Below is the actual content of the Summer 2019 final exam.

Keep in mind that the actual questions on your exam WILL BE DIFFERENT. Use this document to read on areas of interest, self-study key topics, and discuss with your peers what reasonable solutions to these problems might be. You are expected to study techniques, not the answers.

Take note that this document is **confidential**, and it should not be shared or used beyond our course.

Notes: This is a closed book exam worth 25% of the final grade. Do not collaborate or copy other people's work. Please write legibly – the graders can give marks only for correct, clear answers they are able to read. During the exam no queries about the exam paper will be entertained; if you are not clear about a question, or found a mistake in the code, write down your assumption clearly on the side; it will be taken into consideration during marking.

Unless explicitly stated, you are:

- Required to use C++11 or newer.
- Required to use STL templates and algorithms where possible.
- Required to ensure `const`-correctness of parameters where possible.
- Not allowed to use explicit loops (`for`, `while`, `do while`).
- Not allowed to use `auto` if a type is known upfront.

1. Indicate the truthfulness of the following statements

Given the statements below, indicate whether each of them is true ("**T**") or false ("**F**"):
(1 mark each; 10 marks total)

a)	A class marked with the `final` specifier can be inherited from, but none of its member functions can be overridden.	
b)	When using a *covariant return type* feature, an overriding member function can return a type different from the return type of an overridden member function.	
c)	Following the OCP (the "O" in S.O.L.I.D.), we should strive to create code open for modification, but closed from extension.	
d)	`reinterpret_cast` is an expression that generally does not produce new instructions, but assumes a different interpretation of the underlying bit pattern.	
e)	Locking a weak pointer `std::weak_ptr` can successfully produce a non-null shared pointer, even when it is the last smart pointer to a given object.	
f)	A recursive tuple implementation (as discussed in the classroom) stores data members in the order reverse to their declaration as template parameters.	
g)	If a function's return type must be deduced from the function's parameters, the function should use `auto` return type or a trailing return type.	
h)	As containers (including `std::deque` or `std::tuple`) are unable to store references to objects, you can store these as an `std::ref` type.	
i)	An r-value reference object that is given a name can be bound to an r-value object, but itself behaves as an l-value object.	
j)	In the type erasure pattern, the `Concept` interface class should implement at least one virtual function (other than a virtual destructor) to be useful.	

2. Define required member functions

Given a partial definition of a class template `my_vector` that encapsulates a dynamic array of elements of type `T`, provide the definitions of required member functions. *(10 marks total)*

```
1  template<typename T>
2  class my_vector
3  {
4      size_t _count;
5      T* _data;
6  public:
7      //...
8  };
```

a) Define the copy constructor outside of the class definition. *(2 marks)*

b) Define the move constructor outside of the class definition. *(2 marks)*

c) Define the copy assignment operator outside of the class definition; make sure it offers a **basic exception safety guarantee** without using `try`/`catch`. *(3 marks)*

d) Define the move assignment operator outside of the class definition. *(2 marks)*

e) Define the destructor outside of the class definition. *(1 mark)*

3. Write the output of functions

Given the following definitions, identify the **output** printed out by each function call provided below. Write **NC** if the statement does not compile.

```
 1  struct B {
 2    virtual void f1()
 3    { std::cout << "B_1"; }
 4    void f2()
 5    { std::cout << "B_2"; }
 6  };
 7  struct D1 : public B {
 8    void f1(double i = 4.0)
 9    { std::cout << "D1_1" << i; }
10    void f2(int x = 5)
11    { std::cout << "D1_2" << x; }
12    virtual void f3()
13    { std::cout << "D1_3"; }
14  };
```

```
15  struct D2 : public D1 {
16    void f1()
17    { std::cout << "D2_1"; }
18    void f2(float)
19    { std::cout << "D2_2"; }
20    void f3(short s = 1)
21    { std::cout << "D2_3" << s; }
22  };
23  B b; D1 d1; D2 d2;
24  B* bp_b = &b;
25  B* bp_d1 = &d1;
26  B* bp_d2 = &d2;
27  D1* d1p_d1 = &d1;
28  D2* d1p_d2 = &d2;
```

Answers: *(1 mark each; 10 marks total)*

a) `bp_b->f1();` _____ b) `bp_b->f2();` _____

c) `bp_d1->f1();` _____ d) `bp_d1->f2();` _____

e) `bp_d1->f3();` _____ f) `bp_d2->f1();` _____

g) `bp_d2->f2(1.0f);` _____ h) `d1p_d1->f1();` _____

i) `d1p_d1->f1(2.0f);` _____ j) `d1p_d2->f3();` _____

4. Analyse the function template calls

Given the following function templates, identify the **output** printed out by each function call provided below. Write **NC** if the statement does not compile.

```
 1  template <typename T1, typename T2>
 2    void f(T1&&, T2&&) { std::cout << "1" << std::endl; }
 3  template <typename T>
 4    void f(T&&) { std::cout << "2" << std::endl; }
 5  template <typename T>
 6    void f(T&&, T&&) { std::cout << "3" << std::endl; }
 7  template <>
 8    void f<float&&>(float&&) { std::cout << "4" << std::endl; }
 9  template <>
10    void f<float&>(float&) { std::cout << "5" << std::endl; }
11  template <>
12    void f<float&, float>(float&, float&&)
13    { std::cout << "6" << std::endl; }
14  template <>
15    void f<float, float&>(float&&, float&)
16    { std::cout << "7" << std::endl; }
17  float x;
18  int y;
```

Answers: *(1 mark each; 10 marks total)*

a) `f(x);` _____ b) `f<float&&>(std::move(x));` _____

c) `f(10, x);` _____ d) `f(x, std::move(x));` _____

e) `f(std::move(x));` _____ f) `f(std::move(x), x);` _____

g) `f(std::move(y), x);` _____ h) `f(std::move(y), std::move(y));` _____

i) `f(y);` _____ j) `f<float, float&>(std::move(x), x);` _____

5. Analyse template parameter deduction

In the following function template `foo()`, we define `T` to be the template type parameter, and `T&&` to be the function parameter.

Given the following calls, indicate the deduced T and resultant T&& after type deduction is done by the compiler. Indicate whether the resulting call is compile-able by writing "**C**"/"**NC**" in a required column. The first example is already done for you.

Assume that all appropriate headers have been included.

```
 1  template <typename T>
 2  void foo(T&& t)
 3  {
 4      T bar{std::move(t)};
 5  }
 6
 7  float x = 5.0f;
 8  float* px = &x;
 9  float& rx = x;
10  float&& rrx = std::move(x);
```

Answers: *(1 mark each row; 10 marks total)*

#	Template function call	Template param. `T`	Function param. `T&&`	Compiles? **C/NC**
-	*foo(x);*			
a)	`foo(std::string("a"));`			
b)	`foo(rx);`			
c)	`foo(rrx);`			
d)	`foo<float>(*px);`			
e)	`foo(px);`			
f)	`foo<float&>(std::move(x));`			
g)	`foo(std::move(rx));`			
h)	`foo<float&>(rrx);`			
i)	`foo<float&&>(std::move(rrx));`			
j)	`foo(px+7);`			

Student notes:

6. Assess memory allocation operations

Consider the code snippets **A**, **B** and **C**. Assuming they are a part of a working program, answer the questions *(1 mark each; 10 marks total)*:

Snippet A:

```
1  template<typename T, size_t N>
2  void bar(int x)
3  {
4      T* p1 = static_cast<T*>(malloc(N * sizeof(T)));
5      for (size_t i = 0; i < N; ++i) { new (p1 + i) T{x}; }
6      free(p1);
7  }
```

a)	Write down the expected number of memory **allocation calls** as a specific number or in terms of the parameter `N`.	
b)	Write down the expected number of memory **deallocation calls** as a specific number or in terms of the parameter `N`.	
c)	Write down the expected number of **constructor calls** as a specific number or in terms of the parameter `N`.	
d)	Write down the expected number of **destructor calls** as a specific number or in terms of the parameter `N`.	

Snippet B:

```
1  template<typename T, size_t N>
2  void foo(int x)
3  {
4      T* p1 = new T[N];
5      for (size_t i = 0; i < N; ++i) { new (p1 + i) T{x}; }
6      delete[] p1;
7  }
```

e)	Write down the expected number of memory **allocation calls** as a specific number or in terms of the parameter `N`.	
f)	Write down the expected number of memory **deallocation calls** as a specific number or in terms of the parameter `N`.	
g)	Write down the expected number of **constructor calls** as a specific number or in terms of the parameter `N`.	
h)	Write down the expected number of **destructor calls** as a specific number or in terms of the parameter `N`.	

Snippet C:

```
1  class Dyn
2  {
3      double* _data; // 64-bit pointers
4  public:
5      Dyn(int n = 0) : _data{new double[5]} { (void)n; }
6      ~Dyn() { delete[] _data; }
7  };
```

i)	Write the amount of memory allocated in the free store by this code: `Dyn* a = new Dyn[10];`	
j)	Write the amount of memory allocated in the free store by this code: `Dyn* a = new(preallocAddr) Dyn[10];`	

7. Write the output of the program

The following program compiles successfully, and it produces 10 lines of output.

```
 1  #include <iostream>
 2  struct A {
 3      A() { std::cout << "A" << std::endl; }
 4      virtual ~A() { std::cout << "~A" << std::endl; }
 5  };
 6  struct B {
 7      B() { std::cout << "B" << std::endl; }
 8      virtual ~B() { std::cout << "~B" << std::endl; }
 9  };
10  struct C {
11      C() { std::cout << "C" << std::endl; }
12      virtual ~C() { std::cout << "~C" << std::endl; }
13  };
14  struct D : A, public B {
15      D() { std::cout << "D" << std::endl; }
16      virtual ~D() { std::cout << "~D" << std::endl; }
17  };
18  struct E : public D, C {
19      E() { std::cout << "E" << std::endl; }
20      virtual ~E() { std::cout << "~E" << std::endl; }
21  };
22  int main() {
23      E e; (void)e;
24  }
```

State the expected output printed in each line: *(1 mark each; 10 marks total)*

a)	`Line 1`	_____	b)	`Line 2`	_____
c)	`Line 3`	_____	d)	`Line 4`	_____
e)	`Line 5`	_____	f)	`Line 6`	_____
g)	`Line 7`	_____	h)	`Line 8`	_____
i)	`Line 8`	_____	j)	`Line 10`	_____

8. Analyse the following code

Given the following complete program, answer the questions below.

```
 1  #include <iostream>
 2  #include <utility>
 3
 4  template<typename T>
 5  struct Holder
 6  {
 7    T _value;
 8    template<typename U>
 9    Holder(U&& value) :
10      _value{std::forward<U>(
11        value)}
12    {
13    }
14
15    Holder(Holder& rhs) :
16      _value{rhs._value}
17    {
18    }
19  };
20  template<typename T>
21  auto make_holder(T&& rhs)
22  {
23    return new Holder<T>(
24      std::forward<T>(rhs));
25  }
26
27  int main()
28  {
29    std::string abc("abc");
30    auto p = make_holder(abc);
31    auto q = make_holder(
32      std::string("def"));
33    abc = "ghi";
34    std::cout << p->_value; // C
35    std::cout << q->_value; // D
36    delete p;
37    delete q;
38  }
```

Answer the following questions:

a) What is the full C++ type of `p`? *(2 marks)*

b) What is the full C++ type of `q`? *(2 marks)*

c) What is the printout of the line marked with a comment "`C`"? *(2 marks)*

d) What is the printout of the line marked with a comment "`D`"? *(2 marks)*

e) Does this program have a memory leak? *(2 marks)*

9. Implement the code fulfilling the requirements

Implement a function template `make_shared_and_weak<T>(args)` that returns a shared pointer `std::shared_ptr` and a matching weak pointer `std::weak_ptr` to a newly created object of type `T`, forwarding arguments `args` to the object's constructor. Additionally, implement the code that uses this function template. The detailed requirements are provided below.

The requirements for the **function template**: *(1 mark each; 5 marks total)*

a) The function template must accept a type `T` as its first template parameter.
b) The function template must accept a varying number of function parameters, as required by a constructor of a type `T`.
c) The function template must instantiate `std::shared_ptr` with the new object of a type `T` in an optimal manner.
d) The function template must instantiate a related smart pointer `std::weak_ptr`.
e) The function template must return a tuple with both smart pointers.

Answer: *(5 marks)*

The requirements for the individual **code snippets**: *(1 mark each; 5 marks total)*

f) This code snippet must demonstrate how to use the function template to instantiate smart pointers to a type `MyClass` with its default constructor, and split a returned tuple into individual variables named `s_ptr` and `w_ptr` using **C++17** structured binding.

Answer: *(1 mark)*

g) This code snippet must demonstrate how to use the function template to instantiate smart pointers to a type `MyClass` with its default constructor, and split a returned tuple into variables using **C++11** approach, so that the shared pointer is passed over as `s_ptr`, but the weak pointer is ignored.

Answer: *(1 mark)*

h) This code snippet must demonstrate how to use the function template to instantiate smart pointers to a type `MyClass` with its default constructor, and split a returned into variables using **C++11** approach, so that the shared pointer and the weak pointer are passed over as `s_ptr` and `w_ptr`, respectively.

Answer: *(1 mark)*

i) This code snippet must demonstrate how to overwrite a value of an existing shared pointer `s_ptr` with a value of an existing unique pointer `u_ptr`.

Answer: *(1 mark)*

j) This code snippet must demonstrate how to manually force destruction of an object pointed by the last shared pointer `s_ptr` while the shared pointer is still within the scope of its definition.

Answer: *(1 mark)*

10. Implement the code that matches specification

Write down the code that deals with callable objects or the standard library algorithms, and matches the specification. *(10 marks total)*

a) Define a functor `Division` that takes in `int` arguments `x` and `y`, and if `y` is different than 0, it computes the result of division `x/y`. It must support statements like the following: *(3 marks)*

```
1  Division division;
2  auto result = division(4, 2);
3  if (result) std::cout << *result;
```

b) Define a lambda expression `power` that takes two `double` arguments, and computes the result of raising the first one to the power given by the second (possibly using `std::pow`). It must support statements like the following: *(3 marks)*

```
1  double x = power(4.0, 2.0);
```

c) Using the `power` lambda expression from the previous question, write a **single statement** that fills the `results` container with the results of taking each item in `bases` container and raising them to the power of the corresponding item in `exponents` container.
For example, if `bases` contains values `{2.0, 3.0, 5.0}`, and `exponents` contains values `{2.0, 3.0, 2.0}`, then `results` should be filled with `{4.0, 27.0, 25.0}`. *(2 marks)*

```
1  std::vector<double> bases{2.0, 3.0, 5.0};
2  std::list<double> exponents{2.0, 3.0, 2.0};
3  std::deque<double> results; // to be filled with {4.0, 27.0, 25.0}
```

d) Using the `power` lambda expression from the earlier question, write a **single statement** that takes `bases` and overwrites it with the result of squaring each value.
For example, if `bases` started with the original values `{2.0, 3.0, 5.0}`, then `bases` should store the values `{4.0, 9.0, 25.0}` afterwards. *(2 marks)*

Student's notes: