

CDA 3101 -- Lecture #15+ -- 200806.16

Elaboration on the pointers handout

What to do with a variable when translating C code to MIPS

First thing to recognize is that, in general, a variable (whose *value* may be constant) will either be implemented using a *register* or *main memory* to represent the value. We can *always* use main memory (though accessing the variable's value will be slower); but there are many times when we have the option of using a register.

Pointers and arrays

An *array* is a chunk of contiguous memory where a collection of like typed items can be stored. A *pointer* is a memory address that is associated with a type (e.g., int, char, structure, etc.). The size (in bytes) of the type is used to compute offsets when using array notation/pointer arithmetic. A pointer may be pointing to a single instance of that type, or it may be pointing to one of many instances of that type that are stored in an array: **there is *nothing* in C or MIPS syntax that differentiates between a pointer to a single item vs. a pointer to one of many items.**

A pointer, whether to a single item or to the first element of an array, might be implemented using either:

- * a dedicated register (holding the address of the item/array's first element) *or*
- * a memory location (holding the address of the item/array's first element).

C specifics

An array name is a constant pointer to the first element of that array.

Legal C:

```
int i = 99;           /* initialized int      */
int ia1[3];          /* 3 uninitialized slots */
int ia2[3] = { 11, 13, 77 }; /* 3 initialized slots */
int ia3[] = { 6, 133, -7 }; /* 3 initialized slots */
int *upi;            /* uninitialized pointer */
int *ipi = &i;       /* initialized pointer   */

pi = ia1;            /* point to element 0 of ia1 */
pi = &ia2[2];        /* point to element 2 of ia2 */
```

Illegal C—cannot use an initializer list with pointer:

```
int *pi = { 3, 4, 1 }; /* WILL NOT COMPILE! */
```

Legal C—C-style strings are special:

```
char str1[] = "hi!";  
char *str2  = "bye";
```

As mentioned in class, there is a big difference between those two statements:

str1

is an *array's name*

its value is *constant*

the chars in the string are *mutable*

str2

is a *pointer's name*

its value is *mutable*

the chars in the string are *constant*

Handout

In the handout, I started off by assuming that all *variables* would be stored in *memory*; thus the translations. For example:

```
char sa1[][] = { "bat",  
                "cat",  
                "rat" };  
sala: .asciiz "bat"  
salb: .asciiz "cat"  
salc: .asciiz "rat"  
sa1:  .word   sala, salb, salc
```

The *symbol* **sa1** is a constant. It denotes the starting address of an array containing 3 char*'s. It (**sa1**) is not being stored anywhere!

The following is *illegal* in C; it's here to illustrate "if it *were* legal, how it could be implemented":

```
char **sa2 = { "hat",  
              "mat",  
              "sat" }; [nvc]  
anon3a: .asciiz "hat"  
anon3b: .asciiz "mat"  
anon3c: .asciiz "sat"  
anon3:  .word   anon2a, anon2b, anon2c  
sa2:    .word   anon3
```

The *symbol* **sa2** is also a constant. It is not being stored anywhere! It denotes the address where a *pointer to* an array containing 3 char*'s is stored.

Let's put those 2 notes side by side so you can better see the difference:

sa1 denotes the starting address of an array containing 3 char*'s.

sa2 denotes the address where a *pointer to* an array containing 3 char*'s is stored.

C vs. MIPS

As long as we perform non-mutating operations, `sa1` and `sa2` are used the *same* way in C. However, as we've already seen, they are *implemented* differently in MIPS. Consider:

```
char **x = sa1;                la    $t0, sa1
```

while (not actually in the handout):

```
char **q = sa2;                la    $t1, sa2
                                lw    $t1, 0( $t1 )
```

Why? Because `sa1` is the name of an array (*symbol*); but as we have implemented `sa2` as a pointer *to* an array. That's why the C code looks the same, but the MIPS code and box and pointer diagrams are different.

The comments in the following code fragments are to help you understand the mechanics of how the two statements are implemented; they do not describe the semantic intent and thus would be inappropriate in a program you were turning in.

```
*(*(x + 1) + 2) = 'm';
```

```
# recall, we are already using $t0 to represent x
li  $t5, 'm'          # constant 'm'
lw  $t6, 4( $t0 )    # tmp := MEM[ x + (1 * 4) ]      4 bytes/pointer
sb  $t5, 2( $t6 )    # MEM[ tmp + (2 * 1) ] := 'm'   1 byte/char
```

note: we used `lw` to get the *base destination address*

```
*(sa1 + 2) = *(sa2 + 1);
```

```
la  $t1, sa2          # get address where the pointer is stored
lw  $t1, 0( $t1 )    # get the pointer
lw  $t2, 4( $t1 )    # tmp = *(sa2 + 1) = MEM[ *sa2 + (1 * 4) ]
```

the C variable

the MIPS symbol

the dereferenced MIPS symbol is equivalent in meaning to the C variable

```
la  $t0, sa1          # same as char** x = sa1;
sw  $t2, 8( $t0 )    # MEM[ sa1 + (2 * 4) ]
```

All that said: what the preceding really demonstrates is that when a pointer is stored in *memory*, we have an extra layer of indirection in order to use it (compared to when a pointer is a *symbol*). In either case, *once the pointer is loaded into a register, we use it the same way.*

Exercise 1

Given the following array declarations, write the C and MIPS code to

- 1) count the number of lowercase E's that are in **strings**
- 2) use **strings** to print the textual value of each of the **ints** stopping on encountering an int outside the range of "known" textual values.

```
char strings[][] = {    "zero", "one", "two", "three", "four",
                        "five", "six", "seven", "eight", "nine",
                        "ten", "eleven", "twelve", "thirteen",
                        "fourteen", "fifteen", "sixteen",
                        "seventeen", "eighteen", "nineteen"    };
int ints[] = { 7, 13, 3, 9, -3 };
```

Exercise 2

Given the following array declaration, write the C and MIPS code necessary to add 3 to each element int value. Then sum the elements in each int[] and store that sum in the *last* element of the same array.

```
int i3d[][][] = { { { 0, 1, 2 },
                    { 3, 4, 5 } },
                  { { 6 },
                    { 7, 8 },
                    { 9, 10, 11 } } };
```