CSE4100: Compilers **Fall 2014**

Homework 3: LLVM Part 2

Out Date:	09/25/2014
Due Date:	10/02/2014

Objectives

The intent of this homework is to wrap up with the core concepts of LLVM, namely: learn to handle arrays, structures and pointers[1]. Once again, the algorithms involved are quite trivial and the focus should be solely on your assembly language skills. You will surely remember that the material at `http://llvm.org/docs/LangRef.html` can be extremely useful and I strongly recommend that you take another close look at it, focusing on the array and structure type as well as the `getelementptr` instruction. The homework has two main questions.

As before, hand-in via moodle a single zip archive with a suitably named folder (follow the same convention as for the two previous homeworks). Again, you will be using a Makefile and the GNU suite compilation tools.

1 Handout

To ease your task, there is a small handout that you should download from moodle.

2 Binary Search

Your first task is to write in LLVM assembly a binary search routine operating on a sorted array of n 32-bit wide integers. Namely, write an LLVM function that takes are input an array of integers a as well as its size n and a key k (a 32-bit integer) and returns -1 if k does not appear in a and a non-negative integer r between 0 and $n-1$ if and only if $a_r = k$.

The main program is responsible for setting up a sorted array of n integers and testing your llvm routine.
For instance, the following:

```
1  @msg = constant [9 x i8] c"got:␣%d\0A\00"
2  @data = constant [10 x i32] [i32 10,i32 11,i32 12,i32 13,i32 14,i32 15,i32 16,i32 17,i32 18,i32 19]
3
4  define i32 @main() {
5          %t0 = bitcast [10 x i32]* @data to i32*
6          %t1 = call i32 @binSearch(i32* %t0,i32 10,i32 14)
7          %t2 = call i32(i8*,...)* @printf(i8* bitcast ([9 x i8]* @msg to i8*),i32 %t1)
8          ret i32 %t1
9  }
```

is a perfectly adequate test. You should call your array `@data` and place its definition *on a single line* to help with the grading (as we will modify your source via scripts to try them with different arrays.). You will noticed that, once again, the program must rely on a bitcast instruction to convert the pointer to the constant data (an array of *fixed* size) into a plain pointer to 32-bit wide integers. The call to `@binSearch` is straightforward.

[1] The next homework will return to LR parsing, right in time after we complete the theory in class.

Key instructions to master include

- `load` to read a piece of data in memory based on an address `http://llvm.org/docs/LangRef.html#i_load`
- `store` to write a piece of data into memory at a specific address `http://llvm.org/docs/LangRef.html#i_store`
- `getelementptr` to compute an effective address within a contiguous block of memory `http://llvm.org/docs/LangRef.html#i_getelementptr`. This third function is often the cause of some confusion, so it is worth to carefully read the manual. The instruction does not perform any load or store. Its sole purpose is to figure out the address at which a load / store will be subsequently performed. It cannot follow pointers, it merely strings together a number of offsets within a fixed contiguous memory block. For instance, to compute the address of the third character in a string whose address is in $\%s$, one could do
  `%a = getelementptr i8* %s,i32 3 .`
  Note that several indexing steps can (and often are) combined within a single `getelementptr` instruction. This is perfectly fine as long as the memory in question is completely contiguous and there is no pointer "chasing" taking place.

3 Bubble Sort

You must write a program that manipulates a simple data type represented by a C structure with two integer arrays of the same size. Namely, the data type is:

```
1  struct AIP_tag {
2      int* _a;
3      int* _b;
4  };
5
6  typedef struct AIP_tag AIPair;
```

and it can be found in the source file `misc.c`. Note that that two fields $\_a$ and $\_b$ are pointers to arrays allocated on the heap via calls to `malloc`. Two utility functions are defined for you in `misc.c`. `initPair` is meant to fill the two arrays of a structure whose address you receive with integers (the array $\_a$ is filled with random values whereas $\_b$ is filled with n consecutive integers in $0..n-1$). The `printPair` routine prints the content of the two arrays (as a sequence of pairs).

3.1 Setup and Allocation

You must first create a main program in LLVM that allocates an LLVM pair structure (named $AIPair$)

```
1  %AIPair = type { i32* , i32* } ; a struct with two integers
```

and allocates two arrays of a chosen size (say 50) from the C heap and stores their pointer in the newly allocated structure. It then calls the C function $initPair$ passing the pointer to the structure and the size of the two arrays to initialize them.

3.2 Sorting

The second task is to implement an LLVM routine to sort the content of the structure. Consider that the first array is an array of keys whereas the second array is an array of values. The idea is to permute the pairs so that the keys are listed in increasing order. For instance, the output for a setup with 10 pairs is

```
<32,0>,<32,1>,<54,2>,<12,3>,<52,4>,<56,5>,<8,6>,<30,7>,<44,8>,<94,9>
<8,6>,<12,3>,<30,7>,<32,0>,<32,1>,<44,8>,<52,4>,<54,2>,<56,5>,<94,9>
```

where the first line is the array before the sorting and the second is after the sorting. The `@bubbleSort` routine has the following simple API

```
1 define void @bubbleSort(%AIPair* %tab,i32 %n)
```

the crux of the problem is, of course, to make proper use of the `load,store` and `getelementptr` instructions. Note that your implementation must be sensible and use the correct layout since two functions (`initPair` and `printPair`) are implemented in plain C. Any error in data layout with be quite visible from the C functions.

Have fun!