Week 4: II

The files pertaining to exercise 26 were already checked, so only the relevant additions for exercise 27 were included here.

../27–tko/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  {
5    Base **bp = derivedFactory(10);
6
7    for (size_t idx = 0; idx != 10; ++idx)  // Returning allocated memory
8      delete bp[idx];
9
10   delete[] bp;
11 }
```

../27–tko/derivedFactory.cc

```
1  #include "main.ih"
2
3  Base **derivedFactory(size_t size)
4  {
5    Base **base = new Base *[size];
6
7    for (size_t idx = 0; idx != size; ++idx)
8      base[idx] = new Derived;
9
10   return base;
11 }
12 //We set the array of pointers to base objects to point to derived objects
13 //instead such that the function returns a pointer to size pointers to Derived
14 //objects.
```

../27–tko/derived/derived.h

```
1  #ifndef INCLUDED_DERIVED_
2  #define INCLUDED_DERIVED_
3
4  #include "../base/base.h"
5
6  class Derived: public Base
7  {
8    private:
9      std::string d_string = "";
10
11   public:
12     Derived();
13     Derived(std::string const &input);
14
15   private:
16     void vHello(std::ostream &out) override
17     {
18       out << d_string << '\n';
19     }
20 };
21
22 #endif
```

../27–tko/derived/c_derived2.cc

```
1  #include "derived.ih"
2
```

```
3  Derived::Derived(string const &input)
4  :
5    d_string(input)
6  {
7  }
```

Week 4

Exercise 28

Note that all of the operator functions are not implemented here, as the point of this code is to show virtual inheritance.

../28/handler/handler.h

```
#ifndef INCLUDED_HANDLER_
#define INCLUDED_HANDLER_

#include "../msg/msg.h"

class Handler: public virtual Msg
{
  public:
    Handler();

  private:
};

#endif
```

../28/msg/msg.h

```
#ifndef INCLUDED_MSG_
#define INCLUDED_MSG_

#include <cstddef>

class Msg
{
  private:
    enum message                        // Original enum
    {
      NONE    = 0,
      DEBUG   = 1 << 0,
      INFO    = 1 << 1,
      NOTICE  = 1 << 2,
      WARNING = 1 << 3,
      ERR     = 1 << 4,
      CRIT    = 1 << 5,
      ALERT   = 1 << 6,
      EMERG   = 1 << 7,
      ALL     = (1 << 8) - 1
    };

    size_t valueOf(message theEnum);  // Only these functions are implemented
    void show(message theEnum);       // to illustrate the use of virtual
};                                    // inheritance
#endif
```

../28/msg/msg.ih

```
#include "msg.h"

#include <iostream>

using namespace std;
```

../28/msg/show.cc

```
#include "msg.ih"

```

```
3  namespace
4  {
5    char const *name[] =
6    {
7      "DEBUG",
8      "INFO",
9      "NOTICE",
10     "WARNING",
11     "ERR",
12     "CRIT",
13     "ALERT",
14     "EMERG",
15   };
16 }
17
18 void Msg::show(message theEnum) // Taken from the solutions of week 1
19 {
20   if (theEnum == Msg::NONE)
21   {
22     cout << "NONE\n";
23     return;
24   }
25
26   for (size_t test = valueOf(Msg::DEBUG),              // iterates over Msg
27        endTest = valueOf(Msg::EMERG) << 1,
28        msgValue = valueOf(theEnum),
29        idx = 0;                                  // for name[idx]
30
31        test != endTest;
32        test <<= 1, ++idx
33       )
34   {
35     if ((test & msgValue))
36       cout << name[idx] << ' ';
37   }
38
39   cout << '\n';
40 }
```

../28/msg/valueof.cc

```
1  #include "msg.ih"
2
3  size_t Msg::valueOf(message theEnum)
4  {
5    return static_cast<size_t>(theEnum);
6  }
```

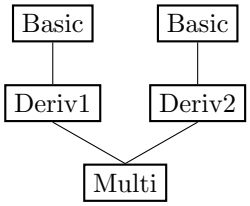
../28/processor/processor.h

```
1  #ifndef INCLUDED_PROCESSOR_
2  #define INCLUDED_PROCESSOR_
3
4  #include "../msg/msg.h"
5
6  class Processor: public virtual Msg
7  {
8    public:
9      Processor();
10
11   private:
12 };
13
14 #endif
```

Exercise 30

- Draw `Multi`'s class hierarchy

Below is the class hierarchy of `Multi` at this point of the assignment.



- Explain the compiler's error message after the addition of the `static_cast`

As can be seen from the illustration above, due to the way that `Deriv1` and `Deriv2` are constructed, `Basic` is included twice by the time `Multi` inherits from `Deriv1` and `Deriv2`. Hence, the compiler indicates that it does not know which `Basic` to cast to.

- Change the statement so that there is no compilation error

First, the cast can be done in a two-step fashion: first to either `Deriv1` or `Deriv2`, and then to its associated `Basic`. By doing so, the compiler knows which `Basic` parent is applicable. This is achieved as follows:

Listing 1: c_multi.cc

```
1 Multi::Multi()
2 {
3   cout << static_cast<Basic *>(static_cast<Deriv1 *>(this)) << '\n';
4 }
```

Secondly, a `reinterpret_cast` can be used instead, as follows. What this does is blindly interprets the `Multi` pointer (`this`) as a `Basic` pointer. Note that this is a very dangerous practice, and should be used with extreme caution, as there are a myriad of problems that can arise from this.

Listing 2: c_multi.cc

```
1 Multi::Multi()
2 {
3   cout << reinterpret_cast<Basic *>(this) << '\n';
4 }
```

- Show the required modifications to allow the compiler to compile the statement without errors

The best way to solve the compilation error without altering the statement would be to make use of virtual inheritance. As such, the class interface of `Deriv1` and `Deriv2` should be changed as follows:

Listing 3: deriv1.h

```
1 ...
2 class Deriv1: public virtual Basic
3 {
4 };
5 ...
```

Listing 4: deriv2.h

```
1 ...
2 class Deriv2: public virtual Basic
3 {
4 };
5 ...
```

- How do you realize that this 2nd constructor is the only Basic constructor that's called

Without specifying otherwise, `Multi` directly calls the default constructor of `Basic`. To change this behaviour, explicitly calling the integer constructor in the initialisation list of `Multi`, as shown below, is the correct approach. Of course the actual integer used here is just an example.

Listing 5: c_multi.cc

```
1  #include "multi.ih"
2
3  Multi::Multi()
4  :
5    Basic(10)
6  {
7    cout << static_cast<Basic *>(this) << '\n';
8  }
```

Exercise 32

../32/fork/fork.h

```
1  #ifndef INCLUDED_FORK_
2  #define INCLUDED_FORK_
3
4  #include <cstdlib>
5  #include <unistd.h>
6  #include <sys/types.h>
7  #include <sys/wait.h>
8
9  class Fork
10 {
11   private:
12     pid_t d_pid;
13
14   public:
15     void fork();
16
17   private:
18     virtual void parentProcess() = 0; // Pure virtual functions
19     virtual void childProcess()  = 0;
20
21   protected:
22     int waitForChild() const;
23     pid_t pid() const;
24 };
25
26 #endif
27
28 inline pid_t Fork::pid() const
29 {
30   return d_pid;
31 }
```

../32/fork/fork.ih

```
1  #include "fork.h"
2
3  using namespace std;
```

../32/fork/fork.cc

```
1  #include "fork.h"
2
3  void Fork::fork()
4  {
5    if ( (d_pid = ::fork()) == -1 )      // In case fork() fails
6      throw "System fork() failed \n";
7
8    if (d_pid == 0)
9    {                       // Hence, child process
10     childProcess();
11     exit(1);              // Bad practice, but must stop
12   }
13
14   parentProcess();
15 }
```

../32/fork/waitForChild.cc

```
1  #include "fork.ih"
2
```

```
3  int Fork::waitForChild() const  // Taken from assignment instruction
4  {
5    int status;
6    waitpid(d_pid, &status, 0);
7    return WEXITSTATUS(status);
8  }
```