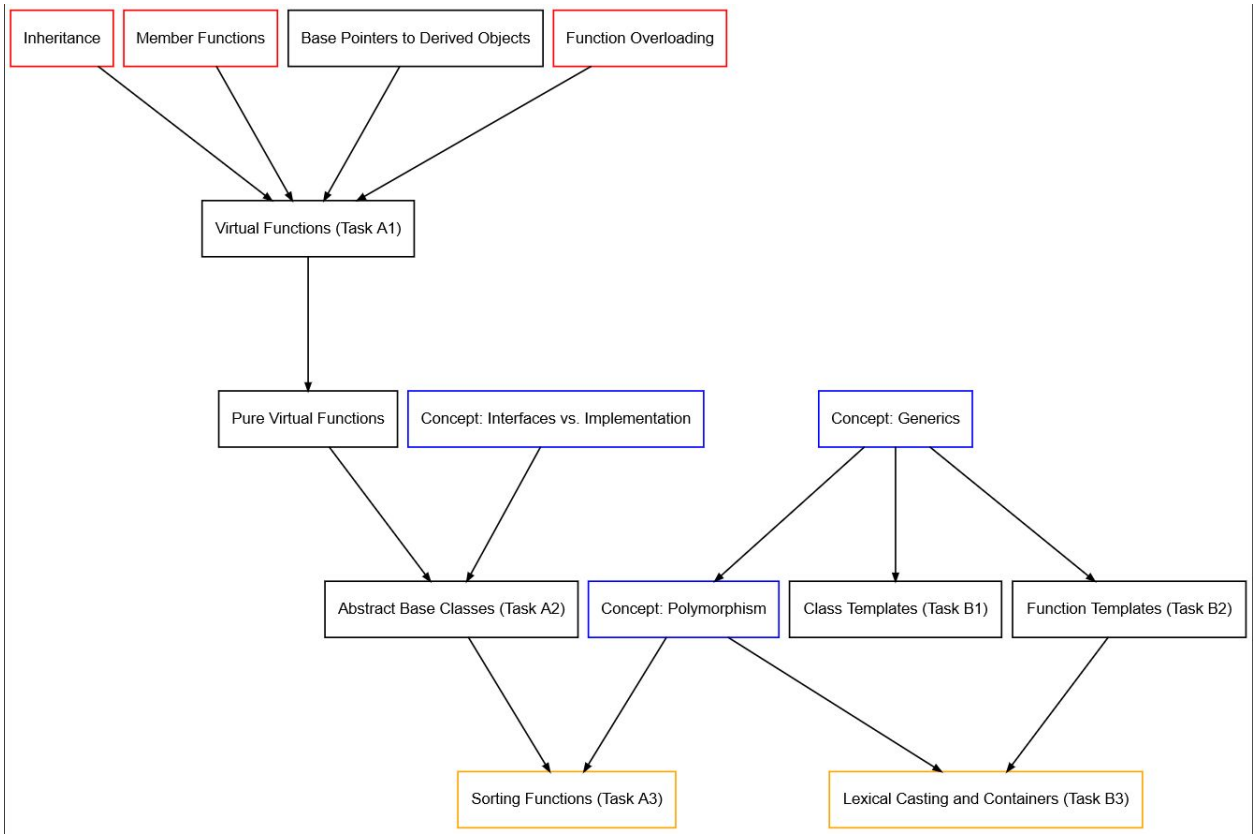
Lab 5 - Polymorphism and Templates

Goals

- In this lab, you will experience:
    - Virtual functions
    - Pure virtual functions and abstract base classes
    - Interfaces implemented using abstract base classes
    - Class templates
    - Function templates
    - Sorting algorithms (Bubble Sort)
    - Lexical casting over arbitrary types



Forewarning

- **This lab is split into 6 tasks, with an A-line and a B-line.** You can complete the first two in each line independently, but Tasks A3 and B3 require knowledge from both lines to complete.
- Each program should complete its run with no errors, with its last line printing "**SUCCESS**" once you have completed a program successfully, without any segmentation faults or failed assertions.

- You can run **make** in order to test all programs, or type in the number prefix of the source file name after make (etc.) in order to test a task (e.g. **make a1** will test **a1-virtual-functions.cpp**).

Task A1 - Virtual Functions (src/minitasks/a1-virtual-functions.cpp)

- There are three classes: Employee, TempEmployee, and Contractor.
- TempEmployee and Contractor inherit from Employee.
- The **get_type()** member function exists on all 3 classes.
- In main(), it is desired to store a container of "employees," whether they be regular Employee objects or TempEmployee or Contractor objects. One of the possible things that can be expected to be done is to query each object for what they *truly* are.
    - However, each item reports themself to be of the type "EmployeeType::EMPLOYEE" which is clearly not correct.
- Your task is to make it so that get_type() will correctly call the correct derived class member functions when using an Employee* by making all the get_type() functions into virtual functions.

Task A2 - Abstract Base Classes (src/minitasks/a2-abstract-base-classes.cpp)

- **Interfaces** are sets of behavior that a type (here, class) may implement. Implementing behaviors according to an interface is called *"implementing an interface."*
- In this task, there is an IPrintable abstract base class which wants to support one behavior (member function): **print(std::ostream&)** which presumably prints out a string representation of a class out to an **std::ostream&** object (the same type as the familiar **std::cout**).
- Employee inherits from IPrintable, and implements **print()**. However, while Executive inherits from Employee, it does not override **print()**, and thus, Executives will call **Employee.print()** when using polymorphic calls with it using an **IPrintable***.
    - In short: the problem is that Executive is being printed as an Employee because there is no overload of **print()** on Executive.
- Your task is to create an virtual function that overrides **print()** for Executive.
    - The format to print out as is like so:
        - `<job_title>: <name>\n`

Task A3 - Sorting Functions - (src/minitasks/a3-main.cpp, include/BubbleSortVector.tcc)

- In the **include** directory, two classes **BubbleSortVector** and **InsertionSortVector** inherit from **std::vector** and the **ISortable** interface (abstract class) in **include/ISortable.hpp**.
    - Each derived vector class implements the required **ISortable** behavior called **sort()**. It simply sorts the elements of the vector in ascending order.
        - Note: because **std::vector** is actually a **class template**, it must implement its member functions in a header file. So, the two classes that zinherit **std::vector** must implement **sort()** in a header file.
            - **sort()** is placed in a ".tcc" file that is included in .hpp files.

- We implemented **InsertionSortVector**, which uses insertion sort, a sorting algorithm that relies on repeatedly picking and moving the lowest-valued item in an array until the array is sorted.
- **BubbleSortVector** needs to use bubble sort.
- Your task is to implement **BubbleSortVector.sort()** such that it correctly sorts the vector.

Task B1 - Class Templates (src/minitasks/b1-class-templates.cpp)

- **Templates** are C++'s answer to "How do I create code that can work with many types?"
- **Class templates** accept types when they are used so that they can fill in "type variables" with "real" types.
    - e.g. Point2D<int> will create a struct whose x and y values are of the type **int**.
    - e.g. Point2D<float> will create a struct whose x and y values are of the type **float**.
    - Each type that is generated after what is called **template instantiation** (when you provide the missing type) is still unique.
        - So, Point2D<int> and Point2D<float> are distinct types (and ***do not use inheritance in order to achieve this***).
- The underlying float and int types for Point2D<float> and Point2D<int> are still different.
- Your task is to implement **float_to_int()**, a conversion function from Point2D<float> to Point2D<int> that performs floor() (rounds down) on its float parameters to turn them into integers.

Task B2 - Function Templates (src/minitasks/b2-function-templates.cpp)

- **Function templates** use templates to accept a variety of types that "satisfy the interfaces used within the function."
    - Informally, "if you can copy and paste the argument and its type into the function template and it compiles, it will compile."
        - Otherwise, you'll get a lot of errors related to "instantiation."
- In this task, we deal with **std::vector<T>**, **std::deque<T>**, and **std::array<T, std::size_t size>**. These are 3 different sequence container types that have the member functions:
    - **std::size_t size() const**
    - **T& operator[]()**
- We use function templates to create functions that still work across the three container types. This is a form of polymorphism.
    - **print()** prints out each item in a sequence container, and since it only uses **size()** and **operator[]**, it will theoretically work on any type or class that provides these.
- Your task is to implement **is_equal()**, a function template that will compare two sequence container types and return true if the two containers contain the same items in the same order, or false otherwise.

Task B3 - Lexical Casts and Containers (src/b3-main.cpp, include/string_conversion.hpp)

- **std::stringstream** is a class that implements most of the behavior of **std::ostream** but stores the data output to it as a **std::string**.
    - You can use this class to convert any class that overloads **operator<<** to print to get a string representation of whatever is printed using **operator<<.**
        - Converting objects to their string form is sometimes called a **lexical cast.**
- Using your knowledge from Task B2, it is also possible to have function templates take all types of **std::vector** and access their elements generically.
- Your task is to implement **lexical_cast()** as well as **vector_to_string()** that perform a lexical cast on each element of a vector, with the string forms of each element separated only by a single comma.

Submission

- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**
- In a file called **team.txt**, write your group number on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, “da Silva”), you may keep the whitespace.
- Place your source code files into a directory called **project**, using the following directory structure:
    - `project/`
        - `Makefile`
        - `src/`
            - `minitasks/`
                - `<source files for a1, a2, b1, b2>`
            - `<source files for a3, b3>`
        - `include/`
            - `<header files for a3, b3>`
        - `docs/`
            - `team.txt`
        - `bin/`
            - `NO EXECUTABLES SHOULD BE HERE UPON SUBMISSION!`
- Archive and compress your **project** directory using the command:
    - `tar czf name_of_archive_file.tar.gz project`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file under your group’s entry on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.

Supplementary Material

Polymorphism

- Duplicate code or repeated source code is considered extremely undesirable in software engineering.
    - A software principle Don't Repeat Yourself (DRY) expresses this common feeling.
- One of the causes of duplicate code is code that is repeated for similar types that requires only small changes.
    - Polymorphism or, more generally, generic programming, allows you to write code that will work across multiple types without having to create type-specific code.

Types of Polymorphism.

- Let us think in theoretical terms bigly:
    - There are 3 types of polymorphism:
        - Ad hoc polymorphism -- basically, function overloading
        - Parametric polymorphism -- basically, generics (templates in C++)
        - Subtyping -- basically, inheritance
    - Notice that traditional C doesn't support any of these, while C++ supports all 3.

Virtual Functions

- Virtual functions are C++'s answer to subtyping polymorphism. Essentially, you can treat an entire family of types as if they were all objects of the base class type while also using their unique individual derived behaviors through virtual calls.
- How does C++ do this? C++ and other language may implement this behavior by creating what is called a vtable, or a "virtual method/member function table."
    - All of the functions that implement/override a virtual function have their function pointers put into a table that contains the addresses of all the virtual functions.
        - At runtime, the program will know the type of the pointer that it wants to call a virtual function on, and use that to look up the corresponding virtual function to call. **Therefore, there is a runtime cost for virtual functions in exchange for subtyping polymorphism.**
- For many, the runtime cost of virtual functions is worth it in exchange for the reduction in duplicate code afforded by subtyping polymorphism. You may not appreciate the speed cost, but many have grown, in time, to appreciate the gain in programmer productivity and readability of code -- a dimension not often talked about in a traditional algorithms-heavy view of the world, but often talked about in a software engineering context.

Templates

- Templates are C++'s answer to parametric polymorphism, or code that can be used with any number of new types. It can also be used to generate new types with class templates.
    - Notice: functions that operator on **std::vector** will be able to work on any *type* of **std::vector<T>** as long as it compiles.
- Templates are actually not just a simple feature to copy and paste types -- it is actually in the same computability class as any normal programming language -- theoretically, it can simulate a Turing machine, which can simulate any other usual programming language. This notion of being able to simulate this very important theoretical machine model is called Turing completeness.
    - In short: templates form their own programming language (and yes, you can sometimes crash the compiler itself by taking advantage of templates.)
        - This is important because there are many powerful things you can do with templates at compile-time: see template metaprogramming. It is also sometimes considered a bane to C++ due to the sheer unreadability of template metaprogramming code.
- Note: since templates embody computation at compile-time, it can also ridiculously bloat C++ compilation times. (Templates is one of the primary reasons why C++ compilation is slower than C compilation).

So now what?

- With this lab, you complete learning the 3 (or 4) pillars of object-oriented programming:
    - **Composition** (Lab 2, Lab 4)
    - **Encapsulation** (Lab 2, Lab 3)
    - **Inheritance** (Lab 4)
    - **Polymorphism** (Lab 5)
- You will find that many of the possible programming languages you will encounter in both industry and academia will support these features:
    - **Java** and **C#** are classic object-oriented programming languages. In fact, in Java, *ALL* code must be attached to a class.
    - **Rust** is a new systems programming language focused on speed and safety from Mozilla that trades classes in exchange for traits, which are a form of interfaces. While it goes against OOP by not including classes, it still keeps 3 out of the 4 major OOP principles.
    - **Python** is a common programming language that has full support for OOP. However, its execution model and class system will prove to be much different than C++, but it is powerful (and possibly more powerful) in its own way.
- You may choose to explore other types of programming, and explore their unique views on how to express programs:
    - **Functional programming** (Haskell, Racket)
    - **Logic programming** (Prolog)

- **Concurrent programming** (learn about threads, mutexes, etc.)