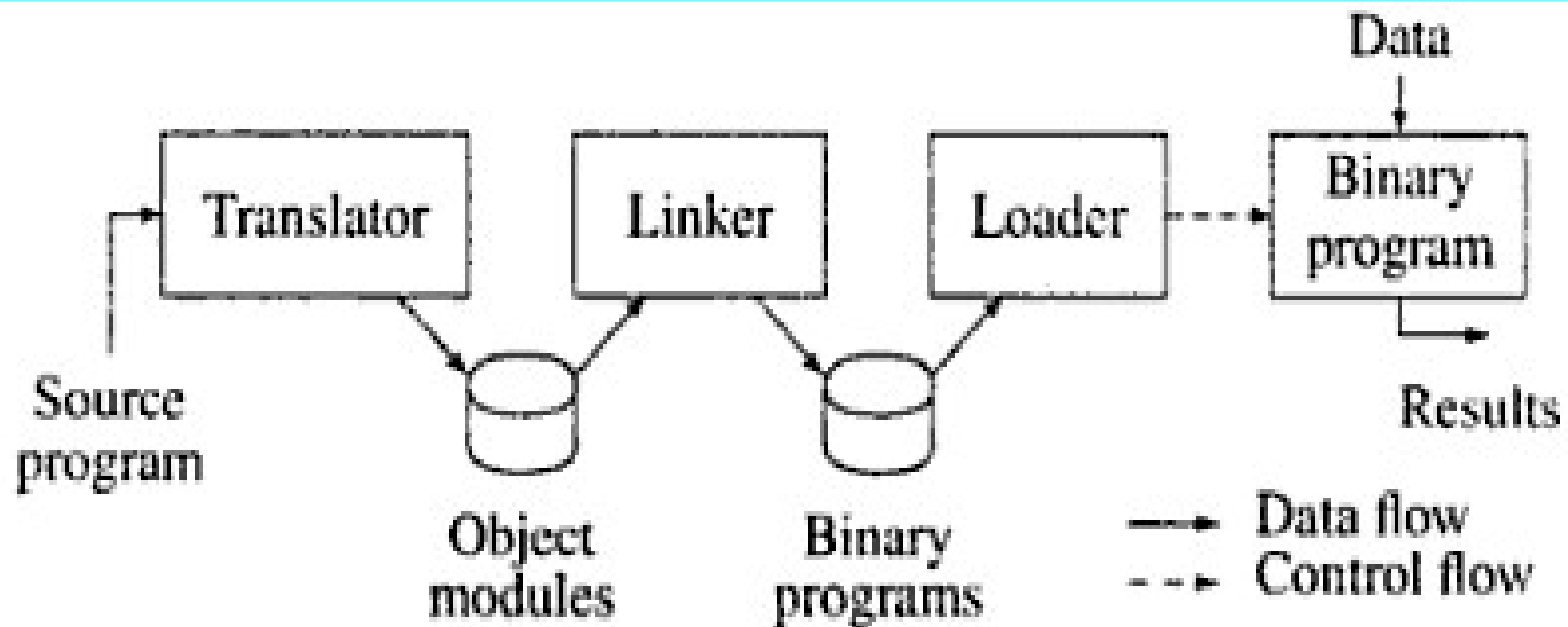


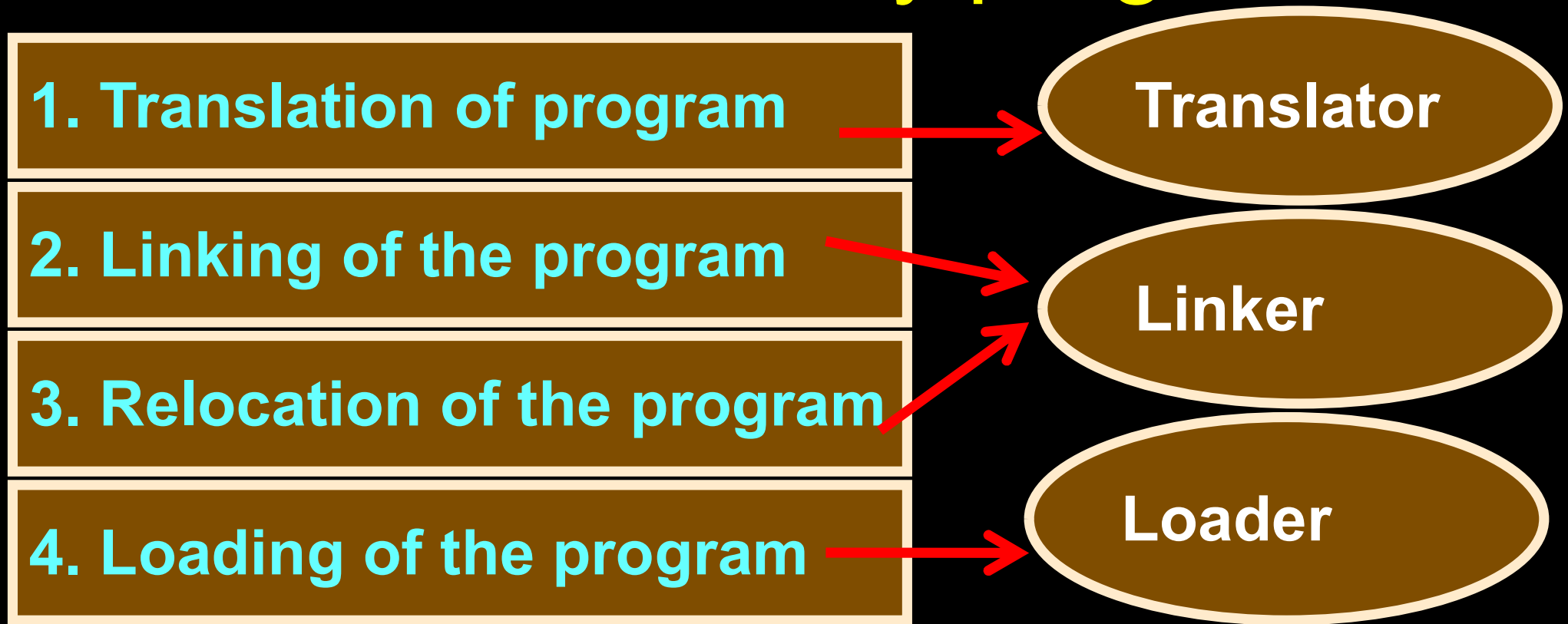
# Chapter 5

## Linkers and Loaders

# Execution of a program



# Execution of any program



# Four steps for execution

- **Translation** : A program is translated into a target program
- **Linking** : Code of target program is combined with codes of those programs and library routines that it calls.
- **Relocation** : is the action of changing the memory address used in the code of the program so that it can execute correctly in the allocated memory area.
- **Loading** : the program is loaded in a specific memory area for execution

- **Translator** – generates a program form called the object module for the program.
- **object module** - which contains target code and information about other programs and library routines

- **Linker** - program performs linking and relocation of a set of object modules to produce a ready-to-execute program form called a **binary program**.
- **Loader** – program loads a binary program in memory for execution.

# Translated, linked and load time addresses

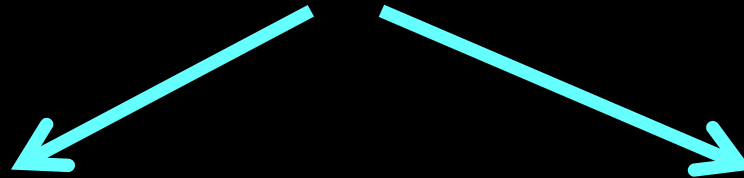
- While compiling the program P, a translator is given an origin specification for P.
- That is called the **translated origin** of p.
- Translator uses the values to perform memory allocation of the symbols declared in P

**Translation time address tsymb**

- **Execution start address** – is the address of the instruction from which its execution must begin.
- **Translated start address** – the start address specified by the translator.



- The origin of the program may have to be changed for one of the two reasons.



**1. The same set of translated addresses may have been used in different object module**

**2. OS may require that a program should execute from the specific memory location**

- The change in the origin of a program leads to changes in its *execution start address*  
and  
in the addresses assigned to the symbols defined in it.

What happens when we change the origin of the program?

1. execution start address changes( jya thi start thai ne tamaru execution)
2. addresses assigned to the symbols defined in it.

# Terminology used to refer address of program

1. *Translation time (or translated) address*: Address assigned by the translator.
2. *Linked address*: Address assigned by the linker.
3. *Load time (or load) address*: Address assigned by the loader.

1. *Translated origin*: Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
2. *Linked origin*: Address of the origin assigned by the linker while producing a binary program.
3. *Load origin*: Address of the origin assigned by the loader while loading the program for execution.



## TRANSLATED ORIGIN

1. kone address apyu 6e = programmer
2. kyare aa address ape 6e = in the ORIGIN statement of the source code
3. konu address apvanu 6e = address of origin

# Example – translated origin

## LOAD ORIGIN

1. kone address apyu 6e = loader
2. kyare aa address ape 6e = while loading the program for execution
3. konu address apvanu 6e = address of origin

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START 500		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
LOOP	READ A	500)	+ 09 0 540
	:	501)	
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	:		
	BC LT, LOOP	538)	+ 06 1 501
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

## LINKED ORIGIN

1. kone address apyu 6e = linker
2. kyare aa address ape 6e = while producing the binary program
3. konu address apvanu 6e = address of origin

# Example

## Translation origin of program

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START	500	
	ENTRY	TOTAL	
	EXTRN	MAX, ALPHA	
	READ	A	500) + 09 0 540
LOOP		501)	
	:		
	MOVER	AREG, ALPHA	518) + 04 1 000
	BC	ANY, MAX	519) + 06 6 000
	:		
	BC	LT, LOOP	538) + 06 1 501
	STOP		539) + 00 0 000
A	DS	1	540)
TOTAL	DS	1	541)
	END		

## Translation time address of Loop

# Address Sensitive Instruction

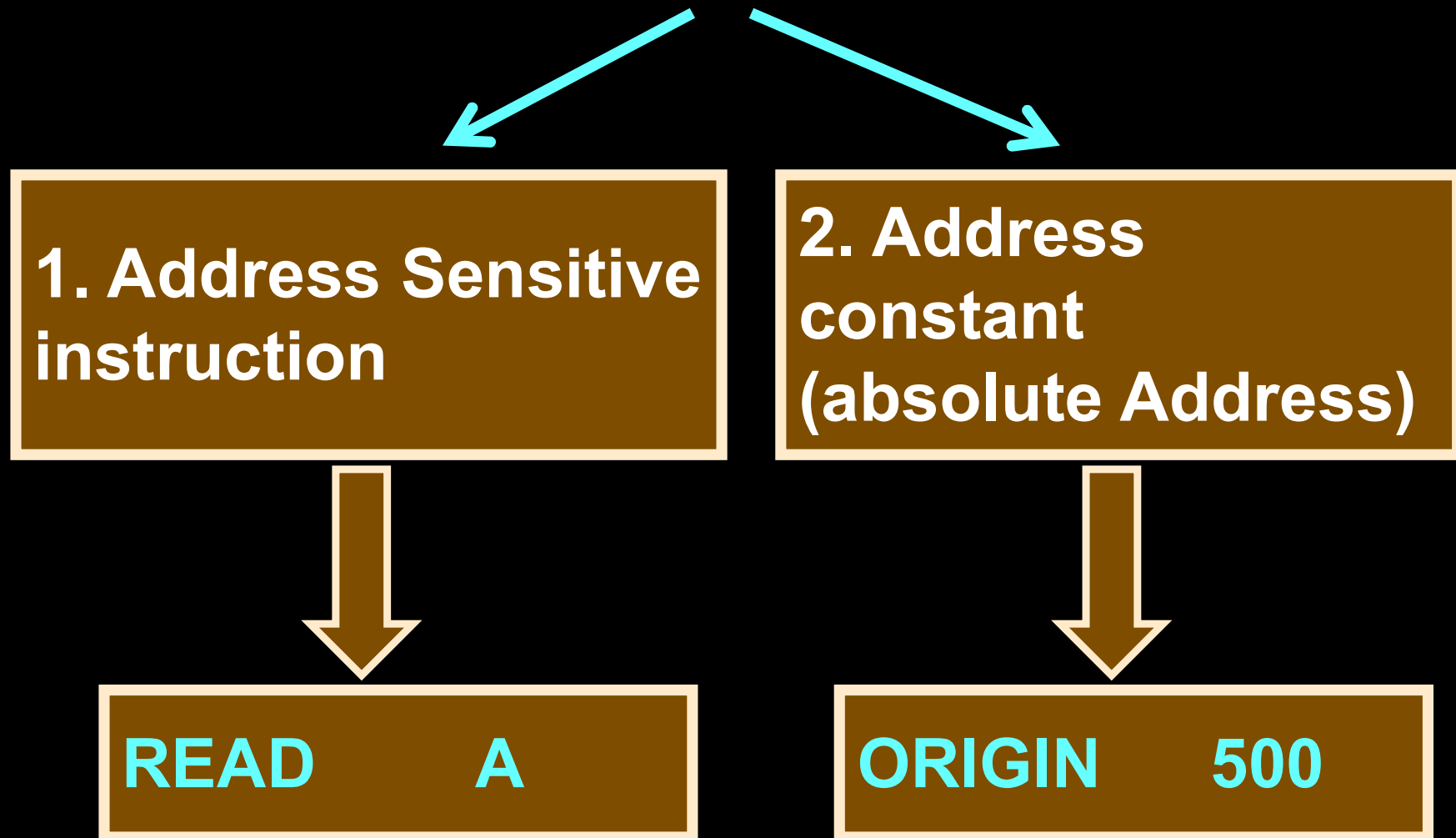
	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START <del>500</del> <sup>900</sup>		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
	READ A	500)	+ 09 0 540 <sup>940</sup>
LOOP		501)	
	⋮		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	⋮		
	BC LT, LOOP	538)	+ 06 1 501 <sup>901</sup>
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

If the linked origin is 900, then what about the address of A and LOOP?

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START 500		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
	READ A	500)	+ 09 0 540
LOOP		501)	
	:		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	:		
	BC LT, LOOP	538)	+ 06 1 501
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

It should be corrected to 940 and 901 respectively if linked origin is 900

# Address sensitive program contains...





- An address sensitive program can only execute correctly if the start address of the memory area allocated to it is same as the translated origin

ADDRESS SENSITIVE PROGRAM  
hu pan tya avi = translated origin  
tame pan tya j hata = start address of the memory area allocated.

- To execute correctly from any other memory area, all the address sensitive instructions need to be **corrected** To su karvu joie?
- That's called **program relocation**

**Definition 7.1 (Program relocation)** Program relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.

If linked origin  
and translated  
origin **are same**,  
then...



No need to perform  
**Relocation**

If linked origin  
and translated  
origin **are not  
same**, then...



**Relocation**  
Is performed by **linker**

# Relocation process

- $t\_origin$  = translated origin hal to ahiya 6e.
- $l\_origin$  = linked origin ahiya to hovu joie
- $Symb$  = any symbol in program P
- $T\_symb$  = translation time address
- $L\_symb$  = link time address
- $Relocation\_Factor = l\_origin - t\_origin$

ketle javanu baki 6e.

$$relocation\_factor_p = l\_origin_p - t\_origin_p$$

$$l_{symb} = t\_origin_p + d_{symb}$$

$$l_{symb} = l\_origin_p + d_{symb}$$

$$l_{symb} = t\_origin_p + relocation\_factor_p + d_{symb}$$

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START 500		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
	READ A	500)	+ 09 0 <del>540</del>
LOOP		501)	
	⋮		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	⋮		
	BC LT, LOOP	538)	+ 06 1 <del>501</del>
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

- Relocation factor=  $900-500 = 400$
- Now address contained in operand field 540 is now  $540 + 400 = 940$
- The instruction with translated address 538 contains the address 501 in the operand field. Adding 400 to this address makes it 901.
- who does the designation of the set of instructions? Let  $IRR_P$  designate the set of instructions requiring relocation in program P. Following formula, relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address(es) in every instruction *i belongs to*  $IRR_P$  .

$IRR_P$  = badhi j aevi instructions ke jemne relocation joie 6e.

to kevi ritna karisu.... relocation factor + translation time addresses ne add karine.  
 $= \text{relocation\_factor} + t\_symbol = \text{relocation\_factor} + t\_origin + d\_symbol$

# Linking

- A program unit is any program or routine that is to be linked with another program or routine.
- Let an application consists of a set or program units SP.
- Program unit  $P_i$  that requires another program unit  $P_j$  during its execution.
- As it uses the address of some instruction of in  $P_j$  in one of the instructions,

- To form a binary program combining  $P_i$  and  $P_j$ ,
- It is achieved by following linking concepts.

Linking concepts are based on?

1.where it is defined 2.where it is been referenced 3.which keyword is used to do it.

- **Public definition** : A symbol defined in a program unit that may be referenced in other program units
- **External References** : A reference to a symbol that is not defined in the program unit containing the reference.



# Example

	<u>Statement</u>		<u>Address</u>	<u>Code</u>
	START	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
	READ	A	500)	+ 09 0 540
LOOP			501)	
	:			
	MOVER	AREG, ALPHA	518)	+ 04 1 000
	BC	ANY, MAX	519)	+ 06 6 000
	:			
	BC	LT, LOOP	538)	+ 06 1 501
	STOP		539)	+ 00 0 000
A	DS	1	540)	
TOTAL	DS	1	541)	
	END			

# ENTRY statement

- Lists the public definition
- Which may be referenced in other program units

	<u>Statement</u>		<u>Address</u>		<u>Code</u>
	START	200			
	ENTRY	ALPHA			
	- -				
	- -				
ALPHA	DS	25	231)	+	00 0 025
	END				

	<u>Statement</u>		<u>Address</u>	<u>Code</u>
	START	200		
	ENTRY	ALPHA		
	- -			
	- -			
ALPHA	DS	25	231)	+ 00 0 025
	END			

	<u>Statement</u>		<u>Address</u>	<u>Code</u>
	START	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
	READ	A	500)	+ 09 0 540
LOOP			501)	
	:			
	:			
	MOVER	AREG, ALPHA	518)	+ 04 1 000
	BC	ANY, MAX	519)	+ 06 6 000
	:			
	:			
	BC	LT, LOOP	538)	+ 06 1 501
	STOP		539)	+ 00 0 000
A	DS	1	540)	
TOTAL	DS	1	541)	
	END			

# EXTERN statement

- Lists the symbols to which **external references are made** in the program unit

	<u>Statement</u>	<u>Address</u>	<u>Code</u>
	START 500		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
	READ A	500)	+ 09 0 540
LOOP		501)	
	⋮		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	⋮		
	BC LT, LOOP	538)	+ 06 1 501
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

# Resolving external references

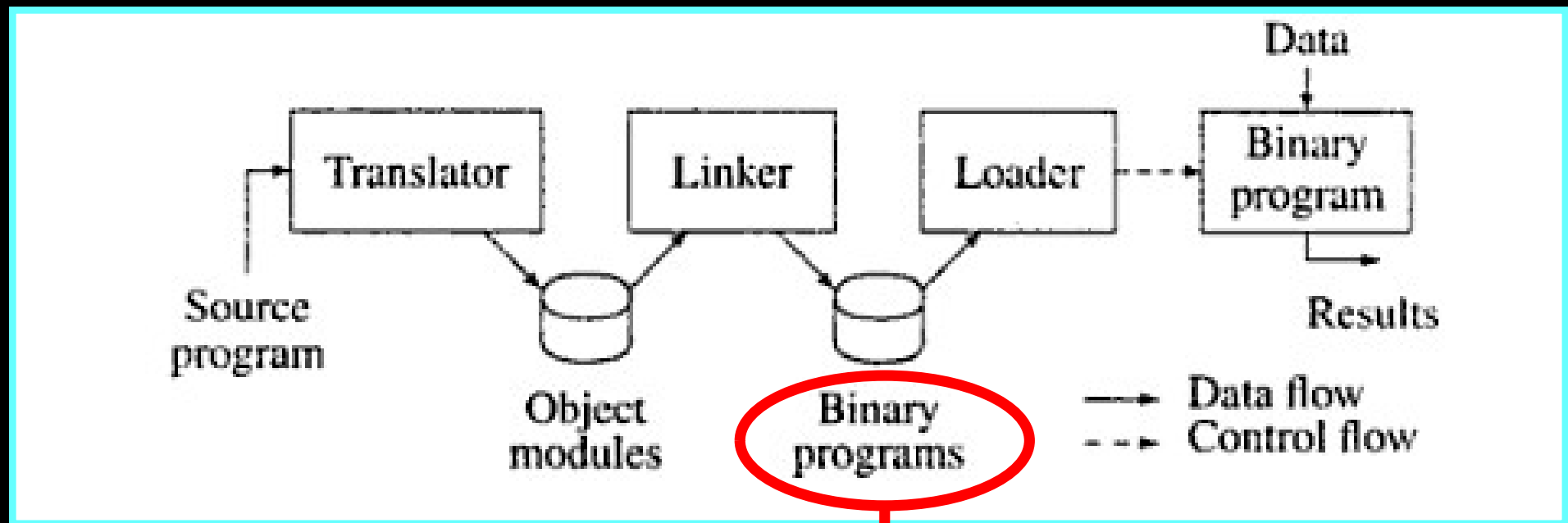
- **Linking** is the action of putting the correct linked addresses in those instructions of a program that contain external references.
- An external reference in an instruction is said to become **resolved** when the correct linked address has been put in the instruction.

- Program P contains external ref to ALPHA in the instruction with Translation time address 518
- Linked origin of P is 900
- Its size is 42 words

what is the Q program?

- Hence linked origin of Q program is 942
- And linked address of ALPHA is 973
- External ref in the inst. Address 518 is resolved by putting the linked address of ALPHA i.e. 973

# Execution of the program



**Ready to execute form of program**

# Binary Program

**Definition 7.3 (Binary program)** *A binary program is a machine language program comprising a set of program units  $SP$  such that  $\forall P_i \in SP$*

1.  *$P_i$  has been relocated to the memory area starting at its link origin, and*
2. *Linking has been performed for each external reference in  $P_i$ .*



from where we form a binary program?

- To form a binary program from a set of object modules. The programmer invokes linker by using the command how to form a binary program

- Linker <link origin> , <object module names>

[,<execution start address>]

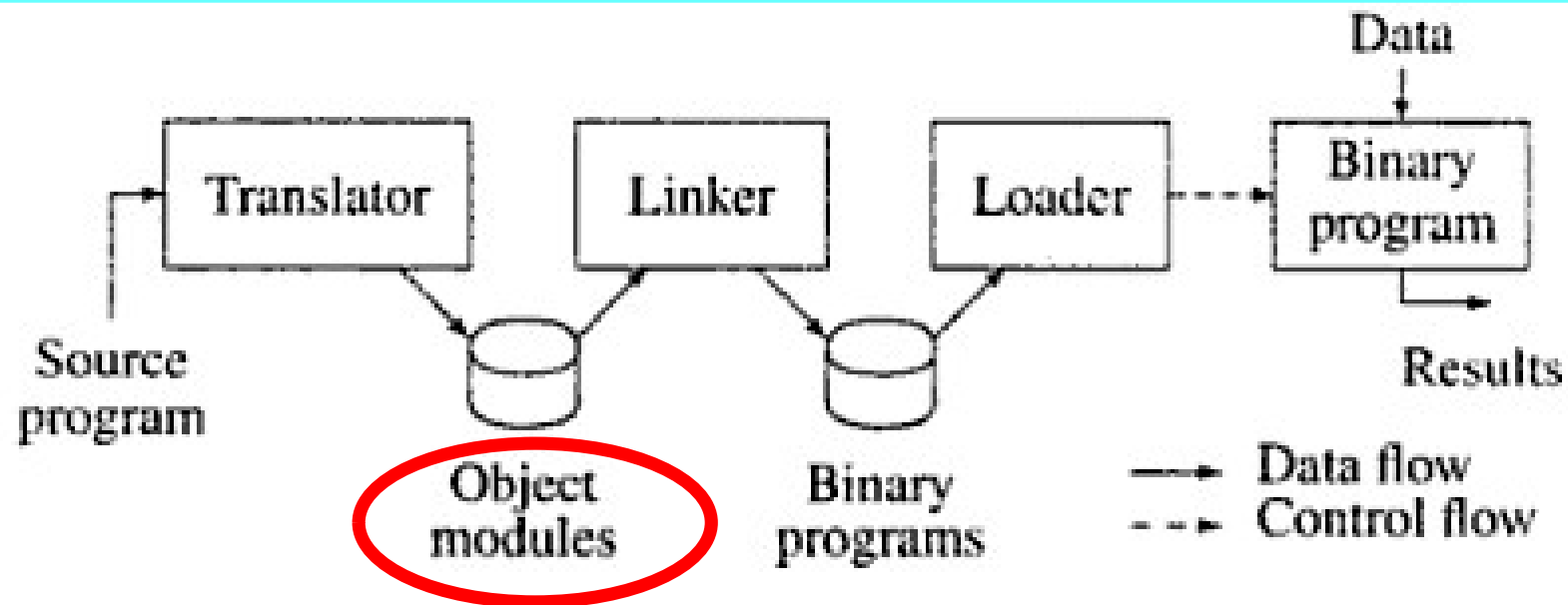
<link origin> - specified the memory address to be given to the first word of <sup>33</sup>

- <execution start address> - is omitted as since we have assumed that load origin = linked origin.
- Linker 900 P,Q

# Object Module

- The object module of a program unit contains all the information that would be needed to relocate and link the program unit with other program units.

# Execution of the program



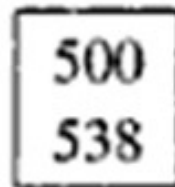
# Object module of program P - four components

1. *Header*: The header contains *translated origin*, *size* and *execution start address* of P.

2. *Program*: This component contains the machine language program corresponding to P.

3. *Relocation table*: (RELOCTAB) This table describes  $IRR_P$ . Each RELOCTAB entry contains a single field:

*Translated address* : Translated address of an address sensitive instruction.



500  
538

4. *Linking table (LINKTAB)*: This table contains information concerning the public definitions and external references in P.

Each LINKTAB entry contains three fields:

- Symbol* : Symbolic name
- Type* : PD/EXT indicating whether public definition or external reference
- Translated address* : For a public definition, this is the address of the first memory word allocated to the symbol. For an external reference, it is the address of the memory word which is required to contain the address of the symbol.

ALPHA	EXT	518
MAX	EXT	519
A	PD	540

# For previous example

1. *translated origin = 500, size = 42, execution start address = 500.*
2. Machine language instructions shown in Fig. 7.2.
3. Relocation table

500
538

4. Linking table

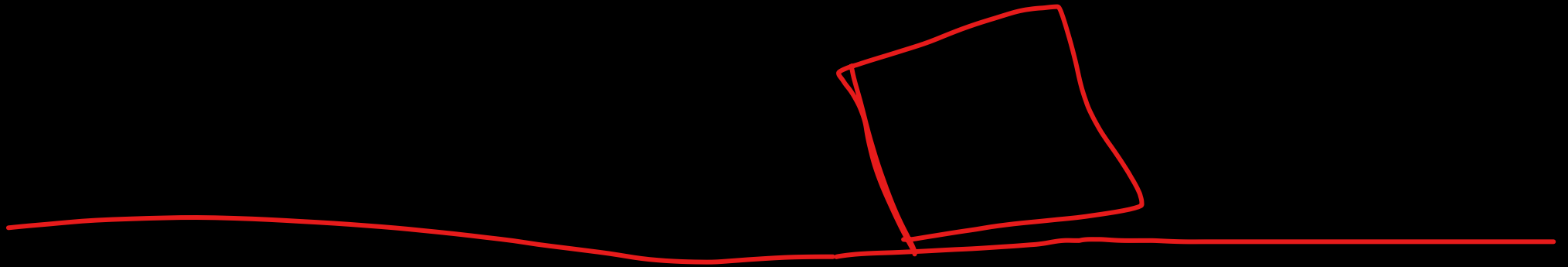
ALPHA	EXT	518
MAX	EXT	519
A	PD	540

# Algorithm: Program Relocation

1.  $program\_linked\_origin := \langle link\ origin \rangle$  from linker command;
2. For each object module
  - (a)  $t\_origin := translated\ origin$  of the object module;  
 $OM\_size := size$  of the object module;
  - (b)  $relocation\_factor := program\_linked\_origin - t\_origin$ ;
  - (c) Read the machine language program in  $work\_area$ .
  - (d) Read RELOCTAB of the object module.
  - (e) For each entry in RELOCTAB
    - (i)  $translated\_addr :=$  address in the RELOCTAB entry;
    - (ii)  $address\_in\_work\_area :=$  address of  $work\_area$  +  
 $translated\_address - t\_origin$ ;
    - (iii) Add  $relocation\_factor$  to the operand address in the word  
with the address  $address\_in\_work\_area$ .
- (f)  $program\_linked\_origin := program\_linked\_origin + OM\_size$ ;



# Program linking algorithm



su- su avi sake 6e ? = symbolic name ...konu?= external reference or an object module.

# NTAB – name Table

*Symbol* : symbolic name of an external reference or an object module.

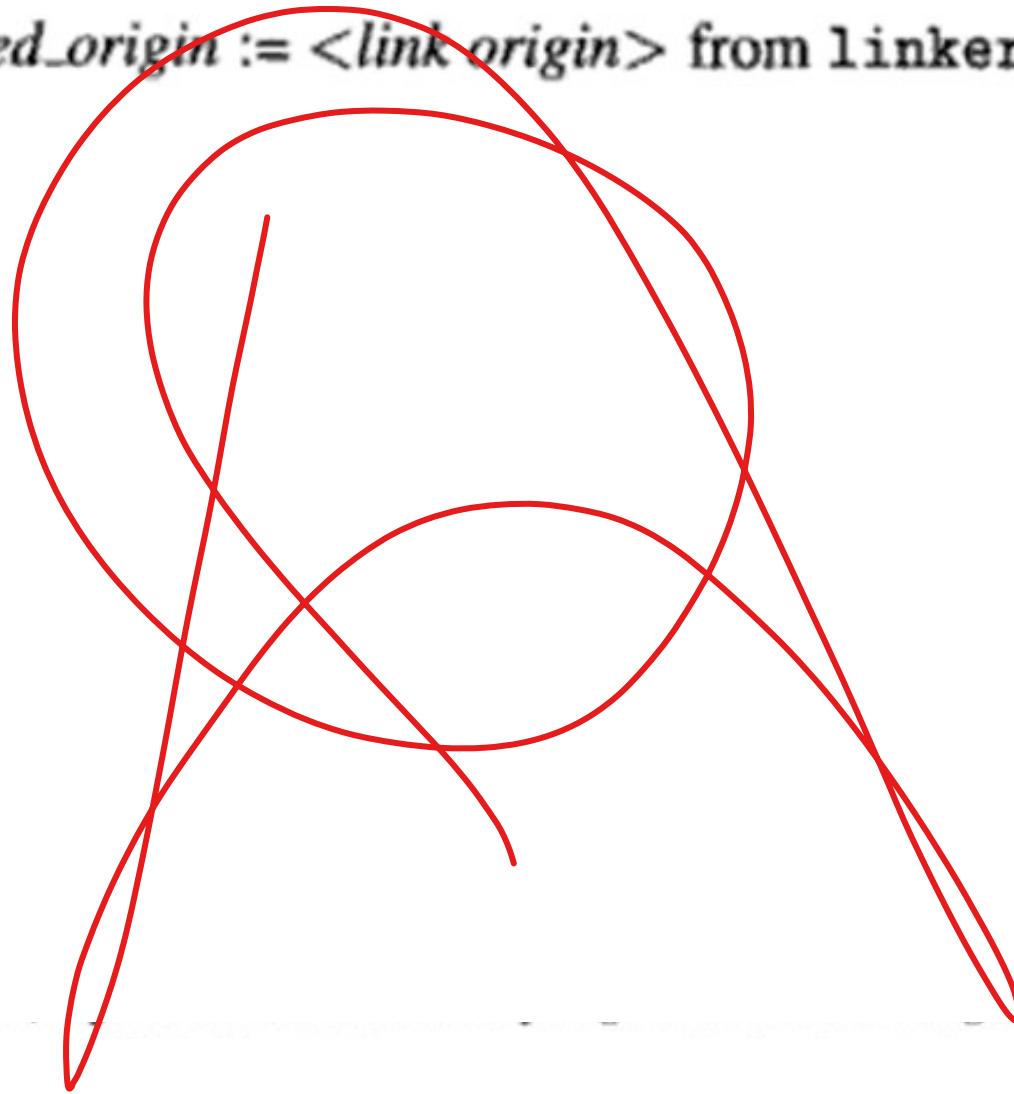
*Linked\_address* : For a public definition, this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.

Linked address : .  
for public definition ->  
this field contains the linked address  
of the symbol  
for an object module ->  
this field contains the linked origin  
of the object module

<i>symbol</i>	<i>linked address</i>
P	900
A	940
Q	942
ALPHA	973

## Algorithm 7.2 (Program Linking)

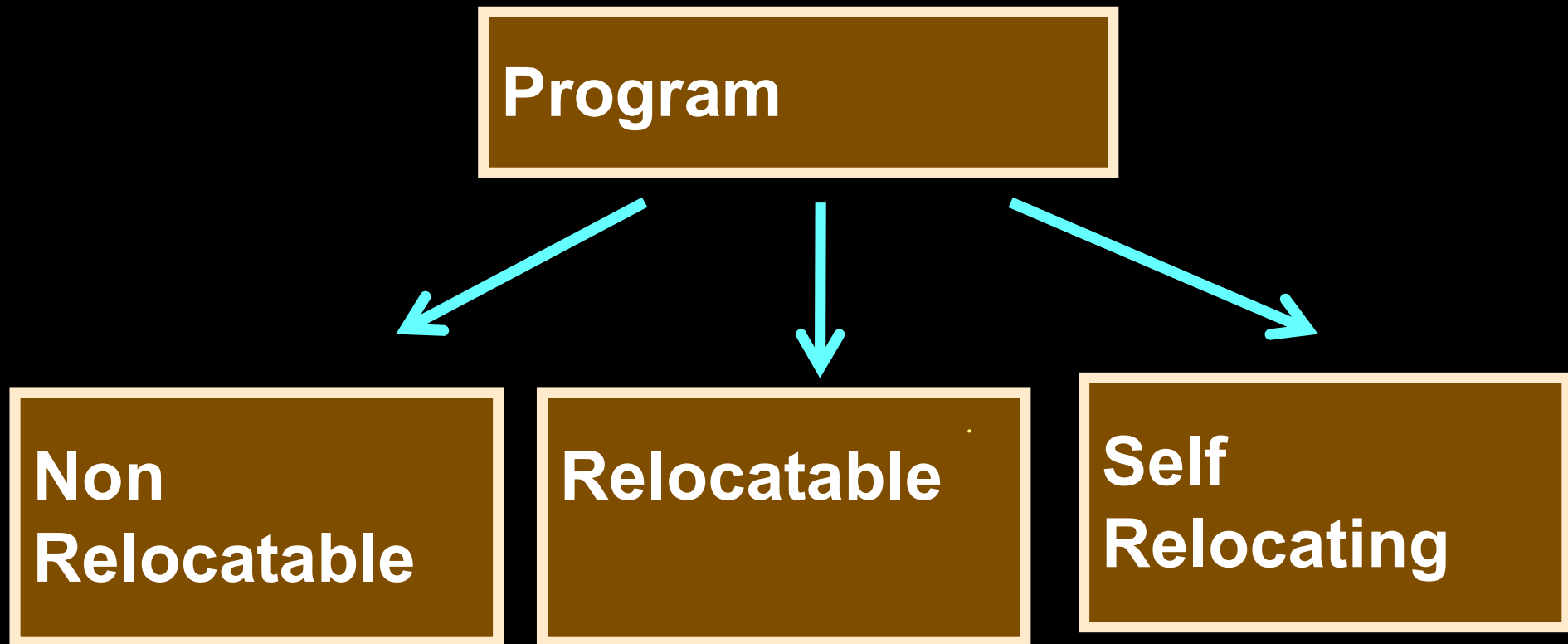
1. *program\_linked\_origin* := <link origin> from linker command.



3. For each object module

	<u>Statement</u>		<u>Address</u>	<u>Code</u>
	START	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
LOOP	READ	A	500)	+ 09 0 540
			501)	
	:			
	MOVER	AREG, ALPHA	518)	+ 04 1 000
	BC	ANY, MAX	519)	+ 06 6 000
	:			
	BC	LT, LOOP	538)	+ 06 1 501
	STOP		539)	+ 00 0 000
A	DS	1	540)	
TOTAL	DS	1	541)	
	END			

	<u>Statement</u>		<u>Address</u>	<u>Code</u>
	START	200		
	ENTRY	ALPHA		
	- -			
	- -			
ALPHA	DS	25	231)	+ 00 0 025
	END			



i.e linked nathi thai sakta ...translated origin thi j start karvu padse.

# Non-relocatable program

- **Non- relocatable** : program can not be executed in any memory area other than the starting from its translated origin
  - e.g. hard coded machine language program; information concerning its address sensitive instructions is not available.

1. hard coded machine language programs
2. ae program na address sensitive instructions na related ni information available nathi hoti.

- **Relocatable program** : The program can be relocated by a linker or loader to have linked address or load address
  - An object module can be relocated because it contains information about address sensitive instructions.
- **A self-relocating program** : The program can be loaded in any area of memory for execution.
  - At the start of execution it would perform its own relocation



- A self-relocating program is designed to have following two components :
  - A table of information concerning address sensitive instructions. This table resembles the RELOCTAB. *ek aevo table je jena andar badhi address sensitive information available rehse.*
  - Code to perform the relocation of address sensitive instructions ; this code is called the relocating logic.
    - A start address of relocating logic is specified as the execution start address of the program.

*1. and aenu relocation kevi ritna karso ??=>*

*to aena mate hase relocating logic...*

*2. kaya form ma relocating logic hase ??= code na forme ma*

*3.what is the start address of the relocating logic ??= execution start address of the program.  
(kya thi tame start karso relocation ne perform karvanu).*

# Architecture of Intel 8088

- Data registers – AX, BX, CX, DX
- Index registers – SI, DI
- Stack pointer register – BP and SP
- Segment register Code, Stack, Data and Extra

- Data register – 16 bits – upper and lower halves
- Index register – used to index source and destination address in string manipulation
- SP – points into the stack used to store subroutine and interrupt return address
- BP – can be used by programmer

# Segment based addressing

- Memory is not accessed through absolute address but through address contained in segment registers
- A program may contain many components – called segments
- Each segment may contain a part of the program's code, data or stack

- CS, DS and SS registers are used to contain start address of program's code, data and stack, respectively
- Extra segment (ES) can be used to contain address of any other memory area.
- An instruction uses a segment register and a 16 bit offset to address a memory operand.

# Addressing modes

- 24 addressing modes
- 1) immediate addressing mode : the instruction itself contains the data
- MOV SUM, 1234H
- 2) Direct addressing mode : may contain a 16 bit displacement which is offset from the segment register.
- MOV SUM, [1234H]

- 3) Indexed mode : content of index register added to displacement in instruction, which is taken to be offset from the segment base of data segment
- MOV SUM, 34H[SI]
- 4) Based mode : contents of base register added to displacement and taken to be offset from the segment base of data segment

MOV SUM, 10H[BP]

- 5) Based – and – indexed mode :  
combine effect of both modes
- MOV SUM, 56H[SI] [BX]



# Assembler Directive

- Declarations:
- A DB 25 : reserves a byte & initialize it
- B DW ? : reserves a word but do not initialize
- EQU and PURGE (make name 'undefined')
  - XYZ DB ?
  - ABC EQU XYZ – ABC represents the nameXYZ

# SEGMENT, ENDS

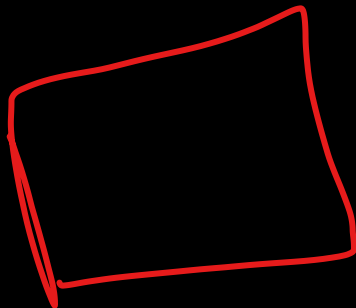
- Start and End of a segment
- ASSUME <segment register> :  
<segment name>
- Informs assembler that the address of segment would be in segment register
- ASSUME DS:SAMPLE\_DATA

agar jo segment register na andar segment nu address hase then to j aena segment na andar no operand tamaro addressable hase else tarmra thi ae to memory operand in the instruction nahi addressable hoi.

- Memory operand in an instruction is addressable only if the address of that segment that contains the operand is present in one of the segment register.

- PROC, ENDP
- Delimit the body of a procedure
- NEAR , FAR
- Appear in operand field of PROC indicate whether the call to the procedure is to be assembler as a near call or far call.
- If it is FAR procedure , need not to be addressable at the CALL statement

- PUBLIC , EXTRN
- PUBLIC – if symbol is to be accessed in other programs
- EXTRN – any other program wishing to use this symbol must specify it in an EXTRN directive



# Linking in MS DOS

- Segment based addressing has two components – address of segment and offset within the segment
- No absolute addressing is used , so instructions are not address sensitive
- It reduces relocation requirement of a program
- Only value of segment register is changed during relocation and effective address is found

- It does not completely eliminate need for relocation
- Instructions like `MOV AX,DATA_HERE`
- `DATA_HERE` is name of segment
- Assembler does not know linked address of `DATA_HERE`
- So it uses `RELOCTAB`
- Inter-segment calls and jumps also handled same way

# To handle such problem

- Solution : assembler use MOV instruction in the immediate operand format,
- so that it could put 16 bits of the address of DATA\_HERE as the immediate operand
- Puts zero in the operand field
- Makes entry in RELOCTAB so that linker would put the appropriate address in operand field



# Object Module Format

- An object module is a sequence of object records
- 14 types of object records –five kinds of info
  - 1) Binary image
  - 2) External references
  - 3) Public Definitions
  - 4) Debugging information like line number in source program
  - 5) Miscellaneous information like comments

# Some object record type

- Record type    Id(Hex)    Description
- THEADR 80        Translator header record
- LNames 96        List of names record
- SEGDEF 99        Segment definition record
- EXTDEF 8C        External names def rec
- PUBDEF 91        Public names def rec
- LEDATA A1        Enumerated data (bin.img)
- LIDATA            A3        Repeated data (bin.  
Img)

- FIXUPP      9D    relocation record
- MODEND    8B    Module end record

# THEADR, L NAMES and SEGDEF record

- The module name in the THEADR record is typically derived by the translator from the source file name.
- This name is used by the linker to report errors. An assembly programmer can specify the module name in the NAME directives.
- L NAMES record lists the names for use by SEGDEF record.
- A SEGDEF record designated a segment name using an index into this list.
- The attributes field of a SEGDEF record indicates whether the segment is relocatable or absolute, whether it can be combined with other segments, as also the alignment requirement of its base address.
- Stack segments with the same name are concatenated with each other, while common segments with the same name are overlapped with one another. The attribute field also contains the origin specification for an absolute segment.

- THEADR record

80H	length	T-module name	check-sum
-----	--------	---------------	-----------

- LNAMEs record

96H	length	name list	check-sum
-----	--------	-----------	-----------

- SEGDEF record

98H	length	attributes (1-4)	segment length (2)	name index (1)	check-sum
-----	--------	---------------------	-----------------------	-------------------	-----------

# EXTDEF and PUBDEF records

- The EXTDEF record contains a list of external references used by the program of this module.
- A PUBDEF record contains a list of public names declared in a segment of object module. *base segment su specify kare 6e.*
- The base specification identifies the segment. Each (name, offset) pair in the record defines one public name, specifying the name of the symbol and its offset within the segment designated by the base specification. *kone apyu*
- EXTDEF record

8CH	length	external reference list	check-sum
-----	--------	-------------------------	-----------

- PUBDEF record

90H	length	base (2-4)	name	offset (2)	...	check-sum
-----	--------	---------------	------	---------------	-----	-----------

## LEDATA records

- An LEDATA record contains the binary image of the code generated by the language translator. Segment index identifies the segment to which the code belongs and offset specifies the location of the code within the segment.

<b>A0H</b>	<b>length</b>	<b>segment index</b> (1-2)	<b>data offset</b> (2)	<b>data</b>	<b>check-sum</b>
------------	---------------	-------------------------------	---------------------------	-------------	------------------

## MODEND records

- The MODEND record signifies the end of the module, with the type field indicating whether it is the main program.
- This record also optionally indicates the execution start address. This has two components:
  - a) the segment, designated as an index into the list of the segment names defined in SEGDEF record(s)
  - b) an offset within the segment.

<b>8AH</b>	<b>length</b>	<b>type</b> (1)	<b>start addr</b> (5)	<b>check-sum</b>
------------	---------------	--------------------	--------------------------	------------------

# FIXUPP records

- A FIXUPP record contains information for one or more relocation and linking fixups to be performed. The locat field contains a numeric code called **loc code** to indicate the type of a fixup. The meaning of these codes are given in the following table:

Loc code	Meaning
0	low order byte is to be fixed
1	offset is to be fixed
2	segment s to be fixed
3	pointer (i.e., segment : offset) is to be fixed

- locat** also contains the offset of the fixup location in the previous LEDATA record.
- The **frame datum** field, which refers to a SEGDEF record, identifies the segment to which the fixup location belongs.
- The **target datum** and **target offset** fields specify the relocation or linking information.



- Target datum contains a segment index or an external index, while target offset contains an offset from the name indicated in target datum.
- The **fixdat** field indicates the manner in which the target datum and target offset fields are to be interpreted.
- The numeric codes used for this purpose are given as per the following:

see

code	contents of target datum and offset field
0	segment index and displacement
2	external index and target displacement
4	segment index (offset field is not used)
6	external index (offset field is not used)

9CH	length	locat (1)	fixdat (1)	frame datum (1)	target datum (1)	target offset (2)	...	check sum
-----	--------	--------------	---------------	-----------------------	------------------------	-------------------------	-----	--------------

# Static Linking

- In **static linking** , the linker links all modules of a program before its execution begins; it produces binary program with no unresolved external references
- If statically linked programs use the same module from the library, each program will get a copy of the module.

- Now if many programs that use the same module are in execution at the same time, many copies of the module might be present in memory.

linker badhi j modules of a program ne linke kare 6e... kyare = before it's execution begins ....link kare 6e that means su ?= ke it produces a binary code ...ae binary code kevo hovo joiee? = jena andar ek pan unresolved external references nahi hoi.  
..... pan agar jo many programs were using the same module ...to ae j same module ni multiple copies tamara memory na andar rehse.

aeno matlab su 6e....that you awill perform the linking during the time of execution.

# Dynamic Linking

- It is performed during **execution of binary program** linker nu kam kyare padse...jyare we encounter an unresolved external reference.
- The linker is invoked when an **unresolved external reference is encountered** during its execution
- The linker resolves the external reference and **resumes execution of the program** to linker su karse.... ke it will resolve the external reference and will resume the execution of the program.

# Benefits of Dynamic linking

je pan modules nu kam j nathi(not invoke) aemne link karine kai kam pan nathi.

- Modules that are not invoked during execution of program need not be linked to it.
- If a module referenced by a program , has already been linked to another program that is in execution, a copy of the module would exist in memory
  - The same copy of module could be linked to this program as well, thus save memory.

agar jo ae module ne koi program ne link karyu hase...that means ke ae module ni copy tamara memory ma hse and so tamare to just aene link j kari devani 6e.... taki you same the memory and don't end in creasting the mutiple copies of the same module.

- When a library of module is updated – a program that invokes a module of the library automatically starts using the new version of the module !
- DLL (dynamic linked libraries ) use some of these features

Jyare pan tamari library update thase...at that time...to je pan time par tamaro program tamari module ne invoke karse...to ae to automatically tmari updated module ne j link karse...DLL(dynamic linked libraries ) ae aa vado feature use kare 6e.

DLL kevi ritna aa features ne use kare 6e?? =

## How ?

- 1) Each program is first processed by the static linker.
- 2) The static linker links each external reference in the program to a dummy module whose sole function is to call the dynamic linker and pass the name of the external symbol to it.
  - This way dynamic linker would be activated when such external reference is encountered during execution of the program.

darek program si first processed by the static linker => aa static linker link kare 6e each external reference ne in the program ...kya reference kare 6e ?= to a dummy module ..dummy module su 6e?= aeno sole function is to call the dynamic linker.... and aena dynamic linker ne call karya pachi su karvanu? = pass the name of the external symbol to it.

=> this way dynamic linker would be activated when such external reference is encountered during the time of execution.

- It would maintain a table like the name table (NTAB) , which would have public definitions and their load addresses.

static linker su karse...NTAB = public references, load addresses.

If external symbol is present in the table, it would use the *load address of the symbol* to resolve the external reference.

agar jo ae external symbol maro present in the table ne ae to aeno load address na madad thi aene resolve kari dese.



agar jo ae symbol nathi madto then it would go on the search the library of the object modules jena andar aene to sodhse ek kayo module 6e aena andar aa required symbol ni public defination 6e.

=> pachi su karisu ke ae object module ne binary program thi link karisu and then je apn information madi hase aeni public definations mathi aene add karisu table ma.

- Otherwise, it would search the library of object modules to find a module that contains the required symbol as a public definition.
- This object module is linked to binary program and information on its public definitions are added in the table.

# Overlay structured programs

- Some parts of programs may be executed **only briefly** , or not at all, during an execution.
- Memory requirement of a program can be reduced **by not keeping these parts** in memory at **all times**.
- So, some parts of program are given the **same load address during linking**.

- An **overlay** is a part of a program that has the *same load origin* as some other part(s) of the program.
- A program containing overlays is called an **overlay structured program**
- It has : 1) A permanently resident part , called the **root**
- 2) A **set of overlays** that would be loaded in memory when needed

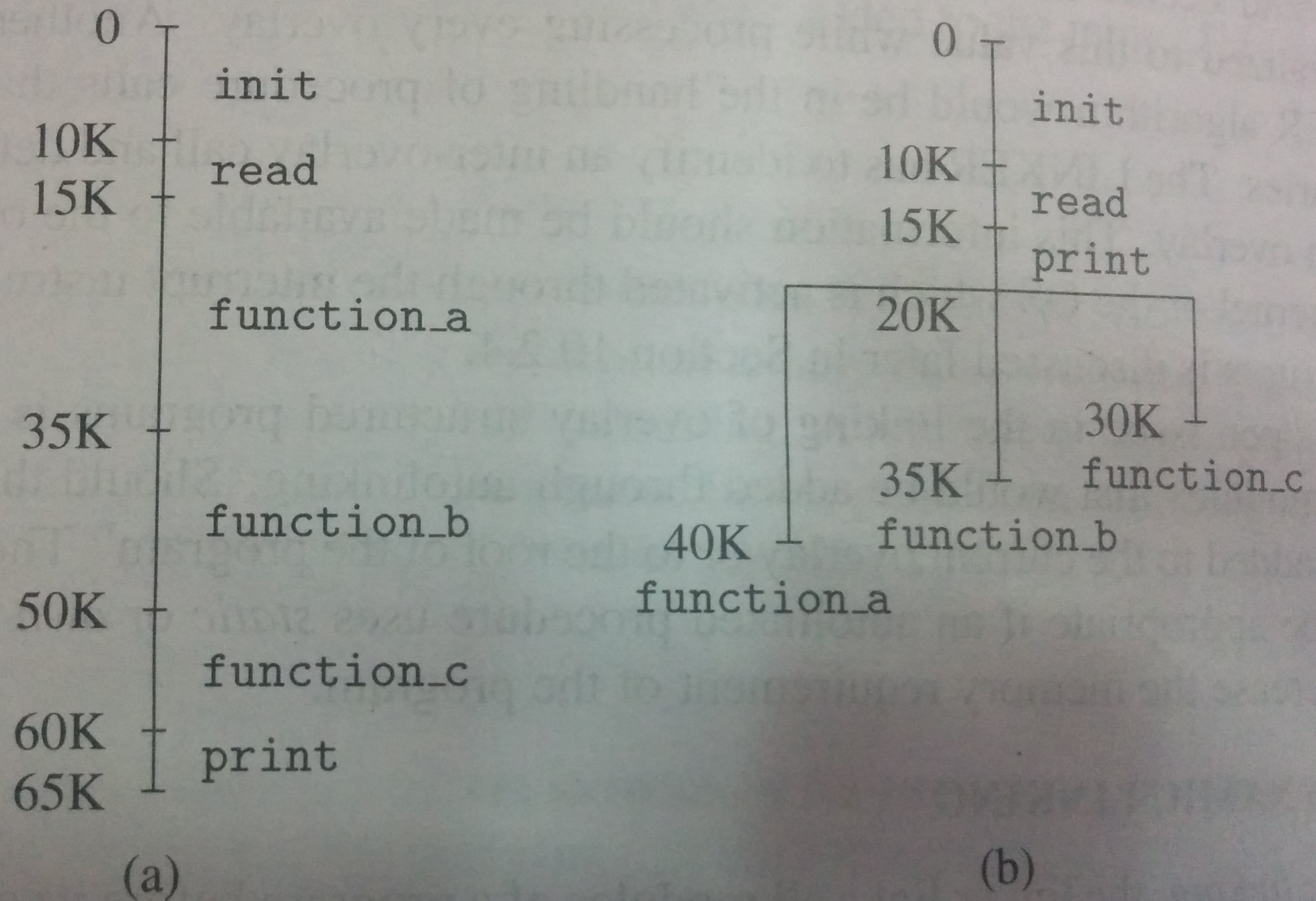
# Overlay manager

- An overlay manager is linked with root
- To start with , the root is loaded in memory and given control of execution
- It invokes overlay manager when it needs to refer data located in an overlay
- Overlay manger would overwrite a previously loaded overlay with same load origin
- Example is assembler : Passes of the assembler would form different overlays, data structures would exist in the root

# Design of an overlay structured program

- Consider a program with 6 sections
  - Init
  - Read *Rest*
  - Function\_a
  - Function\_b
  - Function\_c
  - Print *Rest*
  - Here read reads one set of data and invokes one of the function a b or c depending on values of data

- Here function\_a, b and c are mutually exclusive
- They can be made into separate overlays
- Here read and print are put in the root of the program
- See the diagram – the overlay structured program can execute in 40k bytes though it has a total size of 65 k bytes



**Figure 5.9** An overlay tree



# Linking for Overlays - Summary

- An overlay is a part of a program (or software package) which has the same load origin as some other part(s) of the program.
- Overlay are used to reduce the main memory requirement of a program.

## Overlay Structured Program

- A program containing overlays is an overlay structured program. It consists of:
  1. A permanently resident portion called the root.
  2. A set of overlays.
- ~~The overlay structure of the program is designed by identifying mutually exclusive modules- that is, modules which do not call each other. Such modules do not need to reside simultaneously in memory. Hence they are located in different overlays with the same origin.~~

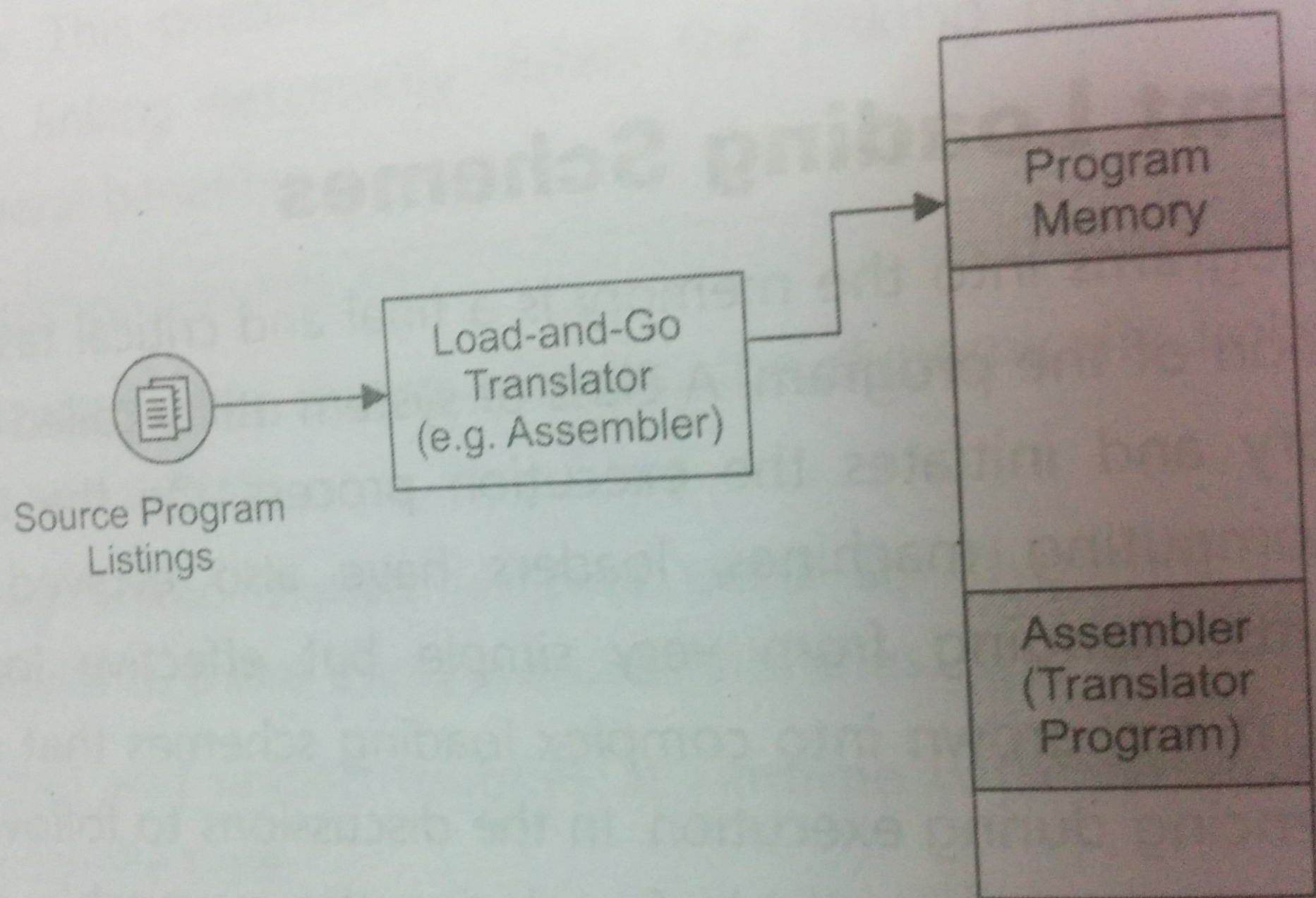


# Different loading schemes

- Compile and go loading scheme:
  - Translator is placed in one part of memory
  - Translator assembles the instructions one by one and puts it along with the data in the assigned part of the same memory.
  - when assembly is over, the assembler transfers the control to the first instruction.
  - It is known as assemble-and-go or load-and-go loading scheme.

assemble-and-go loading is

Core/Memory



▲ Figure 6: Compile-and-Go Loading Scheme

improve kare 6e tamaro compile time. ne ek j sathe tame to multiple source programs ne translate kari sako 6o into their respective object programs.

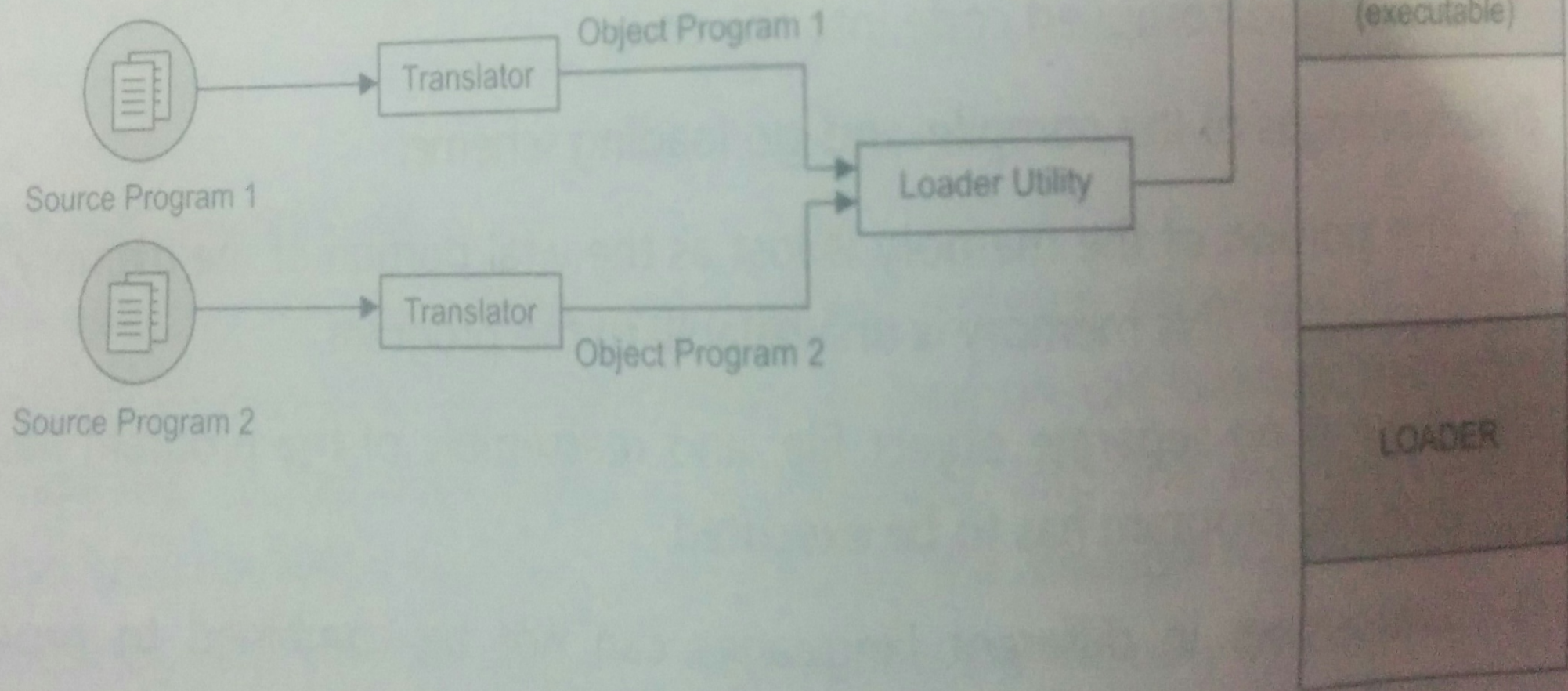
- General loading scheme:

- It improves the compile and go by allowing different source programs to be translated separately into their respective object programs  
object code ne kya store karso ?= secondary storage na andar.
- The object code(modules) is stored in secondary storage area and then they are loaded  
pachi ae object modules ne tame load karso.
- The loaders combines modules and loads in memory  
loader su karse....badhi modules ne combine karse and aemne load karse kya ? memory na andar.
- Rather than entire assembler , a small loader does the job.

akha assemble jeva mota defa ni kai jarur j nathi ...nanu amtu loader pan kam chalavse. 91



General loading scheme is shown below



▲ Figure 7: General Loading Scheme

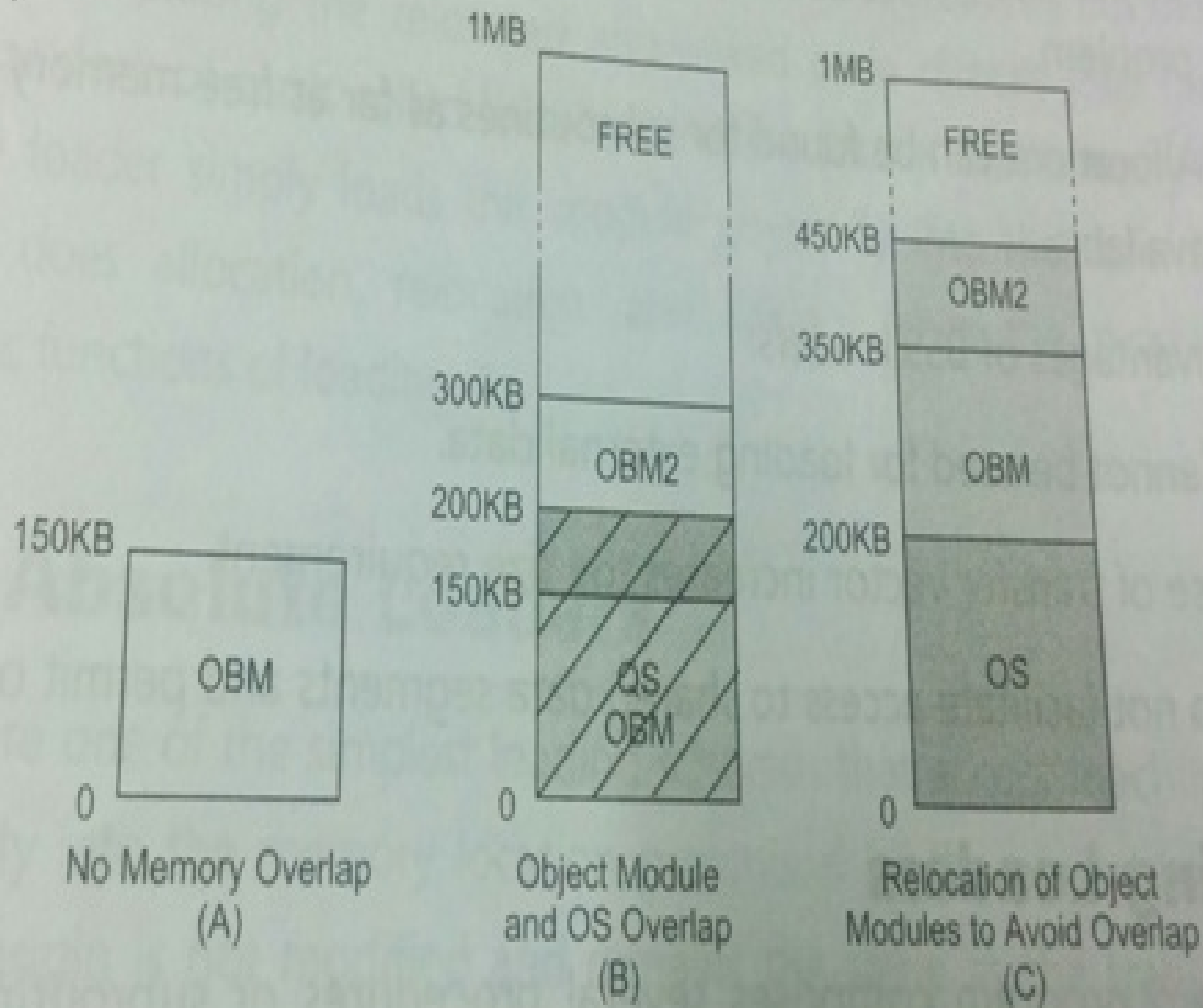
su karse...program relocation or relocation of the various modules ne karse by the programmer.

- Relocating loaders:

- Allow program relocation or relocation of various modules for the programmer

what is relocation ??

- Relocation is moving object code from a memory location already occupied to a new location in the memory for some reason
  - See the concept in diagram



▲ Figure 9: Concept of Relocation (Performed by Relocating Loaders)

- Practical relocating loaders
  - Example is binary symbolic subroutine(bss) loader
  - The assembler translates and assembles each subroutine or procedure segment independently and passes it onto the loader together with
    - Translated text of each subroutine
    - Intersegment references for each subroutine
    - Information about the relocation of each subroutine

assembler to tamaru translate kare 6e and darek subroutine ne assemble kare 6e...or segment ne procedure kare 6e independently and pass kare 6e ae badha loader ne ek j sathe.

=> darek subroutine nu translated text hovu joie.

=> darek subroutine nu intersegmented reference hovu jie ..s

=> darek subroutine na relocation ne information hvi joie.

- A transfer vector is associated with each source program text generated by the assembler.
- It consist of addresses containing the names of subroutine that are referenced by the source program.
- Assembler passes info such as length of program and transfer vector to loader
- Loader will load each subroutine identified by transfer vector

ek transfer vector => je associated hoi dare source program na jode...that is generated by the assembler.

transfer vector na andar su hoi 6e?= addresses of the names of the subroutine j pan tamara source program na andar referecence thaya 6e.



assembler to tamaro pass karse .... length of the program ne .... transfer the vector to the loader.

loader tamaro load karse .... darek subroutine ne ... kevo subroutine ne je pan identified hoi 6e ne tamara transfer vector na andar.

- Output of BSS loader:
  - The object program
  - Information about all subroutine and their references
  - Information about locations that need to be changed

BSS =>

object program avse

2. badha subroutines and aemna vishe ni information ne

3. information about locations that needs to be changed/

various object modules nu linking tamaru linker kare 6e.  
2.

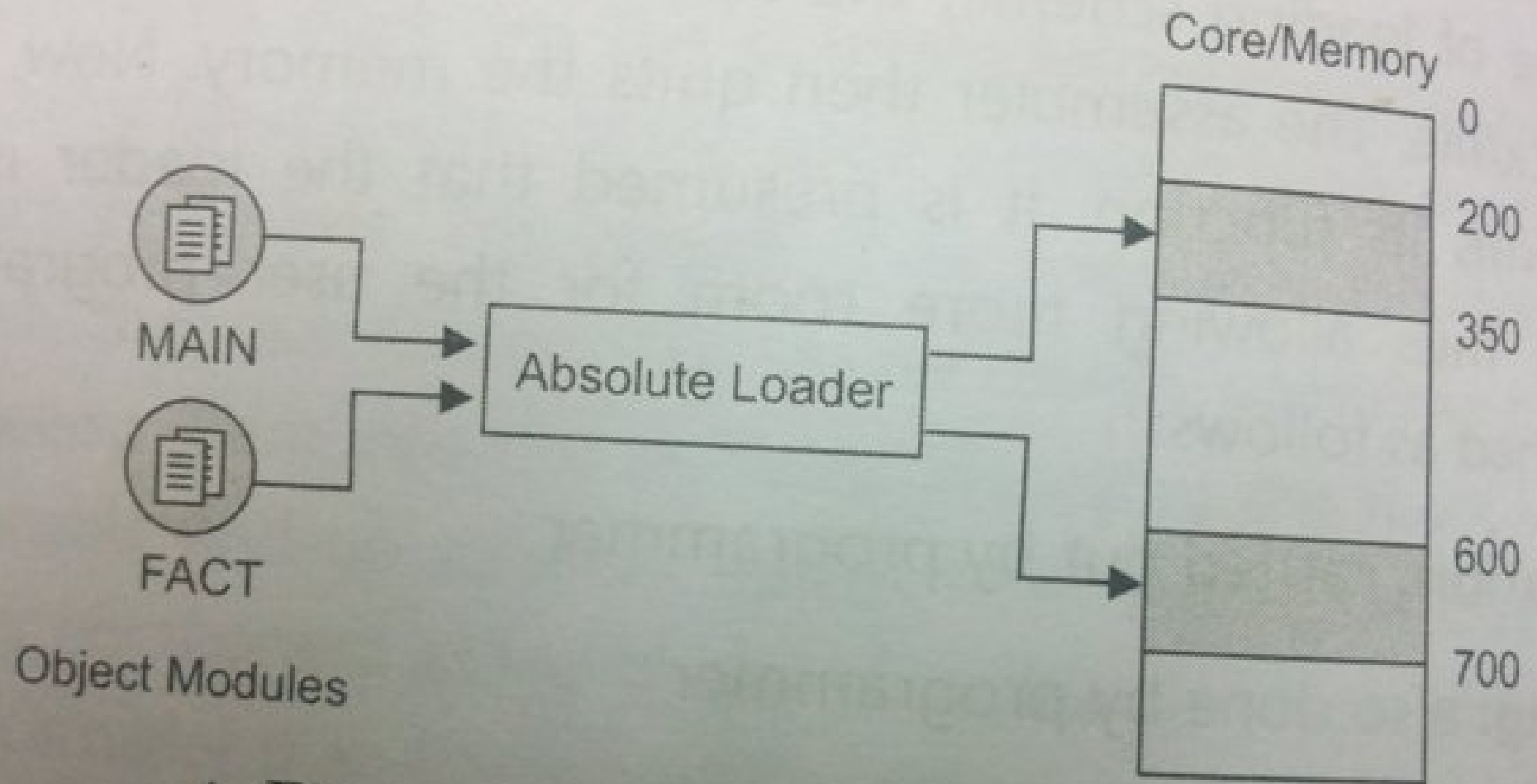
- Linking loaders:
  - Linking of various object modules is done by the linker.
  - Special system program called **linking loader** gathers various modules, links them together to produce single 'executable binary program'
  - and loads them into the memory
  - It is also called **direct-linking loaders**

# Absolute Loading scheme

- Assembler assembles the program into intermediate object file
- It quits the memory
- Loader sits in memory and perform its function
- Loader is very small compared to assembler, so give more room to user programs
- Loader functions:
  - Allocations
  - Linking

o-o r-o

- Relocation
- Loading
- Advantage:
  - simple to implement and efficient in execution
  - Saves the memory



▲ Figure 8: Absolute Loading Scheme

- 1) absolute loader notes load origin and length of the program ( header record)
- 2) enters a loop that reads a binary image record and moves code to the memory area starting on the address ( in binary image record)
- 3) transfers control to the execution start address of the program

# Limitation

- Its use is limited to loading of programs that either have load origin = linked origin or are self relocating

what is the disadvantage of the absolute loading scheme?

- Design of absolute loaders
- Design of direct-linking loaders
- (not included in syllabus )