CS212

Pointers and Objects in Bali
Spring 2006

Announcements

- P3B: Due on Friday, 11:59 PM
- P3A: Graded. If you were asked to meet with a TA, please do so ASAP.
- P4: Due Sunday 5/7, 11:59 PM. More from DIS later. No document – just finish your compiler.
- Regrades: If anyone still has anything, send them to me (Levitan) ASAP.
- Part 3 templates posted
- Minor grammar clarifications, see e-mail/newsgroup
- Prototypes and Main (Newsgroup post later)
- Pointers in Part 3 Grammar (not supposed to be there)

Review of Pointers

- Pointers point to an address in memory
- Operated on by **`*`** and **`&`**
- l-values and r-values
- Two types of pointers:

Stack:
```
int a; int *b;
a = 10;
b = &a;
print *b;
```

Heap:
```
int *p;
p = <int *> malloc(3);
print *(p+1);
free(<void *> p);
```

- Differences?

Pointers in Bali

- Stack pointers
  - Your compiler should already know address of pointer for referencing/dereferencing.
- Heap pointers
  - Remember that **`malloc`** returns a **`void*`** pointer and **`free`** takes a **`void*`** pointer. More on casting on next slide.
  - **`free`** always returns 0. Note that the **`FREE`** instruction does not put anything on the stack, so you must push 0 onto the stack.
  - Offset is 0 (i.e. data should start on address provided by **`MALLOC`** instruction)
  - Use **`PUSHIND`**/**`STOREIND`** for data access

Casting in Bali

- Casting:
  - You can cast from any pointer type to any other pointer type
  - Do not worry about converting types – let SaM handle this
  - This means the following code is valid:

```
// prints 65 (ASCII for 'A')
char c;
char *d;
c = 'A';
d = &c;
print *(<int *> d);
```


Pointer Arithmetic in Bali

- Not all pointer arithmetic is valid in Bali
- Addition: pointer + int = pointer
  - the pointer can be on either side of the +
- Subtraction: pointer – int = pointer
  - this is the only legal orientation. int – pointer is not valid.
- Subtaction: pointer – pointer = int
  - This gives the difference in addresses between two pointers
- No other arithmetic is allowed

Type Size

- The **`sizeof`** function returns the size of the type provided:
  - **`sizeof(int) == 1`** (same for all primitives)
  - **`sizeof(int*) == 1`** (same for all pointers)
  - **`sizeof(class a*) == 1`**
  - **`sizeof(class a)`** – discussed a bit later
- All integers in pointer arithmetic refer to a section of memory corresponding to the size of the object pointed to (same as in C, except that in Bali primitives all size 1).
  - If **`sizeof(class a) == 4`** and the address of **`class a* p;`** is **`1000`**, then **`p+2 == 1008`**

Accessing Data

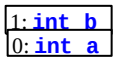
- We have a pointer to a block of heap memory. What do we do with it now?
  1. Use **`*(p+i)`** to access the memory. This may work for list, but its ugly and not convenient if we want to store different data in one block of memory.
  2. Use arrays: **`p[i]`**. In C, **`p[i]`** is the same as **`*(p+i)`**. This is simpler than option 1, but it still doesn't allow us to structure the data in any meaningful way.
  3. Use records (structs in C): **`p->name`**. This gives us a structured way of accessing data without needing to know its location in memory. It also lets us easily have multiple data types in one block of memory.

Structs

- C has structs. Structs only store data in predetermined fields:

```
struct cs212struct{ int a; int b; };
```

Memory Structure:



- Java/C++ objects are an extension of structs, but are much more powerful.
- For this reason, Bali has objects/classes.
- In extra credit you will be able to add methods, etc...
- From now on, any reference to objects/classes is to the Bali sort of object/class (C style structs), not C++/Java

Defining Bali Objects

- The full grammar specifies program to be:

  program -> [*decClasses*] [*decVars*] [*decFuncs*] *classDef* function**

- *decClasses* is optional list of classes for recursive class definitions (i.e. `struct a` contains field of type `struct a`)
- *classDef* is a list of class definitions:

```
classes{ class b; }

classdef a { class b* v1; int v2; }
classdef b { int v1; int v2; }
```


Storing Objects

- Only structs on the heap are allowed. Thus, a variable definition like `class cs212struct a;` is not valid. A definition like `class cs212struct *a;` is valid.
- First, determine how much space is needed. The space is:

$$\text{space} = (\#\,\text{primitive/pointer fields}) + \sum_{\substack{\text{object}\\ \text{fields}}} \text{sizeof(obj ect type)}$$

- Now assign a memory location for each field.
- A struct does not store memory, it just breaks up memory starting at a particular address into logically accessible chunks.

Data in Objects

- You must allocate space for a struct before using it. Use the `sizeof` function to find out how much space you need.
- Remember the pointer arithmetic rules mentioned earlier
- Only use the `->` operator:

```
classdef cs212struct{ int a; int b; }

int main(){ class cs212struct *a; }
{
   a = <class cs212struct *>malloc(
      sizeof(class cs212struct));
   a->a = 4;
   a->b = a->a;
   print 3;
}
```

More Complicated Access

- Suppose we have the following code:

```
classdef cs212struct{ int a;
     class cs212struct *b; }

int main(){ class cs212struct *s; }
{    s = <class cs212struct*> malloc(
          sizeof(class cs212struct));
     (s->a) = 5;
     (s->b) = <class cs212struct*> malloc(
          sizeof(class cs212struct));
     ((s->b)-> a) = s->a;
     print (s->b)->a; }
```

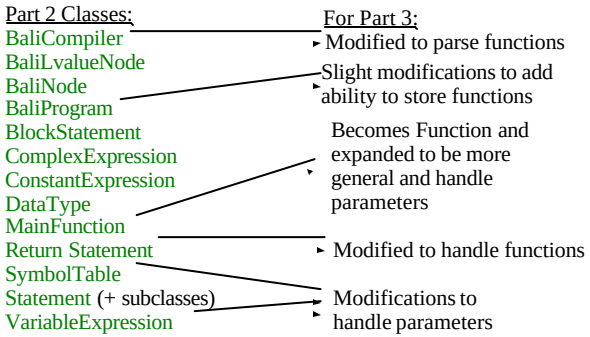
- Note the location of the parentheses for multi-level access and for l-values.

Object Limitations

- In C (not in Bali), `(*a).a` is the same as `a->a`
- Note that `a->b` is an expression, so parentheses must be used as appropriate: `5+(a->b)`
- Remember fields in an object have no access control.
- You cannot dereference a variable/expression which is a pointer to an object. The following code is invalid:

```
int main(){
   class cs212struct *a;
   class cs212struct **b;
} {
   *a; // this is invalid
    b = &a; *b;  //this is valid
}
```


Our Solutions



+ add FunctionCallExpression class

You will get Part 3 solutions after all submissions are in

Adding Pointers

- BaliCompiler
  - Add applicable parsing
- CastExpression
  - Note that a cast expression is a compile time expression
  - It is purely for semantic handling
- ComplexExpression
  - Now handles ***** and **&**, becomes a BaliLvalueNode
  - Additional type checking
  - Handle pointer arithmetic correctly

Adding Pointers

- DataType
  - This is a class to allow you to create custom point data types.
  - Need to add appropriate methods/fields for this.
  - Remember that you can have pointers to pointers.
- FreeExpression, MallocExpression, SizeOfExpression
  - Self-explanatory

Adding Objects

- Modify BaliCompiler to handle appropriate parsing.
- Modify ComplexExpression to handle object references.
- Modify VariableExpression to handle object fields.
- Create a BaliClass class to represent the actual objects.
  - Please note that Class and Object are classes in java.lang, so make sure you are careful with this.

What you could do with this…

```
//Generic queue implementation.
classdef element {  void* item;  class element* next; };
classdef dlist {class element* head, tail; int count; };

//Return an empty queue, or null on failure.
class dlist* queue_new() { class dlist* q; }
{
   q = <class dlist*> malloc(sizeof(class dlist));
   (q->head) = < class element*> null;
   (q->tail) = < class element*> null;
   (q->count) = 0;
   return q;
}
…
```

A full queue implementation like this will work with the grammar you are working with (this is only one function).