Lecture 16

Type Casting & Exception Handling

Yohan Jo

Midterm Exam

- **Coverage**
  - Lectures 3–17
- **Question Types**
  - 10 conceptual questions (true/false, MCQs)
  - 3 coding questions

Midterm Exam

- **Accessible Resources**
  - Editor of your choice
  - eTL
  - cppreference.com
  - Cheat sheet (2 pages) – A print-out is allowed
- **Prohibited Resources**
  - Any AI-based plugins and suggestion tools (e.g., IntelliSense) must be disabled
  - The use of such tools will be considered cheating
  - Using a debugger is allowed

Midterm Exam

- **Screen Sharing**
  - Every student should share their screen via Zoom
  - Redirect your screen to the webcam using OBS
- **Rehearsal**
  - If you want to check your computer settings, attend the exam rehearsal at the end of the next class
- **Instructions**
  - Refer to the exam instructions document

Midterm Exam

- **Review Session**
  - We will be having a review session in the next class
  - Post your questions to the eTL forum by 10/27

Lecture Overview

- Type casting
- Exception handling
- Exercises

Type Casting

Type Casting

- Type casting is the process of converting a variable from one data type to another
- Various kinds of type casting
  - C-style casting
  - Static cast
  - Dynamic cast
  - Const cast
  - Reinterpret cast

C-Style Casting

- C-style casting is the original casting mechanism inherited from C, using the `(type)` syntax (e.g., `int intVal = (int)floatVal`)
- C-style casting can be problematic in C++ due to its lack of specificity (in intents) and safety checks

```
ElectricPokemon* elecPokemon = new Pikachu();
Charmander* charmander = (Charmander*)elecPokemon;  //
No compile-time error
```

Static Cast

- `static_cast` is a compile-time cast based on static types declared by the programmer
- `static_cast` performs an explicit conversion between compatible types
- Syntax: `static_cast<NewType>(expression)`
- It provides compile-time checking, reducing the risk of runtime errors

```
// Converting fundamental types
double pi = 3.14159;
int intPi = static_cast<int>(pi);  // 3

// Upcasting in class hierarchies
Pikachu pikachu;
ElectricPokemon* elecPokemon =
static_cast<ElectricPokemon*>(&pikachu);

// Downcasting in class hierarchies
ElectricPokemon* elecPokemon = new Pikachu();
Pikachu* pikachu =
static_cast<Pikachu*>(elecPokemon); // OK
Charmander* charmander =
static_cast<Charmander*>(elecPokemon); //
Compile-time error (vs. C-style casting)
Pikamander* pikamander =
static_cast<Pikamander*>(elecPokemon); //
Wrong but would compile
```

Dynamic Cast

- `dynamic_cast` is a runtime cast to convert pointers and references to related classes
- Syntax:
  `dynamic_cast<NewType>(expression)`
- It checks the validity of a cast that might be difficult to check at compile time, especially a downcast
- If the cast fails, it returns `nullptr` (for pointers) or throws an exception (for references)

```
ElectricPokemon* elecPokemon = new
Pikachu();
Pikachu* pikachu =
dynamic_cast<Pikachu*>(elecPokemon); // OK

Pikamander* pikamander =
dynamic_cast<Pikamander*>(elecPokemon); //
Returns nullptr
if (pikamander) { /* Ignored */ }
else { /* Run */ }
```

Dynamic Cast

- `dynamic_cast` cannot or may not be used in the following scenarios:
  - The source class or the destination class has no virtual tables – in order to identify the actual class of an object at runtime (Run-Time Type Information or RTTI), a virtual table is needed, as it includes or is associated with type information for the class
  - For upcasting or casting to unrelated types (e.g., `void*`), `dynamic_cast` can be used but is unnecessary because a compile-time check via `static_cast` is enough

Const Cast

- `const_cast` removes the const qualifier from const pointers and references, allowing the programmer to modify their value
- Syntax: `const_cast<NewType>(expression)`
- `const_cast` should be used sparingly, as it can lead to undefined behavior (e.g., the compiler caches a const value in read-only memory or replaces references to a const variable with its literal value)

```
void func(const std::string& str) {
    str += "_suffix"; // Error
    std::string& nonConstStr =
const_cast<std::string&>(str);
    nonConstStr += "_suffix"; // OK

    str.append("_suffix"); // Error
    nonConstStr.append("_suffix"); // OK
}
```

Reinterpret Cast

- `reinterpret_cast` is used to convert a pointer type to another pointer type
- Syntax: `reinterpret_cast<NewType>(expression)`

```
// Example memory address used for
communication with external modules
#define REG_ADDRESS 0x40021000

uint32_t* regUint =
reinterpret_cast<uint32_t*>(REG_ADDRESS);
*regUint = 1432;

...

int32_t* regInt =
reinterpret_cast<int32_t*>(REG_ADDRESS);
*regInt = -384;
```

Exception Handling

Exception Handling

- Exception handling is a mechanism that allows a program to deal with exceptional situations that may occur during the execution of a program
- The primary goal of exception handling is to provide a mechanism to detect and handle such errors gracefully without crashing the program

Exception Handling

```
#include <iostream>
#include <new>

int main() {
    try {
        int* myArray = new int[10000000000]; // Throws an
exception if allocation fails
        ...
        delete[] myArray;
    } catch (std::bad_alloc& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Exceptions

- An exception is typically an object of a class that inherits, directly or indirectly, from the `std::exception` class, ensuring a consistent interface
- Exceptions can contain detailed error information, including messages, error codes, etc.
- Standard exceptions are defined in the C++ standard library (`<stdexcept>`, `<new>`, `<typeinfo>`, etc.), tailored for common error scenarios
- Errors vs. Exceptions
  - Errors: Typically not recoverable and often lead to program termination
  - Exceptions: Expected and recoverable conditions that a program can handle

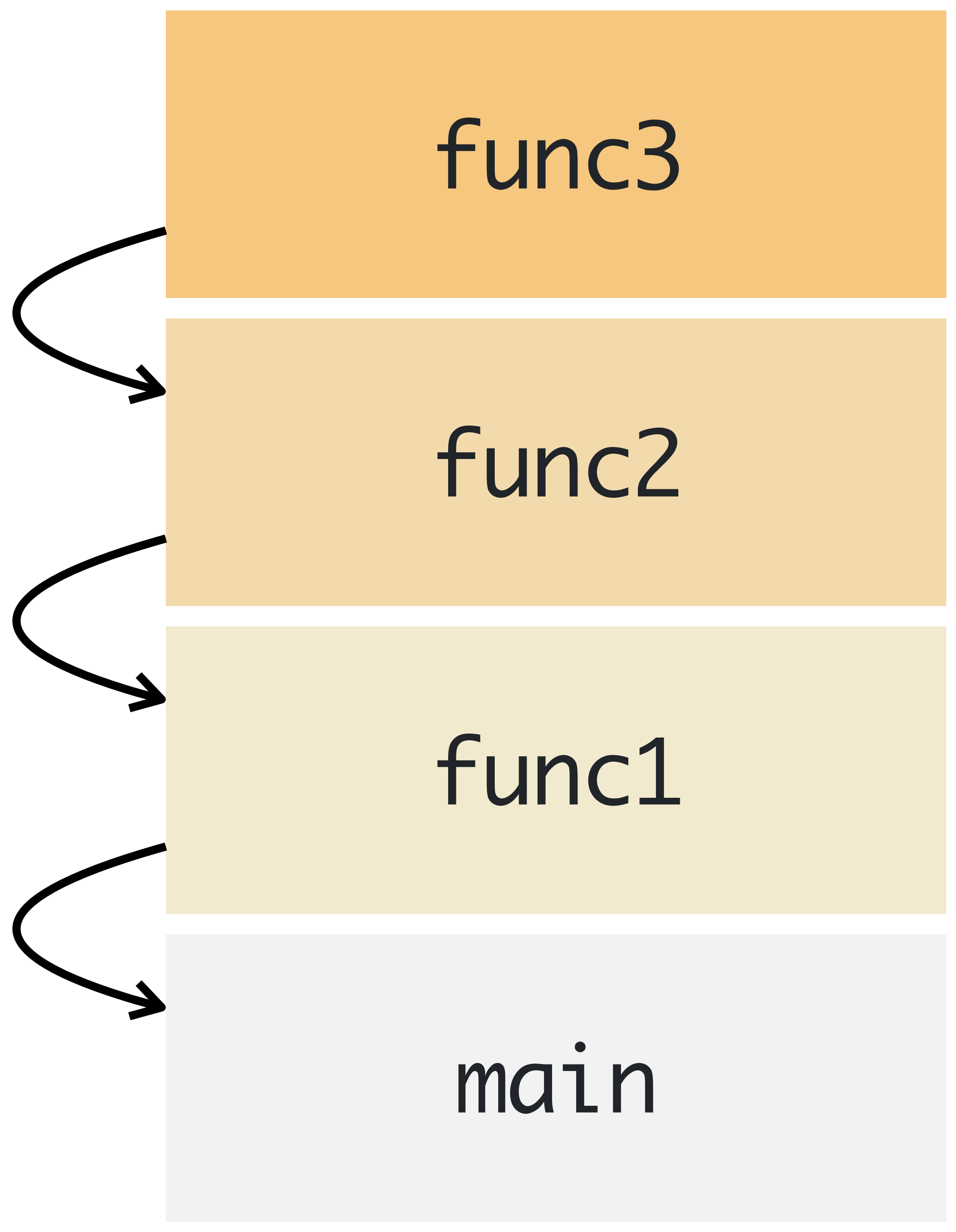
Syntax

- `try`
  - A `try` block contains code that you suspect might throw an exception
  - If an exception is thrown, execution of the code inside stops immediately
- `catch`
  - A `catch` block follows a `try` block and catches exceptions of the specified types
  - Multiple `catch` blocks are allowed

```
try {
    ...
} catch (std::runtime_error& e) {
    cout << "Runtime exception: "
<< e.what() << endl;
} catch (std::bad_alloc& e) {
    cout << "Memory allocation
exception: " << e.what() << endl;
}
```

Syntax

- `throw`
  - The `throw` directive initiates the exception handling mechanism by creating an exception object
    - E.g., `throw std::exception();`
  - The runtime system unwinds the call stack, exiting each function, until a suitable `catch` block is found



Call Stack

Syntax

```
#include <iostream>
#include <stdexcept> // std::out_of_range

template <typename T>
T& SimpleVector<T>::operator[](int index) {
    if (index < 0 || index >= size) {
        std::string message = "Index " +
std::to_string(index) + " is out of the
array of size " + std::to_string(size);
        throw std::out_of_range(message);
    }
    return array[index];
}
```

```
int main() {
    SimpleVector<int> vec{1, 2};
    int index, value;
    cin >> index >> value; // 10 5

    try {
        vec[index] = value;
    } catch (std::out_of_range& e) {
        std::cout << "Out of range
exception: " << e.what() << std::endl;
    }
    // Output: Out of range exception: Index
10 is out of the array of size 2
    return 0;
}
```

Custom Exceptions

- Custom exceptions provide a way to define error conditions specific to an application's logic
- Recommendations for a custom exception class:
  - Inherit from `std::exception` or any class derived from it
  - Override the `what()` method to return an error message

```
#include <exception> // std::exception

class MyException : public std::exception {
    int index, size;
    std::string message;
public:
    MyException(int index, int size) :
index(index), size(size) {
        message = "MyException: Index " +
std::to_string(index) + " is out of the array of
size " + std::to_string(size);
    }
    const char* what() const noexcept override {
        return message.c_str();
    }
};

template <typename T>
T& SimpleVector<T>::operator[](int index) {
    if (index < 0 || index >= size) {
        throw MyException(index, size);
    }
    return array[index];
}
```

Best Practices

- Catch an exception by reference not by value
  - Catching exceptions by value is slower
  - Object slicing can happen if an exception is caught by a more generic type (i.e., loses the parts specific to the derived object, including vptr and function overrides)

```
try {
    throw MyException(5, 3);
} catch (std::exception e) {
    std::cout << e.what() <<
std::endl;
    // Output: std::exception
}
```

- Avoid catching generic exceptions
  - Catching generic exceptions may leave unexpected exceptions unnoticed
  - Catching specific exceptions ensures meaningful error handling

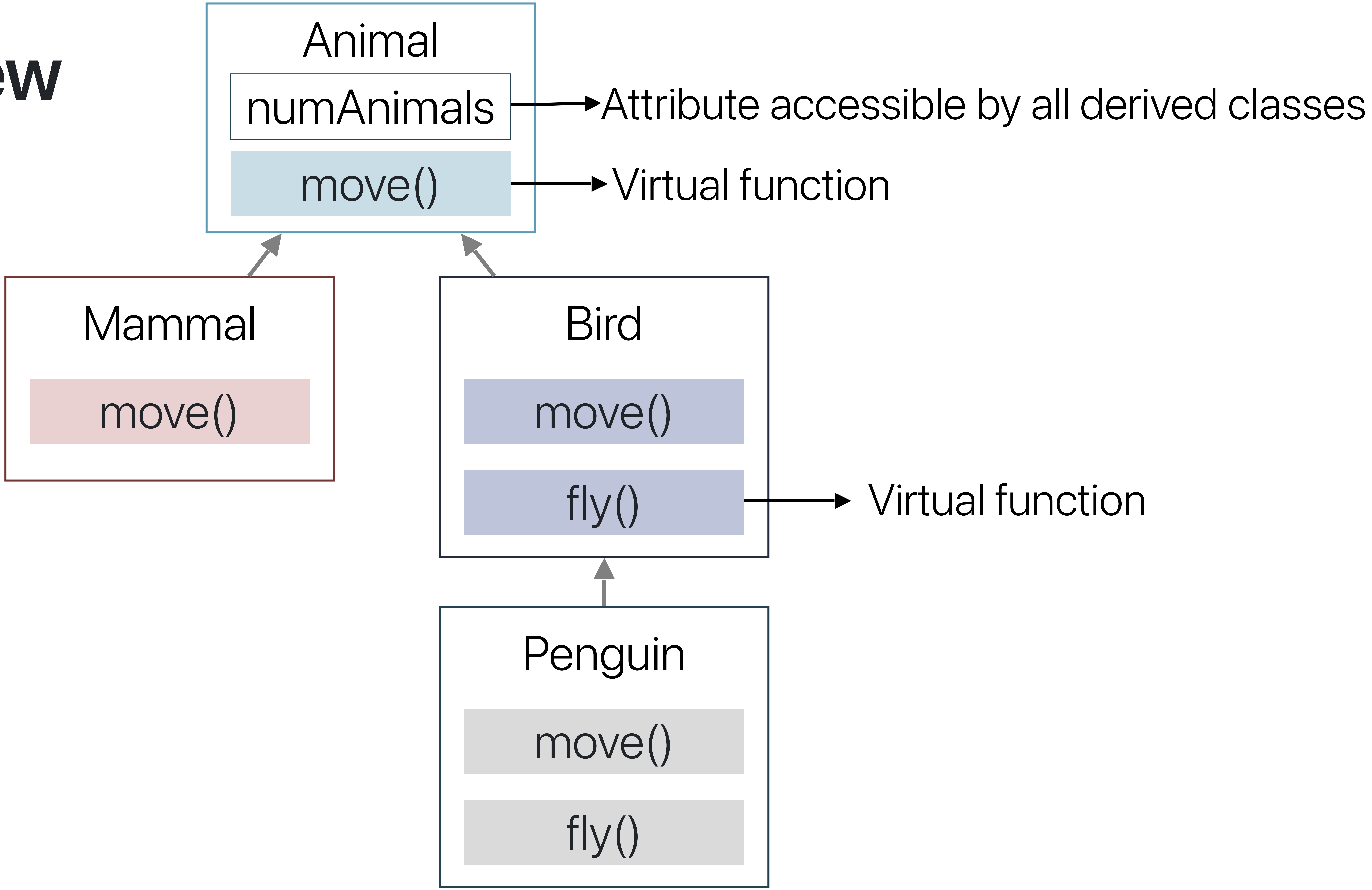
Exercises

Doyoun Kim

Overview

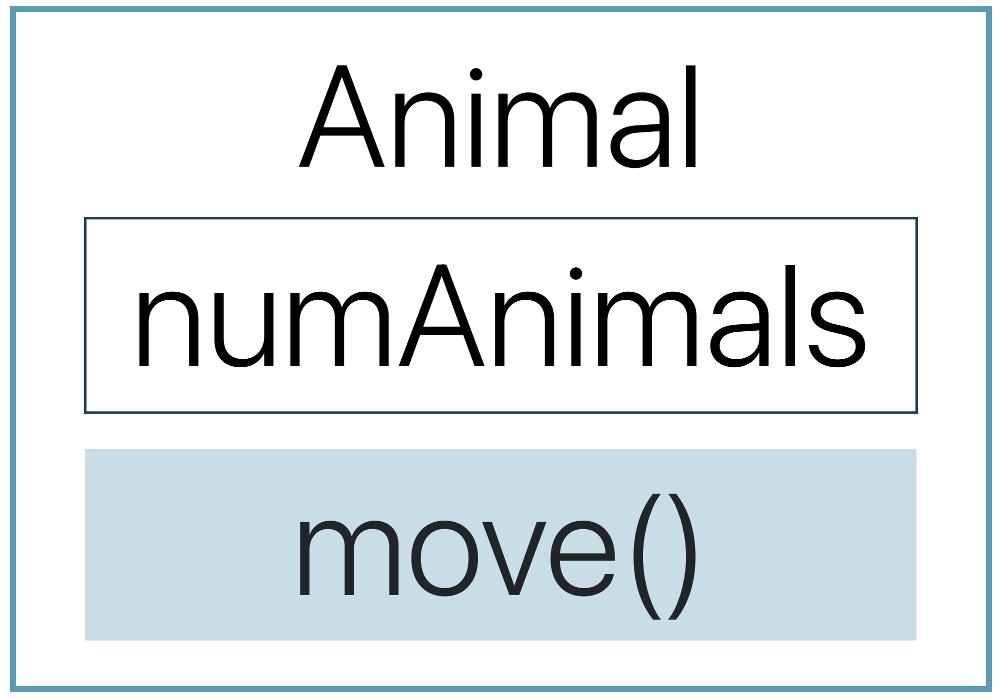
- A goal of this exercise is to understand class inheritance in C++.
- In this exercise, you will…
  - implement a hierarchy of animal classes.
  - use virtual functions to allow derived classes to override the base class's function with specific behaviors.

Overview



Animal
numAnimals
move()
Attribute accessible by all derived classes
Virtual function
Mammal
move()
Bird
move()
fly()
Virtual function
Penguin
move()
fly()

Overview - Animal



Key Attributes and Functions

- Static member: numAnimals
    - Tracks the total number of Animal objects.
    - This attribute is shared across all derived classes and external code can access this directly.
    - It should be incremented in the constructor and decremented in destructor.
- Pure virtual function: move()
    - A virtual function that must be implemented by any derived class.
    - It forces derived classes to implement specific behaviors for movement.

Overview - Mammal

Mammal

move()

Inheritance from Animal

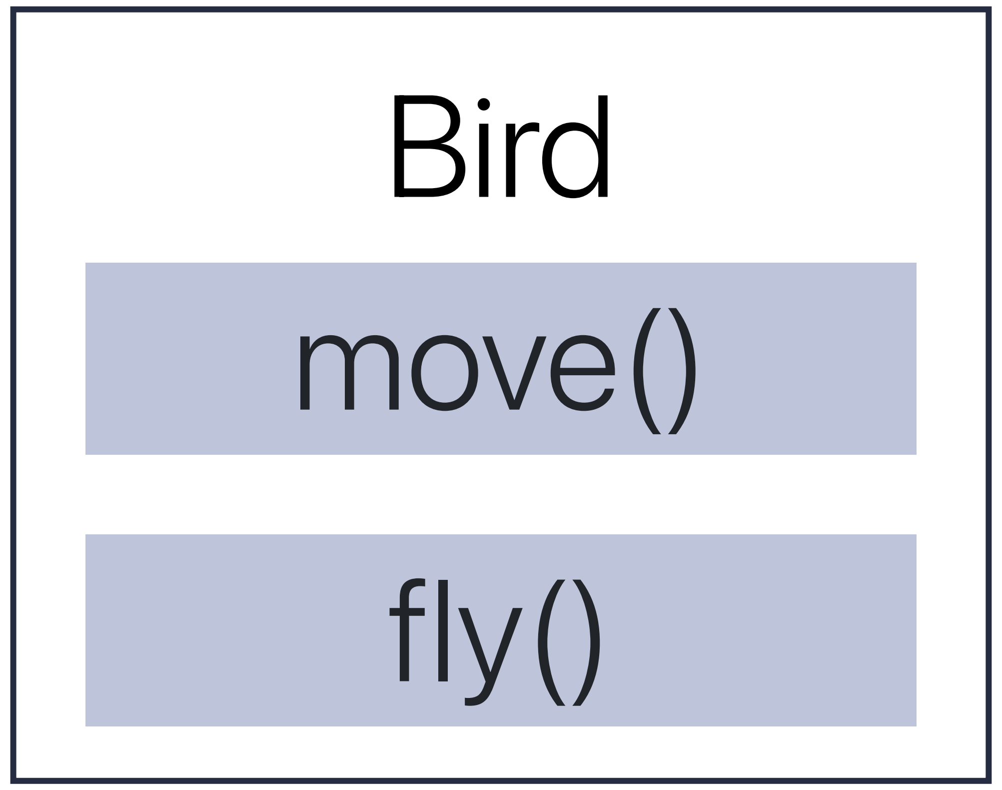
- The Mammal class inherits all the properties and functions from the Animal class.

Overriden move() function

- The Mammal class overrides the move() function to provide a specific implementation that prints "Mammal walks on land!".

```
Mammal* mammal = new Mammal(); // numAnimals + 1
mammal->move();  // Outputs: "Mammal walks on land!"
delete mammal; // numAnimals - 1
```

Overview - Bird



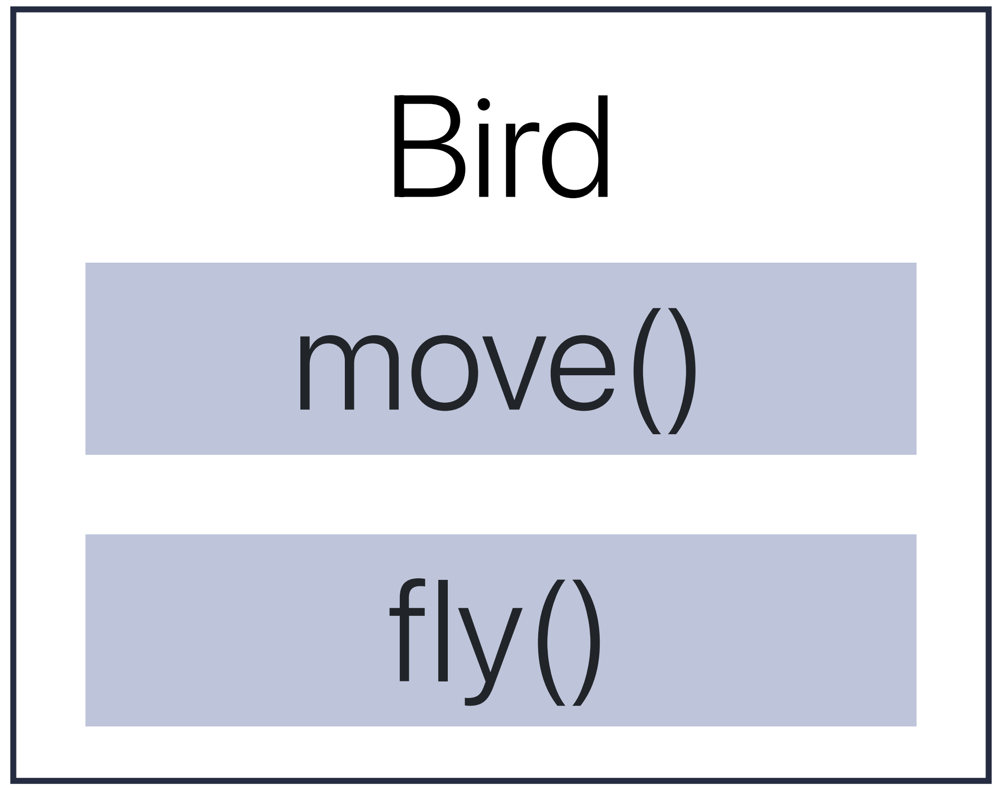
Inheritance from Animal

- Like Mammal, Bird class also inherits the static numAnimals variable and the pure virtual move() function from Animal.

Overriden move() function

- The Bird class overrides the move() function to provide a specific implementation that prints ”Bird files in the sky!".

Overview - Bird

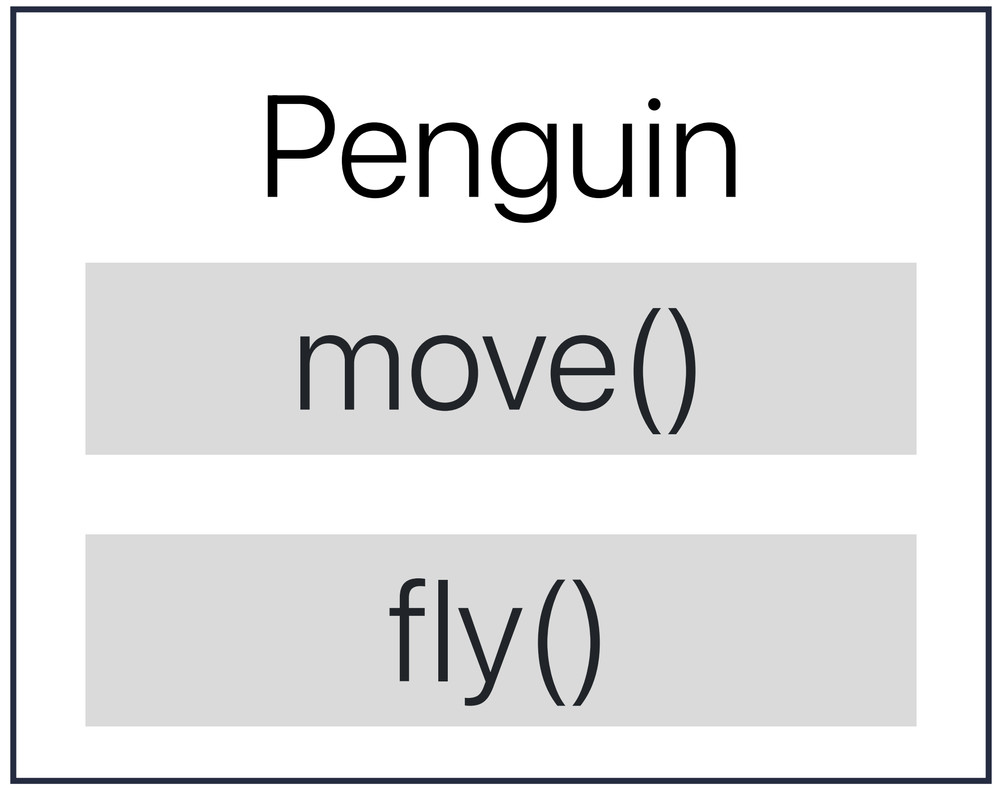


Additional fly() method

- The Bird class introduces an additional fly() method, which prints "Bird is flying!".

```
Bird* bird = new Bird(); // numAnimals + 1
bird->move();  // Outputs: “Bird files in the sky!”
bird->fly();   // Outputs: “Bird is flying!”
delete bird; // numAnimals - 1
```

Overview - Penguin



Inheritance from Bird

- The Penguin class inherits from Bird, but overrides the move() and fly() functions to provide penguin-specific behavior.

Overridden move() function

- Penguins don't fly, so the move() function is overridden to print "Penguin swims but cannot fly.".

Overview - Penguin

Penguin

move()

fly()

Overridden fly() function

- Since penguins can't fly, the fly() function is overridden to throw a logic_error and the function should print “Penguins cannot fly!”.
- This ensures that when fly() is called on a Penguin object through a Bird* or Penguin* pointer, the invalid operation is properly handled by throwing an exception.

```
Penguin* penguin = new Penguin(); // numAnimals + 1
penguin->move();  // Outputs: “Bird files in the sky!”
penguin->fly();   // Outputs: “std::logic_error: Penguins cannot fly!”
delete penguin; // numAnimals - 1
```

Run exercise.cpp

- Find the TODO sections in exercise.cpp files and implement them correctly based on the instructions.
- There are 6 TODOs.
- $ g++ exercise.cpp -o exercise -std=c++11
- $ ./exercise

```
Number of animals: 3
Testing Mammal:
Mammal walks on land!

Testing Bird:
Bird flies in the sky!
Bird is flying!

Testing Penguin:
Penguin swims but cannot fly.
Error: Penguins cannot fly!

Number of animals: 0
```

Expected output

Solutions – TODO 1

```
class Animal {
public:
    static int numAnimals;
    Animal() { numAnimals++; }
    virtual void move() const = 0;
    virtual ~Animal() { numAnimals--; }
};


int Animal::numAnimals = 0;
```

Solutions – TODO 2

```cpp
class Mammal : public Animal {
public:
    void move() const override {
        std::cout << "Mammal walks on land!" << std::endl;
    }
};
```

Solutions – TODO 3

```cpp
class Bird : public Animal {
public:
    void move() const override {
        std::cout << "Bird flies in the sky!" << std::endl;
    }
    virtual void fly() const {
        std::cout << "Bird is flying!" << std::endl;
    }
};
```

Solutions – TODO 4

```cpp
class Penguin : public Bird {
public:
    void move() const override {
        std::cout << "Penguin swims but cannot fly." << std::endl;
    }
    void fly() const override {
        throw std::logic_error("Penguins cannot fly!");
    }
};
```

Solutions – TODO 5

```
Animal* mammal = new Mammal();
Animal* bird = new Bird();
Animal* penguin = new Penguin();

std::cout << "Number of animals: " << Animal::numAnimals << '\n';
```

Solutions – TODO 6

```
delete mammal;
delete bird;
delete penguin;

std::cout << "Number of animals: " << Animal::numAnimals << '\n';
```

Thank you