CSE 340 Fall 2011

Activation Records and parameter passing

Warning! these notes are not meant to replace your notes from class. They are not meant to be exhaustive. If you see a topic that is mentioned here and not elaborated upon, you should go back to your class notes or the book for details. Also, if a topic is not here, it does not mean that I did not cover it.

1 Environments and their pointers

During execution, a name refers to different locations depending on the scope that is active at the point of reference. Even with static scoping, in which name references are resolved at compile time, we still need to determine the locations associated with these names. In general, these locations might not be available until runtime as we have already seen for names local to function. In what follows I will assume we are talking about static scoping.

For global variables, the locations of variables are determined at compile time (I am assuming these variables are not external variables declared in other compilation units). For such global variables, the address of the variable can be determined at the time the reference is resolved (recall that resolving a reference consists of identifying the corresponding declaration whether explicit or implicit).

For non-global variables, the location is not available at compile time. The space allocated to variables of a scope is created dynamically (the space is created dynamically but we are assuming static scoping). I call the pointer to that space is called the *local environment pointer* (this is the *frame pointer* in the case of stack-based allocation and execution of functions, which is the model we will be considering in what follows).

2 Functions definitions and calls

Many programming languages allow the definition of functions that can be *invoked* in a program. The discussion that follows assumes a programming language with a sequential execution model in which one statement is executed after another. I will not make the distinction between functions and procedures, but you should read that in the textbook.

A function definition can allow for the specification of

- Function name
- parameter names and types
- function return type

The parameter specified in the function definition are called *formal parameters*. Functions have function bodies which define a scope for the function. The scope consists of the header of the function together with its body. So, the declarations of formal parameters are in the scope of the function.

A function is invoked in the program using special syntax. The form $f(x_1, x_2, \ldots, x_k)$ is commonly used. $x_1, \ldots, x_k$ are expressions that are visible at the point of the invocation. They are called *actual parameters*.

The compiler generates code that gets executed when a function is called (see below).

In order to determining memory binding at runtime, we need the following information: 1. Global region start address, 2. Frame pointer to resolve local references, and 3. access link to resolve non-local references.

For our purposes, we consider the following information needed to Call (C) a function, Execute (E) the function, and Return (R):

1. (C+E) The address of the entry point of the function
2. (E) The address of the global region
3. (E) The access link that points to the activation record of the defining environment at runtime
4. (E) The frame pointer which points to the begining of the activation record
5. (R) The return address which is the address of the instruction to be executed in the callee immediately after the return
6. (R) The frame pointer of the caller, this is the control link

Calling a function includes the following step by the caller:

1. determining the access link of the callee
2. Saving the frame pointer of the caller (note that this is not known to callee)
3. Saving the return address of the caller (not known to callee)
4. Initialize bindings of instantiated formal parameters. These bindings can involve initializing values (pass by value) or initializing space for for instantiated parameters (pass by reference)
5. Initialize the environment pointer of the callee (for example by setting the value of *fp*, the frame pointer)
6. Jump to entry point of the function code (this changes the program counter)

Calling a function involves creating an *activation record* for the function by the callee[1]. The activation record contains allocates space needed to execute the function:

- space for return value of the function
- space for the old local environment pointer, the local environment pointer of the calling scope (the frame pointer of the calling scope if it is a function or global variable region for the `main()` function).
- space for the return address (the point in the calling environment where execution will proceed after the function finishes execution)
- space for the old stack pointer. The stack pointer points to the top of the execution stack (this is also equal to the new frame pointer and might not be needed)
- space for formal parameters. We will call the formal parameters in a given activation *instantiated formal parameters*. We should note here that for the pass by reference model (see below), *at the language abstraction level*, space for the instantiated formal parameter is not created in activation record (see below). for local variables
- space for temporary variables of the function

Creating all this space is achieved by changing the value of the stack pointer. The space between the old stack pointer and the new stack pointer is the space allocated for the function.

Note that the caller cannot create this space because the caller does not know the space requirement for the local variables and the temprorary variables of the callee.

3 Pass by value and pass by reference

Definition. Parameter passing is *by value* if the values of the actual parameters are calculated and then copied to the space allocated for the instatiated formal parameters at the time of the call.

Definition. Parameter passing is *by reference* if the formal parameters are bound (from binding) to the locations associated with the actual parameter (the actual parameters must be l-values in this case)

We give examples in C and C++

```
// pass by value in C
 void f(int x)
```

[1]This assumes that the function exits after the call ends

```
{  x = x + 5;
}

y = 4;
f(y);
```

In this example, the value 4 is stored in the location allocated to x. The value of x is used in $x+5$ which evaluates to 9 and 9 is stored in x, then 9 is returned by the function. The execution of f does not affect the value in the location associated with y (value of y) which remains 4.

```
// pass by reference in C++
void f(int & x)
 {  x = x + 5;
 }

 y = 4;
 f(y);
```

In this example, the location associated with y (location of y) is associated also with x. Executing $x = x+5$ changes the value in the location associated with x (which is the location of y). When f returns, the value in the location associated with y (value of y) is 9.

4 Pass by name

I explained this in detail in class and gave a couple examples. I will ask about it in the test. You can also read it in the book.

5 Java: pass by value or pass by reference?

I explained in class that according to our terminology, passing objects in Java is by reference. This means that when an actual parameter is passed it location is associated with instantiated formal parameter. In Java, executing $a = newA$, where a is a formal parameter, does not affect the actual parameter corresponding to a. The reason is that in Java, assignment is by sharing, so executing $a = newA$ will have the effect of changing the location binding of a from the actual parameter corresponding to A to the new location created by the call to new.

This terminology is not standard and is not the one used by all authors. Some authors call the parameter passing in Java pass by sharing.

If I ask a question about pass by reference, we will use the definition given in these notes.