Programming Languages: Lecture 15
I have no clue

Rishabh Dhiman

4 February 2022

1 Static

Before execution (runtime)

- compile time
- linking time
- loading time

Context-sensitive analysis (static semantics) Type, size, requirements, scopes, etc.
Static memory allocation.
FORTRAN 4: All data sizes are known at compile-time, there is no nesting of sub-routine or recursion.
Memory allocation for all sub-routines and the main routine allocated at compile-time. Memory is divided up into essentialy disjoint units so that there is no overlap.

1.1 Static allocation

+ faster execution because there are no overheads due to fixed predefined address for all data
- inflexible
- does not allow for recursion or nested scoping
- wasterful over time because a lot of memory is simply lying idle and is not used.

Lot of weather-prediction programs currently used use FORTRAN because of *speed*.

2 Dynamic

At execution-time (runtime), meaning of program.

2.1 Dynamic Memory Allocation

Almost all other languagws which support

- nested scoping
- recursion
- dynamic data-structures
- all OO, functional and declarative languages

Most algol languages, starting from ALGOL60.

+ Flexible allocation at runtime
+ Memory may be more optimally used on demand.
- Impossible to ascertain how much memory is actually required at any time, because it depends on the calling-chain at any instant of program execution.
- Impossible to ascertain how many recursive activations of a procdure or function may be required at any instant of time during execution.
- Impossible to ascertain how many different objects of various sizes need to be present simulataneously during runtime.

3 Run-Time Environment

Memory for running a program is divided up as follows,

Code segment. This is where the *object* code of the program resides.

Runtime stack. Required in a *dynamic* memory management technique. Especially required in languages which support *recursion*. All data whose sizes can be determined *statically* before loading is stored in an appropriate *stack-frame* (activation road).

Heap. All data whose sizes are not determined statically and all data that is there is genreated at run time.

Consider the following program,

```
// Main Program
// global vars
{
        // Procedure P2
        // Locals of P2
        {
                // Procedure P21
```

```
                // Locals of P21
                // Body of P21
        }
        // Body of P2
}

{
        // Procedure P1
        // Locals of P1
        // Body of P1

        // Call P2
}
// Main body
```

with the following calling chain,

$$\text{Main} \rightarrow P1 \rightarrow P2 \rightarrow P21 \rightarrow P21.$$

At the end of P2, all global variables must be present, P1 variables should still exist but shouldn't be accessible. P21 made a recursive call to P21, the same variables should be allocated again however, these should be allocated at a different location as before.

The memory is divided up as follows,

When stack overflows heap, segmentation faults occur.

The stackfram or activation record looks as follows,

```
Return address to last of P21.
        Dynamic link to last P2.
Locals of P21.
Static link to last P2.
Formal par of P21.

-----------
Return address to last of P2.
        Dynamic link to last P2.
Locals of P21.
Static link to last P21.
Formal par of P21.

-----------
Return address to last of P1.
        Dynamic link to last P1.
Locals of P2.
Static link to main.
```

```
Formal par of P2.

-----------
Return address ot Main
        Dynamic Link to main
Locals of P1
Static Link to Main
Formal par of P1

--------------
Globals
```

Address are all relative to the base address of an activation.

The static link specifies the non-local referencing environment. It points to its 'parent' in the scoping structure and tells what variables other than the local variables it's accessing and to get all these non-local variables we have to go through all these static links to the root, the number of transitions specify the nesting depth, 0 transitions – local variables, etc.