

## The Store Model

- Each programming language has its own conceptual model of *store*.
  - Pascal is a typical *Algol-like* language & has:
    - » A *store* to hold the code representing the program
    - » A *stack* to hold the static and semi-static variables
    - » A *heap* to hold the dynamic and semi-dynamic variables
- We will use the RPAL machine in lectures as the target for our code generation – as you will in your compilers.

### The Code Store

- The code store is conceptually an *array*.
  - » We refer to a given code *location* by giving its *address*.
- The structure of the code store is *simple and regular*

## Lifetime of a Variable

- This is the interval of time storage area (for value) is bound to a variable
- Acquiring a storage area is called *allocation*
- *static allocation* - performed before run-time
- *semi-static allocation* - at run-time in the activation record
- *semi-dynamic allocation* - at run-time above the activation record
- *dynamic allocation* - at run-time in the heap
- Java is interesting – *most* Java Objects can be semi-static!

## The Stack (Run-Time Stack)

- The run-time stack is organized as a series of *activation records*:
  - There is one activation record for each *instance* of a procedure/function.
  - Local variables are stored in the activation record.
- An activation record consists of three basic parts:
  - The *stack mark*.
  - Space for local variables.
  - The *expression stack*.
- The stack mark maintains some basic "housekeeping" information:
  - The dynamic link (lists the blocks in the order they were called).
  - The static link (defines the referencing environment).
  - Return address information.

## Calling Conventions

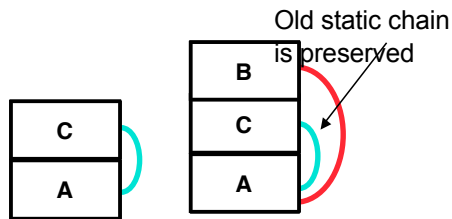
- caller startup:
  - mark stack top (start new AR)
  - push parameters
  - call (pushes RA)
- callee prologue:
  - create space for locals
- callee epilogue:
  - destroy locals & return (RA at TOS)
  - if a function, return value left at TOS
- caller cleanup:
  - none
- Note that these conventions are somewhat unusual
  - AR construction is normally done by the callee, for example

## The Static Chain

- When procedure B is called, the static chain links all those AR's which contribute to the scope
  - This means the AR for C is *excluded*

```

procedure A;
  var i : integer;
  procedure B;
    var i : integer;
    begin ... end;
  procedure C;
    begin ...B;... end;
  begin ...C;... end;
    
```



- This also shows us how we access *non-local* variables: we need to traverse the chain. How far?
  - We compute the *level difference*: current level – level of declaration and traverse that many links down the static chain.

## Accessing Variables

- We can locate the activation record for a variable, but how do we get the full address?
  - At compile time, we know how many variables we have in a block *and* we know their size.
  - Compute an *offset* for each variable from the start of the activation record.
  - Internally, the compiler represents the *address information* about a variable by a pair: (level, offset).
- Address assignment is easy: maintain a counter which is incremented by the size of the variable every time a variable is declared.
  - The type manager needs to be modified to keep the sizing information.
  - The counter (called the location counter) is zeroed every time we enter a new scope level.

## Address Assignment

- Adding address assignment to our previous code is relatively simple.
  - Implementing `size_of`:
    - If we don't keep the `size_of` information in the record that describes the type we will need to:
      - » Traverse the type description and propagate the information back up the tree
- ```

array [ 1 .. 10 ] of T
    => 10 * Size_of(T)
    
```
- It doesn't matter which strategy is chosen
  - Sometimes you *must* defer computation of the size
  - Some languages require the size information to be computed *on block entry* – eg an array sized by a formal parameter

## The Heap

- A *heap* is a pool of storage from which dynamically sized objects can be retrieved.
  - The "new" function in Pascal retrieves store.
  - The "dispose" procedure returns store.
- Pascal & C are languages that exhibit *explicit* storage allocation – some languages have *implicit* storage management.
  - Store is allocated from the heap automatically.
  - When the store is no longer accessible, it can be re-used by the heap.
  - The gathering up of the inaccessible store is called *garbage collection*.
- Garbage collection (typically) involves traversing all the pointers the program has access to, marking all the storage thus encountered and then collecting all the *unmarked* store as garbage.
  - This can be expensive!
  - Must be able to identify all pointer objects.
  - Compiler writer must make this task possible.

## The Expression Stack

- The top of the run-time stack is called the expression stack because that is where arithmetic expressions are evaluated.
- $2 + 3 * 4$  will be translated into:
  - push(2)
  - push(3)
  - push(4)
  - multiply
  - add
- The expression stack is transitory – it should be empty after each "expression" has been computed.
  - In Pascal, this means whenever a statement has been executed.

## Values in the Stack Machine

- Unlike most computers, the store in our stack machine is *typed*.
  - A *tagged architecture*.
  - Each storage cell in the system has a type field or tag.
    - » Only a value with a matching tag can be stored in a cell.
  - The machine implements some of the dynamic type rules of the language for us, simplifying the task of writing the compiler!
  - The tags only represent the base-types of the language.
    - » Recall that there are an infinite number of possible types!
- Unlike most computers, we don't have to worry about the issue of *alignment of data*:
  - Modern RISC processors, for example, usually only deal with 32 bit data aligned on a 32 bit boundary.
  - Our stack machine is more sophisticated and easier to generate code for.