UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program design: Introduction

Practical 6 Specification
Templates and Exceptions

Release Date: 03-10-2022 at 06:00

Due Date: 07-10-2022 at 23:59

Total Marks: 4100

Contents

1 General instructions

- This practical should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Read the entire assignment thoroughly before you start coding.
- **To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at http://www.ais.up.ac.za/plagiarism/index.htm.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will not be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements **Please ensure you use C++98**

2 Overview

This practical will test your ability to:

- Use exceptions in a few basic use cases
- Create generic template classes

Ensure that you are familiar with theses concepts either from the lecture slides or the textbook (preferably both).

Templates can be applied to both functions and classes in order to create a generic method/class once, and have it apply to many different types at compile time.
By specifying a type in a template parameter, the class is compiled by using that specific datatype in the way specified. While multiple template parameters can be used (more than one type can be put in a template parameter list) for this prac, you will only use single-parameter templates.

Exceptions provide opportunities to both get more information from an error, and to potentially recover from an error during runtime. In this prac, you will create a few exception classes for certain use cases and handle them appropriately.

Important: Something to note for this specification is the following with regards to UML: a rectangle at the top right of the diagram indicates that the class is a template class, and it takes the parameter in the dotted square. Also take note that functions that are *const* are not labeled as such on the UML diagrams, so read the descriptions of methods as well.

For templates: Implementation will be done in the cpp file corresponding to the header file. Since templates can't actually be compiled, you need to include the cpp file in the header file. That is, instead of including both the h and cpp files in a file where they will be used, only the h file is included and the cpp file is included by the h file. e.g say you have *TemplateClass.h* and *TemplateClass.cpp*. Inside *TemplateClass.h* you will have the line *#include "TemplateClass.cpp"*. See lecture slides for details.

3 Your Task:

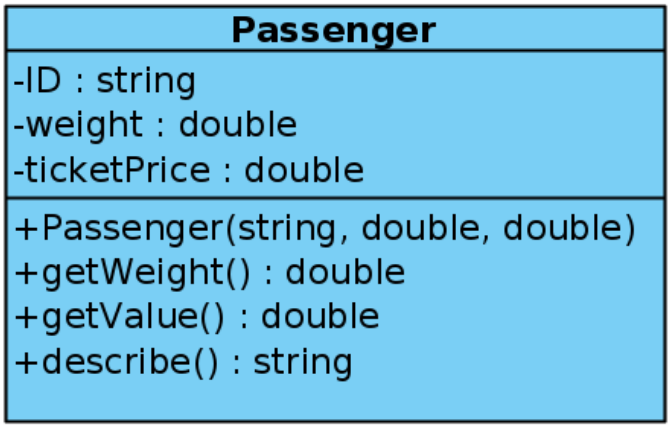
You are tasked with creating a small, basic loading system for a shipping company that uses planes to transport goods and people (as passengers). You need to create a generic plane class, which will store instances of some sort of cargo. You will also implement some operations for the plane class, and exceptions which are thrown under certain conditions. A controller class will control a plane's loading, and handle any exceptions that occur.

You have been provided with the necessary files, in which you must complete the practical.

3.1 Part 1

First you need to create the Passenger, Cargo and SecretCargo classes. These are the what will be transported by the planes.

3.1.1 Passenger



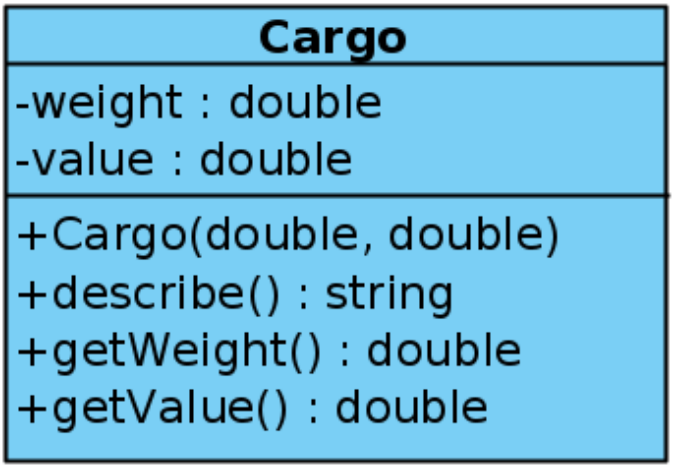
Variables:

- ID : string
  Id of the passenger
- weight : double
  The weight of the passenger
- ticketPrice : double
  The price paid for a ticket

Functions:

- **Passenger(id : string, w : double, t : double)**
  A constructor which takes in the passenger's id, weight and the price they paid for a ticket and assigns the values to member variables.
- **describe() : string**
  Returns a description of the passenger as a string. The description is in the form *id <ticketPrice> (weight)*. e.g ID0 <300> (80)
- **getWeight() const : double**
  A getter for the weight member variable
- **getValue() const : double**
  A getter for the ticketPrice member variable. The reason it is not called *getTicketPrice* will be clearer later, when you start using templates

3.1.2 Cargo



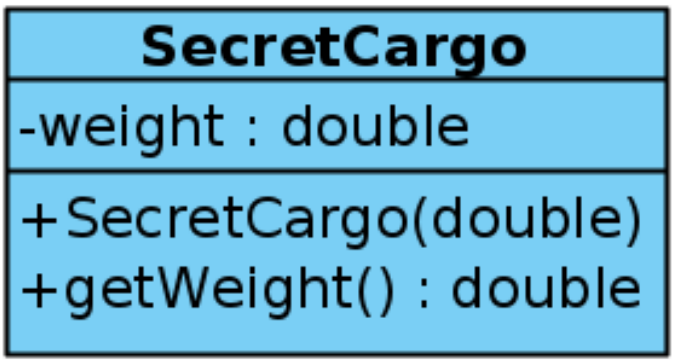
Variables:

- weight : double
- value : double

Functions:

- **Cargo(w : double, ppw : double)**A constructor which takes in the weight of the cargo and the price per weight of the cargo (working on the assumption that the price scales linearly with the weight). Calculates the value of the cargo, then assigns the appropriate values to the member variables.
- **describe() : string**
  Returns a description of the cargo as a string, in the format Generic cargo $<$ *value*$>$(*weight*). e.g Generic cargo <52874.7> (1000.0)
- **getWeight() const : double**
  A getter for the weight member variable
- **getValue() const : double**
  A getter for the value member variable

3.1.3 SecretCargo



This is a special type of cargo, with the supplier wanting as little information about it revealed as possible. For logistic purposes, its weight is still required.

Variables:

weight : double

Functions:

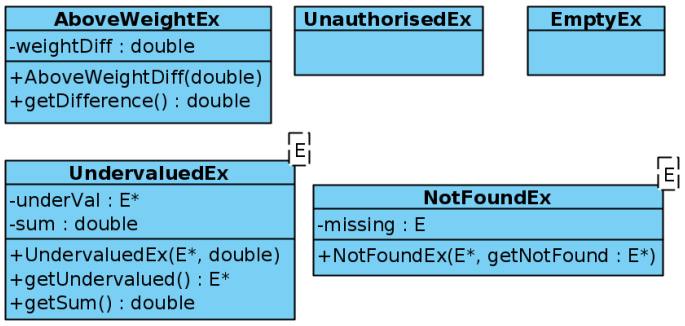
SecretCargo(w : double)

- A constructor which takes in the weight of the secret cargo.

getWeight() const : double

- A getter for the weight member variable
  The plane company wishes to make it clear that any items with the *Secret Cargo* classification should be left alone. For that reason, only its weight is accessible.

3.2 Part 2



Before moving on, you will first create a few custom exception types. For convenience, they will all be in the same files (*Exceptions.h* and *Exceptions.cpp*). The exact uses of each exception class will become clear in the next section

The exception classes are as follows:

AboveWeightEx

Variables:

- weightDiff : double

Functions:

- AboveWeightEx(wd : double)
  A constructor which takes in a single parameter, to assign to its member variable

- getDifference() const : double
  A getter for the member variable

EmptyEx

This class has no members

UnauthorisedEx
This class has no members

UndervaluedEx
This exception needs a template parameter, called E
Variables:

- underVal : E*
- sum : double

Functions:

- UndervaluedEx(uv : E*, s : double)
  A constructor which takes in two parameters, to assign to its member variables
- getUndervalued() const : E*
  Get the underVal member variable
- getSum() const : double
  Get the sum member variable

NotFoundEx
This exception also uses a template parameter, called E
Variables

- missing : E*

Functions:

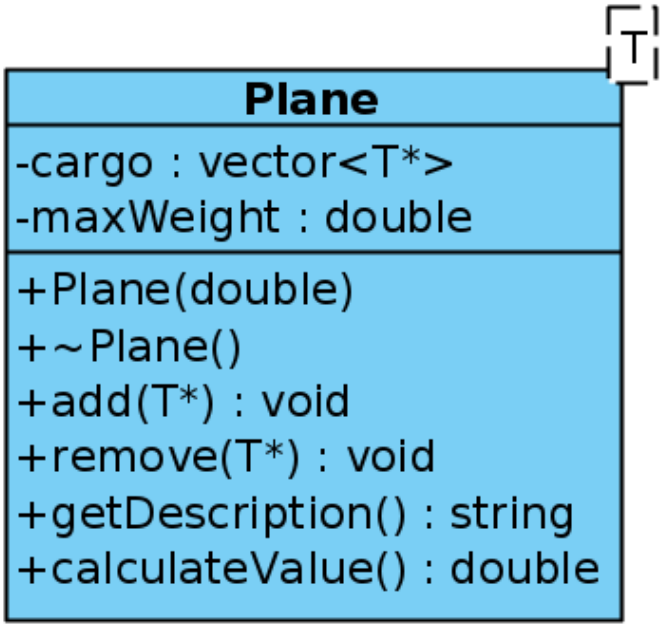
- NotFoundEx(nf : E*)
  A constructor which takes in one parameter, to assign to its member variable
- getNotFound() const : E*
  Get the *missing* member variable

3.3 Part 3

Now you will create the plane class. A plane can only carry one type of item to transport, i.e if it carries passengers, it cannot carry Cargo or SecretCargo. The plane will thus be implemented generically with a single template parameter called T.
Important: to simplify implementation, all the items to be transported by the plane will be stored in a *vector*, which is part of C++ STL. For this prac, you only need to know this:

- Create a vector with template parameters, e.g *vector<int> vec;*
- Add an item to the back of a vector using *vec.push_back(item);*
- Access an item like you would with an array, with [ ]

- To get the number of items in a vector, use the *vec.size()* method
- To remove an item at an index i, use *vec.erase(vec.begin() + i)*



Variables:

- cargo : vector<T*>
- maxWeight : double

Functions:

- **Plane(max : double)**
  A constructor which takes in the max weight that the plane can support. The vector will automatically initialise to empty (as it should be, since a plane starts of unloaded)
- **~Plane()**
  Destructor, which deletes everything in the vector *cargo*
- **add(item : T*) : void**
  Adds an item of the template type to the back of the vector. If the plane would be put over its max weight, an *AboveWeightEx* is thrown instead and the item is not added.
- **remove(item : T*) : void**
  Removes and deletes the first instance of a specified item from the vector (erase calls delete itself). Note: we define two objects to be the same if they share the same memory address, not the same characteristics. If there are no items aboard, en *EmptyEx* should be thrown. Similarly, if the item is not found, a *NotFoundEx* is thrown
- **getDescription() : string**
  ]Return a description of all items on the plane, as a string. The description of each item in the *cargo* vector is added to the description, in order, and separated by a newline (the last item is not terminated by a newline). If there are no items, an *EmptyEx* instance should be thrown.

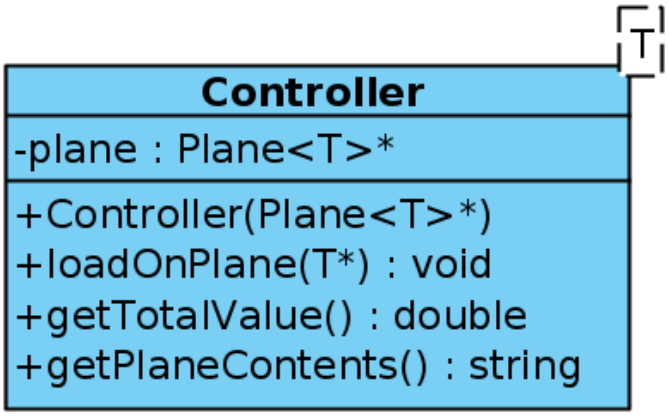
- **getDescription<SecretCargo>() : string**
  This is a specialisation of the *getDescription* function. Since no one is allowed access to any additional information about SecretCargo, and *UnauthorisedEx* is thrown whenever this method is called, unless it is empty. In that case, and *EmptyEx* is thrown.
  HINT: look at the lecture slides on an explanation of template specialisation. You should also note that for a specialised function, the header file is left unchanged....

- **calculateValue() : double**
  Calculate the total value of all the items on the plane. If there are no items on the plane, an instance of *EmptyEx* should be thrown.
  In addition, the plane company has decided on the following to save on fuel: if an item with the smallest value is found to have a value less than half the value of the next lest valuable item, it should be noted in the form of an exception. The *UndervaluedEx* exception should be thrown, with the least valuable item and the sum of all values (excluding the least valuable item) as parameters. This way, when the exception is handled, the handler will know both the least valuable item and the sum of all values up until that point. If there are no items, an *EmptyEx* instance should be thrown.

- **calculateValue<SecretCargo>() : double**
  Another specialisation. This one also throws *UnauthorisedEx* (if not empty.)

3.4 Part 4

Now that there is a system in place to create a plane that carries a certain type of item, a plane Controller class needs to be created to handle plane operations. This will act essentially as a 'layer above' the plane, to handle any exceptions thrown. This way, certain exceptions can be handled differently if the plane company changes its mind about certain things down the line. Since the controller needs to know which type is used in the plane template for its member variables, it too is a template class.



Variables:

- plane : Plane<T>*
  A pointer to an instance of Plane<T>

Functions:

- Controller(p : Plane<T>*)
  Instead of creating a plane itself, a controller is bound to a specific plane through the constructor

- loadOnPlane(t : T*) : string
  Load an item onto the plane, and return "Success". If an item puts the plane over max weight, it is simply not loaded, and an error message is returned in the form "Could not load item, above max weight by *amount-by-which-over-max-weight*"

- getTotalValue() : double
  Get the total value aboard the plane. After some debate, the plane company decided that if the controller is not authorised, the value is simply reported as 0 (remember that an exception is thrown if the controller is not authorised. By catching it, you can determine if you are authorised to view that information). If an *UndervaluedEx* is thrown by the plane, then the item with the smallest value should be removed from the plane. This is only done once (the total value is not checked afterwards). The value returned is the total value on the plane excluding the item removed. If the plane is empty, 0 is returned.

- getPlaneContents() : string
  This returns the entirety of the planes' contents (using its *getDescription* method). If the plane is empty, then "Nothing to report" is returned. If the controller is not authorised to view the contents of the plane, "Not permitted" is returned.

Remember to test your code well. Feel free to use and extend the given main.cpp file.

Templates can be used in many more advanced ways, including specifying inheritance. In general any number of template parameters can be added, and any part of a class where a certain class or type is used can be customised to be a template parameter. The only real requirement is that within the generic template class, the types are used in a valid way. For instance, you won't be able to make a plane that can store *int*, unless you create a specialisation that only uses an *int* value in valid ways.

Also the reason the prac is out of 4100 marks, is due to Fithfork not being able to assign fractional marks. So, instead of a mark being 0.25, it is now 25 etc.

4 Uploading

Upload the following files in an archive:

- Cargo.h , Cargo.cpp
- SecretCargo.h, SecretCargo.cpp
- Passenger.h, Passenger.cpp

- Plane.h, Plane.cpp
- Controller.h, Controller.cpp
- Exceptions.h, Exceptions.cpp