Week 3

Exercise 21

0 21
? 23
8 25

../21/text.txt

Explain why, when Derived is derived from the Base the Base class is constructed before the Derived class, and the Base class is destroyed only after the Derived class has been destroyed.

Since the derived class expands upon the base class first the base class has to be there and then extra can be added in the form of a derived class. Then when destroying the class, first the outer layer (The derived class) has to be destroyed and then the center (The Base class) can be destroyed.

Why?

An example:

With the vehicle class as a base class, and car as a derived class. The derived class may add ,for example, more members. These members may use existing members from the base class. If the car class is constructed before the vehicle class, these members are not declared yet.

When destroying the class, one first has to destroy the car class before one destroys the vehicle class since car class also contains the vehicle class so if the vehicle class is destroyed first, the pointer to where the car class "starts" is also gone, and where the rest of the class (The derived part) is stored is lost.

W

When using placement new, or in the context of unrestricted unions, explicit destructor calls are encountered. Explain why code using such an explicit destructor calls for objects of a class Derived, which was derived from Base doesn't have to call explicitly the Base class's destructor.

The base destructor is called by default by the compiler after the derived destructor has done its job.

they are. See the interface

What pointer?

Exercise 23

../23/main.cc

```
#include "main.ih"

void caller(Base &obj)
{
  Derived &tmp = static_cast < Derived &> (obj);
  tmp.hello();
  cout << tmp.test() << '\n';
}

//Bad practise since now we have a tmp derived class which isnt properly
//initialised, the derived test function should return zero since it should
//have been intitialised to zero, it however returns a random size_t.


int main(int argc, char const **argv)
{
  Base myBase;
  caller(myBase);
}
```

Why not?

why? isn't this your fault?

≈ ? 23

../23/base/base.h

```
#ifndef INCLUDED_BASE_
#define INCLUDED_BASE_

#include <iostream>

class Base
{
    public:
        Base();

        void hello()
        {
          std::cout << "Base says hello \n";
        };

    private:
};

#endif
```

../23/derived/derived.h

```
#ifndef INCLUDED_DERIVED_
#define INCLUDED_DERIVED_

#include "../base/base.h"
#include <iostream>

class Derived: public Base
{
  size_t d_test = 0;

    public:
        Derived();
        void hello();

        size_t test();

    private:
};

#endif

inline void Derived::hello()
{
  std::cout << "Derived says hello \n";
};
```

../23/derived/test.cc

```
#include "derived.ih"

size_t Derived::test()
{
  return d_test;
}
```

Exercise 25

../25–2/factory/factory.ih

```
#include <string>

using namespace std;
```

8 25

../25–2/factory/factory.cc

```
#include "factory.ih"

string *factory(string const &str, size_t size)
{
  static string staticString = str;              // Can't access str otherwise

  struct ExString: public string                // Local struct extending string
  {
    ExString()
    :
      string(staticString)
    {}
  };
  return new ExString[size];
};
```

../25–2/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

#include <iostream> // Just for testing correct initialisation of array
#include <string>

std::string *factory(std::string const &str, size_t size);

using namespace std;
```

../25–2/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
{
  string const myString{ "test" };               // Testing
  string *myStringArray = factory(myString, 10);
  cout << myStringArray[5];                      // Testing
  delete[] myStringArray;                        // Freeing memory
}

// Note: all this could be implemented in a proper class with a deconstructor,
// this is just a proof of concept.
```

?