Advanced Object Oriented System Design Using C++

Assignment 5

Overview

Object-oriented in C++ is different than in other object-oriented languages since, in C++, all objects are values (not pointers or references) by default. To have a pointer or a reference to something one has to ensure the type of the object is such by properly adding *, &, or && to the type declaration. Object-oriented programming relies on objects being pointers/references for a reason: that is how it achieves some of its polymorphism. Specifically by using a pointer/reference to an object, one can vary the sizeof() the object pointed/referrred to. To represent objects as values, requires having a fixed sizeof() for that type and varying types will require using pointers and/or unions.

This assignment is straight-forward: allowing you to explore how to define a couple of simple classes and to use them polymorphically. It is not intended to be challenging, rather, it is intended that this will allow you to explore how to use some of the "traditional" object-oriented language features as they exist in C++.

Task

A Piece of Code to Start With...

Start by writing this code at the top of an a5.cxx file:

```
#include <vector>
#include <memory>
#include <string>
#include <type_traits>
#include <utility>
#include <iostream>

class animal
{
public:
  virtual ~animal() { }
  virtual std::string kind() const = 0;
};

using animal_ptr = std::unique_ptr<animal>;

template <typename T>
animal_ptr make_animal(T&& t)
{
  using type = std::remove_cvref_t<T>;
  return animal_ptr{ new type(std::forward<T>(t)) };
}
```

i.e.,

- <vector> will be needed for std::vector
- <memory> will be needed for std::unique_ptr
- <string> will be needed for std::string
- <type_traits> is be needed for std::remove_cvref_t (see above code)
- <utility> is needed for std::forward (see above code)

The **kind()** function in animal is called **pure virtual** due to **virtual** and the **= 0**. This is C++'s way of defining an "abstract" object-oriented function. This also implicitly makes the animal class an **abstract class** as C++ has no special keyword to make a class abstract. This will be the base class used with the classes defined in the assignment below.

The **make_animal()** function is a "convenience" function that does the following:

- The **typename T** parameter is **type-deduced** by the compiler from the argument passed to the function. The **T&&** is **not** an rvalue reference in this instance since **T** is a **template type parameter**: T&& instead can be any valid type (value, lvalue reference, or rvalue reference). (Like const& function parameters being able to accept rvalues, this is a special case when a template parameter type is used with &&.)
- Since the **T** can be a reference, it is important to **remove the reference from T's type** so it can be used with **operator new** to dynamically-allocate an object. (Note (a) one wants a value dynamically-allocated so the type must represent a value --not a reference, and, (b) one cannot dynamically allocate a reference type in C++.)
- Ideally one would want to **perfectly forward** the argument **t** to **operator new** to avoid unnecessary copying overhead. (If it was passed by value, then unnecessary copying would occur.) The std::forward function template allows this to occur: std::forward<ArgType>(arg) passes an argument as-is if ArgType is an lvalue reference; if ArgType is a value then std::forward<ArgType>(arg) casts arg to an rvalue reference; and if ArgType is an rvalue reference it is passed as-is.

> ASIDE: Not unrelated is **std::move(expression)**. A call to std::move(v) will **unconditionally cast** v to be **an rvalue reference** of the same type. The **std::forward<T>(t)** function is required when one cannot be unconditional with respect to the casting and wishes to perfectly forward such to another function call's argument. This normally occurs when template types are used with function parameters in a way that one does not know whether or not the type is a value, lvalue reference, or an rvalue reference. std::move is used to explicitly allow an object to be moved, and, std::forward is used to pass an object "perfectly" (i.e., effectively by reference) to a function/constructor argument regardless of its type.

Why have a make_animal() function? In C++, objects are handled by-value unless you are manipulating pointers or references to such. Traditional style bject-oriented programming, in general

Writing the Cat Class

Define a **class** called **cat** as follows:

- the class needs to inherit publicly and virtually from **animal**,
- the class has a single public member **kind()** that overrides the **kind()** (abstract) function in **animal** and returns **"Felis catus"s**.

Tips:

- The **s** following **"Felis catus"** involves the C++ literal operator. The **s** makes the quoted string literal a **std::string** object instead of a **const char [12]** object. To use **s** like this, ensure you have **using namespace std::literals;** inside this function (or some suitable place elsewhere).
- When writing **kind()** don't write **virtual** in front of it BUT do add **override** after the **const**. A function that has the same signature name as a virtual function in the parent class is automatically virtual, thus, you don't need to add **virtual** again. By adding **override** (which is optional) you are telling the compiler you intend to override a virtual function. If for some reason your function doesn't actually override a virtual function, the compiler will report an error. (NOTE: If you don't use override, the compiler has no way of knowing what you intended and cannot report an error. This is why override was added to C++.)

The cat class is intended to be very simple: it simply outputs its genus and species with kind().

Writing the Dog Class

Define a **class** called **dog** as follows:

- the class needs to inherit publicly and virtually from **animal**,
- the class has a single private data member of type **std::string** (choose a variable name to use),
- the class has a deleted default constructor,
- the class has a defaulted copy constructor,
- the class has a defaulted copy assignment operator,
- the class has a **constructor** accepting a **std::string const&** parameter which is the name of the dog being constructed, (This constructor must also set the value of the private data member std::string using direct initialization BEFORE the opening '{' of the constructor.)
- the class has a **kind()** member function that overrides **animal::kind** and returns **"Canis lupus familiaris"s**, and,
- the class has a **name()** member function that returns the type **std::string const&**, i.e., it returns the private data member defined earlier. (This function must also be const-qualified).

Tips:

- Visit the linked-to cppreference.com pages for more information. Look up the corresponding item in the ATOCPP textbook.

- By design there is no way to change the value of a dog's name once it has been set via the constructor.

Writing the dogcat Class

Unfortunately you have to deal with hybrid organisms too --so you need to define a **class** called **dogcat** as follows:

- the class (multiply) inherits both publicly and virtually from both **dog** and from **cat**,
- the class has a **default constructor** that **invokes the parent dog constructor** with the name "hybrid" and the **default cat constructor**,
- the class has a **constructor** that has **two parameters** (**dog const& d** and **cat const& c**) and invokes the corresponding parameter class' copy constructor passing in d or c respectively,
- the class has a **defaulted copy constructor**,
- the class has a **defaulted copy assignment operator**, and,
- the class has a **kind()** member function that overrides **animal::kind** and returns the result of calling **dog's kind()**, + (i.e., string append), the string **" + "**, + (i.e., string append), and calling **cat's kind()**.

Tips:

- A dogcat's kind() function returns a string that is a concatenation of its dog's kind(), " + ", and its cat's kind().
- A dogcat's name is always "hybrid" if default constructed.

Writing operator+(dog const& d, cat const& c)

To make it easier to create **dogcat** hyrbids, write an **operator +**() overload as follows:

- the function has two parameters: **dog const& d** and **cat const& c**,
- the function return type is **dogcat**, and,
- the return expression must create a **dogcat** object from **d** and **c** passing in **d** and c to a **dogcat** constructor.

Writing main()

The above classes will now be used in main() as follows:

- declare a **std::vector<animal_ptr>** variable called **v**,
- declare a variable, **d**, of type **dog** passing in the name **"Fido"**,
- **default construct** a variable of type **cat** called **c**,
- call **v.push_back()** with the result of calling **make_animal(d)**,
- call v.emplace_back() with the result of **dynamically allocating** a **cat** value using **operator new**,
- call **v.push_back()** with the result of calling **make_animal(d+c)**.

The variable v is effectively an array of pointers to animal objects. Its pointers are, however, stored in a "smart pointer" type called std::unique_ptr which will automatically call **operator delete** when it is destroyed. Another useful smart pointer type is called std::shared_ptr which allows multiple **std::shared_ptr** variables to refer to the same object. Like **std::unique_ptr**, **std::shared_ptr** also ensures that the dynamically allocated memory associated with its contents is also destroyed. This is why this program does not directly call **operator delete** to free allocated memory. So the fact that C++ does not have garbage collection is not really an issue: one can make use of C++ destructors in types to "clean up" resources. When this is done, this coding pattern is called RAII (Resource Acquisition Is Initialization).

With the above, let's process polymorphically the animals in the vector object as follows:

- open a (multi-line) **range-for loop** over all elements e of **v** (each element type should be **auto const&**, or if you prefer, **animal_ptr const&**) and inside the for loop:
- output to **std::cout** the output of **e**'s **kind()** member function,
- create a variable called **p** of type **dog*** and assign the result of dynamically casting the result of calling **e.get()** to **dog***,
- if **p** is not **nullptr** then output to **std::cout** the string **", name = "** (without quotes), followed by the result of calling the **name()** function pointed to by **p**, and,
- output the newline character to **std::cout**.

Tips:

- In C if you had a pointer to a struct, how would you access a member of that struct through a pointer? Similarly, in this C++ program, ask yourself what is the type of e? If e is a pointer / smart pointer how would you access the kind() member function through such?

Program Output

Once the program has been written, the output it will produce is as follows:

```
Canis lupus familiaris, name = Fido
Felis catus
Canis lupus familiaris + Felis catus, name = Fido
```

Submission

When finished, upload your a5.cxx file for this assignment. :-)

Submission status

Attempt number	This is attempt 1.
Submission status	Submitted for grading
Grading status	Not marked
Due date	Thursday, 4 March 2021, 11:59 PM
Time remaining	8 hours 13 mins

Grading criteria

Normally the marker will adhere to the rubric, but, the marker reserves the right to add/subtract marks based on the quality of the work provided. (Such overrides will be noted in the feedback comments.)

Compiler Errors (Deduction)	Code fails to compile due to compiler errors. ***-5 points***	No compiler errors. ***0 points***		
Possible Run-Time Errors	Too many run-time errors occur or can occur. ***0 points***	No run-time errors occur or can occur. ***1 points***		
Code Structure, Names, and Comments	Poor code structure, meaningful symbol names, and comments. ***0 points***	Satisfactory code structure, meaningful symbol names, and comments. ***1 points***		
cat Class Definition	There are issues with the cat class definition. ***0 points***	No issues with the cat Class definition. ***1 points***		
dog Class Definition	There are issues with the dog Class Definition. ***0 points***	There are some (minor) issues with the dog Class Definition. ***1 points***	There are no issues with the dog Class definition. ***2 points***	
dogcat Class Definition	There are too many issues with the dogcat Class definition. ***0 points***	There are issues with the dogcat Class definition. ***1 points***	There are some (minor) issues with the dogcat Class definition. ***2 points***	There are no issues with the dogcat Class definition. ***3 points***
Assignment Requirements Conformance	Not all instructions were followed. ***0 points***	Most instructions were followed. ***1 points***	All instructions were followed. ***2 points***	
Late Penalty (Deduction)	Late. All marks deducted. ***-10 points***	Late but submitted within 24 hours of deadline or late without penalty deadline. ***-2 points***	Not late or late without penalty (as decided by or with permission of instructor). ***0 points***	

Last modified Thursday, 4 March 2021, 3:44 PM

File submissions a5.cxx 4 March 2021, 3:44 PM

Submission comments Comments (0)

Edit submission Remove submission

You can still make changes to your submission.

◄ Week 05 Lecture Files

Jump to...

Week 06 Lecture Files ►