1.1 Lambda Functions

1. (First Encounters with Lambda Functions)

The objective of this exercise is to get you acquainted with lambda function syntax by examining some simple examples. Lambda functions are very important and useful and we shall be using them in other examples during the course.

In this exercise we concentrate on defining lambda functions that process arrays of double precision numbers (or integers) in various ways. In general, you need to think about a lambda's *signature*, that is its return type and input arguments.

Answer the following questions:

a) Create a lambda function that multiplies each element of an array by a number. This number is a *captured variable*; test the function for the cases when it is copy-by-value and a reference. The original array is modified after having called the function.

b) Print the values of the modified array using `auto` to initialise the iterator variable instead of declaring it explicitly.

c) Write a lambda function to compute both the minimum and maximum of the elements in an array. The return type is an `std::pair` (or if you prefer, `std::tuple` with two elements) containing the computed minimum and maximum values in the array.

d) Compare the approach taken in part c) by calling `std::minmax_element`. For example, do you get the same output? How easy is to understand and reuse the code?

2. (Comparing Lambda Functions with Function Objects and Free Functions)

The objective of this exercise is to examine the application of lambda functions in C++ code. In this case we test some STL algorithms. In particular, we examine `std::accumulate` (by the way, it has two forms) that is a special *fold function* that iterates over the elements of an array and accumulates them in some way to produce scalar output. Consider the code to accumulate the elements of an array in some way. In other words it is a scalar-valued function that maps a vector to a scalar value.
The start code uses three functions, two of which are user-defined and the other is a predefined STL function object. The first two functions are:

```
// N.B. Generic lambda
auto MyMultiply = [] (auto x, auto y) { return x*y ;};

struct FOMultiply
{
    template <typename T>
            T operator () (const T& x, const T& y)
{ return x * y; }
};
```

and the algorithm is:

```
std::vector<int> vec { 1,2,3,4,5 };
int acc2 = std::accumulate(vec.begin(), vec.end(), initVal,
                        std::multiplies<int>());

int accA = accumulate(vec.begin(), vec.end(), initVal,
                        FOMultiply());
```

Answer the following questions:

a) Implement and test the algorithm using the *generic lambda*:

```
auto MyMultiply = [] (auto x, auto y) { return x*y ;};
```

Can the algorithm also be used with complex numbers, for example? Can we use function objects to multiply the elements of an array of complex numbers?

```
template <typename T>
    T MyMultiply2(const T& x, const T& y) { return x*y; };


// Using complex numbers
using Complex = std::complex<double>;
std::vector<Complex> complexArray{Complex(1.0, 1.0), Complex(2.0, 2.0) };
Complex initCVal(1.0, 1.0);
// auto acc6C = accumulate(complexArray.begin(), complexArray.end(),
// initCVal, FOMultiply());
Complex acc6C = accumulate(complexArray.begin(), complexArray.end(),
                                     initCVal, MyMultiply2<Complex>);
std::cout << "Sum 6, complex numbers: " << acc6C << std::endl;
```

Experiment with this kind of code in order to reach your conclusions.

b) Implement the accumulation algorithm using an embedded lambda function in combination with `std::for_each()` and captured variables.

c) Give an example of a stored lambda function that may be called from an STL algorithm (may be any relevant STL algorithm). Demonstrate using the STL algorithm with your stored lambda.

3. (Review/Reflection for Yourself – no Submission required)

Consider the code that you have written and consider the cases in which the use of lambda functions lead to more understandable, maintainable code than with function objects. For example, are lambda functions suitable for small chunks of code and system configuration? Some other tips:

- More general function signatures.
- Other data types.
- Lambda versus free functions versus function object.

1.2 New C++ Language Features ("Improving Your Classes")

Summary

The objective of the exercises in this *pivotal* section is to improve the robustness, predictability and reliability of individual C++ classes. In particular, we wish to address some issues in C++98:

- How to avoid the compiler implicitly generating *special member functions*, that is the default constructor, destructor, copy constructor and copy assignment. These functions are generated only if needed.
- Preventing generation of special functions by the compiler; preventing calls to these functions.
- Creating move constructors and move assignment operators.
- Explicit constructors.
- Constructors and `const` expressions.
- The `noexcept` keyword.

Not all topics have been discussed yet, so you may need to do some research (www.cppreference.com). The objective is to introduce these new features into your code, test the code and make sure you are happy with the results.

First, you need to create a class (call it C) with an instance of `std::vector` as data member. Create public default constructor, copy constructor, assignment operator and destructor. Test the code. Place print statements in the bodies of each member function to show that it has been called. Furthermore, create some member functions (for example a *`print()`* function or a modifier function to `scale(double)` the values in C's data member) that you know will not throw an exception.

This class will now be used as input in this exercise. You should spend some time on these exercises until you understand the bespoke core features.

1. (Basic Improvements in Classes)

a) Modify class C so that 1) its default constructor is absent and 2) copy constructor and assignment are private. To this end, use keyword `default` to explicitly tell the compiler to generate a default constructor. Furthermore, use the keyword *delete* to mark the copy constructor and assignment operator as *deleted functions*. Deleted functions may not be used in any way, even by friends. Test your code again, including calling the defaulted and deleted functions. What is the resulting behavior?
b) Use the `explicit` keyword in the constructors to disallow implicit type conversion.
c) Use `constexpr` keyword for those functions in which input arguments are known at compile-time (for example, constructors and setters). Then the data members will also be known at compile-time.
d) Use the keyword `noexcept` for those member functions which you know will not throw an exception.

Run and test your code.

2. (Move Semantics 101)

The objective of this exercise is to get you used to *move semantics* and *rvalue* references, a warming-up session if you like. Answer the following questions:

a) Create a string and move it to another string. Check the contents of the source and target strings before and after the move.
b) Create a vector and move it to another vector. Check the contents of the source and target vectors before and after the move. Compare the time it takes to move a vector compared to a copy constructor or a copy assignment statement.

c) Consider the following user-defined code to swap two objects:

```
template < typename T >
    void SwapCopyStyle(T& a, T& b)
{ // Swap a and b in copying way; temporary object needed

    T tmp(a);     // Copy constructor
    a = b;        // Copy all elements from b to a
    b = tmp;      // Copy all elements from tmp to b
}
```

How many temporary copies are created? Now create a function based on move semantics to swap two objects. Compare the relative performance of the two swap functions by timing the swap of two very large vectors. Use `std::chrono`.

3. (Improvements in Classes, Part II)

We now return to class C in exercise 1. Answer the following questions:

a) Create the move constructor and the move assignment operator for class C.
b) Test these new functions. How can you ensure that a move constructor is called instead of a copy constructor?
c) What happens if you use an *lvalue* as the source of a move operation?

1.3 Advanced Language Features

1. (Fundamentals of Variadics)

In general terms, *variadics* in C++11 is concerned with defining functions having a variable number of template parameters. The syntax can be confusing but it can be made more understandable when we realise that it is similar to how recursion in computer science works. In particular, a function of *arity n* processes the first argument in its argument list and then it calls itself as a function of *arity n-1*. The chain of function call terminates when the arity becomes one and then the function returns.
The objective of this exercise is to create a variadic print function for objects that support overloading of the standard ouput stream operator <<.

Answer the following questions:
a) Create the template variadic function with its *parameter pack*.
b) Create the *termination/tail function*, that is the print function accepting a single input argument.
c) Test the function. A typical example is:

```
// Variadic function templates
int j = 1; double d = 2.0;
print(j); print(d); print(j, d); print(d, j);
std::cout << "\n3 params \n";
print(d, j, std::bitset<8>(233));
```

2. (Variadics and Polymorphic Behaviour)

In this exercise we apply variadic functions to allow us to call polymorphic and non-polymorphic functions on a fixed set of objects. This avoids us having to create a composite or aggregate object and then call one of its polymorphic functions. To this end, consider the class hierarchy:

```
class Shape
{
public:
    virtual void rotate(double d) = 0;
};

class Circle : public Shape
{
public:
    void rotate(double d) override { std::cout << "c\n"; }
};

class Line : public Shape
{
public:

    void rotate(double d) override { std::cout << "l\n"; }
};
```

and an unrelated class that also has a `rotate` member function:

```
class Valve
{
public:
    void rotate(double d) { std::cout << "v\n"; }
};
```

Answer the following questions:

a) In some situations we wish to apply a rotation to a finite but otherwise unknown number of objects but we do not want to have to create a special function for each possible combination of the number of arguments to the new free function `rotate()`. This would lead to combinatorial explosion and instead we use a variadic function which is created once and used many times. There is no code duplication. An example of use is:

```
Circle circle;
Line line;
Valve valve;

double factor = 2.0;

// We can in essence create compile-time aggregates/whole
// part objects on the fly. We only need to write one function.
rotate(factor, circle);
rotate(factor, circle, line);
rotate(factor, circle, valve);
rotate(factor*2, circle, valve, line);
```

Create this function.

b) Based on the result in part a) create a rotation function that works with any object whose class implements the member function `rotate()`. Test your code.

c) Implement the alternative solution to this problem:

```
// The other way of doing it.
std::array<Shape*, 5>
    shapeList{ &circle, &line, &circle, &circle, &line };
double factor2 = 20.0;
for (auto& elem : shapeList)
{
    elem->rotate(factor2);
}
```

Why can we not include valves in this array? And why can we include valves in the variadic approach?

1.4 Some Data Structures

Summary

The exercises in this section introduce *universal function wrappers* (`std::function`) and Bind (`std::bind`) in C+11. We start with relatively simple examples to get used to the syntax and we then move to design-level cases. We will be using these types in later sections.

The prototypical function type is one that accepts a scalar as input argument and that has a scalar as return type. The type is generic in order to allow specialisations for various built-in data types:

```
template <typename T>
    using FunctionType = std::function<T (const T& t)>;
```

In the following exercises we shall use double as data type for convenience.

1. (Function Wrappers' Fundamentals 101)

In this exercise we examine how to use function wrappers with a range of C++ function types as target methods.

Answer the following questions:

a) Create an instance of a function wrapper:

```
FunctionType<double> f;
```

and a function to print the value of `f` (after it has been assigned to a *target method*) for a given input argument:

```
template <typename T>
    void print(const FunctionType<T>& f, T t)
{
    std::cout << f(t) << '\n';
}
```

b) We now wish to assign the newly created function wrapper to each of the following functions having compatible signatures:
   - A free function.
   - A function object.
   - A stored lambda function.

Create the corresponding code for these functions and test your code.

2. (Function Wrapper and partial Function Evaluation 101)

Here we show how to reduce functions with a given arity to functions with a lower *arity*.
In this case we take a simple example to get you used to `std::bind` and *placeholders* in conjunction with the test function:

```
double freeFunction3(double x, double y, double z)
{
    return x+y+z;
}
```

Answer the following questions:

a) Define functions of arity 3, 2,1 and 0 based on the above function.
b) Test these functions by *binding* the appropriate variables and checking that the output is correct.

3. (Function Wrapper and Bind)

In this exercise we modify function arguments so that function signature conforms to `FunctionType`. In this way we resolve the mismatches between functions that we have created and that have the wrong signature for the current problem. To this end, consider the class:

```
class C
{ // Function object with extra member functions

    private:
            double _data;
    public:
            C(double data) : _data(data) {}

    double operator () (double factor)
    {
            return _data + factor;
    }

    double translate (double factor)
    {
            return _data + factor;
    }

    double translate2 (double factor1, double factor2)
    {
            return _data + factor1 + factor2;
    }

    static double Square(double x)
    {
            return x*x;
    }
};
```

Answer the following questions:

a) Bind the function wrapper to C's static member function.
b) Bind the function wrapper to C's member functions using `std::bind` and placeholders.
c) Test the function.

4. (Function Wrapper and Inheritance)

We modify a class hierarchy containing polymorphic functions to a class hierarchy that uses a function wrapper. Thus, instead of a public pure `virtual` function we define a protected function wrapper in the base class which will be implemented in derived classes. Take the following starting point of a traditional C++ class hierarchy:

```
// Class hierarchy
class Shape
{
public:
    virtual void rotate(double d) = 0;
};

class Circle : public Shape
{
public:

    void rotate(double d) override { std::cout << "c\n"; }
};

using VoidFunctionType = std::function<void (double)>;
```

The objective of this exercise is to achieve the same (or better) level of flexibility of the polymorphic function *void rotate (double d).* To this end, we 'emulate' pure virtual functions by replacing them with an instance of VoidFunctionType (call it f) as a protected data member in the base class.

The steps are:

- Create a constructor in the base and derived classes to initialize f with a target method, having a compatible signature with *void rotate (double d).*
- Modify *void rotate (double d)* in the base class so that it calls ***f***.
- The derived classes may override void rotate (double d).
- Create a test program with various examples of ***f***.

(Remark: the examples in this exercise will be extended into a larger *next-generation design pattern* that we discuss in Module 5.)

5. (Function Wrapper and Composition)

We modify a class hierarchy containing polymorphic functions to a *single class* that uses a function wrapper. The approach is similar to that taken in exercise 4, except that we now have a single class C that is composed of a function wrapper. There are no derived classes anymore, just instances of C.

This design is a *next-generation design pattern* (to be described in more detail in Level 5) in the sense that we do not need to create a class hierarchy in combination with (pure) virtual functions, in order to create flexible software. In this particular case, we have a single class with one or more function wrappers as members (composition again). They must be initialised in the class constructor, but they can be assigned to another target method (function).

We take an example of a class, with what we call a 'customisable algorithm'. The class interface is:

```
template <typename T>
class NextGenPolymorphism
{
 private:
     FunctionType<T> _func;
     T _fac;

 public:
     NextGenPolymorphism (const FunctionType<T>& function,
                          const T& factor) : _fac(factor)
     { _func = function; }

     T execute(double d) { return _func(d) * _fac; }
     void switchFunction(const FunctionType<T>& function)
     { _func = function; }
};
```

Here is a test case to show how to use the new design idea:

```
// Next gen stuff
auto square = [](double d) {return d*d; };
auto expo = [](double d) {return std::exp(d); };
NextGenPolymorphism<double> ng(square, 1.0);
std::cout << "Square: " << ng.execute(2.0) << '\n'; // 4
// square is not cool, switch to expo
ng.switchFunction(expo);
std::cout << "Exponential: " << ng.execute(5.0); // 148.413
```

This approach is not necessarily a competitor to OOP polymorphism, although it could be under certain circumstances.

Answer the following questions:

a) The goal of the exercise is to create a single `Shape` with an embedded command-style function wrapper (such as `rotate()` or other ones).

b) Test the new code.

1.5 Tuple/Span

Summary

The exercises in this section are concerned with the C++ library `std::tuple`. Some C++ developers may not be familiar with this library and we show how it can be used as a better solution to problems that have been resolved differently in the past (for example, using structs to group related data). We also show how to use tuples as part the GOF *Builder design pattern* that we shall meet again in later sections on design and applications.

1. (Tuple/Span 101)
   https://en.cppreference.com/w/cpp/utility/tuple
   https://en.cppreference.com/w/cpp/container/span

In this exercise we create code that creates and manipulates tuples and we endeavor to emulate similar functionality when working with databases.

Answer the following questions:

a) Create a tuple that models some attributes of a *Person* consisting of a name, address (both `std::string`) and date of birth (as a Boost `date`). Create some instances of Person and modify their elements.

b) Create a function to print the elements of *Person*.

c) Create a list/vector of *Person* and add some instances of *Person* to it.

d) Write a function to sort the list based on one of the elements (for example, we can sort by name, address or date of birth).

e) Create fixed-sized and variable-sized arrays and create span views of them. Are these views read-only? Prove or disprove this question.

f) Create 3 print functions for spans using a) range-based for loops, b) iterators and c) indexing operator `[]`.

g) Write functions to return the first and last N elements of a span.

h) Write a function to test `std::subspan`.

i) Investigate and extend the following code to create "byte views of spans":

```
// Bytes stuff
   float data[1]{ 3.141592f };
   auto const const_bytes = std::as_bytes(std::span{ data });

   for (auto const b : const_bytes)
   {
         std::cout << std::setw(2)
              << std::hex
              << std::uppercase
              << std::setfill('0')
              << std::to_integer<int>(b) << ' ';
   }

   // Exx. test
   //std::as_writable_bytes
```

This material is intended for students of the "Advanced C++11/C++14 and Multidisciplinary Applications" certificate

2. (Application I, Numeric Algorithms with Tuples)

In this exercise we develop a number of functions to aggregate the elements of variadic tuples whose underlying types are numeric. It is necessary that you know some of the technical details of `std::tuple`.

Answer the following questions:

a) Create a template class with static member functions to compute the maximum, sum and average of the elements in the tuple. A typical class is:

```
template <typename T, typename Tuple, std::size_t N>
     struct Calculator
{

     // TBD
};
```

b) Test the code on tuples with two and three elements whose underlying type is `double`.

c) Compute the sum and average of a tuple whose element type is `std::complex<int>`.

3. (Application II, Tuples as Return Types of Functions)

In many applications we may wish to create functions that return multiple values and we decide to encapsulate these values in tuples. Some examples are:

- A function that returns a value and a status code.
- The STL algorithm `std::minmax_element` that returns a `std::pair` instance.
- A tuple representing and IP address and a communication endpoint.

In this exercise we create functions to compute computed numerical properties of instances of `std::vector` and the STL-compatible `boost::numeric::ublas::vector`.

Answer the following questions:

a) Create a function that returns the following *statistical properties* of a numeric vector as a tuple: mean, mean deviation, range (the difference between the largest and smallest numbers in the vector), standard deviation and variance.

b) Test the function from part a). In order to make the code more readable you can use *tied variables* and `std::ignore`.

c) Write a function that computes the *median* of a sorted vector (the median is either the middle value or the arithmetic mean of the two middle values) and the *mode* which is the value that occurs with the greatest frequency.

   When there are two or more values that are candidates to be the mode, then the smallest value will be chosen. For example, the time between eruptions of Old Faithful geyser (in minutes, to nearest minute) on 1 August 1978 were: {78,74,68,76,80,84,50, 93,55,76,58,74,75}. Both values 74 and 76 are candidates for mode, so we choose the smaller value, namely 74.

   (Remark: the following are outside the scope of this exercise: The mode's associated frequency (for example, both 74 and 76 have frequency two), and the arrays candidate's modes and possibly their associated frequencies).

4. (Adapters for STL algorithms)

We need to create special functionality when working with vectors in numerical analysis, for example:

- Given a sorted numeric vector v, find the first index i such that **v[i] <= x < v[i + 1]** for a given target value x.
- Find the maximum error between two vectors v1 and v2 in a given index range [i; j].

To this end, we use the functionality in STL (in particular, `std::find` and `std::distance`). However, there is an *interface mismatch* between what we want (integral indexes) while STL works with iterators. We morph the STL algorithms by creating *adapters*.

Answer the following questions:

a) Given a sorted numeric vector v and a target value x, find the first index i such that **v[i] <= x < v[i + 1]**. Determine the return type.

b) Find the maximum error between two vectors $v1$ and $v2$ in a given index range $[i;j]$. We wish to compute the difference in some (customisable) norm, specifically the absolute error, relative error and the index values where these errors occur.

Ideally, you should use the following C++ functionality if possible:

- Move semantics instead of copy constructors.
- Smart pointers (if applicable).
- Function objects and lambda functions.
- Use as much of STL Algorithms as you can (investigate what's in the library).

5. (A simple next-Generation *Builder* Pattern)

In this exercise we consider how to create and package objects that can be used when configuring an application. In the interest of *separation of concerns* (and *Single Responsibility Principle*) we deploy a special GOF factory pattern called the *Builder* to create the application objects in a step-by-step fashion. The added value in the current case is that the objects will be contained in a tuple that clients can use at will. This is a major improvement on the traditional *Builder* pattern that does not provide a clear entry point to the objects that clients need.
The problem description is as follows: We take the case of a two-dimensional *Shape* CAD C++ hierarchy. It must be possible to present these graphics objects in various media (base class *IODevice*) and the coupling between CAD objects and their IO driver should be as loose as possible. The relevant interfaces are:

```
class IODevice
{ // Interface for displaying CAD objects

public:
    virtual void operator << (const Circle& c) = 0;
    virtual void operator << (const Line& c) = 0;
};
```

and

```
// Class hierarchy
class Shape
{
public:
    virtual void display(IODevice& ioDevice) const = 0;
};
```

Answer the following questions:

a) Create derived classes `Circle` and `Line`. Create I/O device classes to display CAD shapes in different ways.
b) We wish to configure different shapes to work with various I/O devices. To this end, we create the well-known interface for the *Builder* pattern but in this case the newly created objects are elements of a tuple:

```
using ShapePointer = std::shared_ptr<Shape>;
using IODevicePointer = std::shared_ptr<IODevice>;

class Builder
{ // A Builder hierarchy that builds shapes and io devices

public:
    std::tuple<ShapePointer, IODevicePointer> getProduct()
    { // GOF (!) Template Method pattern

     // TBD

    }

    // Hook functions that derived classes must implement
    virtual ShapePointer getShape() = 0;
    virtual IODevicePointer getIODevice() = 0;
};
```

Create and test a number of builders and apply them in the current context.

Remark
We shall extend this example and concept to larger designs in later sections.

1.6 Other C++ Features

Summary

The exercises in this section deal with some *remaining* syntactical features in C++11. They are probably not the most critical things to know but we include them for completeness.

1. (Using Sealed Classes and Sealed Member Functions)

Below is C++ code that does not compile for a number of reasons:

```
struct Base
{
    virtual void draw() { std::cout << "print a base\n"; }
    void print() {}
    ~Base() { std::cout << "bye base\n"; }
};

struct Derived final : public Base
{
    Derived() {}
    void draw() override
           { std::cout << "print a derived\n"; }
    void draw() const override {}
    void print() override {}
    ~Derived() { std::cout << "bye derived\n"; }
};

class Derived2 : public Derived
{

};
```

Answer the following questions:

a) Fix this code without removing any of the (perceived) functionality.
b) Once fixed, create instances of `Derived` using `Base` smart pointers. Check that you get the expected output.

2. (Implementing the GOF *Template Method Pattern == TMP*)

In software engineering, the template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure. This use of "template" is unrelated to C++ templates. (Source: Wikipedia).

The objective of this exercise is to implement TMP in C++11 using a combination of sealed/final member functions and universal function wrappers. The problem description is: design an algorithm to multiply each of the elements of a vector by a given value (from an abstract input source) and then send the modified vector to some abstract output device. In short, the *algorithm* consists of the following steps:

- *Preprocessor*: get the value that multiplies the elements of the vector.
- *Computation*: use `std::transform` and a lambda function to perform the steps in the algorithm.
- *Postprocessor*: send the data to the output device.

The problem is similar to data flow. The I/O interfaces are defined as:

```
using InputFunction = std::function<double ()>;
using OutputFunction = std::function<void (const std::vector<double>&)>;
```

Answer the following:

a) Create a single class `TMPClass` consisting of a vector, input device and output device. Build the code for the algorithm.
b) Instantiate `TMPClass` by the constructor:

```
TMPClass(std::size_t n, double startValue,
        InputFunction iFunction, OutputFunction oFunction);
```

The input device produces a hard-coded value while you should implement the output device by a range-based loop applied to the modified vector.

3. (*auto* Deduction From Braced Init-List)

The different ways to initialise objects and data are:

- *Default initialisation*: object is constructed without an initializer.
- *Value initialisation*: object is constructed with an empty initializer.
- *Direct initialisation*: object is initialised from an explicit set of constructor arguments.
- *Copy initialisation*: object is initialised from another object.
- *List initialisation*: object is initialised from braced-init-list.
- *Aggregate initialisation*: initialise an aggregate from a braced-init-list.
- *Reference initialisation*: bind a reference to an object.

Answer the following questions:

1) Give a code example of use of each of the above cases. Take a user-defined class as test. Pay attention as to whether default, copy and move constructors are needed for your code to work.
2) Can you identify ill-formed cases?
3) Apply aggregate initialisation to the following class:

```
struct S
{
            int x;
            struct Foo
{
                   int i; int j; int a[3];
            } b;
};
```

How many different ways can you think of to initialise this class?

4. (Alias Template and its Advantages Compared to `typedef`)

The `typedef` is essentially deprecated. It does not work with templates (at least not directly) and we need to do a lot of contortions when working with templatised classes and structs. To this end, we create a class that is composed of a container:

```
template <typename T>
    class Client
{ // An example of Composition
private:
    typename Storage<T>::type data; // typename mandatory
public:
    Client(int n, const T& val) : data(n, val) {}

    void print() const
    {
           std::for_each(data.begin(), data.end(), [](const T& n)
                        { std::cout << n << ","; });
```

```
                std::cout << '\n';
            }
    };
```

where the storage ADT is:

```
    // C++ 98 approach
    // Data storage types
    template <typename T> struct Storage
    {
        // Possible ADTs and their memory allocators
        // typedef std::vector<T, CustomAllocator<T>> type;
        // typedef std::vector<T, std::allocator<T>> type;

        typedef std::list<T, std::allocator<T>> type;
    };
```

An example of use is:

```
    // Client of storage using typedef (C++ 98)
    int n = 10; int val = 2;
    Client<int> myClient(n, val); myClient.print();
```

Answer the following questions:

a) (re)Create the class using alias template instead of `typedef`.

b) Test your code. Do you get the same output as before?