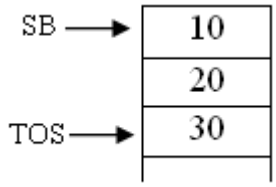
MODULE 3 – STORAGE ALLOCATION

Storage Allocation

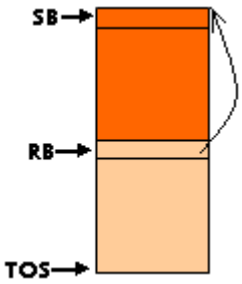
It involves three important tasks.
- Specifies the amount of memory required
- It is a model to implement life time and scope
- Perform memory mapping to access non scalar values

Stack
- Linear Data Structure
- Allocation and deallocation in LIFO order
- One entry accessible at a time, through the top



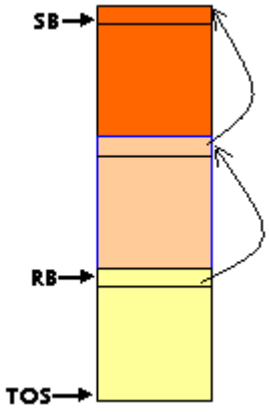
A stack holding 3 elements are given above. Here the top element is 30 pointed by the pointer TOS.

Extended Stack
Extended stack differ from a normal stack due to the following reasons.
- Entries are of different size
- Two additional pointers
  - *Record base pointer (RB)*, points to first word of last record
  - First word in any record is a *Reserved Pointer* for house keeping



Allocation actions
Push operation

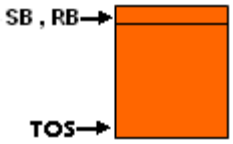
TOS = TOS + 1
TOS* = RB
RB = TOS
TOS = TOS + n



Pop operation

TOS = RB – 1

RB = RB*



There are two methods of storage allocation, Static Allocation and Dynamic Allocation.

Static Allocation

Allocation performed before execution, during compilation. Usually, the global and static constants and the garbage collection information are allocated statically.

Dynamic Allocation

Allocation performed during execution. There are two types of dynamic execution. Automatic Dynamic Allocation and Program controlled Dynamic allocation

Automatic: Memory is allocated to the variables declare in a program unit when the program unit is entered during execution and deallocated when the program unit is exited. This uses the stack.

Program controlled: Allocation performed during execution at arbitrary points. This uses a heap. Usually, the malloc () and new statements are allocated in this way.

Memory allocation in block structured languages

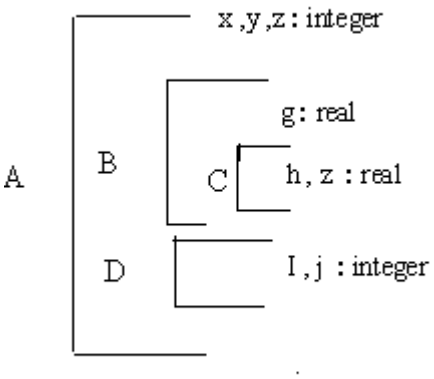
- A block is a program segment that contains data declarations.
- There can be nested blocks.
- Uses dynamic memory allocation.

Scope Rules

If a variable 'var' is created with the name 'n' in a block b.

1) 'var' can be accessed in any statement situated in block b.
2) 'var' can be accessed in any statement situated in a block b| ,which is enclosed in b, unless b| contains a declaration using the same name.

Consider the sample program given below.

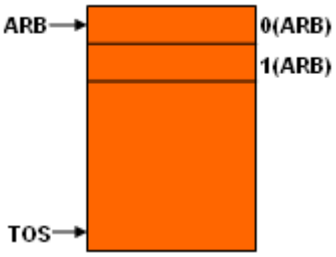


Block	Accessible variable	
	Local	Non-local
A	x_A, y_A, z_A	----------
B	g_B	x_A , y_A , z_A
C	h_C, z_C	x_A , y_A , g_B
D	i_D , j_D	x_A , y_A , z_A

Variable z_A is not accessible inside block C since C contains a declaration using the same name. Thus z_A and z_C are two distinct variables.

Memory allocation and access

- Automatic memory allocation implemented using extended stack model with variation
- Each record is called an *Activation record (AR)* and it contains variables for one activation of block
- During execution *Activation Record Base(ARB)* points to start of TOS record
- Local variable 'x' can be accessed as <ARB> + d_x, where d_x is the displacement from starting point to the position where variable x is stored.



Dynamic pointer

First reserved pointer in activation record is called dynamic pointer. It points to the activation record of its dynamic parent.

Allocation actions

Push operation

TOS = TOS + 1
TOS* = ARB
ARB = TOS
TOS = TOS + 1
TOS* =
TOS = TOS + n

Pop operation

TOS = ARB – 1
ARB = ARB*

Static pointer

Second reserved pointer in activation record is called static pointer. It points to the activation record of its static ancestor. A record can have any level of ancestors.

Array allocation and access

A 1-D array can be allocated as a single sequence of contiguous locations. But for a 2-D array, this is not the case. Array elements can be arranged either in row major form or in column major form. Given below is the case were elements are arranged in column major form. Consider a 2 dimensional array with m rows and n columns. Address of the element $a[s_1,s_2]$ can be written as
$Ad.a[s_1,s_2] = Ad.a[0,0] + \{ (s_2-0) * m + (s_1-0) \} * k$, where k is the amount of memory for storing a word.

If the lower limit and upper limit are l_i and u_i then we can rewrite the above formula as
$Ad.a[s_1,s_2] = Ad.a[l_1, l_2] + \{ (s_2- l_2) * (u_1–l_1+1) + (s_1- l_1) \} * k$

If $range_i$ represents the range of i^{th} subscript, we have
$range_1 = u_1–l_1+1$, $range_2 = u_2–l_2+1$. Then
$Ad.a[s_1,s_2] = Ad.a[l_1, l_2] + \{ (s_2- l_2) * range_1 + (s_1- l_1) \} * k$

$= Ad.a [l_1, l_2] – (l_2 * range_1 + l_1) * k + (s_2 * range_1 + s_1) * k$
$= \underline{Ad.a [0, 0] + (s_2 * range_1 + s_1) * k}$

Dope vectors

An array descriptor called dope vector (DV) is used to store dimensions, lower and upper bounds and range values of the array. A dope vector is like:

Address of a[1,1,…,1]		
No. of dimensions, m		
l_1	u_1	$range_1$
l_2	u_2	$range_2$
l_n	u_n	$range_n$

Code generation for expressions

The major issues in code generation for an expression are as follows:

- An evaluation order for the operators in an expression is required.
- Selection of instructions to be used in the target code. This depends on the type, length and addressability of the operand.
- Use of registers and handling partial results.

The evaluation order of operators depends on operator precedences.The choice of an instruction depends on the following.

1. The type and length of each operand.
2. The addressability of each operand.

An operand descriptor is used to maintain the type, length and addressability of each operand. An operand descriptor is built for every operand participating in an instruction.

A partial result is the value of some subexpression compiled while evaluating an expression. Partial results are maintained in CPU registers.

Operand Descriptor

An operand descriptors has 2 fields:

ATTRIBUTE: Type, length and miscellaneous information.
ADDRESSABILITY: Specifies where the operand is located. It has 2 fields:

Addressability code: Takes the values M (operand is in memory) and R (operand is in register)

Address: Address of a CPU register or memory word.

1. An operand descriptor is built for every operand participating in an expression.

Let us see an example: a*b. The operand descriptor for that is:

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	R, addr(AREG)

Register Descriptor

It has 2 fields:

STATUS: “Code-free” or “occupied” to indicate register status.

OPERAND DESCRIPTOR: If status is “occupied”, this field contains the descriptor for the operand contained in the register.

Occupied	#3

This indicates that the register AREG contains the operand described by descriptor #3.

Intermediate code for expressions
This helps for easier generation of assembly code. The 3 forms are:

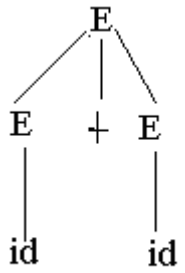
1. Postfix
2. Syntax Tree
3. Three address code

Postfix
General form of a postfix string is operators immediately after the operands

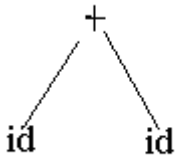
Example
Consider the expression ‘a+b*c+d*e^f’. The corresponding postfix string is ‘abc*+def^*+’. Usually, stacks are used to perform this conversion. The operand descriptors are pushed into and popped from the stack as needed. Descriptors for partial results are also used. The main advantage is that it can evaluate as well as generate the required code. But stacks are slower.

Syntax Tree

It is a type of intermediate representation, which is similar to parse tree. The syntax tree will have operators as roots but the parse trees will have the non terminals as their roots.
Eg: id+id



Parse Tree



Syntax Tree

Three Address Codes

The three address code is a type of intermediate code, which are popularly used in compilers. Mostly these are helpful in optimizing compilers.

Three address codes is sequence of statements of the general form as given below.

A: =B op C.

Where A,B and C are identifier names, constants or compiler generated temporary variable names and "op" represents operator, which may be any binary operator like arithmetic operator, or relational operator, or a Boolean operator, and the symbol ":=" stands for assignment operator.

This representation is called three address codes because it has three addresses, two for the operands and one for the result.

Eg 1: The expression is i: =i+1

The three address code is i: = i+1

2. The expression is i: =i+j+k

The three address code will be slightly different

t: =i+j

i: =t+k where t is a compiler generated temporary variable.

The three address codes are generally implemented using any one of the following representations.

1. Triples
2. Quadruples

Triples

They represent elementary operations in the form of pseudo machine instructions. Each operand of the triple is either a variable or a constant or a result of some other evaluation.

General form of a triple is

Operator Operand1 Operand2

The expression given in the postfix example can be written in triple form as given below

.

	Operator	Operand1	Operand2
1	*	b	c
2	+	1	a
3	^	e	f
4	*	d	3
5	+	2	4

Quadruples

They are similar to triples, except that the result has a field of its own and it is stored in a location temporarily. So, for subsequent use, the location is referred.

General form of a quadruple is

Operator Operand1 Operand2 Result name

Here, the result name is a name. Temporary memory locations are not allocated for this. The expression 'a+b*c+d*e^f' can be written in quadruple form as given below.

Operator	Operand1	Operand2	Result name
*	b	c	t1
+	t1	a	t2
^	e	f	t3
*	d	t3	t4
+	t2	t4	

Expression trees

- Abstract syntax tree which shows the structure of an expression.
- Simplifies analysis of expressions to determine best evaluation order.

Consider the expression '(a+b)/(c+d)'. This can be evaluated in two different ways.

Method 1

```
MOVER AREG, A
ADD AREG, B
MOVEM AREG, temp_1
```

```
MOVER AREG, C
ADD AREG, D
MOVEM AREG, temp_2
MOVER AREG, temp_1
DIV AREG, temp_2
```

Method 2

```
MOVER AREG, C
ADD AREG, D
MOVEM AREG, temp_1
MOVER AREG, A
ADD AREG, B
DIV AREG,  temp_1
```

To select the best evaluation method, we can use the following algorithm. The algorithm is performed as two steps. First we draw the expression tree and calculate the register required for each node without using temporary locations in the bottom up order. Then we evaluate by scanning the tree in the top down order.

Finding best evaluation method

1. Associate register requirement label with each node
2. Evaluate as

Visit all nodes in post order

For each node n_i

i. If n_i is a leaf node

If n_i is left operand $RR(n_i) = 1$

Else $RR(n_i) = 0$

ii. If n_i is not a leaf node

If $RR(l\_child_{ni}) \neq RR(r\_child_{ni})$

$RR(n_i) = \max(RR(l\_child_{ni}), RR(r\_child_{ni}))$

Else

$RR(n_i) = RR(l\_child_{ni}) + 1$

```
evaluation_order (node)
{
    If node is not a leaf node
        If RR(l_child_node) <= RR(r_child_node) then
            evaluation_order(r_child_node);
            evaluation_order(l_child_node);
        else
            evaluation_order(l_child_node);
            evaluation_order(r_child_node);
    print(node);
}
```

Example
Consider the expression f + (x+y) * ((a+b) / (c-d))