Name Resolution, Overloading, Allocation, and Lifetimes

CS152

Chris Pollett

Oct. 8, 2008.

Outline

- Symbol Table Organization
- Name Resolution
- Name Resolution and Overloading
- Allocations, Lifetimes and the Environment.

Introduction

- We talked about lexical versus dynamic scoping.
- Lexical scoping is the scoping obtained by reading the program from the top down in a static setting before execution.
- Dynamic scoping results from keeping track of scope based on what in the program has run so far.
- We gave examples of how the symbol tables of a C program would evolve if lexical versus dynamic scoping was used.
- C, C++, Java mainly uses static scoping. Interpreted languages like Lisp, Snobol, Perl, PHP, etc tend to use dynamic scoping.

Stretching the Symbol Table

- What does the symbol table look like when we have a struct? Consider:

```
void p() {
    struct {
        double a; int b; char c;
    } y = {1.2, 2, 'b'};
    /* some more code */
}
```

- To handle a struct we need a local symbol to store its fields (a, b, c) in the above.
- The local symbol table cannot be delete until the struct variable itself goes out of scope.

Symbol Table for a Struct

- The symbol table for the previous example might look like:

(p, (void function))

(y, (struct local to p

symbtab(

(a, (double=1.2)), (b, (int = 2)), (c, (char = 'b'))

)

)

)

- In general, any scoping structure that can be referenced directly in a language must have its own symbol table.
- Examples: named scopes in Ada, structs and namespaces in C++, classes and packages in Java.

Name Resolution and Overloading

- An important question about declarations and the operation of the symbol table is to what extent the same name can be used to refer to different things in a program?
- It is often reasonable to allow some overloading of names.
- For example, '+' in C refers to a function that both can be used to do both integer and double addition. It would be painful to have to write ADDI for one, ADDF for the other; as one does in assembly.
- If we allow overloading, how does the translator tell the different uses apart? It needs to be able to do this to map to machine code.

Disambiguation

- Notice the problem with + is a problem about how functions are looked up in the symbol table.
- Which + we are talking about can be determined by looking at the arguments of the function. For instance, if I see 2.4 + 3.6, I know I am referring to the + on doubles.
- So when we store a function name in the symbol table we need to store roughly its function prototype.
- So we might store:

```
int operator+ (int, int); //in C++ like language
double operator+ (double, double);
```

More on Disambiguation

- Consider the function max(x,y) which returns the larger of x and y or x if they are equal. This can be defined on int's and on double's.
- What do you do in an **ambiguous** situation like: max(3.1, 3); //?
- C++ allows both conversion from integer to double and vice versa. So is the answer 3 or 3.1.
- The language spec does not say which to prefer.
- In Ada, the above would be illegal because no automatic conversions are allowed.
- Java takes the approach that conversion to doubles should be preferred since converting a 3 to a double doesn't lose information but converting 3.1 to an int does.
- In C++, we could overload

  double max (double, int) //to be explicit

The Environment

- We next consider the environment which maintains the bindings of names to locations.
- The environment can be constructed statically (at load time), dynamically (at execution), or a mixture of the two.
- For example, in Fortran all locations are bound statically; in Lisp all locations are bound dynamically; in C++, Java, Ada, etc.; we have a mixture of the two, some allocation is dynamic some static.
- Not all names in a program may be bound at all. For example, const int MAX = 10; can be replaced throughout a program by 10 by the compiler. So the name will have disappeared entirely by execution time.

Declarations and Allocations

- Declarations are used to construct the environment as well as the symbol table.
- In a compiler, they are used to indicate what allocation code the compiler is to generate as the declaration is processed.
- In an interpreter the symbol table and environment are combined, so attribute binding by a declaration includes the binding of locations.
- In block structured languages, global variables are typically allocated statically since they will be available for the whole program.
- Local variables on the other hand are allocated dynamically when execution reaches the block in question.

Environments and Stacks

- We saw that a stack-like mechanism was used by the symbol table to maintain the bindings of declarations.
- Similarly, the environment in a block-structured language uses a stack-like mechanism to bind locations to local variables.
- Consider the labeled blocks in C:

```
A:{ int x; char y; /* (1) */
    B:{double x; int a; /* (2) */ } /* 3*/}
```

- As execution progresses, as each block is entered, the variables declared at the beginning of the block are allocated, when the block is exited they are de-allocated.
- So the runtime stack used to holds this part of the environment might look like (x, y) at (1); (x, y, x, a) at (2), and (x, y) again at (3), where we imagine the end of the list is the top of the stack.
- I wrote the stack like this because usually, the stack grows from the top of memory down.

Allocation for Procedures; Lifetimes

- Consider:

```
void p(){
   int x;
   double y;
}
```

- During execution this declaration will not itself trigger execution of the block of p.
- The local variables x and y of p will not be allocated at the point of declaration.
- Instead x, y will be allocated when p is called.
- Also, each time p is called p is called, new local variables will be allocated.
- We refer to each call to p as an **activation** of p and the corresponding allocated memory and **activation record**.
- We call the allocated area of storage associated with with the processing of a declaration an **object.**
- The **lifetime** or **extent** of an object is the duration of its allocation in the environment.