Object Oriented Programming 1

Homework 1

What is it about?

- Lambdas
- Inheritance
- Virtual Functions
- Operator Overloading
- Containers
- Clang-Format

Lambdas

What are lambdas?

- Functions without a name (anonymous functions)
- Often used to pass a short function into other functions like `std::for_each`

Basic lambda structure

```
[ capture clause ] ( parameters ) -> return_type {
  // Function body
};
```

Capture clauses (`[]`)

- Specifies which variables from the surrounding scope should be "captured"
- Common captures:
  - `[]` : Capture nothing
  - `[&]` : Capture all variables by reference
  - `[=]` : Capture all variables by value
  - `[some_var]` : Only capture the variable `some_var`

Captured variables are then available inside the lambda

Parameters (`()`)

- Defines the parameters of the lambda function
- Like in any other function or method
  - Can be left empty if none are needed
  - Separated by a comma

Return Type (`-> return_type`)

- Like in any other function or method
  - Tells the compiler and programmer what the function returns
- Can be omitted
  - Is then inferred from the body of the lambda

Body (`{}`)

- Like in any other function or method
- The code that is executed when the lambda is called

Simple Examples

```
auto add = [](int a, int b) {
  return a + b;
};

int x = 10;
auto add_x = [x](int num) -> int {
  return num + x;
};

int normal_result = add(1, 2);
int result_with_x = add_x(1);

// normal_result is 3
// result_with_x is 11
```

Example with `std::for_each`

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5};

auto print = [](int x) {
  std::cout << x << std::endl;
};

std::for_each(vec.begin(), vec.end(), print);

// Output:
//  1
//  2
//  ...
```

Inheritance

What is inheritance?

- A class can use another as a base
  - This class is called a **derived** class
  - The base class is often also called **parent** class
- The derived class "copies" (inherits) from the base class:
  - Functions
  - Member variables
- The derived class can add new functions and variables on top of the parent

Why use inheritance?

- Reusability:
  Write code once, use it in multiple places
- Maintainability:
  Update the base class and changes apply to derived classes
- Extensibility:
  Easily extend functionality of existing classes

Basic Syntax

```
class Base {
  // Base class members
};

class Derived : public Base {
  // Added class members
};
```

What does the "public" mean?

```
class Derived : public Base {
```

- Change which parts of the base class are "copied" (inherited) to the derived class
- Options: `public`, `private` and `protected`
- These are called "access modifiers"
- Are also used to control access to members

- Public:
  - Inherited members keep the same modifier as in the base class
- Protected
  - Inherited `public` members become `protected`
  - Inherited `protected` members stay the same
- Private
  - All inherited members are now `private` inside the derived class

`private` members of the base class are **not** accessible inside the derived class

Example

```
class Animal {
  public:
    void eat() {
      std::cout << "The animal eats." << std::endl;
    }
};

class Dog : public Animal {
  public:
    void bark() {
      std::cout << "The dog barks." << std::endl;
    }
};
```

Every class derived from `Animal` can `eat`
but only the `Dog` can `bark`

```
auto animal = Animal();
animal.eat(); // The animal eats.

auto dog = Dog();
dog.eat();  // The animal eats.
dog.bark(); // The dog barks.
```

Virtual Functions

What are `virtual` functions?

- Functions in the base class can be marked as `virtual`
- These `virtual` functions *can* then be overwritten by a derived class
- The function defaults to the implementation of the base class *(if it doesn't get overwritten)*

Virtual Functions

```
class Animal {
  public:
    virtual void eat() {
      std::cout << "The animal eats." << std::endl;
    }
};

class Dog : public Animal {
  public:
    void eat() override {
      std::cout << "The dog eats." << std::endl;
    }
};

class Cat : public Animal {
  public:
    void eat() override {
      std::cout << "The cat eats." << std::endl;
    }
};
```

Virtual Functions

```
// ------------- Wrong ---------------

Animal dog = Dog();
Animal cat = Cat();

dog.eat(); // The animal eats. ❌
cat.eat(); // The animal eats. ❌

// ------------ Correct -------------

Animal* dog = new Dog();
Animal* cat = new Cat();

dog->eat(); // The dog eats. ✅
cat->eat(); // The cat eats. ✅

delete dog;
delete cat;
```

Pure `virtual` functions

- Is a function that **has** to be overwritten by the derived classes
- Methods that are already `virtual` can additionally be marked as **pure** `virtual`
  - *(Also known as **Abstract Functions**)*
- Compiler error if not all derived classes override this function

Difference between `virtual` and pure `virtual`

```
class Base {
  // Function that *can* be overwritten. Defaults to printing "Hello".
  virtual void virtual_hello() {
    std::cout << "Hello" << std::endl;
  }

  // Function that *has* to be overwritten. No default implementation.
  // Compiler error if it is not overwritten.
  virtual void pure_virtual_hello() = 0;
};
```

Virtual Functions

```
class Shape {
  public:
    virtual void pure_virtual_hello() = 0;
};

class Circle : public Shape {
  public:
    void pure_virtual_hello() override {
      std::cout << "Hello from a Circle" << std::endl;
    }
};

class Square : public Shape {
  public:
    void pure_virtual_hello() override {
      std::cout << "Hello from a Square" << std::endl;
    }
};
```

Virtual Functions

```
void say_hello(Shape* shape) {
  shape->pure_virtual_hello();
}

int main() {
  Shape* shape1 = new Circle();
  Shape* shape2 = new Square();

  say_hello(shape1); // Hello from a Circle
  say_hello(shape2); // Hello from a Square

  delete shape1;
  delete shape2;

  return 0;
}
```

Operator Overloading

Operator Overloading

```
class Vector2 {
  public:
    Vector2(int x, int y): x_(x), y_(y) {}

    int x_;
    int y_;
};
```

How you have done it until now

```
Vector2 add_vectors(Vector2 v1, Vector2 v2) {
  Vector2 result(0, 0);

  result.x_ = v1.x_ + v2.x_;
  result.y_ = v1.y_ + v2.y_;

  return result;
}

Vector2 subtract_vectors(Vector2 v1, Vector2 v2) {
  Vector2 result(0, 0);

  result.x_ = v1.x_ - v2.x_;
  result.y_ = v1.y_ - v2.y_;

  return result;
}
// And so on...
```

How you have done it until now

```
Vector2 vec1(1, 2);
Vector2 vec2(2, 3);

Vector2 result = add_vectors(vec1, vec2);

std::cout << "X: " << result.x_ << std::endl;
std::cout << "Y: " << result.y_ << std::endl;

// Output:
// X: 3
// Y: 5
```

How you can do it with Overloading

```
class Vector2 {
  public:
    Vector2(int x, int y): x_(x), y_(y) {}

    int x_;
    int y_;

    Vector2 operator+(const Vector2& other) const {
      return Vector2(x_ + other.x_, y_ + other.y_)
    }
};
```

How you can do it with Overloading

```
Vector2 vec1(1, 2);
Vector2 vec2(2, 3);

// Add the two vectors like any other integer
Vector2 result = vec1 + vec2;

std::cout << "X: " << result.x_ << std::endl;
std::cout << "Y: " << result.y_ << std::endl;

// Output:
// X: 3
// Y: 5
```

Bonus: Handle different data types

```
class Vector2 {
  public:
    Vector2(int x, int y): x_(x), y_(y) {}

    int x_;
    int y_;

    Vector2 operator+(const int number) const {
      return Vector2(x_ + number, y_ + number)
    }
};
```

Bonus: Handle different data types

```
Vector2 vec(1, 2);
int num = 3;

// Add a integer to a vector
Vector2 result = vec + num;

std::cout << "X: " << result.x_ << std::endl;
std::cout << "Y: " << result.y_ << std::endl;

// Output:
// X: 4
// Y: 5
```

Why use Operator Overloading?

- Reuse common code
- Make code cleaner
- Make code more readable

Why not use Operator Overloading?

- Prone to logic errors:
  What would be the result of a vector multiplication?
  The dot product or the scalar product?

What can you override?

- Basically everything:
  - Addition, Subtraction, Division, Multiplication, ...
  - Assignment operators (`=` , `+=` , `-=` , ...)
  - Increment / Decrement operators (`++` , `--`)
  - Bitwise operators (`&` , `|` , `^` , `~` , ...)
  - Member access operators (`[]`)

Full list: en.cppreference.com/w/cpp/language/operators

Extending the example even more

```
class Vector2 {
  public:
    Vector2(int x, int y): x_(x), y_(y) {}

    int x_;
    int y_;
};

std::ostream& operator<<(std::ostream& os, const Vector2& vec) {
  os << "X: " << vec.x_ << ", Y: " << vec.y_;
  return os;
}
```

Extending the example even more

Now you can easily append your `Vector2` to `std::cout` or any other output stream:

```
Vector2 vec1(1, 2);
std::cout << vec1 << std::endl;

// Output:
// X: 1, Y: 2
```

So basically this often used C++ syntax for printing is just an overloaded "bitwise left shift" operation

Containers

What are Containers?

- Data Structures that store and organize collections of objects
- Allow for easy access, modification and iteration
- Many are part of the standard library

Why use Containers?

- Easy way of storing multiple elements
- Simplify memory management & organization
- Build-in functionalities like searching, sorting, and iterating

Some examples:

- `std::array`
  - Fixed size list of elements
  - Safe alternative to C-Style Arrays
- `std::vector`
  - Dynamically allocated
  - Can grow in size
- `std::map`
  - Stores key-value pairs
  - Very fast lookups

Common operations

- Insertions: `push_back()`, `insert()`
- Search Elements: `find()`
- Access Elements: `at()`

Safer alternatives to C

```cpp
// Make an array of integers with the fixed size of 5 elements
std::array<int, 5> arr = {1, 2, 3, 4, 5};

// Safer lookups in the array, will throw an error if the index doesn't exist
// In C you would use arr[2] but this will return invalid data if the index doesn't exit
int element = arr.at(2);
std::cout << element << std::endl;

// Output:
// 3
```

Safer alternatives to C

```cpp
// Make an vector of integers with a starting size of 5 elements
std::vector<int> vec = {1, 2, 3, 4, 5};

// Add a new element to the vector, it will be added on the end
vec.push_back(6);

// Add a new element at the beginning of the vector
vec.insert(vec.begin(), 0);

// Loop through the vector in an foreach loop
// Safer since you cannot have an off-by-one error
for (const auto& element : vec) {
  std::cout << element << std::endl;
}

// The vector will be deallocated automatically, no need to worry about memory management

// Output:
// 0
// 1
// ...
```

More Lambda examples

Lambda examples

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5};

std::for_each(vec.begin(), vec.end(), [](auto& element) {
  // Add one to each element
  return element += 1;
});

for (const auto& element : vec) {
  std::cout << element << std::endl;
}

// Output:
// 2
// 3
// ...
```

Lambda examples

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5, 18, 101, 7, 420};

// Find the first element that is above 100
auto element_it = std::find_if(vec.begin(), vec.end(), [](const auto& element) {
  return element > 100;
});

// Check if a number was found
if (it != vec.end()) {
    std::cout << "The first number greater than 100 is: " << *it << std::endl;
} else {
    std::cout << "No number greater than 100 was found." << std::endl;
}

// Output:
// The first number greater than 100 is: 101
```

Clang-Format

Clang-Format

- A tool that automatically **formats C/C++** code
- Ensures **consistent style** throughout the project
- Improves code **readability and maintainability**

Once configured, you no longer need to worry about our coding standard and can focus entirely on writing code

Guide: Specification / clang-format.md