CPSC 213: Assignment 6

Updated Oct 27 5:00pm: fixed typo is list of files to submit (A3-6.s) and clarified that partners are allowed. Revised due date.

Due: Thursday, October 30, 2014 at 11:59pm

After an eight-hour grace period, no late assignments accepted.

Goal

The goal of this assignment is to explore dynamic control flow from several perspectives.

First, you will implement and test the double-indirect jump instructions. You will also examine snippets that implement polymorphism.

Then you will expand on the polymorphism-in-C example we have discussed in class and that is contained in snippet A to implement a pair of Java classes in C.

Then you will use function pointers to implement a generic iterator by modifying a C program provided to you.

Finally, you will translate a C program that uses function pointers into an equivalent SM213 assembly program.

Then you will explore function pointers in C by modifying three C programs. You will replace switch statements with jump tables, you will generalized a procedure by adding a function-pointer parameter, and you will modify the polymorphic-C example described in class to implement a simple program that uses polymorphism, modelled after a Java program.

Groups

You may do this assignment in groups of two if you like. If you do, be sure that you both contribute equally to the assignment and that you each work on every part of the assignment. Do ***not*** split up the assignment in such a way that one of you does one part and the other does another part. Keep in mind that the core benefit to you of doing the assignment is the learning that happens while you do it. Each assignment is worth only around 1.8% of your grade, but what you learn while doing the assignment goes a long way to determining the other 85%.

If you choose to do the entire assignment with a partner, submit one solution via **`handin`** and list both of your names, student numbers, and computer-science login ids in the **`README`** file.

Alternatively, you can also choose to collaborate with another student for a portion of the assignment and have your work marked individually. Do do this, simply submit your own version via **`handin`** and list the other student as a collaborator in your **`README`** file. Just don't do

this if you and your partner are turning in identical solutions, as we would like to realize marking efficiency where possible. You may have at most one collaborator for the assignment; i.e., you can not collaborate with one student for one part of the assignment and another student for another part.

Code Provided this Week

The file www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a6/code.zip contains the following files that you will use with this assignment

- **SA-dynamic-call.**{**c**,**s**,**java**}
- **A6_1.java**
- **A6-2.c**
- **A6-3.c**

Part 1: Implement and Test Double-Indirect Jump Instructions

There are two remaining instructions to implement in the simulator, described below. Implement them and augment your **test.s** to test them.

Instruction	Assembly	Format	Semantics
dbl ind jmp b+d	`j *o(rt)`	`dtpp`	pc ← m[r[t] + (o == pp*4)]
dbl ind jmp indx	`j *(rb,ri,4)`	`ebi-`	pc ← m[r[b] + r[i]*4]

Part 2: Snippet SA

Examine the Java, C and assembly code for **SA-dynamic-call**. Run the assembly version in the simulator and observe its behaviour. Unlike previous assignments you do not need to document anything.

Part 3: Using Function Pointers for Polymorphism in C

The file **A6_1.java** contains a simple example of polymorphism using two classes: **Person** and **Student**, which extends **Person**. Write a new C program called **A6-1.c** that does the same thing in C. Start by copying **SA-dynamic-call.c**. Then make the necessary changes to replace **A** and **B** with **Person** and **Student.**

Note that the name of the Java file has an underscore instead of a dash, because Java does not allow dashes in class names. Your C program should use a dash, just as you have in other assignments and as you do for the other parts of this assignment.

Hints: Emulating Java Objects

When extending the example code, you will need to consider a three additional issues related emulating Java objects.

First, you will note that an instance of a subclass must contain the instance variables declared in its superclass(s). For example, a **`Student`** object has both a **`name`** and an **`sid`**. Recall that like the jump table, super-class variables need to be located at the same offset in the subclass as they are in the super class. There are just two instance variables in this case, so its probably pretty obvious that this is what you would do.

Second, as shown in class, instance methods in Java have an implicit, hidden parameter, a pointer to the object on which the method is invoked that the method can access via "**`this`**" or implicitly by naming instance variables with the "**`this.`**" suffix. In your C emulation, you will need to make this parameter explicit. For example, a Java instance method declared this:

```
class A {
    void foo() {…}
}
```

And called like this:

```
anA.foo();
```

In C would be declared something like this:

```
A_foo (struct A* this) {…}
```

And called like this:

```
A_foo (anA);
```

Finally, you will need to consider how to implement a subclass constructor. In Java, the subclass constructor calls the superclass constructor using "**`super (…)`**". But since we have not listed the constructors (e.g., **`new_Student`**) in the class jump table, this isn't possible in our C emulation. If you'd like to add constructor to the jump table and go the Java way, that is fine. However, it is also okay to simply have the subclass constructor just repeat the work of the superclass constructor. So, for example, if the Java code looked like this:

```
class Super {
    int x;
    Super (int x) { this.x = x; }
}

class Sub extends Super {
    int y;
    Sub (int x, int y) { super(x); this.y = y; }
}
```

It would be fine if your C constructor for Sub did something like this:

```
struct Sub* new_Sub (int x, int y) {
    struct Sub_object* this = (struct Sub_object*) malloc…;
    this->class = &sub_class_obj;
    this->x     = x;
    this->y     = y;
}
```

Hints: Manipulating Strings in C

In addition to all of this, you will also be manipulating strings in your C program and strings in C are more troublesome, by a long way, than strings in Java, due entirely to the dynamic-allocation issues we talked about a couple of weeks ago; i.e., deciding what part of code is responsible for freeing something that is malloced from the heap.

A C string is stored in an array of characters. The string itself, which can be smaller than the array that contains it, is terminated by the first **`null`** (i.e., 0). You'll need to consider a couple of issues.

First, when a procedure receives a string as an input parameter, if it procedure stores that string, it should make its own copy of the string rather than storing the string pointer. Storing the pointer is dangerous because the caller could free the object following the call and thus turn this stored value into a dangling pointer. By copying, the procedure ensures that its copy of the string is safe from whatever its caller does with the string after the call returns.

You will want to use the standard method called **`strdup`** to do this (see its man page; e.g., via google or by typing "man strdup" at a unix command line). You will also need to add "**`#include <string.h>`**" to the beginning of your file.

For example your code should look like this:

```
void bar (struct X* anX, char* string) {
    anX->string = strdup (string);
}
```

And **not**, as it would in Java, like this:

```
void bar (struct X* anX, char* string) {
    anX->string = string;
}
```

Because if you did this, a caller that does the following creates a dangling pointer:

```
void foo (struct X* anX) {
    char string[] = "Hello World";
    bar (anX, string);
}
```

Similarly, as we examined a couple of weeks ago, a C procedure should not return a pointer to an object it dynamically allocates. Instead, it should leave it to its caller to perform the dynamic allocation and simply copy its result to that location.

For example, Java code that looks like this:

```
String getString () {…}
```

Would in C look like this:

```
void getString (char* buf, int bufSize) {…}
```

Where **buf** is a pointer to memory provided by that caller into which **getString** copies its result up to the limit of **bufSize** bytes.

Finally, you will need to convert numbers to strings and to concatenate strings. The easiest way to do this is with the standard procedure called **snprintf** that uses **printf** formatting to write to a string. So, for example if you wanted to create the string "**Hello World 42**" from the string "**Hello World**" and the integer **42**, you would do something like this:

```
char buf [1000];
snprintf (buf, sizeof (buf), "%s %d", "Hello World", 42);
```

Part 4: Using Function Pointers as Parameters in C

The file **A6-2.c** computes the sum of an array using a procedure called **iterate**. Modify this program so that it computes the following functions on the array: **sum**, **min**, **max**, **size**, **mean**, and **median**. The trick is, that all of these functions must use the same **iterate** procedure. That is, you must modify **iterate** to add a function-pointer parameter that **iterate** uses in place of **sum** and that allows callers of **iterate** to specify which operation they would like it to perform on each list elements. Then implement the six array-summary listed above.

For extra credit change iterate so that it can iterate over lists of arbitrary-type elements and then implement an iterator that concatenates a list of strings. If you do the extra credit, say so in your README and place the extra credit version of the program in the file **A6-2x.c**. *That is, you must submit two versions of your solution.*

Part 5: Function Pointers in Assembly

Examine, compile and run **A6-3.c**. Be sure you understand what it does and how.

Write an SM213 program called **A6-3.s** that might be what the compiler would produce when compiling this C program. You program should treat the variables the same. Notice that **compute()** takes three parameters, two of which are global variables. You code must do that too. Do not name the global variables directly in **compute()**. In other words, you should be able to call **compute()** on a different array without modifying **compute()**. Also note that the

functions **`add()`**, etc. are not called directly. You code must use function pointers in the same way that this program does.

Requirements

Here is a summary of the requirements for each part of this assignment.

1. Revise **`CPU.java`** and **`test.s`**.
2. Examine snippet **`SA`**.
3. Modify **`SA`** to provide a simple example of polymorphism in a C in **`A6-1.c`**.
4. Write a generalize an iterator using function pointers by modifying **`A6-2.c`**.
5. Write the sm213 program **`A6-3.s`**

What to Hand In

Use the **`handin`** program. The assignment directory is **a6**.

1. A single file called "**`README.txt`**" that includes your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like **`a0b1`**), and all written material required by the assignment as listed below.
2. If you had a partner for the entire assignment, turn in only one copy of your joint solution under one of your ids and list both student's complete information in the **`README.txt`** file and include the text "**`PARTNER - MARK JOINTLY`**".
3. If, on the other hand, you collaborated with another student for a portion of the assignment, but your solutions are not identical and you want them marked individually, each of you should include the other student's complete information in your **`README.tx`**t file, include the text "**`COLLABORATOR - MARK SEPARATELY`**", and turn in copies separately via **`handin`**.
4. You are not required to answer any questions in the **`README.txt`** file this week.
5. The following files **`CPU.java`**, **`test.s`**, **`A6-1.c`**, **`A6-2.c`**, and **`A6-3.s`**. *If you do the extra credit also include the file* **`A6-2x.c`**.