Assignment 5
Virtuals & Abstract Classes

Benjamin Zofcin
zofcin@unlv.nevada.edu
CS 202 - Fall 2023

Assignment Topics

The core concepts of this assignment are the culmination of your knowledge surrounding pointers and object oriented programming. In this assignment you will implement your own library of header files and enforce your knowledge of the following concepts:

1. Pointers and Pointer arithmetic
2. Dynamic memory management (allocation and deallocation)
3. Dynamic Dispatch
   (a) Virtual Functions
   (b) Pure Virtual Functions and Abstract Classes

You don't need to read the following block of text, as it is just my ramblings about proper Object Oriented Design and why I don't agree with the use of pointer casting for object pointers.

There are several schools of thought surrounding pointer management as it relates to pointers-to-classes. For this assignment we will abide by the following idealogy: "A well designed and structured object oriented program requires no casting". Casting non-primitive pointers is a dangerous task and can often lead to horrendous runtime errors that can take hours to resolve and is coincidentally the reason why the virtual keyword and dynamic dispatch were created in the first place. Therefore, this assignment will not utilize pointer casting of any kind (static, dynamic, or reinterpret) and instead focus on developing a rigid class hierarchy that implements abstract classes to restrict and enforce object creation. That being said, if you don't agree with my approach, then you are more than welcome to implement the systems class hierarchy in your own way and utilize the respective casting. Be sure to notify your TA as the test bed I provide to you will not work under such an approach. That being said, let's get into the content and goal of the assignment.

Introduction and Outcome

In the field of computer programming you will often find yourself designing programs that are meant to emulate currently existing technology or simulate some tangible system. Entities like: the Department of Defense, SpaceX, or Chemical Giants like Samsung, will utilize simulation engines in place of real world testing due to the cost to benefit ratio associated with the real world testing. Simulation engines (specifically those in virtual reality) can also be used to train professionals in any sort of field.

In this assignment we will explore writing a simulator for professional bowling. While most people who bowl, bowl as a recreational sport, there are those who do so professionally and are experts in their craft. Let's assume you are a freelance programmer who was approached by a professional bowler. This individual wants you to design and create for them a program that simulates a bowling scenario. Just like any other simulation engine you will need to familiarize yourself with all aspects of the system you are attempting to simulate. In your case, this will involve delving into the fundamental properties of bowling. This will lead you into a in-depth conversation with the client about all the aspects regarding the sport of bowling. After which, you would come away with the following big picture items:

1. Bowler Statistics:
   (a) Ball speed - This is related to how fast the bowler will roll the ball down the lane and is dependent on ball weight and overall bowler style.
   (b) Ball Revolution Rate - The average amount of revolutions a bowler can apply to their bowling ball to induce "hook" as the ball transitions down the lane.

       Hook is said to be a bowling balls transition away from its current trajectory as it encounters friction.
2. Bowling Ball Components:
   (a) Core - Cores are the internal engine of the ball and can be symmetric or asymmetric. The geometry of the core and can drastically change the way a ball moves down the lane.
   (b) Coverstock - This is the exterior of the bowling ball. Depending of the type of material, the ball will transfer oil down the lane at different rates. The coverstock is also a major factor in determining a bowling balls hook potential and is mainly dictated by the grit of the bowling ball.
   (c) Drill Layout - A trio of values that effect the hook potential in some round about way. Even the professional bowler sounded a little unsure of their knowledge surrounding this subject. You decide to integrate the behavior into the simulation engine and learn more on it as you go.
3. Lane
   (a) Oil Pattern - The layout of oil on the lane will dictate if a bowling ball will glide over the lane or generate enough friction to hook and change its trajectory.

From there, you will need do more research on your own to familiarize yourself with the world of bowling and see if anyone else has implemented bowling simulators. Wii bowling and many other mobile bowling apps may come to mind. However, to your surprise, a company has developed a robot called EARL. When the robot was put up against a professional bowler, the robot designers determined that while the robot was capable of launching the ball consistently; however, the robot lacked the knowledge of how the oil pattern was changing on the lane after every ball was thrown. Now that you have an idea of what has been implemented and what could use improving you set out with the following goal:

"Write a program that, given a selection of bowling balls and a given lane condition, determines the best bowling ball to throw to get a strike."

Implementation Files

The following sections will outline each file that must be implemented and handed in. Proper object oriented design in C++ often relates itself to an implementation of a library of header files. Therefore, the only files you will implement in this assignment are .hpp files. All of the header files are provided to you (some are more complete than others) with accompanying .cpp unit tests for each. The following (in recommended completion order) is the list of header files that will be expanded upon in later sections:

1. `core_type.hpp`
2. `symmetric.hpp`
3. `asymmetric.hpp`
4. `coverstock_type.hpp`
5. `plastic.hpp`
6. `performance.hpp`
7. `urethane.hpp`
8. `reactive.hpp`
9. `bowling_ball.hpp`
10. `oil_pattern.hpp`
11. `lane_type.hpp`

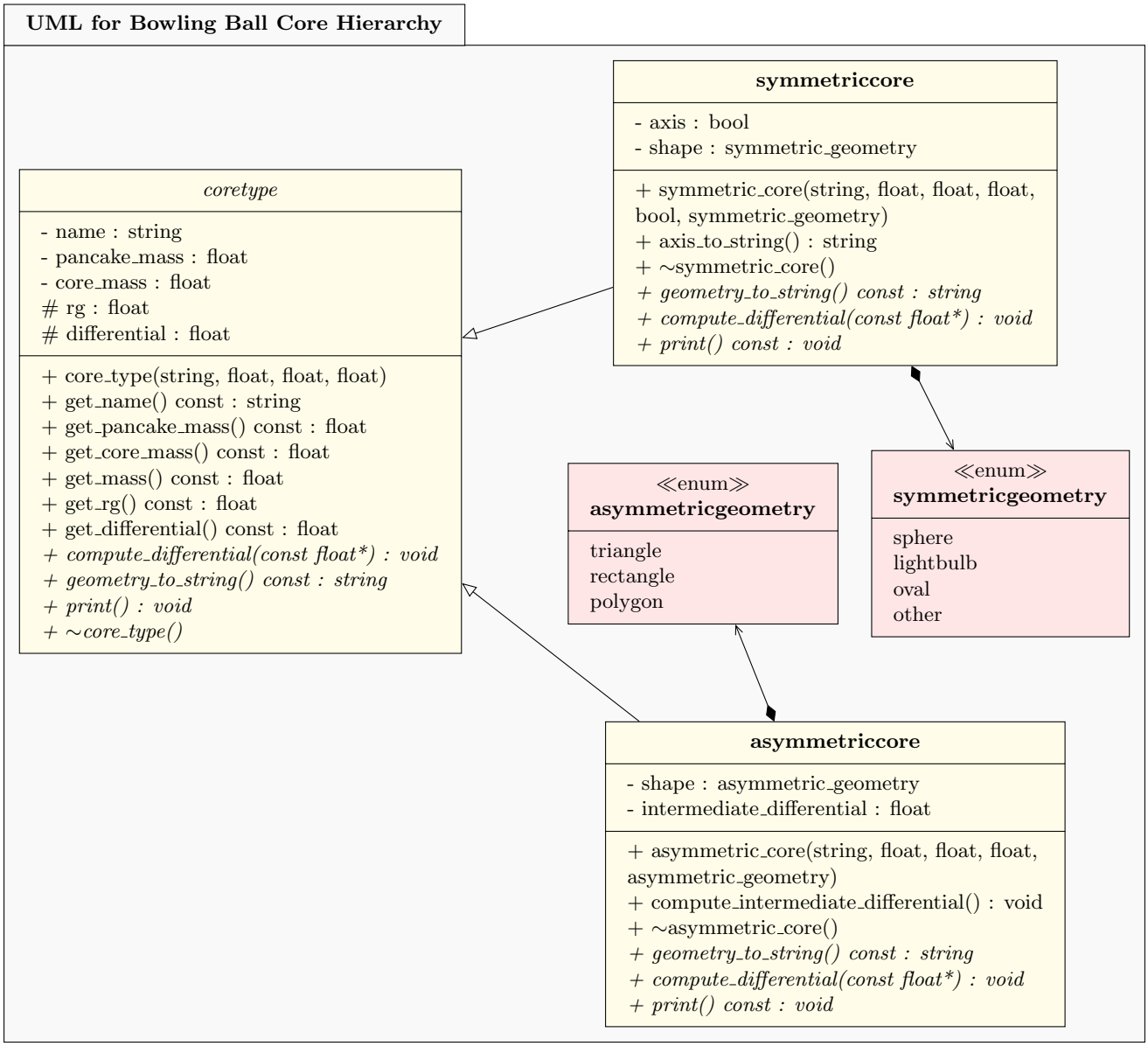
Program Design

Variables are in green.
Functions that are to be implemented as part of this assignment are given in red.
Functions that are partially implemented are in magenta
Functions that have been implemented for you are in blue..

Helpful Tip

Abstract classes and (pure) virtual functions are shown in *italics*.
+ is public, - is private, # is protected
Class and Enum titles are omitting their underscores because this Latex package is awful.

Bowling Ball Cores



class core_type

The `core_type` class should be abstract as no bowling ball core is "just a core"

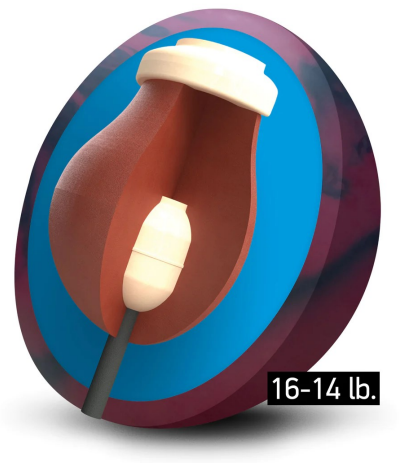
- `name` - Name of the core.
- `pancake_mass` - Pancake mass.
- `core_mass` - Core mass.
- `rg` - Radius of gyration (2.4 - 2.8, lower value → earlier hook).
- `differential` - 0 - 0.8, higher value → aggressive hook.
- `get_name() const` - Inline accessor to return name.
- `get_pancake_mass() const` - Inline accessor to return `pancake_mass`.
- `get_core_mass() const` - Inline accessor to return `core_mass`.
- `get_mass() const` - Inline accessor to return the combination of core and pancake masses.
- `get_rg() const` - Inline accessor to return rg.
- `get_differential() const` - Inline accessor to return differential.
- `compute_differential()` - Should be declared as a pure virtual function and will be implemented in the child classes.
- `geometry_to_string() const` - Should be declared as a pure virtual function and will be implemented in the child classes.
- `print() const` - Should be declared as a pure virtual function and will be implemented in the child classes.
- `~core_type()` - Best practices dictates that all abstract classes must have their destructors declared as virtual

0.0.1 Unit Tests

There is only 1 test to validate that the `core_type` class cannot be instantiated. Note, that this test is designed to fail at compile time. See code grade for expected output.

class symmetric_core

Symmetric cores are the most popular among beginner bowlers due to their predictability and smooth movement down the lane.



Example of a lightbulb core

Example of a spherical core with a pancake

The `symmetric_core` is a child of the `core_type` and will implement all of its pure virtual functions to become an object that can be instantiated.

- `axis` - Axisymmetric or non-axisymmetric
- `shape` - Physical shape of the core
- `symmetric_core(string, float, float, float, bool, symmetric_geometry)`
  Must invoke the parent class constructor with the first 4 parameters. Then initialize the differential to zero and set the axis and shape instance variables using the last 2 parameters.
- `axis_to_string()`
  Based on the value of axis, this function will return a descriptive message about the axis.
- `~symmetric_core()`
  Declare and implement the destructor for the `symmetric_core` class so the parent class destructor may also be invoked.
- `print() const`
  Provides a descriptive message about the core
- `geometry_to_string() const`
  Depending on the value of shape, this function will return the string equivalent of the `symmetric_geometry` enumeration.

  (NOTE: While implementing the logic with if statements will work here, I recommend using a switch statement any time you are testing an enumerated value. This is best practices when working with any enumeration and will simplify many of the functions in this assignment to a copy/paste mentality)
- `compute_differential(const float*)`
  Through your research you found that the calculations for differential are based on how the ball is drilled, is often complex, and vary from ball manufacturer to manufacturer. Therefore, the majority of the function has been implemented by the professional bowler to fit a particular ball manufacturer. The only thing you will need to do is provide the final calculations for PTP (pin to positive axis point).

  PTP should be scaled by the first value pointed to by the parameter and then divided by the provided `OFFSET_MAX`.

  Feel free to ask the professional bowler how the calculation actually works.

0.0.2 Unit Tests

There are 3 tests to validate the `symmetric_core`

1. Basic creation of a `symmetric_core` and passing pointers to a common sink function
2. Validate `axis_to_string()` and `geometry_to_string()`
3. Validate the `compute_differential(const float*)`

class asymmetric_core

Asymmetric core bowling ball are those whose cores have no symmetry and thus operate a bit different from the traditional bowling ball.

Example of an Asymmetric polygonal core (Black Widow)

Example of an Asymmetric rectangular core (Reality)

The `asymmetric_core` is a child of the `core_type` and will implement all of its pure virtual functions to become an object that can be instantiated.

- `shape` - Physical shape of the core
- `intermediate_differential` - The intermediate differential is a secondary pivot point that only asymmetric bowling balls will have
- `asymmetric_core(string, float, float, float, asymmetric_geometry)`
  Must invoke the parent class constructor with the first 4 parameters. Then initialize the differential and `intermediate_differential` to zero, set the shape instance variables using the last parameters, and finally invoke the `compute_intermediate_differential()`.
- `compute_intermediate_differential()`
  Based on the value of shape, this function will increment the `intermediate_differential` by a fixed value.
  1. triangle +0.05
  2. rectangle +0.1
  3. polygon +0.2

  (HINT: Use a switch statement)
- `~symmetric_core()`
  Declare and implement the destructor for the `asymmetric_core` class so the parent class destructor may also be invoked.
- `print() const`
  Provides a descriptive message about the core
- `geometry_to_string() const`
  This function will behave identically to the `geometry_to_string` in `symmetric_core` only using the `asymmetric_geometry` shape. (HINT: Use a switch statement);

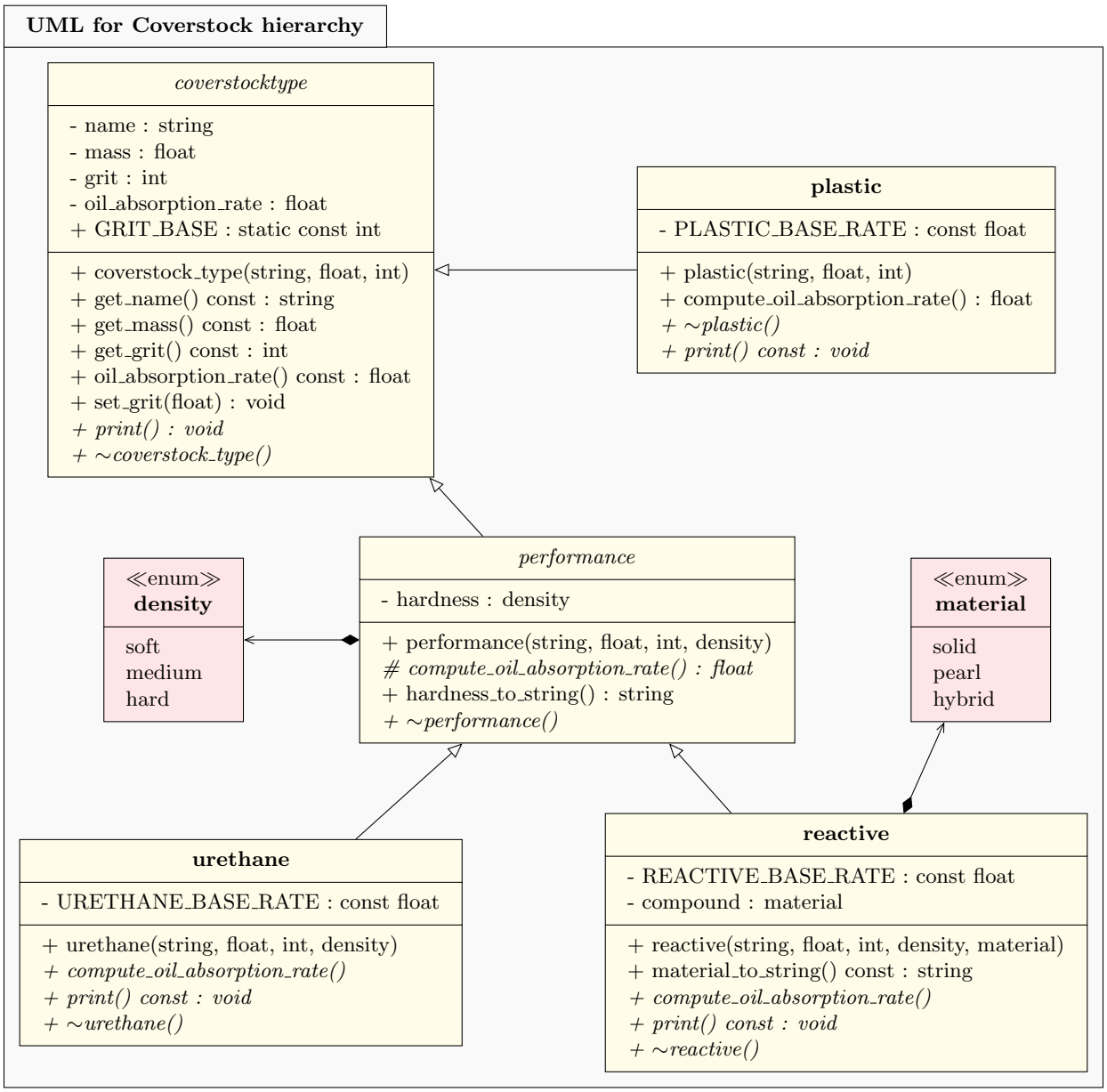
- `compute_differential(const float*)`
  This function will behave identically to the `compute_differential` in `symmetric_core` only the final differential should also include the `intermediate_differential`. (HINT: You should just be able to copy and paste)

0.0.3 Unit Tests

There are 3 tests to validate the `symmetric_core`

1. Basic creation of a `asymmetric_core` and passing pointers to a common sink function
2. Validate `geometry_to_string()` and `compute_intermediate_differential()`
3. Validate the `compute_differential(const float*)`

Bowling Ball Coverstocks



coverstock_type

The coverstock is the thing that every one sees when the ball is rolling down the lane. Predominantly we are interested in the different ways to compute 2 important values.

1. Grit level
2. Oil absorption rate

This class should be abstract and thus should not be able to be instantiated.

- `name` - Name of the coverstock
- `mass` - Total weight of the coverstock
- `grit` - Grit level of the ball [500 → 4000]
- `oil_absorption_rate` - Rate at which the track flares will absorb oil off the lane.
- `GRIT_BASE` - Minimum Grit amount for any bowling ball (500 grit)
- `coverstock_type(string, float, int)`
  Use the parameters in order to set the name, mass, and grit.
- `get_name() const`
  Inline accessor function to retrieve the name
- `get_mass() const`
  Inline accessor function to retrieve the mass
- `get_grit() const`
  Inline accessor function to retrieve the grit
- `oil_absorption_rate() const`
  Inline accessor function to retrieve the `oil_absorption_rate`
- `set_oil_absorption_rate(float)`
  This function will simply use the parameter value to set the value of `oil_absorption_rate`.

- `set_grit(int)`
  This function will set the grit to the value of the actual parameter. Before doing so, some error checking will need to take place. Produce an error message using the provided `lesser_grit` and `greater_grit` strings and exit the program (exit(1)) for the following:

  1. if parameter is lesser than 500
  2. if parameter is greater than 4000

  Then compare the current grit value to the new one and use the provided sanding and polishing strings if the new grit is lower or higher respectively.

- `print()`
  This function should be declared as a pure virtual function to implement `coverstock_type` as an abstract class.

- `~coverstock_type()`
  Best practices dictates that all abstract classes must have their destructors declared as virtual

0.0.4 Unit Test

There is only 1 unit test for this class and it validates that objects of `coverstock_type` cannot be instantiated and is designed to fail at compile time.

plastic

The plastic coverstock is easily one of the most common and is the one most recreational bowlers will use. They are known as "house ball's".

- `PLASTIC_BASE_RATE` - Base rate at which all plastic coverstock's will absorb oil.

- `plastic(string, float, int)`
  Use the parameters to invoke the constructor of the base class and then use the `compute_oil_absorption_rate` to invoke the parent class's `set_oil_absorption_rate`

- `compute_oil_absorption_rate() const`
  This function will scale the `PLASTIC_BASE_RATE` proportional to the `GRIT_BASE` with respect to the current grit of the coverstock.
  (e.g., if current grit is 2000 $\Rightarrow$ `oil_absorption_rate` = 0.1 * 500 / 2000)
  From this example, we can see that the lower the grit the more oil is absorbed. This will be true for all cover stocks so keep this equation in mind.

- `print()`
  This function will produce a formatted output for the plastic coverstock. This implementation of the parent classes pure virtual function allows plastic coverstock's to be instantiated.

- `~plastic()`
  Virtual destructor

0.0.5 Unit Test

There are 4 unit tests for the plastic coverstock. One will tests the ability to create an instance and call that instances print. The other 3 will test the `set_grit()` function; one is designed to work and test the ability of the function to produce output and update the grit; the other 2 are designed to fail.

performance

The performance style of coverstocks are the ones predominantly used by casual and professional bowlers alike. The main component of any performance coverstock is their density when manufactured. The density will dictate the rate of oil absorption in most cases.

This class should be abstract and thus should not be able to be instantiated.

- `hardness` - Variable based on the density enumeration. The harder the ball the less oil it will absorb.
- `performance(string, float, int, density)`
  Use the first 3 parameters to invoke the constructor of the base class then use the final parameter to initialize the hardness instance member.
- `compute_oil_absorption_rate() const`
  This function will return a floating point scalar value that is directly related to the value of the hardness variable.
  1. `soft` → 1.5
  2. `medium` → 1
  3. `hard` → 0.8

  (HINT: Use a switch statement)
- `hardness_to_string()`
  This function will return the string equivalent of the identifier of the hardness variable.
  (HINT: Use a switch statement)
- `~performance()`
  Virtual destructor

0.0.6 Unit Test

Since the performance class is an abstract class, there is only one unit test for it and that test is designed to fail at compile time.

urethane

The urethane ball is technically a performance ball as we need to keep track of its coverstock density. Urethanes are popular amongst beginners for their ease of use and forgiveness. Their main characteristic is how quickly they can transition oil down the lane.

- `URETHANE_BASE_RATE` - Base rate at which a urethane coverstock will absorb oil.
- `urethane(string, float, int, density)`
  Use the first 3 parameters to invoke the constructor of the base class then use the final parameter to initialize the hardness instance member.
- `compute_oil_absorption_rate() const`
  This function will compute its oil absorption rate in a similar fashion to the plastic coverstock.
  The urethane base rate is first scaled by the parent class's `compute_oil_absorption_rate()` (recall, this value is based on density of the coverstock). Once that value is calculated, apply the grit base proportionality ratio to get the final oil absorption rate
- `print()`
  This function will produce a formatted output for the urethane coverstock. This implementation of the parent class's pure virtual function allows urethane coverstock's to be instantiated.
- `~urethane()`
  Virtual destructor

0.0.7 Unit Test

There is only one unit test for the urethane coverstock. This test will create 3 different urethane coverstock's to test the implementation of the constructor and `compute_oil_absorption_rate()` functions

reactive

The reactive resin coverstock is a bit special compared to the prior 2 coverstock's. No matter how much you sand or polish the ball the oil absorption rate does not change. Instead, there are 3 variants of the reactive coverstock that provide the proportionality ratio.

- `REACTIVE_BASE_RATE` - Base rate at which a reactive coverstock will absorb oil.
- `compound` - Specific type of material that the bowling ball is made out of.
- `reactive(string, float, int, density, material)`
  Use the first 4 parameters to invoke the constructor of the base class, then use the final parameter to initialize the compound instance member, finally use the reimplemented `compute_oil_absorption_rate` to set the `oil_absorption_rate` just like you did on the plastic coverstock.
- `material_to_string() const`
  This function will return the string equivalent of the material value that compound is set to. (HINT: Use a switch statement);
- `compute_oil_absorption_rate() const`
  This function is a 2 step process.
  1. Determine the individual compound rate based on the material of the compound (HINT: Use a switch statement)
     - solid $\rightarrow$ 0.65
     - pearl $\rightarrow$ 0.9
     - hybrid $\rightarrow$ 0.7
  2. Use the compound rate to scale the `REACTIVE_BASE_RATE` and scale that value by the parent classs's `compute_oil_absorption_rate()` (recall, this value is based on density of the coverstock).
- `print()`
  This function will produce a formatted output for the reactive coverstock. This implementation of the parent class's pure virtual function allows reactive coverstock's to be instantiated.
- `~reative()`
  Virtual destructor

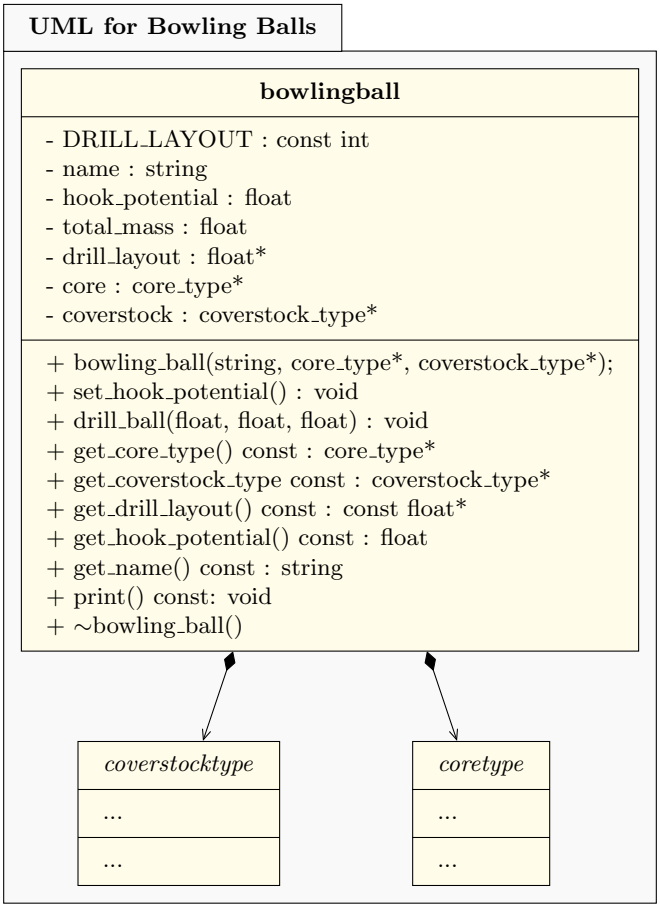
0.0.8 Unit Test

There are 3 unit tests for the reactive class

1. Simple reactive coverstock creation and print
2. Test the implementation of `material_to_string()` and `compute_oil_absorption_rate()`
3. Use base class pointers of type `core_type` and `performance` to create a reactive ball and use the member access modifier to access the dynamically dispatched `print()` function

Bowling Ball

Now that you are able to make cores and coverstocks of many varieties, you can use them to define already existing bowling balls or create entirely new ones all together.



Bowling Ball Members

- `DRILL_LAYOUT` - Amount of floats to allocate for `drill_layout`
- `name` - Name of the bowling ball
- `hook_potential` - Total hook potential of the bowling ball
- `total_mass` - Total mass of the bowling ball
- `drill_layout` - Dynamically allocated array of 3 floating point values to represent the trio of pin layout
- `core` - pointer to dynamically allocated core instance
- `coverstock` - pointer to dynamically allocated coverstock instance
- `bowling_ball(string, core_type*, coverstock_type)`
  The constructor of the will use the the parameters to initialize the name, core, and coverstock instance members respectively. The constructor must also calculate combined mass of the core and coverstock and store that value in `total_mass`. Finally the constructor must also allocate the memory for `drill_layout`.
- `drill_ball(float, float, float)`
  This job of this function is to populate the memory managed by `drill_layout` with the 3 parameters in order. Then invoke the `compute_differential()` function of the core. Recall, `compute_differential()` is an abstract function and the dynamic dispatch should resolve to the correct version of `compute_differential()` depending on the original core type.

- `set_hook_potential()`
  If everything in the core and coverstock was calculated properly and your `drill_ball()` functions invokes the `compute_differential()` function, then computing the hook potential is relatively easy. Hook potential is the culmination of
  60% grit proportionality (the same proportionality used in plastic and urethane differential computations)
  35% core differential and
  5% core radius of gyration.
  (HINT: Utilize the accessor functions of the coverstock and the core to obtain these values)
- `get_core_type() const`
  Accessor function to retrieve the `core_type`
- `get_coverstock_type const`
  Accessor function to retrieve the `coverstock_type`
- `get_drill_layout() const`
  Accessor function to retrieve the `drill_layout`
- `get_hook_potential() const`
  Accessor function to retrieve the `hook_potential`
- `get_name() const`
  Accessor function to retrieve the `name`
- `print() const`
  Print function to display the relevant information of the bowling ball.
- `~bowling_ball()`
  Because the construction of the bowling ball includes being given the address of dynamically allocated cores and coverstocks we must handle that in the destructor of the bowling ball. The `drill_layout` must also be deallocated here.

Unit Tests

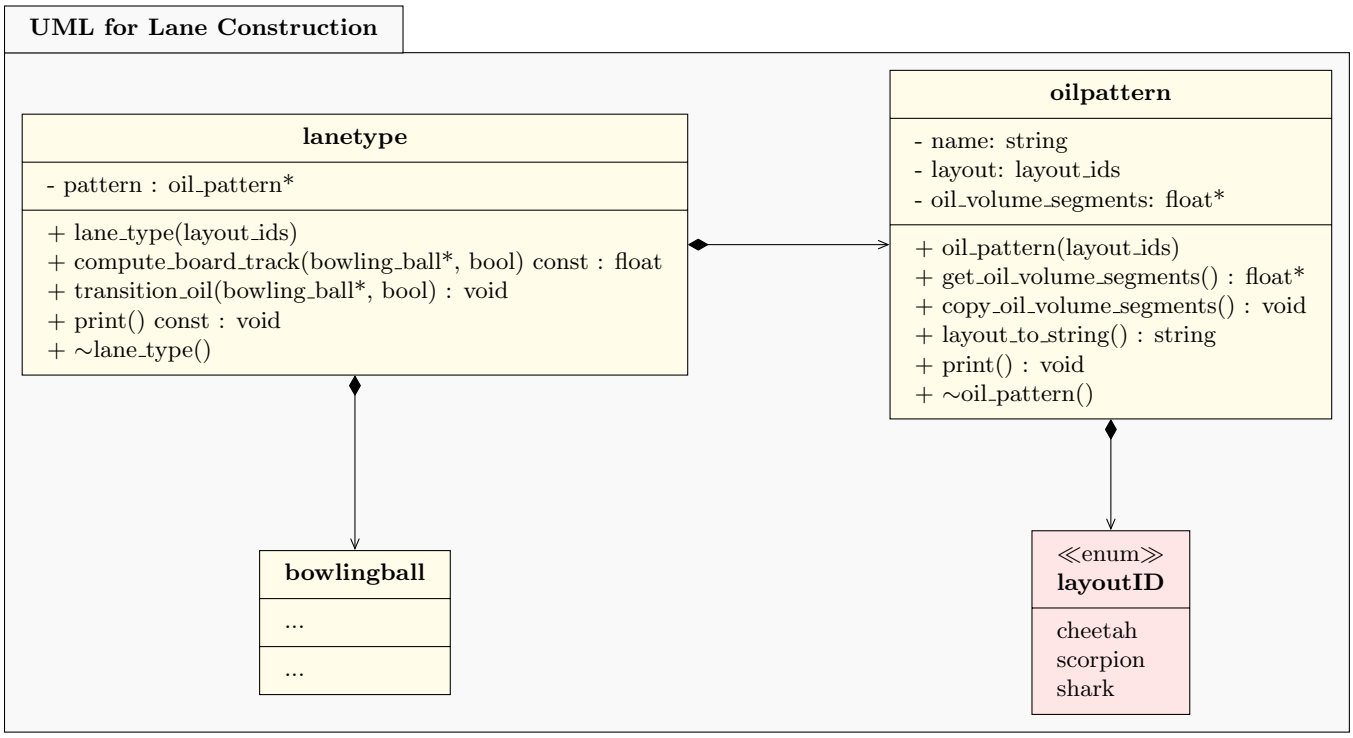
There are 2 unit tests for the `bowling_ball` class.

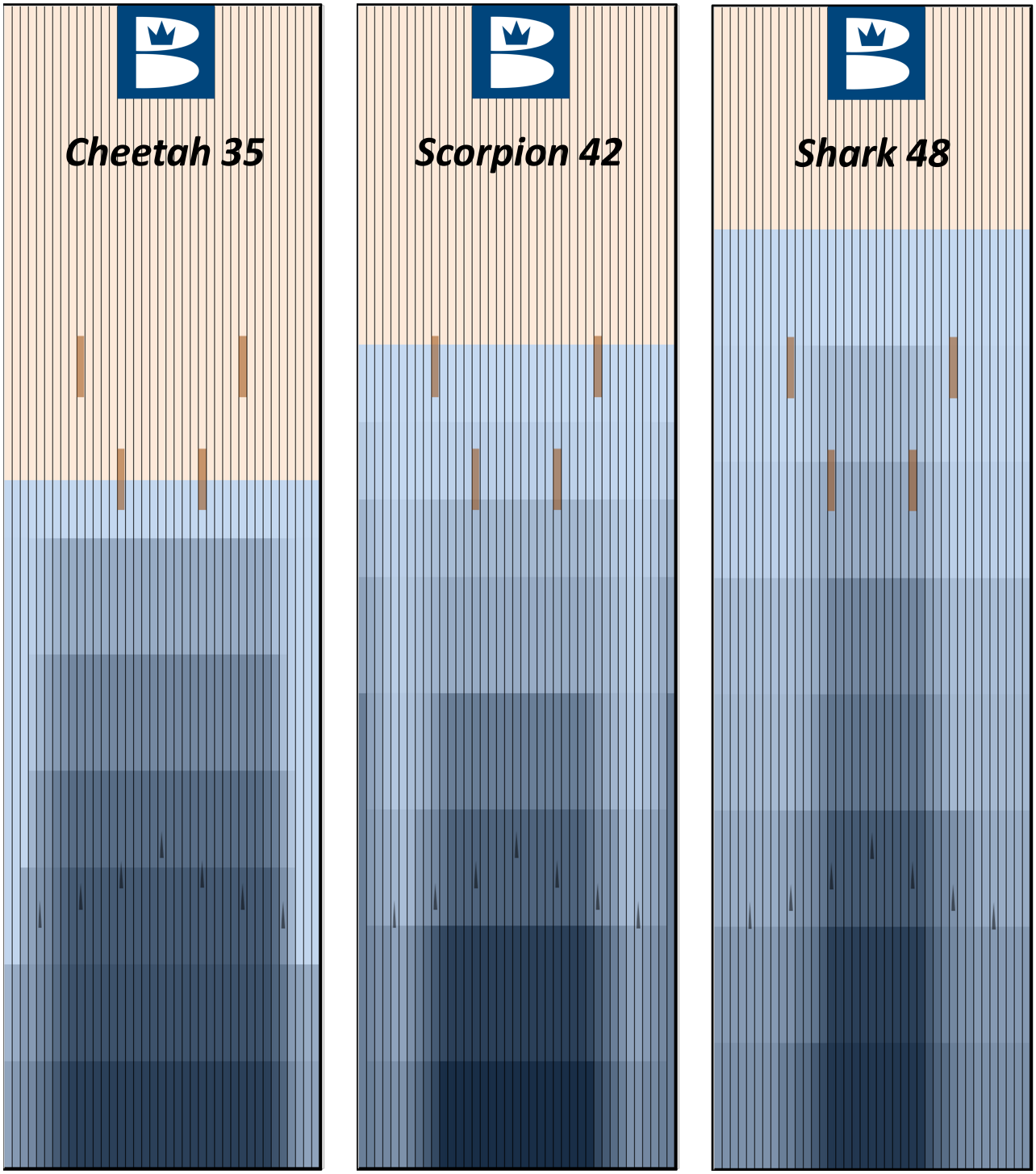
1. Basic bowling ball creation and use of the print function.
2. Validation of the `drill_ball()` function.

Please note, neither of these tests should generate any memory errors or memory leaks. If they do, points will be deducted.

Lane and Oil Patterns

Finally, we have the tools to generate any bowling ball in existence for the customer and "half" of the job is now done. Next up we need to construct the oil patterns and lane construction. Unknown to most, there are dozens of layouts for oil.





Oil Pattern Members

An oil pattern determines the volume of oil per segment on a lane. Some oil patters are leaner or richer and other oil patterns may stop halfway down the lane and others extend further. In the handout code, I have pre-encoded 3 common oil patterns.
The entries in these arrays are percentages that represent the volume of oil on the lane
(e.g., 1. $\Rightarrow$ 100% oil, 0.5 $\Rightarrow$ 50% oil, etc. . .)
The percentage of the volume of oil on the lane dictates if our bowling ball is hooking or not.

- `name` - Name of the oil pattern
- `layout` - `layout_id` tag

- `oil_volume_segments` - float pointer to allocate a deep copy of the defined pre-encoded oil patterns
- `oil_pattern(layout_ids)`
  The job of the constructor is to set the layout to the provided layout and use the `layout_to_string()` function to initialize the name. The instance member `oil_volume_segments` are allocated a segment of memory, that memory will serve as a deep copy for which every layout is defined.
- `get_oil_volume_segments()`
  Accessor function to retrieve the value of `oil_volume_segments`.
- `copy_oil_volume_segments()`
  This function will use the `layout` instance member to decode which oil pattern was defined for the instance. Using that information, the function must make a deep copy the contents of the respective pre-encoded array into the memory allocated for `oil_volume_segments`. (HINT: Use a switch statement)
- `layout_to_string()`
  This function will use the `layout` instance member to return a string equivalent of the `layout_id`. (HINT: Use a switch statement)
- `print()`
  This function will produce a formatted output of the current oil status of the lane.
- `~oil_pattern()`
  Since the constructor allocated dynamic memory for the `oil_volume_segments` in the constructor, the destructor is responsible for deallocating that memory.

0.0.9 Unit Test

There is one unit test for the `oil_pattern` class. This test will validate that your implementation is capable of copying each of the pre-encoded oil patterns and has no memory leaks.

lane_type members

Here we are, the final header file to implement. Every lane will have an oil pattern associated with it and a bowling ball to interact with said oil pattern.

- `pattern` - `oil_pattern` pointer to dynamically allocate a new oil pattern when the lane is created.
- `lane_type(layout_ids)`
  The job of the constructor is to allocate a new oil pattern for the the pattern pointer.
- `compute_board_track(bowling_ball*, bool) const`
  The job of this function is to simulate the ball rolling down the lane on a given oil pattern. Before we start looking at what the ball does on the individual segments, notice that we have some predefined constants and other variables

  1. `OIL_BOARDS_PER_SEGMENT` - Maximum board transition per segment when gliding on oil.
  2. `HOOK_BOARDS_PER_SEGMENT` - Maximum board transition when hooking on a lack of oil.
  3. `hook_potential` - You will get the balls hook potential out of the bowling ball to initialize this variable.
  4. `oil_volume_segments` - You will create a shallow copy the lane's current `oil_volume_segments`.
  5. `current_board` - Current board the bowling ball is on.

  The following images depicts the numbering of the boards and how you can imagine the ball tracing down the lane as `current_board` updates.

Now, for every segment of oil the ball interacts with on the lane, 2 values need to be computed.

1. Transition on glide (oil):
   This value represents the number of boards the ball will glide across the lane in the presence of oil. It is computed by scaling the `OIL_BOARDS_PER_SEGMENT` by the volume percentage of oil on that segment of the lane. (e.g., 85% oil * -2 boards = -1.7 boards)
2. Transition on hook (friction) This value represents the number of boards the ball will hook on given the absence of oil. The calculation will require you to take the complement of the amount of oil on the board, scale that value by the `HOOK_BOARDS_PER_SEGMENT`, and scale that value by the `hook_potential` of the ball. (e.g., 1 - 85% * 2.5 boards = 0.375 boards)

When you add the sum of those 2 values to the `current_board`, you will get the which board the bowling ball is on at the end of that lane segment. This process will be repeated for all 12 segments of the lane. The value of `current_board` at the end is the board where the bowling ball with make contact with the pins and is the value the function returns

- `transition_oil(bowling_ball*, bool)`
  As the bowling ball rolls down the lane it also interacts with the oil on the lane. While the `compute_board_track` tracks the board position of the ball as it rolls down lane, the transition oil function determines how much oil is being transitioned from the lane to the ball and vice versa.

  First, let's take a look at the variables already declared:

  1. `MAX_OIL_ABSORBED` - The maximum percentage of oil (regardless of ball) a bowling ball can absorb off the lane segment.
  2. `MAX_OIL_DISSIPATED` - The maximum percentage of oil (regardless of ball) a bowling ball can dissipate onto the lane segment.
  3. `oil_volume` - The amount of oil either being absorbed or dissipated. This is acting as our "temp" variable
  4. `oil_absorbed` - Total amount of oil absorbed by the ball.
  5. `oil_absorption_rate` - The rate at which the coverstock of the given bowling ball will absorb oil from the lane.
  6. `oil_volume_segments` - Pointer to array of oil segment values.

Now let's take a look at the logic and calculations. For every segment of the lane, 2 checks need to be made and if those checks are successful then oil transition must occur.

1. The current segment of the lane has oil to be absorbed.

   If there is oil on the lane to be absorbed, then compute the volume of oil the be the current amount of oil on that lane segment scaled by the `MAX_OIL_ABSORBED`, and scale that value by the `oil_absorption_rate` of the bowling ball. This value tells you exactly how much oil is being absorbed from the lane.

   Take that value, and do the following:

   (a) Subtract the oil volume from the current lane segment

   (b) Add the oil volume to the ball's absorbed oil

   This will effectively transition oil from the lane to the bowling ball.

2. Bowling ball has oil to be dissipate.

   If the bowling ball has absorbed oil then it must dissipate a fraction of that oil before leaving the segment. To compute the volume of dissipating oil, simply take the `oil_absorbed` value and scale it by `MAX_OIL_DISSIPATED`. This will tell you exactly how much oil will be transferred from the bowling ball to the lane segment.

   Take that value and do the following:

   (a) Subtract the oil volume from the ball's absorbed oil

   (b) Add the oil volume to the current lane segment

The `transition_oil` function will transfer oil down lane only ONCE for every invocation.

- `print() const`
  The job of the print function is to provide the status of the oil pattern.
- `~lane_type()`
  Since the constructor allocates dynamic memory, the destructor should deallocate the same memory

Unit Tests

There is a simple unit test that only validates the creation of a `lane_type` object and prints the data.

Simulation Unit Tests

At this point you should have implemented all of the header files and passed all of the associated unit tests. Now for the grand finale, this is where you get to show off to your client and demonstrate the full power of your simulation engine.

1. `simulation_test_00.cpp`
   This test will validate many criteria; most importantly, that the member function `compute_board_track` of `lane_type` is working correctly. The output will show the ball being tested and the status of where the ball is on the lane for each segment.
2. `simulation_test_01.cpp`
   This test will validate that the member function `transition_oil` will work with a single bowling ball. The output will show a change in oil concentrations as the `transition_oil` function is repeatedly invoked.
3. `simulation_test_02.cpp`
   This test will validate that the member function `transition_oil` will work with a collection of unique bowling balls. The output will show a change in oil concentrations as the `transition_oil` function is repeatedly invoked.

4. `simulation_test_03.cpp`
   This test is the culmination of the entire assignment. Given a collection of bowling balls and a known lane condition, we can determine which bowling ball is the most likely to give us a strike.

After successfully passing those final 4 simulation tests you can finally rest easy knowing you have built a simulation engine from the ground up (and get paid for it :))