Skipped and additional topics

Skipped topics

- Friend class
- Object relationships

Additional topics

- Command line arguments
- cerr, exit, assert
- cppcheck
- Static and dynamic libraries
- Using libraries
- Using CMake

Assignment #4

1. What is polymorphism? Compare and contrast compile-time and run-time polymorphism.
2. What is Virtual function? Why do we need Virtual function?
3. What is Virtual Destructor? Explain how Virtual Destructor avoids memory leakage in the case of Inheritance.
4. Differentiate between Interface class and Virtual Base class?
5. Why do we need to handle exceptions? What is the mechanism in C++ to handle it?
6. What do you mean by Generic Programming? Explain Function Template and Template Class.

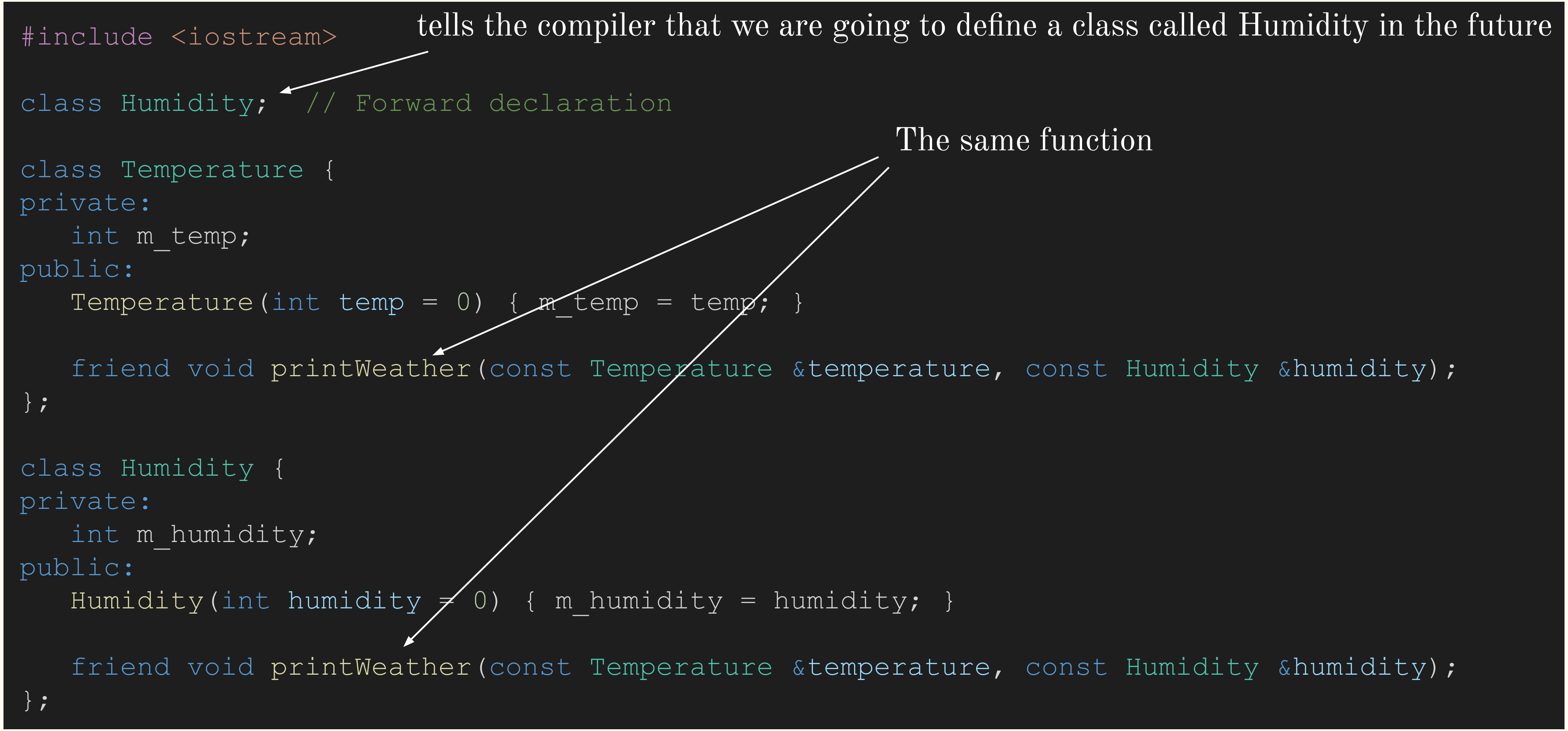
Friend class

Recall

A **friend function** is a function that can access the private members of a class as though it were a member of that class.

A function can be a friend of more than one class at the same time.

Friend function: Example



```cpp
#include <iostream>

class Humidity;  // Forward declaration

class Temperature {
private:
   int m_temp;
public:
   Temperature(int temp = 0) { m_temp = temp; }

   friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity {
private:
   int m_humidity;
public:
   Humidity(int humidity = 0) { m_humidity = humidity; }

   friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};
```

tells the compiler that we are going to define a class called Humidity in the future

The same function

Friend function: Example (Contd.)

```cpp
void printWeather(const Temperature &temperature, const Humidity &humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
                 " and the humidity is " << humidity.m_humidity << '\n';
}

int main()
{
    Humidity hum(10);
    Temperature temp(12);

    printWeather(temp, hum);

    return 0;
}
```

Friend class

An entire class can be a friend of another class.

A friend class can use all the data members of a class (including private ones) for which it is friend.

Friend class: Example

```cpp
#include <iostream>

class A {
private:
    int a1;

    friend class B;
public:
    A(int a1 = 0) : a1(a1) {}
};


class B {
public:
    void print(const A &a) {
        std::cout << "a1 = " << a.a1 << std::endl;
    }
};
```

B is a friend of A

```cpp
int main() {
    A a(10);
    B b;

    b.print(a); // Prints a1 = 10
}
```

Object relationships

Some examples of relationships:

- A square “is-a” shape.
- A car “has-a” steering wheel.
- A student is a “member-of” a class.
- And your brain exists as “part-of” you.

Object Composition

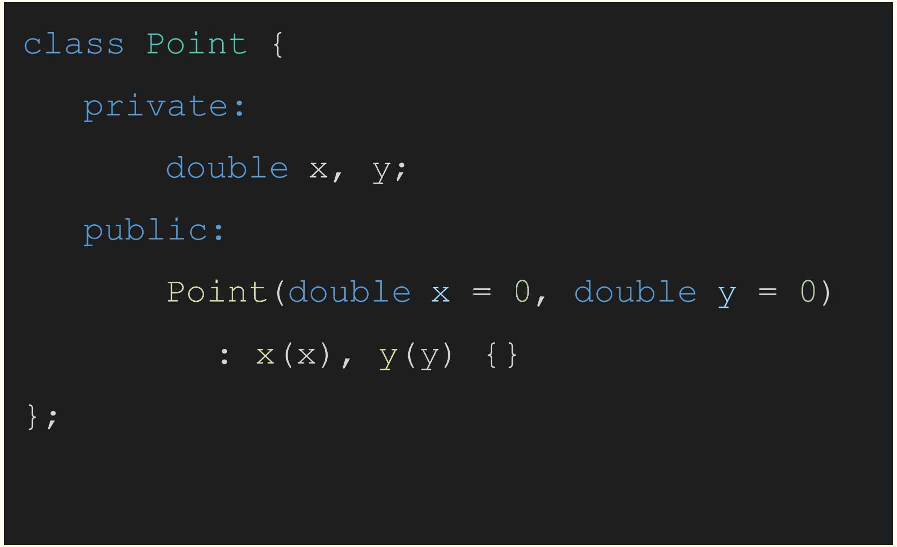
- The process of building complex objects from simpler ones is called **object composition.**
- The complex object is sometimes called the **whole**, or the **parent**. The simpler object is often called the **part**, **child**, or **component**.
- Models a “has-a” relationship between two objects.
  - Examples: A car “has-an” engine. Your computer “has-a” CPU.
- Reduces complexity, and allows us to write code faster and with less errors.
- One OOP design principle: Composition over inheritance
- Two basic subtypes of object composition:
  - Composition and
  - Aggregation.

Composition

To qualify as a composition, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can **only belong to one object (class) at a time**
- The part (member) **has its existence managed by the object (class)**
- The part (member) does not know about the existence of the object (class)

Composition: Example



```
class Point {
   private:
       double x, y;
   public:
       Point(double x = 0, double y = 0)
          : x(x), y(y) {}
};
```

Precisely, composition models “part-of” relationships

- This class has two data members: x and y.
- x and y are part of the Point .
- They cannot belong to more than one Point at a time.
- x and y don’t know they are part of a Point, they just hold doubles.
- When a Point instance is created, x and y are created.
- When the Point instance is destroyed, x and y are destroyed as well.

Aggregation

To qualify as an aggregation, a whole object and its parts must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) **can belong to more than one object (class) at a time**
- The part (member) **does not have its existence managed by the object (class)**
- The part (member) does not know about the existence of the object (class)

Aggregation: Example

```cpp
class Teacher
{
private:
   std::string m_name{};

public:
   Teacher(const std::string &name)
       : m_name{name}
   {  }

   const std::string &getName() const
   { return m_name; }
};
```

```cpp
class Department
{
private:
   const Teacher &m_teacher;

public:
   Department(const Teacher &teacher)
       : m_teacher{teacher}
   {  }
};
```

Aggregation: Example

```cpp
int main()
{
    // Create a teacher outside the scope of the Department
    Teacher bob{"Bob"}; // create a teacher

    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department department{bob};

    } // department goes out of scope here and is destroyed

    // bob still exists here, but the department doesn't
    std::cout << bob.getName() << " still exists!\n";
}
```

Composition vs Aggregation

Compositions

- Typically use normal member variables
- Can use pointer members if the class handles object allocation/deallocation itself
- Responsible for creation/destruction of parts

Aggregations

- Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
- Not responsible for creating/destroying parts

Compositions should be favored over aggregations. Why?

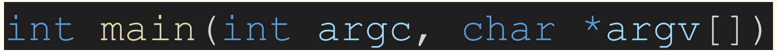
Command line arguments

- Command line arguments are optional string arguments that are passed by the operating system to the program when it is launched.
- The program can then use them as input (or ignore them).

Passing command line arguments

```
$ a.out arg1 arg2
```

Using command line arguments



```
int main(int argc, char *argv[])
```

The number of arguments passed to the program, i.e. the length of argv

An array of C-style strings, that stores the actual argument values

Static and dynamic libraries

A **library** is a package of code that is meant to be reused by many programs.

Typically, a C++ library comes in two pieces:

1. A **header file** that defines the functionality the library is exposing (offering) to the programs using it.
2. A **precompiled binary** that contains the implementation of that functionality pre-compiled into machine language.

Static and dynamic libraries

There are two types of libraries:

- Static libraries
- Dynamic libraries.

Static libraries

- A **static library** (aka an **archive**) consists of routines that are compiled and linked directly into your program.
- When you compile a program that uses a static library, all the functionality of the static library that your program uses becomes part of your executable.
- Typical extension: .lib on Windows, .a (archive) on Linux
- Pros:
  - You only have to distribute the executable in order for users to run your program.
- Cons:
  - Size is big.
  - Can not be upgraded easy

Dynamic libraries

- A **dynamic library** (also called a **shared library**) consists of routines that are loaded into your application at run time.
- When you compile a program that uses a dynamic library, the library does not become part of your executable.
- Typical extensions: .dll (dynamic link library) on Windows, .so (shared object) on Linux.
- Pros:
  - Many programs can share one copy, which saves space.
  - Can be upgraded to a newer version without replacing all of the executables that use it.
- Cons:
  - The executable files are not self-sufficient.
  - If the shared libraries are updated and the updated library is not compatible, the corresponding executable file may not function well.

Using libraries

Once per library:

1. Acquire the library. Download it from the website or via a package manager.
2. Install the library. Unzip it to a directory or install it via a package manager.
3. Tell the compiler where to look for the header file(s) for the library.
4. Tell the linker where to look for the library file(s) for the library.

Once per project:

5. Tell the linker which static or import library files to link.
6. #include the library’s header file(s) in your program.
7. Make sure the program know where to find any dynamic libraries being used.

Using CMake

What's next?

- File handling
- The Standard Template Library (STL)
- The C++ standard libraries: https://en.cppreference.com/w/cpp
- C++ reference: https://en.cppreference.com/w/cpp, https://cplusplus.com/
- Data structures and algorithms (COMP 202, next sem.)
- Learning best practices (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md https://google.github.io/styleguide/cppguide.html)
- Design patterns (https://www.oodesign.com/)

References

1. https://www.learncpp.com/cpp-tutorial/813-friend-functions-and-classes/
2. https://en.wikipedia.org/wiki/Composition_over_inheritance
3. https://www.learncpp.com/cpp-tutorial/102-composition/
4. https://www.learncpp.com/cpp-tutorial/103-aggregation/
5. https://www.learncpp.com/cpp-tutorial/a2-using-libraries-with-visual-studio-2005-express/