

# CD Moodle Assignment 5

**Jovial Joe Jayarson**

**20 June 2020**

**IES17CS016**

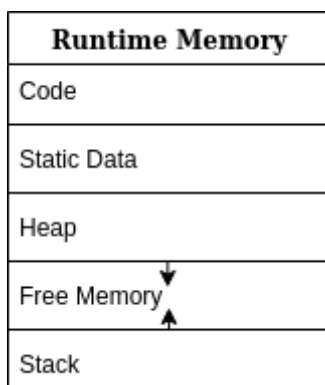
# 1. Explain about storage organization and storage allocation strategies.

Ans 1:

- The compiler creates and manages a run-time environment in which it assumes its target programs are being executed.
- This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program.

## Storage Organization

- When a program is running, dynamic memory is required, while in the editing and compilation stage it is in the HDD.
- Once the target code is generated it needs to be executed in the main memory.
- The following figure shows the runtime memory.



- Each section does the following:
  - Code: Stores actual instructions
  - Static Data: Stores macro, constants global variables etc.
  - Heap: Used for dynamic memory (the heap grows towards higher addresses)
  - Stack: The stack is used to store data structures called activation records that get generated during procedure calls. (the stack grows towards lower addresses)

The figure show the opposite so that we can use positive offsets for notational convenience.

## Storage Allocation

There are mainly two types of memory allocation:

1. Static Allocation: The memory does not change the size during runtime. e.g. C/C++ arrays.
  - Merits
    - It's fast.
    - Memory management is simpler.
    - Less likely to encounter null pointer errors.

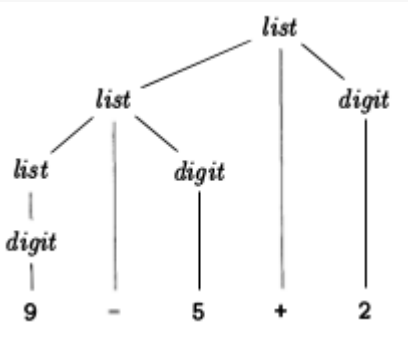
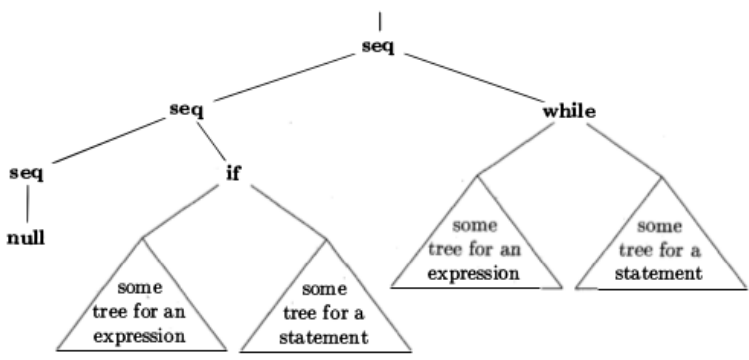
- Demerits
  - We require the prior knowledge of the needed memory.
  - Both memory wastage and shortage can occur.
  - Insertions and deletions can be expensive.

2. Dynamic Allocation: The memory changes during the runtime of a program. e.g. C/C++ vectors.

- Merits:
  - Uses stack or heap which are more structured.
  - It supports recursion.
- Demerits:
  - It's slower.
  - Does not save activation records (garbage collection)
  - Likely to run into null pointer errors.

## 2. Compare parse tree and syntax tree. construct syntax tree for the expression `a - 4 + c`

Ans 2:

Parse Tree	Syntax Tree
The parse tree is a concrete representation of the input.	The syntax tree is an abstract representation of the input.
The parse tree retains all of the information of the input.	The syntax tree does not retain all of the information of the input because the associations are derivable.
Generated as the next step after lexical analysis.	They don't deal with grammars and string generation, but programming constructs.
<p>*</p> 	<p>*</p> 

\*figures are from SO

### 3. Explain quadruple, triple and indirect triple with example.

Ans 3:

Let an expression be `p_new = p + ((p * n * r) / 100)` Its equivalent three address code will be:

```
t1 := n * r
t2 := p * t1
t3 := t2 / 100
t4 := p + t3
p_new := t4
```

1. **Quadruple** is a structure which consists of 4 fields namely operator `opr`, arguments(`arg1`, `arg2`) and result `res`.

@	opr	arg1	arg2	res
(0)	*	n	r	t1
(1)	*	p	t1	t2
(2)	/	t2	100	t3
(3)	+	p	t3	t4
(4)	=	t4		p_new

2. **Triple** is the standard form of representation of any three address code. It consists of fields `opr`, `arg1` & `arg2`.

@	opr	arg1	arg2
(0)	*	n	r
(1)	*	p	(0)
(2)	/	(1)	100
(3)	+	p	(2)
(4)	=	(3)	

3. **Indirect Triple** is an enhancement of the *triple* representation. It uses an additional instruction array to list the pointer to the triples in the desired order.

#	@
1001	(0)
1002	(1)
1003	(2)
1004	(3)
1005	(4)

@	opr	arg1	arg2
(0)	*	n	r
(1)	*	p	(0)
(2)	/	(1)	100
(3)	+	p	(2)
(4)	=	(3)	

#### 4. Explain three address code and types of three address statements.

Ans 4:

##### Three Address Code

- Three-address code (TAC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.
- Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. e.g. `C := A + B`
- Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses or processor registers, but rather symbolic addresses that will be translated into actual addresses during register allocation.
- It is common that operand names are numbered sequentially since three-address code is typically generated by the compiler.

e.g. convert: `a = b * -c + b * -c` to a TAC

```

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4

```

## Types of TAC statements

1. Copy statement: `y := x`
2. Assignment statements: `y := x op z` (`op` = `+` | `-` | `*` etc.)
3. Assignment instruction statements: `y := op x` (`op` = `++` | `--` | `-` etc.)
4. Unconditional jump statement: `goto L`
5. Conditional jump statements: `if x relop y goto L` (`relop` = `<` | `>=` | `!=` etc.)
6. Indexed assignment statements: `x := y[j]`
7. Address & Pointer assignment statements: `x := &y` `x := *y`
8. Argument passing statements: `param x; call p, n`