Procedures Overview
Lecture 19
Chapter 6 - Intel Manual, v. 1

Robb T. Koether
Hampden-Sydney College
Mon, Mar 30, 2009
Outline

1. Function Calls
2. Parameter Passing
3. Accessing the Parameters
4. The Return Value
5. Returning from the Function
6. Managing the Offsets
7. Assignment
Functions and Function Calls

- A function definition includes
  - A return type.
  - A function name.
  - A list of formal parameters.
  - A block of local variables.
  - Statements, including optional return statements.

- Each formal parameter and each local variable is introduced as a declaration.
Functions and Function Calls

Example (Function Definition)

```c
double average(int a, int b)
{
    double sum;
    double avg;
    sum = a + b;
    avg = sum/2;
    return avg;
}
```
Functions and Function Calls

- This is summarized in the productions

\[
\begin{align*}
\text{func} & \rightarrow \ f\text{beg} \ \text{stmts} \\{ \\
\text{fbeg} & \rightarrow \ f\text{name} \ f\text{args} \ \{ \ \text{dcls} \\
\text{fname} & \rightarrow \ \text{type} \ \text{id} \ | \ \text{id} \\
\text{fargs} & \rightarrow \ ( \ \text{args} ) \ | \ ( ) \\
\text{args} & \rightarrow \ \text{args} , \ \text{arg} \ | \ \text{arg} \\
\text{arg} & \rightarrow \ \text{type} \ \text{id}
\end{align*}
\]
A function call includes
- A function name.
- A list of actual parameters.

Each actual parameter is an expression (of the appropriate type).
Functions and Function Calls

Example (Function Call)

```plaintext
int a;
int b;
double c;
int main()
{
    a = 8;
    b = 16;
    c = average(-a, b + 10);
    print c;
    return 0;
}
```
This is summarized in the productions

\[
expr \rightarrow \text{id}(\ ) \mid \text{id}(\ exprs) \\
exprs \rightarrow exprs, expr \mid expr
\]
Functions and Function Calls

- We need to distinguish the *physical* end of the function from the *logical* end.
- The physical end is defined by the right brace }.
- The logical ends are defined by the `return` statements.
- If there is no `return` statement at the physical end, then we insert one.
The actual parameters are pushed onto the stack, where they can be accessed by the called procedure.

It is the responsibility of the calling procedure to push the parameters onto the stack before the `CALL` statement.

Why?
Before pushing the parameters.
After pushing the parameters.

The leftmost parameter is at the top.
The Procedure Call

- When the procedure is called,
  - The instruction pointer $\text{eip}$ is pushed onto the stack.
  - The address of the procedure is loaded into the instruction pointer.
- This is handled automatically when the `CALL` instruction is executed.
The Procedure Call

Before the procedure call.
After the procedure call.
Creating the Stack Frame

To create a new stack frame (activation record) for the procedure,

- Push `ebp` onto the stack to preserve the “old” base pointer.
- Move `esp` to `ebp` to preserve the “old” stack pointer and establish the base pointer of the stack frame.
- Subtract from `esp` the size of the block of local variables.
Creating the Stack Frame

Create a Stack Frame

```assembly
push %ebp        # Save old base ptr
mov %esp,%ebp   # Save old stack ptr
sub $n,%esp     # Create local block
```
The Procedure Call

Just after executing `CALL`.
The Procedure Call

Save Old Base Pointer

push %ebp  # Save old base ptr
The Procedure Call

Save Old Stack Pointer

\texttt{mov \%esp,\%ebp \# Save old stack ptr}
The Procedure Call

Create Local Block

```
sub $n,%esp  # Create local block
```
The Procedure Call

Create Local Block

```
sub $n, %esp  # Create local block
```
Accessing the Parameters

- With each parameter, there is associated an offset.
- The offset refers to the byte of the value that has the lowest address.
- The offset of a parameter is positive since the parameter is “below” the base pointer in the stack.
- For example, to access the parameter with offset 12, we use the indexed addressing mode.
  \[ 12(\%ebp) \]
Accessing the Parameters

le a 12(%ebp),%eax
Accessing Local Variables

- With each local variable, there is associated an offset.
- The offset of a local variable is negative since the local variable is “above” the base pointer in the stack.
- For example, to access the parameter with offset $-4$, we use the indexed addressing mode.

$-4(\%ebp)$
Accessing Local Variables

\[ \text{lea} \ -4(\%ebp), \%eax \]
When we return from a procedure, we first must pass the return value to the calling procedure.

- If the return value is an `int`, then we move it to `eax`.
- If the return value is a `double`, then we move it to `st(0)` in the FPU.

The calling procedure will go to `eax` or `st(0)` to retrieve the return value.
Returning from a Procedure

- When execution returns to the calling procedure, we must
  - Clear the block of local variables from the stack.
  - Restore the old stack pointer.
  - Restore the old base pointer.
Returning from a Procedure

Clear the Stack Frame

```
mov %ebp,%esp  # Restore old stack ptr
pop %ebp      # Restore old base ptr
ret          # Restore old instr ptr
```
Returning from a Procedure

- Just before returning.
Returning from a Procedure

```
mov %ebp,%esp  # Restore old stack ptr
```
Returning from a Procedure

pop %ebp # Restore old base ptr
Returning from a Procedure

ret # Restore old instr ptr
After execution returns to the calling function, the calling function must remove the parameters from the stack.

This is done by simply adjusting the stack pointer.

We must add to the stack pointer the number of bytes in the parameter block.
Returning from a Procedure

Before removing the parameters.
After removing the parameters.
Computing the Parameter Offsets

- When parsing the argument list of a function definition, we must keep count of the number of bytes used by the arguments.
  
  The `SymbolTable` class variable `fArgSize` is used for this.

- When we begin parsing the argument list, we must initialize `fArgSize` to 8.

- Why 8?
Computing the Parameter Offsets

- For each argument, we must
  - Store the value of $f\text{ArgSize}$ as the offset of that argument in its $\text{IdEntry}$ object.
  - Add the argument size to $f\text{ArgSize}$.

- Because the production

  \[ \text{args} \rightarrow \text{args} , \text{arg} \]

  is left-recursive, the arguments are parsed from left to right.
The syntax tree for the actual parameter list is built from the bottom up and later traversed from the top down. Therefore, the parameters will be pushed onto the stack in reverse order, from right to left.
For example, if the parameter list is 
\((\text{int } a, \text{ double } b)\) 
then the offset of \(a\) is 8 and the offset of \(b\) is 12.
Computing the Local Variable Offsets

- When parsing the local variable declarations in the function definition, we must keep count of the number of bytes used by the local variables.
- The `SymbolTable` class variable `dclSize` is used for this.
- When we begin parsing the declarations, we must initialize `dclSize` to 0.
- Why 0?
For each local variable, we must
- Add the argument size to dclSize.
- Store the value of \(-dclSize\) as the offset of that argument in its IdEntry object.

Because the production

\[ dcl s \rightarrow dcl s dcl \]

is left-recursive, the arguments are parsed from left to right.
For example, if the local variables are

```
int a; double b;
```

then the offset of `b` is $-4$ and the offset of `a` is $-12$. 
Assignment

Homework