltype(...) needed { return sum(sum(a,b),c); }
```

Remark: This C++14 feature helps to improve readability of the code and prevents deduction errors (due to forgotten/ inconsistent deduction rule by the programmer) but it does not solve the problem of being able to pass arguments of different types to the same function

Function templates

Template meta programming is the standard technique to deal with arbitrary (= generic) function parameters

Function templates: allow you to implement so-called parameterized functions for generic parameter types

```
template&lttypename R, typename A, typename B> R sum(A a, B b) {return a+b; }
```

Function templates

Types must be specified explicitly when the function is called

int s1 = sum&lt int, int, int>(1, 2); double s2 = sum&ltdouble, double, int>(1.2, 2); double s3 = sum&ltdouble, float, double>(1.4, 2.2); This can be slightly simplified by using the auto specifier

auto s1 = sum&lt int, int, int>(1, 2); auto s2 = sum&ltdouble, double, int>(1.2, 2); auto s3 = sum&ltdouble, float, double>(1.4, 2.2);

Function templates

C++11: automatic return type deduction

```
template&lt typename A, typename B> auto sum(A a, B b) –> decltype(a+b){ return a+b; }
```

C++14: automatic type deduction

```
template&lt typename A, typename B> auto sum(A a, B b){ return a+b; }
```

Usage

```
auto s1 = sum&ltint, int>(1, 2);
```

Function templates

How to convert this function into a templated function

```
int sum(int a, int b, int c)
{
      return sum(sum(a,b), c);
  }
```

Function templates

Use explicit return type parameter (ugly!) -> NO template&lttypename R, typename A, typename B, typename C> auto sum(A a, B b, C c) { return sum&ltR,C>(sum&ltA,B>(a,b), c); } Guess what this function call will return

auto s1 = sum&lt int, double,double,double>(1.1,2.2,3.3)

Function templates

Use a smart combination of templates and auto specifier

template&lttypename A, typename B> auto sum(A a, B b) -> decltype(a+b) // omit in C++14 {return a+b; }

template&lttypename A, typename B, typename C> auto sum(A a, B b, C c) { return sum&ltdecltype(sum&ltA,B>(a,b)),C> (sum&ltA,B>(a,b), c); }

Function templates

But wait, C++ can deduce the type of the template argument from the given argument automatically

template&lttypename A, typename B> auto sum(A a, B b) -> decltype(a+b) // omit in C++14 {return a+b; } template&lttypename A, typename B, typename C> auto sum(A a, B b, C c) { return sum(sum(a,b), c); }

Function templates

We can call the generic sum functions as follows:

```
auto s1 = sum&ltint, int> (1, 2); auto s2 = sum&ltdouble, int> (1.2, 2); auto s3 = sum&ltfloat, double>(1.4f, 2.2);
```

Or, even simpler, as follows:

```
auto s1 = sum (1, 2); auto s2 = sum (1.2, 2); auto s3 = sum (1.4f, 2.2)
```

Function templates

- Since the compiler needs to duplicate code and substitute A,B,C for each combination of templated types both the compile time and the size of the executable will increase
- Template meta programming is simplest if the code resides in header files only; later we will see how to use template meta-programming together with pre-compiled libraries

Implementation time! Task: generic Vector class

Write a Vector class that can store real values and complex values and supports the following operations:

- Addition of two vectors of the same length
- Multiplication of a vector with a scalar
- Dot product of a vector with another one In all operations, the involved objects (vectors/scalar) can be of different types, e.g., double/float

Vector class prototype

Implementation of Vector-of-double class

In [71]:

```
#include <iostream>
class Vector{
private:
    double* data;
    int n;
public:
//----------------------------------- Constructors below -----------------------------------------------
//
    Vector() : n(0), data(nullptr) {} // Default constructor

    Vector(int n) : n(n), data(new double[n]) {} // Constructor with size

    ~Vector(){ n=0;  delete[] data; }// Destructor
//----------------------------------- method and operators ---------------------------------------------
--//

    // Addition (+=) operator
    Vector& operator+=(const Vector& other) {
        if (n != other.n) {
            throw std::invalid_argument("Vectors must be of the same size");
        }
        for (auto i = 0; i < n; i++) {
            data[i] += other.data[i];
        }
        return *this;
    }

    // Scalar multiplication (*=) operator
    Vector& operator*=(double scalar) {
        for (int i = 0; i < n; i++) {
            data[i] *= scalar;
        }
```

```
        return *this;
    }

    // Dot product
    double dot(const Vector& other) const {
        if (n != other.n) {
            throw std::invalid_argument("Vectors must be of the same size");
        }
        double sum = 0;
        for (auto i = 0; i < n; i++) {
            sum += data[i] * other.data[i];
        }
        return sum;
    }
    // Subscript operator
    double& operator[](int index) {
        if (index >= n || index < 0) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
     }
};
```

Brainstorming

Function templates alone will not help since the type of a class attribute needs to be templated -> class templates

Some member functions can be implemented generically, e.g., addition of two vectors and multiplication of a vector with a scalar value since they are the same for all types

Some member functions must be implemented in different manners for real and complex values ->specialisation

$$x \cdot y = \sum_{i=1}^{n} x_i y_i, \quad x, y \in \mathrm{R}, \quad x \cdot y = \sum_{i=1}^{n} x_i \bar{y}_i \quad x, y \in \mathrm{C}$$

Class template: anything vector class

Implementation of Vector-of-anything class -- creating a generic vector that can store elements of any type.
CAUTION: We are going to leave the presentation mode to have an overview of the entire program (without the main function).

In [72]:

```
#include <iostream>
#include <stdexcept>

template<typename T>
class Vector {
private:
    T* data;
    int n;

public:
//-------------------------------------Constructors below----------------------------------------//
    // Default constructor
    Vector() : data(nullptr), n(0) {}

    // Constructor with size
    Vector(int n) : n(n), data(new T[n]()) {}

    // Destructor
    ~Vector() {
        delete[] data;
    }

//----------------------End of constructors, go to next page for explaination---------------------//


//------------------------------------Methods and operators below-----------------------------------/
/
    // Addition (+=) operator
```

```
    Vector<T>& operator+=(const Vector<T>& other) {
        if (n != other.n) {
            throw std::invalid_argument("Vectors must be of the same size");
        }
        for (int i = 0; i < n;i++) {
            data[i] += other.data[i];
        }
        return *this;
    }

    // Scalar multiplication (*=) operator
    Vector<T>& operator*=(T scalar) {
        for (int i = 0; i < n; i++) {
            data[i] *= scalar;
        }
        return *this;
    }

    // Dot product
    T dot(const Vector<T>& other) const {
        if (n != other.n) {
            throw std::invalid_argument("Vectors must be of the same size");
        }
        T sum = 0;
        for (int i = 0; i < n; i++) {
            sum += data[i] * other.data[i];
        }
        return sum;
    }
//------------------Combined class template and function template below------------------------//
    template<typename S>
    Vector<T>& operator*=(S scalar){
        for (auto i=0; i<n; i++)
            data[i] *= scalar;
        return *this;
    }

// Non-const subscript operator
    T& operator[](int index) {
        if (index < 0 || index >= n) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }
```

```
// Const version of subscript operator
    const T& operator[](int index) const {
        if (index < 0 || index >= n) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }
};
```

Class template

Template parameter must be explicitly specified

```
Vector<int> x(10); // Vector-of-int with length 10
Vector<double> y; // Empty Vector-of-double
Vector<float> z(5); // Vector-of-float with length 5
```

Remark: if you want to pass a Vector-of-anything to a function in the templated class Vector you have to write

```
Vector<T>  v
Vector<T>& v
```

instead of

```
Vector  v
Vector& v
```

Class template

With the **class template parameter T** we can do

In [70]:

```
Vector<int> x1(5);
Vector<int> x2(5);

// Initialize x1 and x2 with some values
for (int i = 0; i < 5; ++i) {
    x1[i] = i + 1;  // x1 = [1, 2, 3, 4, 5]
    x2[i] = 2 * (i + 1);  // x2 = [2, 4, 6, 8, 10]
}

x1 += x2; // x1 = [3, 6, 9, 12, 15]
x1 *= 2;  // x1 = [6, 12, 18, 24, 30]
int dot_product = x1.dot(x2);  // Dot product of x1 and x2

std::cout <<"Dot product:" << dot_product;
```

```
Dot product:660
```

How about?

```
Vector<int> x2(5);
x2 *= (double)1.2;
```

Intermezzo

```
template&lttypename T> class Vector { ... template&lttypename S> Vector&ltT>&amp
operator*=(S scalar) { for (auto i=0; i<n; i++) data[i] *= scalar; return *this; } };
```

Intermezzo

At first glance, this seems to be more flexible

```
Vector&ltdouble> x1(5); x1 = x1*(int)(5);
```

But be really careful since strange things can happen

```
Vector&ltint> x1(5); x1 = x1*(double)(5);
```

Rule of thumb: before using extensive template meta programming like this think about all(!) implications

Specialization

The dot product needs special treatment since

```
T dot(const Vector&ltT>&amp other) const { T d=0; for (auto i=0; i<n; i++) d += data[i]*other.data[i]; return d; }
```

- Lacks the complex conjugate of other and yields the wrong return type in case of complex-valued vectors

Remedy: implement a specialised variant for this case

Specialization

Generic dot product implemented in Vector class

```
#include &ltcomplex> template&lttypename T> class Vector { ... T dot(const Vector&ltT>&amp other) const {...} };
```

This function is used whenever no specialized implementation for a concrete type is available

Specialization

Specialized dot product for Vectors-of-complex-complex
template&lt> std::complex&ltfloat> Vector&lt std::complex&ltfloat> >:: dot(const Vector&lt std::complex&ltfloat>>&amp other) const { std::complex&ltfloat> d=0; // special treatment of dot product for (auto i=0; i<n; i++) d += data[i]*std::conj(other.data[i]); return d; }

Specialization

Current implementation yields

```
Vector&ltfloat> x1(5), x2(5); auto x1.dot(x2); // calls generic implementation
Vector&ltstd::complex&ltfloat> x1(5), x2(5); auto x1.dot(x2); // calls generic implementation
Vector&ltstd::complex&ltdouble> > z1(5), z2(5); auto z1.dot(z2); // calls generic implementation
auto x1.dot(y1); // does not even compile
```

Outlook on next session

C++ allows you to **partially specialize** class templates
template&lttypename S> std::complex&ltS> Vector&ltstd::complex&ltS> >:: dot(const Vector&ltstd::complex&ltS> > other) const { std::complex&ltS> d=0; for (auto i=0; i<n; i++) d += data[i]*std::conj(other.data[i]); return d; }
Note that this code will not compile. We will see why and learn remedies. Welcome to where template magic begins!

In [ ]:

Processing math: 100%