CSE 131 – Compiler Construction

Discussion 3: Project 1 – Wrap-up
01/22/2010
01/25/2010

Overview

- Arrays
- Structs
- Pointers
- Function Pointers
- Type Casts
- Address-Of

Array Declaration

- Syntax:

```
type[index] varName;                (int[4] myArray;)
typedef type[index] ALIASNAME;      (typedef int[2] ARR;)
```

- What to check in the declaration:
  - Index is an int
  - Index > 0 (known at compile time → ConstSTO)
- Note that the only way to have an array of array is by using a typedef alias for the inner array (you can only use one [ ] type modifier per variable/type declaration)

Array Usage

- Syntax:

```
myArray[index];
myArray[index][index];
```

- What to check:
  - The designator before the [] must be of Array or Pointer type
  - index must be equivalent to int
  - If index is a constant, you must check the bounds (this only applies when the designator before the [] is an Array type)

Implementation?

- How can one encapsulate the information from the array declaration for later use?
- Remember the Type Hierarchy:
  - One possibility is to store information such as elementType, dimensionSize.
  - In order to do this, you have to add these fields into the ArrayType definition and provide ways to set and read the information from them.

Further Analysis of Arrays

```
int[20] myArray;
int myInt;
int * myIptr;
const int c = 5;

function : void main() {
    myArray[5+c] = myArray[6-c]; // bounds check
    myArray[myInt] = 15;         // no bounds check here
    myArray = 10;                // error, non-modifiable L-val
    myIptr = myArray;            // OK, since array id is ptr
    myIptr[c] = 100;             // no bounds check since ptr
}
```

Struct Declaration

- Syntax:

```
structdef MYSTRUCTALIAS {
        int foo;
        float baz;
        MYSTRUCTALIAS* nextPtr;
        bool foo;
        function : float getBaz() { return this.baz; }
};      // This is the struct definition (similar to a typedef)
```

- What to check in the declaration:
  - Check for duplicate fields (field mySVar.foo in this case)
  - If a field is duplicated multiple times, an error is reported for each duplicate instance (this includes member functions)

Struct Usage

- Syntax:

```
myStruct.myField
```

- What to check:
  - myStruct must be of type StructType
  - myStruct must contain the field myField, in this case
  - After checking, your result will be of the type of myField.

Further Analysis of Structs

```
structdef REC {
  float a, b;
};
REC myRec1, myRec2;

function : void main() {
   myRec2 = myRec1;     // OK, myRec2 mod l-val
   myRec2.b = 3.6;      // assign 3.6 into b field
   myRec2.a = myRec2.b; // assign b field into a field
}
```

Pointer Declaration

- Syntax:

```
int** x;              // pointer to pointer to int
typedef float* PTR;   // alias PTR is a pointer to float
PTR y;                // y is a pointer to float
PTR* z;               // z is a pointer to pointer to float
```

Pointer Usage

- Syntax:

```
*myPtr = 3;                         // pointer dereference
myStructPtr->myStructFunction(23);  // arrow operation
new myPtr;
delete myPtr;
myPtr = NULL;
if(myPtr != NULL && myPtr != myPtr) { /* stuff */ }
```

- What to check:
  - For a dereference, only things of PointerType can be dereferenced. After a dereference, we need to provide an object of the Type pointed to that is ADDRESSIBLE!
  - For an arrow operation, the left-side must be a variable of some pointer to struct. The right-side must be some field/function within the struct.
  - For new/delete, you must check to see if the argument is also of PointerType.

Further Analysis of Pointers

```
typedef float* FPTR;
FPTR x, y;
float z;

function : void main() {
   new x;      // like calloc, no actual allocation in Proj 1
   *x = 7;     // assign value of 7 into where x is pointing
   y = x;      // assign y to point where x points
   z = *y;     // z = 7, if this were in runtime for Proj 2
   delete x;   // like free, no actual deallocation in Proj 1
}
```

Recursive Structs

- Recursive types will have at least one pointer type in a cycle.
- When you encounter a struct alias declaration (structdef), you want that TypeSTO to be in scope immediately, even if you are not finished with the declaration (i.e., it isn't fully complete, but can still be referenced from the Symbol Table).

Recursive Struct Examples

```
structdef LINKEDLIST {
    LINKEDLIST * next;
    int data;
};

function : void main() {
   LINKEDLIST * first;
   new first;
   first->data = 55;
   new first->next;
   first->next->data = 44;
   first->next->next = first;
}
```

Recursive Structs

- Think more about this – it is an important concept (not just for this class).

Refresher on Type Equivalence

- Remember:
  - All types use structural equivalence (except structs)
  - All typedefs/structdefs use name equivalence to resolve down to the lowest-level type
  - Structs-level operations (e.g. assignment, equality, and inequality) use name equivalence. All structs are defined with structdef

Illustrative Example

```
typedef int INTEGER;
typedef int MONTH;

INTEGER i;
MONTH m;
float r;

structdef REC1 { float a; };
structdef REC2 { float a; };
typedef REC1 REC3;

REC1 r1;
REC2 r2;
REC3 r3;

function : int f(REC1 &a) { /* stuff */ }

float[5] a1;
int[5] a2;

function : int g(float[5] &a) { /* stuff */ }

int* p1;
INTEGER* p2;
REC1* p3;
REC2* p4;
REC3* p5;
```

```
function : int main() {
  i = m; // okay, assignable - name equivalent
  i = r; // error, not assignable - float cannot be assigned to int
  r = i; // okay, assignable - int can be assigned to float (coercion)

  f(r1); // okay, same type/equivalent
  f(r2); // error, not name equivalent
  f(r3); // okay, same type/name equivalent

  g(a1); // okay, structurally equivalent
  g(a2); // error, not assignable - not structurally equivalent

  a1 = a1; // error, arrays are not modifiable L-vals

  r1 = r1; // okay, name equivalent and structs are mod L-vals
  r1 = r2; // error, not name equivalent
  r3 = r1; // okay, name equivalent and structs are mod L-vals

  p1 = p2; // okay, structurally equivalent
  p3 = p4; // error, types pointed to (structs) are not name equivalent
  p3 = p5; // okay, structurally equivalent

  return 0;
}
```

Array/Struct Arguments

- Remember that you can pass both Arrays and Structs to functions. For Project I, the following applies:
  - Structs can be passed only by reference (&)
    - Still will depend on name equivalence.
  - Arrays can be passed by reference to array parameters in functions, or be passed by value to pointer parameters in functions
    - You use structural equivalence for checking compatibility.

Function Pointers

- Function pointers are a specific and unique Type.
- They DO NOT require the * to dereference them. Instead, they are dereferenced implicitly by having parenthesis with optional arguments (i.e., a function call).
- But, they do use the assignability and comparison (==/!=) rules like normal pointers.
  - Thus, NULL can be compared to a function pointer, as well as assigned to it.
  - Other functions and function pointers can be assigned to function pointers by specifying the identifier without the parenthesis.

Function Pointers Example

```
typedef funcptr : int (int x, int y) MYPTRALIAS;
MYPTRALIAS myPtr1, myPtr2;

function : int addition(int x, int y) { return x + y; }
function : int subtraction(int x, int y) { return x - y; }

function : int main() {
  if (myPtr1 == NULL) {
    myPtr1 = addition;
  }
  cout << myPtr1(4, 6) << endl;
  myPtr2 = subtraction;
  cout << myPtr2(5, 2) << endl;
  myPtr2 = myPtr1;
  cout << myPtr2(5, 2) << endl;
  myPtr2 = NULL;
  return 0;
}
```

Type Casts

- Type casts are pretty straightforward
  - Take the STO operand and return an appropriate STO (i.e. ExprSTO or ConstSTO) with the type specified in the type cast.
- Some work for casting constants (need to convert the value of the constant appropriately)
- The result of a type cast is always an R-value

Address-Of Operator

- Simply take the operand and make a pointer to that type. This should be an ExprSTO, which is set to be an R-value.
  - Note: if you de-reference the result of an Address-Of, the result of the de-reference will become a modifiable L-value, even if the original object was not.

Address-Of Examples

```
int x, y;
int *z;
const int w = 77;
z = &x; // &x in this example is simply an R-val
&x = NULL; // Error, since not a modifiable L-val
y = *&x; // *&x is essentially just x, so OK.
*&x = y; // The * reverses the &x, making it a modifiable L-val
*&w = y; // The * reverses the &w, making it a modifiable L-val,
         // even though w was originally a constant
&*z = z; // Error, result of address-of is not a modifiable L-val
```

Address-Of Examples

```
function : int foo() { return 0; }
typedef funcptr : int() MYFP;
MYFP MyFuncPtr;
MyFuncPtr = foo;
MyFuncPtr(); // this will be a function call to foo!

MYFP * MyFuncPtrPtr;
MyFuncPtrPtr = &foo; // Error, since 'foo' is constant R-val (name of function)
MyFuncPtrPtr = &MyFuncPtr; // Allowed
(*MyFuncPtrPtr)(); // this will be a function call to foo!
```

Function Overloading (Extra Credit)

```
function : void foo (float x) …
function : void foo (int x) …
function : void foo (int x, float y) …
function : void foo (float x, int y) …

function : int main() {
    foo(1);          // maps exactly to second one
    foo(1.7);        // maps exactly to first one
    foo(4, 8.8);     // maps exactly to third one
    foo(5, 6);       // error, no perfect match
    foo(1, 2, 3);    // error, no perfect match
   return 0;
}
```

Implementation

- The starter code currently places a FuncSTO onto the Symbol Table for each procedure, and uses the procedure name as the unique identifier.
- Here are two possible ways to allow overloading:
  - Name Mangling
  - Function Lookup Table

Name Mangling

- One can mangle function names to incorporate the parameters:
  - foo(float x, int y) can become: foo_float_int in the Symbol Table
  - When you call foo(3.2, 7), you can lookup the FuncSTO by searching for "foo_float_int"

Function Lookup Table

- Create some kind of table (hash, etc) that stores all functions of the same name.
- When you look that name up (say foo), you will get a Vector of all the matches to the name, but the objects will differ by the parameter types.
- Furthermore, you can make the lookup incorporate the arguments in a call and return the single matching procedure if there is one.

What to do Next!

1. Finish up Project I!
2. Write more test programs to verify correctness.
3. Come to lab hours and ask questions.
4. After everything seems to be working, consider working on the Extra Credit.

Topics/Questions you may have

- Anything else you would like me to go over now?
- Anything in particular you would like to see next week?