CPSC320

Tutorials

Robert Pringle

September 25, 2007

L and R-expressions

- L-expressions are expressions that evaluate to a memory location.
  - In an assignment statement in C the variable identifier you are using to assign to would be considered an l-expression as it contains the memory location to write the result back to.
- R-expressions are expressions that evaluate to a value.
  - In an assignment statement in C the expressions/value on the right hand would be considered an R-expression as it evaluates to a value that will stored in the variable on the left hand side.

L and R-expression Practice

Given the following declarations in a C++ style language determine whether or not the expressions below are l-expressions.

```
int j, k, l;
int *p;
int& j;
```

1. $j + k$
2. $*j$
3. $*p + k$
4. $*(p + k)$
5. $*p++$
6. $\&j$

Parameter Passing Methods

- There are a number of different ways to pass parameters to procedures, these include:
  - Pass by value where the actual parameter's value is copied to another memory space. The actual parameter is not affected.
  - Pass by result where the value of the formal parameter is copied back to the actual parameter when the procedure returns.
  - Pass by value-result where the actual parameter's value is copied to another memory space and copied back after the procedure is finished.
  - Pass by reference where the formal parameter refers the memory location of the actual parameter.
  - Pass by name, which used lexical substitution of the formal parameter with the actual parameter and placement within the caller's body.

Pass by name

- Pass by name is slightly different from the other parameter passing methods in that an actual lexical substitution is used for this method.
- Pass by name is performed by the following steps:
  - Replace instances of the formal name parameter with the actual parameter, surrounded in parenthesis in the case of an expression, in the procedure body.
  - Perform lexical substitutions for local variables that conflict variables bound in the caller's scope.
  - Perform lexical substitutions so that free variable's in the procedure body are still valid even if a substitution occurs in such a way that the particular variable is no visible once the substitutions occurs.
  - Place the modified body at the place where the procedure was called.

Parameter Passing Practice

Consider the following code and determine the resulting output for pass by value, result, value-result, reference and name.

```
int g = 20;

void mult(int b, int p) {
    b = g;
    p = b*g;
    return;
}

int main(void) {
    int g = 1, h=2, i=3;
    mult(h,i);
    cout << h << i << endl;
    return 0;
}
```

Declaration Types

- There are two ways in which the visibility of declarations within a given scope can be determined, sequentially or simultaneously.
- If declarations are handled sequentially then a variable becomes visible right after it is declared.
- If declarations are handled simultaneously, then the variables become visible only after all the declaration have been performed for the current scope.

Free Variable Binding

- Within a particular scope a free variable that is not declared within that scope usually refers to a variable declared in a parent scope.
- There are two ways to determine the binding of free variables for a particular scope, statically and dynamically.
- Static binding looks at the lexical/textual structure of the scope and looks at the parent scope the current scope is nested in.
- Dynamic binding looks at the caller, as well as its callers, for a particular procedure at runtime to determine a particular variable binding.
- The scope closest to the current scope, either in the lexical or call order sense, will be used for the binding of free variables.

Binding and Declaration Practice

Given the following program determine its output for sequential and simultaneous declarations. For each declaration type determine what the output would be if static or dynamic binding of free variables was used.

```
int g = 1;

int h = 2;
int i = 3;

int foo() {
    int h = i * 2;
    int i = h;
    return g+i;
}

int main(void) {
    int g = 4;
    cout << foo() << end;
}
```