Class 30: Stack Frames (1)

Held: Wednesday, 9 November 2011

Summary: We begin our transition from the front end to the back end by considering the layout of memory.

Related Pages:
- EBoard.
- Reading: Aho et al. 7.1-7.3.

Notes:
- Are there questions on Project, Phase 4?
- For those not graduating this year: Tomorrow’s CS extra is on summer opportunities in CS.

Overview:
- About the back end.
- Where do variables and values go?
- Stacks and stack frames.
- Function and procedure calls.
- Non-local variables.

Moving to the Back End
- We’ve covered most of the front-end details: lexing, parsing, and basic semantic analysis.
- Now it’s time to move on to the back end of the parser. That is, the generation of code (perhaps intermediate code, perhaps assembly code) from the annotated parse tree.
- We’ll look at the issue in steps.
  - We’ll consider some general issues (run-time environment, assembly code) this week.
  - We’ll start looking at particular translations starting next week.

Storing Variables in Memory
- A first consideration is how to handle the storage of variables and parameters in memory.
- As you know, in most modern languages it is possible to call procedures recursively and create new instantiations of the local variables for those procedures.
- In addition, when a function exits you no longer need access to its local variables.
- However, in languages that support the dynamic allocation of memory (e.g., most object-oriented languages), there are also some values that live beyond the function that created them.
- Typically, values that are only active during the lifetime of a procedure are allocated on a stack and
values that are independent of procedure lifetime are allocated on a heap.

- Most languages assume one stack and one heap.
- In modern architectures, some variables should be stored in registers to improve performance.

The Stack

- At the machine level, the stack is simply an area of memory that is allocated and deallocated in a stack-like manner.
- Typically, the stack starts at the high end of memory and the heap starts at the low end of memory.
  - This design makes it possible to delay the decision of how much memory to use for heap and how much for stack until run time (i.e., you can grow either heap or stack until the two cross).
  - This design suggests that stacks grown downward and shrink upwards, like bungie cords.
- A designated register called the stack pointer keeps track of the end of the stack.

The Heap

- The heap is an even more amorphous area of memory. Parts of the area are allocated by explicit allocate calls (e.g., `new`) although the determination of which area to use is up to the system rather than the program.
- In many languages (including Pascal) programmers must manage the memory they allocate, freeing it when no more memory is available.
  - The system still must do some behind-the-scenes work in keeping track of which memory the programmer has designated as available and free.
- In some more modern languages, the system is in charge of keeping track of which memory is in use and freeing unused memory “automatically”. This technique is commonly referred to as garbage collection.

Stack Frames

- Since a function will often require space for many variables (parameters, local variables, temporaries, etc.) it is more convenient to allocate all of that space at once.
  - This means that we should predetermine the maximum amount of space a function will use.
  - As long as we’ve determined that space, we might as well lay out the data in that space.
- The organization of local data for the invocation of a function is typically called a stack frame.
- A frame pointer indicates the beginning of the frame.
  - Why have both frame pointer and stack pointer? At times, you need to keep track of other frames.
- What goes in a frame (or in accompanying registers)?
  - Any local variables
  - Parameters (subject to the caveats below)
  - The return address (what statement to branch to when the method exits)
  - Any temporaries
  - Saved registers
  - Space for the return value
Other things ...

Function Calls

- How do we call a function?
- The caller places some of the formal parameters on the stack (often, in its own stack frame).
- The caller places some of the formal parameters in registers.
  - If those registers are currently in use, the caller must store their current values on the stack.
- The caller places a return address and static link on the stack (often, in the next stack frame).
- The caller branches to the beginning of the called function.
- The called function allocates a new stack frame, updating the stack pointer.
- The called function executes.
- The called function stores its result in a register (or on the stack, in a more primitive implementation).
- The called function deallocates its stack frame.
- The called function branches back to the return address.
- The caller makes use of the result value.
- The caller restores any registers necessary.

Accessing Non-Local Variables

- In nested languages, like Pascal, it is possible to refer to a variable from a non-local scope. E.g.,

  ```pascal
  procedure zebra(x: integer);
  var
    y: integer;
    procedure stripes(z: integer);
    begin
      ...
    end
  begin
  end;
  ```

- How do you get access to \( x \) and \( y \) from within \( \text{stripes} \)?
  - And what if we have even deeper nesting.
- One possibility is to have every frame include a pointer to the frame of the enclosing scope (not necessarily the caller). This means that you have to trace backward an appropriate amount, but that amount can be computed at compile time. Such a pointer is typically called a static link.
- Another possibility is to use a global display which maps each scope to the appropriate stack frame.
- A third possibility is to pass all of the variables to the function and restore them afterwards. This can be particularly difficult to implement.