Learning Objectives

Understanding the implementation of object-oriented code at the assembly level, and appreciating the parts handled automatically by the compiler. Specifically, translating:

- Classes and objects
- Methods
- Single inheritance
- Virtual method dispatch
- Automatic calls and their sequencing

Additionally, translating higher order functions and understanding their behavior.

Guidelines

- Translate the given C++ using standard calling conventions – we’ll be testing for that.
- Follow the C++ conventions discussed in lecture:
  - The hidden `this` parameter is the first one to each method
  - Members are laid out in the order they’re written in the source
  - The virtual table pointer, when present, is the very first thing in the object
  - Virtual methods are laid out in the vtable in the order they’re written in the source (and as a reminder, only virtual methods go in the vtable)
- The given tests are deliberately insufficient – you should augment them with your own.
- You might find it helpful to first translate the C++ to C, as was done in lecture, and then translate the C to MIPS.
C++ to MIPS translation [70%]

We’ve written a simple C++ class hierarchy in animal.h, pet.h, cat.h and dog.h. crazy_cat_lady.h is another small class, and print_pet_info.cpp and do_stuff_with_things.cpp actually make use of these classes (although they don’t exercise all their functionality). Finally, main.cpp just contains a call to do_stuff_with_things.

Your job is to translate the above files to their MIPS equivalents, using the techniques discussed in the MIPS++ lecture. Skeleton code is provided which you need to complete; there are some helpful comments in both the C++ code and the MIPS skeletons.

You can compile and run the C++ using:

make mipspp-cpp

or

make mipspp-cpp-debug

Make sure you understand its output. There are quite a few calls made automatically in the C++ which you’ll need to do explicitly in your MIPS, so pay attention to their ordering.

You can run your MIPS with:

make mipspp-mips

Make sure the outputs match.

Thought question [10%]

You’ll notice that the classes in our hierarchy have virtual destructors, but our given code never actually needs to do a virtual function dispatch on the destructor. In what situation would you actually need to do so? In other words, why have we declared our destructors to be virtual? Write your answer in thought_answer.txt. (Google is your friend here, and in general).
Higher order functions [20%]

The venerable qsort function in C allows generic sorting of arrays of any element type using a
custom comparison function (and it would be a good exercise to Google qsort and understand how
it works). We’ll implement a simplified version isort which performs insertion sort on an array of
integers using a comparison function:

typedef int (*compare_fn)(int, int);

void isort(int * array, int length, compare_fn compare)
{
    for(int i = 1; i < length; ++i) {
        int curr = array[i];
        int j;
        for(j = i; j > 0 && compare(curr, array[j - 1]) < 0; --j) {
            array[j] = array[j - 1];
        }
        array[j] = curr;
    }
}

The typedef defines a function pointer type to a function which takes two ints and returns
an int. Function pointer syntax in C is fairly nasty, and the [Clockwise/Spiral Rule](http://decl.org) is helpful in
understanding it. If you’re interested, look up std::function, which is a much nicer way of ac-
complishing the same effect in C++11. (I also recommend looking at [decl.org](http://decl.org) if you want to have
the computer do the interpretation work for you).

The isort function then takes a pointer to array of integers, its length, and a comparison function,
and performs a standard insertion sort. The comparison function is expected to take two integers
and return a negative number if the first should be sorted before the second, a positive number if
the first should be sorted after the second, and zero if the two are equal for sorting purposes.

isort.cpp contains the isort function, some comparison functions, and some testing code; you
can compile and run with:

make isort-cpp
or
make isort-cpp-debug

Implement the isort function in MIPS in isort.s. The comparison functions and testing code
are in isort_main.s, so you can run your MIPS with:

make isort-mips

A correct implementation should produce the same output as the C++.

Additionally, write an explanation of how the three comparison functions (ascending, descending,
and even_odd) work in the file comparison_answer.txt. The third in particular deserves some de-
tail, both for what sorting result it achieves and how it achieves it. In particular, make sure you
explain what return x_odd * 2 - 1; is doing.
Fun with vtables [10% Extra Credit]

Look at the class hierarchy in vtable_classes.h and the main function in vtable_test.cpp, which replaces the vtable pointer of a bar object so that it calls the foo version of say_hello instead. Since we’re only changing the pointer of a single object, other objects aren’t affected. There’s an alternative way to achieve this function change which will apply to all bar objects; implement it in vtable_modify.cpp. You’ll find the mprotect function will come in handy – Google it to figure out what it’s used for and how to use it.

You can compile the C++ using:

```
make vtables-cpp
or
make vtables-cpp-debug
```

You should be able to figure out what the output for a correctly implemented vtable_modify should be.

Reminders

- You can use make clean at any time to delete generated executables and debug directories
- The items to submit to git are:
  - animal.s
  - pet.s
  - cat.s
  - dog.s
  - crazy_cat_lady.s
  - print_pet_info.s
  - do_stuff_with_things.s
  - thought_answer.txt
  - isort.s
  - (optional) vtable_modify.cpp
  - (optional) partners.txt

Have Fun!