UNIT 3

Intermediate Code Generation

Need for Intermediate code

1. Port the entire compiler to generate the machine instruction for different system.

2. The second approach involves breaking the compiler into modular elements called Front End and Back End.

Intermediate Forms

1. Postfix Notation
2. Abstract Syntax Tree
3. Three Address Code (TAC)

1.Postfix Notation

- Postfix notation is one of the forms of intermediate code that is used in the construction of compilers.

Infix	Postfix
X=a+b	Xab+=
Y=(a+b) * c	Yab+c*=
X=(a+b)*(c+d)/(e+f)	Xab+cd+*ef+/=

2. Abstract Syntax Tree

- Some of the front ends of compilers translate the input source into an intermediate form known as Abstract Syntax Tree (AST).
- Grammar

$$E \rightarrow E + T \mid T$$

$$\mathsf{T} \rightarrow T * F \mid \mathsf{F}$$

$$\mathsf{F} \rightarrow id$$

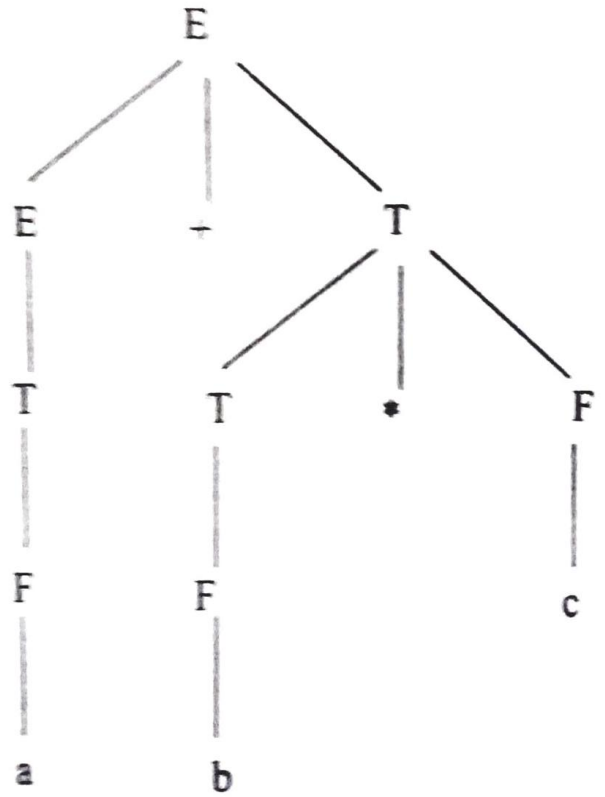

Figure 6.3 a : Parse tree for a + b * c

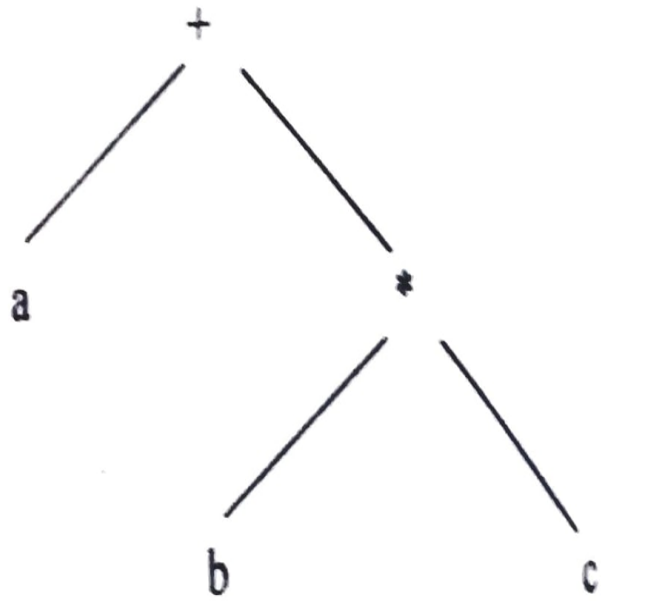

Fig 6.3 b AST for a + b * c

3. Three Address Code

- In three address code, at most three addresses are used to represent any statement.
- The general form of three address code is

$$a = b \; op \; c$$

where

- a, b, c are the operands that can be names, constant or compiler generated temporaries.
- *op* represent the operator

Common Three Address Instruction Forms

- ***Assignment Statements*:-**

  x = y *op* z

  x = *op* y

- **Copy Statement :-**

  x = y

- **Conditional Jump**:-

  if x *relop* y *goto* 1

- **Unconditional Jump**:-

  goto 1

- **Procedure call**:-

  param x call p

  return y

Generation of Three Address Code

- i. a = b + c + d

Solution: Three address code will be

t1 = b + c

t2 = t1 + d

a = t2

- ii. –(a * b) + (c + d) – (a + b + c + d)

Solution:

t1 = a * b

t2 = uminus(t1)

t3 = c + d

t4 = t2 + t3

t5 = a + b

t6 = t3 + t5

t7 = t4 – t6

- iii. Generate Three Address Code (TAC) for

  if a < b and c < d then t = 1 else t = 0

Solution:

(1) if (a < b) goto (3)

(2) goto (4)

(3) if (c < d) goto (6)

(4) t = 0

(5) goto (7)

(6) t =1

(7) next instruction

- iv. Generate Three Address Code (TAC) for

```
c = 0
do
{
    if( a < b)  then
        x++;
    else
         x--;
     c++;
} while (c>5);
```

- *Solution:*

  (1) c = 0

  (2) if (a < b) goto (4)

  (3) goto (7)

  (4) t1 = x + 1

  (5) x = t1

  (6) goto (9)

  (7) t2 = x – 1

  (8) x = t2

  (9) t3 = c + 1

  (10) c = t3

  (11) if (c<5) goto (2)

  (12) Next Instruction

- v. Generate Three Address Code (TAC) for

```
switch(ch)
{
   case 1: c = a + b; break;
   case 2: c = a – b; break;
 }
```

- *Solution:*

```
        if ch = 1 goto L1
        if ch = 2 goto L2
   L1: t1 = a + b
         c = t1
          goto last
   L2: t2 = a – b
         c = t2
          goto last
last:
```

Implementation of Three address Code (TAC)

- Three address code can be implemented as a record with three address fields. There are three representations used for three address code:-

  1. Quadruples
  2. Triples
  3. Indirect Triples

Quadruples

- It is a structure which consists of 4 fields namely *op*, *arg1*, *arg2* and *result*.
- *op* denotes the operator and *arg1 and arg2 denotes two operands and result is used to store the result of the expression.*

Location	Op	Arg1	Arg2	Result

Triples

- In triples representation, the use of temporary variables is avoided and instead, references to instruction are made. The three fields are *op*, *arg1* and *arg2*

Location	Op	Arg1	Arg2

Indirect Triples

- This representation is an enhancement over triples representation. It uses an additional instruction array to list the pointers to the triples in the desired order. Thus it uses pointers instead of position to store results which enables the optimizers to freely reposition the sub expression to produce an optimized code.

Location	Statements

Location	Op	Arg1	Arg2

- *Example:* Translate the following expression to quadruple, triple and indirect triples.

$$X = a + b * c / e \wedge f + b * a$$

Solution:-

t1 = e ^ f

t2 = b * c

t3 = t2 / t1

t4 = b * a

t5 = a + t3

t6 = t5 + t4

Quadruples

Location	Op	Arg1	Arg2	Result
(0)				
(1)				
(2)				
(3)				
(4)				
(5)				

Triples

Location	Op	Arg1	Arg2

Indirect Triples

Location	Statements

Location	Op	Arg1	Arg2

Translation into intermediate Forms

- Syntax Directed Translation(SDT) is the name given to the process of translating the input source to intermediate code based on the syntax of the language.
- The CFG in which the productions are shown along with the semantic rules is called as the Syntax Directed Definition(SDD)

Example

Production	Semantic rules
L → E	L.Val = E.Val
E → E1 + T	E.Val = E1.Val + T.VaL
E → T	E.VaL = T.Val
T →T1 * F	T.Val = T1.Val * F.Val
T → F	T.Val = F.Val
F → (E)	F.Val = E.Val
F → digit	F.Val = digit.lVal

Attributes

- Attributes may be of any kind of numbers, types, table references or strings etc.
- Types of Atrributes:
  - Synthesized Attributes
  - Inherited Attributes

- **Synthesized Attributes**:- A synthesized Attributes at node N is defined only in terms of attributes values at the children of N and N itself.
- **Inherited Attributes**:- An Inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's Siblings.

S-Attributed SDT vs L-Attributed SDT

S-Attributed SDT	L-Attributed SDT
1. Uses only synthesized attributes	Uses both Inherited and Synthesized attribute. Each Inherited attributes is restricted to inherit either from parent or left siblings only. Ex:- A→ XYZ {Y.S=A.S; Y.S=X.S; Y.S=Z.S}
2. Semantic actions are placed at right end of production A→ BC { }	2. Semantic actions are placed anywhere on right end of production A→ { } BC / D { } E / FG { }
3. Attributes are evaluated during BOTTOM UP parsing.	3. Attributes are evaluated by traversing Parse Tree, Depth first , Left to Right .

Write the SDD for the following Grammar:

E → E + E / E * E / (E) / I

I → I digit / digit

The SDD form is given below for the grammar:

E → E(1) + E(2) {E.VAL := E(1).Val + E(2).Val }

E → E(1) * E(2) {E.VAL := E(1).Val * E(2).Val }

E → (E(1)) {E.VAL := E(1).Val}

E → I {E.VAL := I.Val}

I → I(1) digit {I.Val := I(1).Val *10 + digit.LEXVAL}

I → digit {I.Val := digit.LEXVAL}

Syntax Directed Translation for Assignment Statement

- The CFG for the assignment statements is given below:
- A → id:=E

  E → E + E / E * E / -E / (E) / id

CFG	Semantic action
1. A→id : = E Gen (id.Place: =E.Place)	{ A.Code: = E.Code ‖ id.Place ‖ : = ‖ E.Place }
2. $E \rightarrow E^{(1)} + E^{(2)}$ Gen ($E.Place = E^{(1)}.Place + E^{(2)}.Place$)	{ T = NewTemp(); E.Place = T; $E.Code = E^{(1)}.Code \| E^{(2)}.Code$ ‖ $E.Place = E^{(1)}.Place$ ‖ + ‖ $E^{(2)}.Place$ }
3. $E \rightarrow -E^{(1)}$ Gen ($E.Place = -E^{(1)}.Place$)	{ T = NewTemp() E.Place = T $E.Code = E^{(1)}.Code()$ ‖ $E.Place = -E^{(1)}.Place$ }
4. $E \rightarrow (E^{(1)})$	{ $E.Place = E^{(1)}.Place$ $E.Code = E^{(1)}.Code$ }
5. E→id	{ E.Place = id.Place E.Code = null }

Syntax Directed Translation of Boolean Expressions:

- E → E **or** E
  / E **and** E
  / **not** E
  / (E)
  / id
  / id **relop** id

CFG	Semantic action
1. E→E (1) or E (2) Gen (E.Place = E(1).Place or E(2).Place)	{ T = NewTemp(); E.Place = T; E.Code = E(1).Code ‖ E(2).Code ‖ E.Place = E(1).Place ‖ or ‖ E(2).Place }
2. E→E (1) and E (2) Gen (E.Place = E(1).Place and E(2).Place)	{ T = NewTemp(); E.Place = T; E.Code = E(1).Code ‖ E(2).Code ‖ E.Place = E(1).Place ‖ and ‖ E(2).Place }
3. E→ not E (1) Gen (E.Place = not E(1).Place)	{ T = NewTemp() E.Place = T E.Code = E(1).Code () ‖ E.Place = not E(1).Place }
4. E →(E (1))	{ E.Place = E (1).Place E.Code = E (1).Code }
5. E→id	{ E.Place = id.Place E.Code = null }
6. E → id(1) relop id(2)	{ T = NewTemp(); E.Place = T; Gen (if id(1).Place relop id(2).Place goto NextQuad + 3) Gen(T = 0) Gen(goto NextQuad + 2) Gen(T = 1) }

Run Time System

- At run time, the source program should be mapped to storage in the machine.
- This allocation and de-allocation of memory is handled by a **run time support system** typically linked and loaded along with the compiled target code.

Storage Organisation

- It deals with the division of memory into different areas, management of the activation records and layout of the local data.

- A. Subdivision of run-time memory
- B. Activation Records
- C. Compile time layout of the local data.

Subdivision of run-time memory

- **Code Area:** This contains the generated target code.
- **Static Area:** This contains data whose absolute address can be determined at compile time.
- **Stack Area:** This area is for data objects of procedures which can have more than one active procedure at the same time.
- **Heap Area:** This area is for data created at run time. It includes objects pointed to by pointer types.

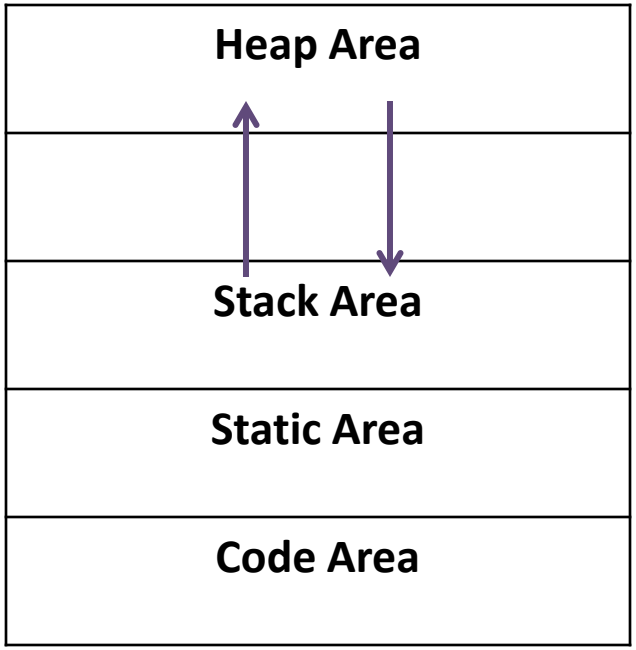

Fig: Initial Division of memory

Activation Records

- An Activation record is a conceptual aggregate of data which contains all information required for a single activation of a procedure.

- Activation records get pushed onto stack when a procedure is called and get popped when a procedure is returned.

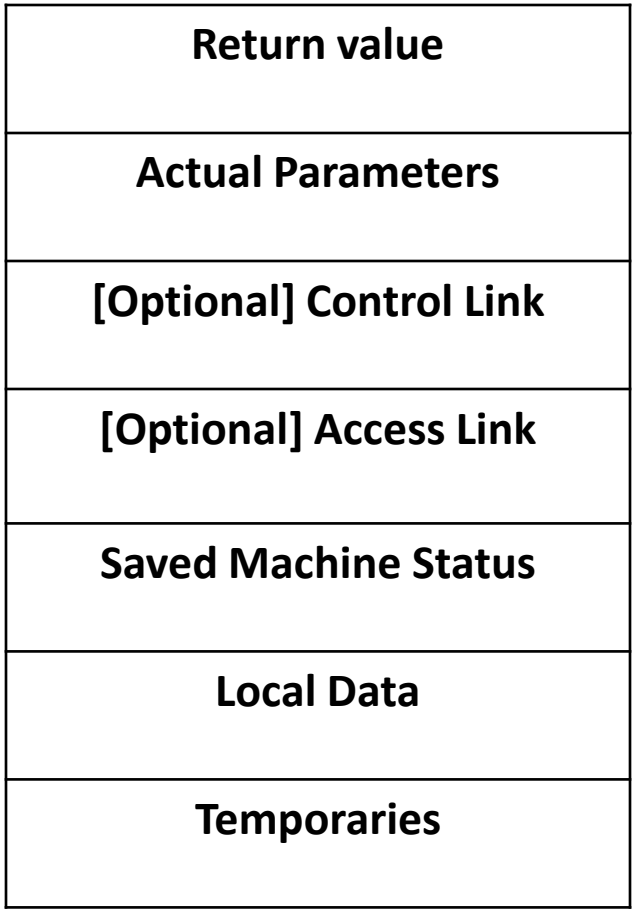

Fig: Format of Activation Record

- **Local Variables:** These variables are a part of the local environment of the currently activated procedure.

- **Parameters/Temporaries:** These are also part of the local environment of the currently activated procedure.

- **Return address:** To return to the appropriate control point of the calling procedure a return address is used.

- **Saved registers/ saved machine status:** If the called procedure wants to use the registers used by the calling procedures, these have to be saved before and restored after the execution of the called procedures.

- **Static Link/access Link:** This link is required in languages which have nesting structure like Pascal, C etc. A procedure in such a language may access variables not locally declared i.e. Variable which are global or declared in surrounding procedures.
- The static link of the procedure points to the latest activation record of the immediately enclosing procedure.

- **Dynamic Link/Control Link:** It is a pointer to activation record of calling procedure.

- **Returned Value:** It is used to store the result of a function call.

Compile time layout of the local data

Type	Size
char	8
Short	16
Float	32
int	32
double	64
char pointer	32
structures	>=8

Example 8.1: Consider the following 'C' program:

```
int  g_var;
main()
{
    int a[100];
    for(int i=0;i<100;i++)
    scanf("%d", &a[i]);
    float avgval = avg(a);
    float stdev = srd(a);
    printf("%f,%f", avgval,stdev);
}
float  avgval(int a[100])
{
    int sum=0;
    for(int i=0;i<100;i++)
        sum+=a[i];
    float avg = sum/100;
    return avg;
}
float std(int a[100])
{
    int sum=0;
    float av = avgval(a);/*position of return after calling
    avgval represented by(*)*/
    for(int i=0;i<100;i++)
        sum+=(a[i]-av)* (a[i]-av);
    float st= sum/100;
    return st;
}
```

The activation record of **avgval** during its invocation from **std** is shown in fig 8.4.

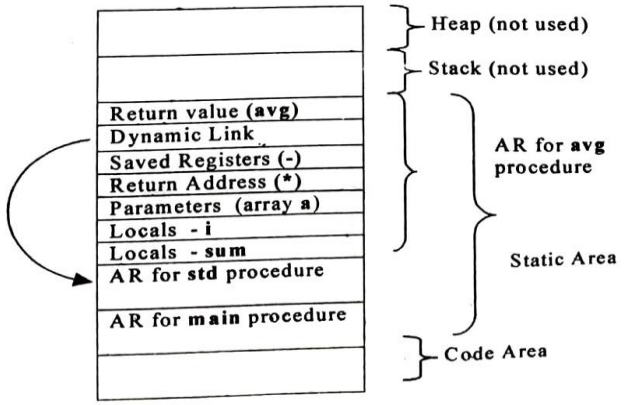

Fig 8.4 Contents of Activation Record

Storage Allocation

- Different types of storage allocation schemes are followed.

1. Static allocation
2. Dynamic Allocation
3. Stack allocation
4. Heap allocation
5. Linked List Allocation
6. Contiguous Allocation
7. Paging and Segmentation
8. Buddy System

1.Static Allocation

- In static allocation, memory is allocated to data structures at compile-time or load-time.
- The size and location of each data structure are fixed and do not change during the program's execution.
- It is simple and efficient but may lead to memory wastage if space is reserved but not fully utilized.

2. Dynamic Allocation:

- Dynamic allocation involves allocating memory at runtime.
- Memory is allocated and released as needed during program execution.
- Common dynamic allocation methods include heap allocation and stack allocation.

3. Stack Allocation:

- The stack is a region of memory used for function call management and local variable storage.
- Memory is allocated and released in a last-in-first-out (LIFO) manner.
- Stack allocation is relatively efficient but limited in size, and its memory is automatically released when variables go out of scope.

4. Heap Allocation:

- The heap is a region of memory used for dynamic memory allocation.
- Memory is allocated on the heap using functions like malloc and released with functions like free (in C/C++) or garbage collection in languages like Java and C#.
- Heap allocation is more flexible than stack allocation but requires explicit management and can lead to issues like memory leaks if not handled properly.

5. Linked List Allocation:

- In linked list allocation, memory is divided into blocks, and each block is managed as a linked list.
- This allows for dynamic allocation and de-allocation of variable-sized memory blocks.
- It can be more memory-efficient than heap allocation for small objects but may suffer from fragmentation issues.

6. Contiguous Allocation:

- Contiguous allocation involves allocating a single contiguous block of memory for a data structure.
- It is commonly used for files and processes in operating systems.
- Fragmentation can be an issue, with both external (unused space between allocated blocks) and internal (unused space within allocated blocks) fragmentation.

7. Paging and Segmentation:

- These memory allocation schemes are commonly used in virtual memory systems.
- Paging divides memory into fixed-sized pages, and segmentation divides memory into logical segments.
- They provide better memory management and protection but can be more complex to implement.

8. Buddy System:

- The buddy system is used in binary buddy memory allocation.
- It involves dividing memory into blocks of various sizes that are powers of 2.
- When a request is made, the system allocates the nearest larger available block or splits a larger block into two smaller "buddy" blocks.

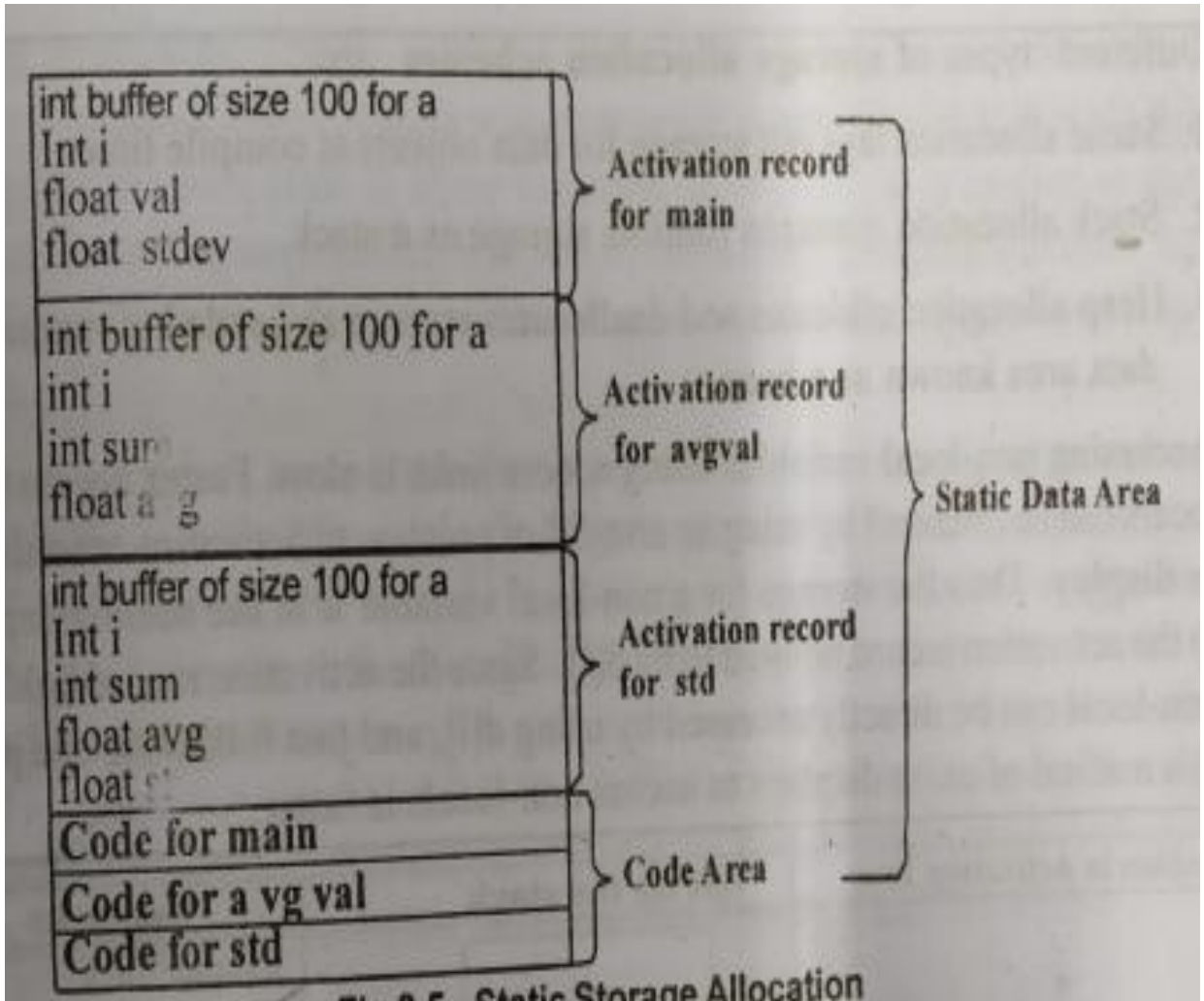

Fig 8.5 Static Storage Allocation

Parameter Passing

- 1. Call by Value
- 2. Call by Reference
- 3. Copy Restore
- 4. Call by Name

- Actual Parameter:
  - Variable specified in the function call
- Formal Parameter:
  - Variables declared in the definition of called function.
- l-value:
  - Refers to the storage
- r-value:
  - Value contained in the storage

Call By Value

- Caller evaluates the actual Parameters
- Passes r-value of actual parameters to formal.

Example:

```
void swap(int x, int y)
{
    int t;
    t=x;
    x=y;
    y=t;
}
void main()
{
    int a=1,b=2;
    swap(a, b);
    printf(“a=%d, b=%d”, a, b);
}
```

Call by reference

- “call by address” or “call by location”
- Caller evaluates actual paramaters
- Passes l-value to formals

Example:

```
void swap(int *x, int *y)
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}
void main()
{
    int a=1,b=2;
    swap(&a, &b);
    printf(“a=%d, b=%d”, a, b);
}
```

Copy Restore

- Hybrid between call by value and call by reference.
- “copy-in copy-out” or “value-result”
- Caller evaluates actual & passes r-value to formals
- l-value of actual parameters are determined before the call.
- *When control returns, r-value of formals are copied back into l-values of the actuals.*

Example:

```
    main()
{
    int a=10,b=20;
    copyrestore_swap(a,b);
    print(a,b);
}
copyrestore_swap(int c, int d)
{
    int temp;
    temp=c;
    c=d;
    d=temp;
}
```

Call by name

- Actual parameters are literally substituted for the formals
- Macro expansion or inline expansion.

Example:

```
int i=100;
void callbyname(int x)
{
	print(x);
	print(x)
}
main()
{
	callbyname(i=i+1);
}
```