Chapter 5
Linkers and Loaders

Execution of a program
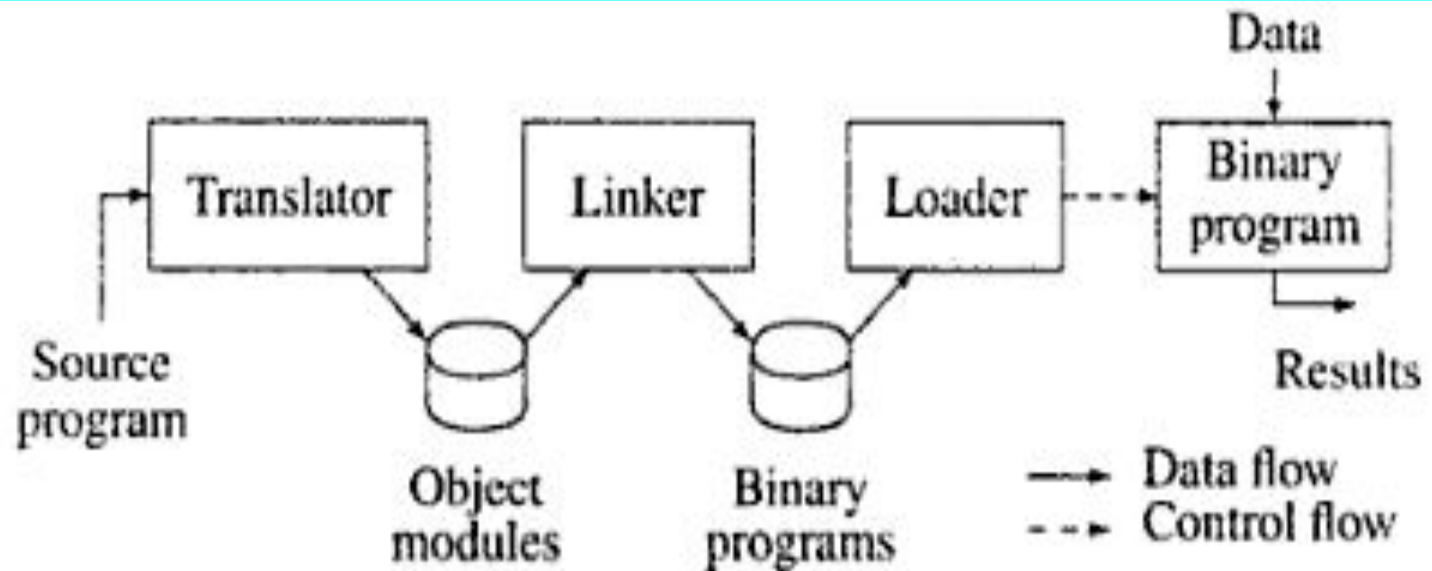
Execution of any program

1. Translation of program → **Translator**

2. Linking of the program → **Linker**

3. Relocation of the program

4. Loading of the program → **Loader**
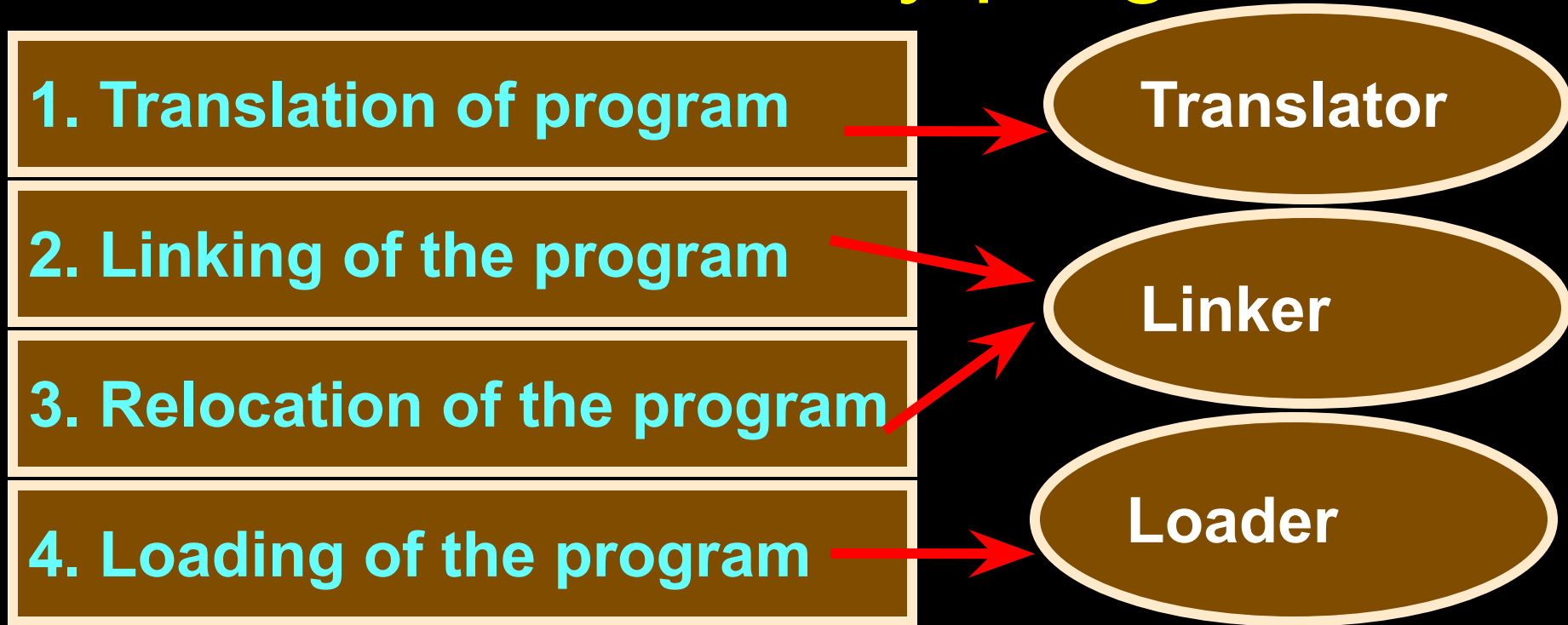
Four steps for execution

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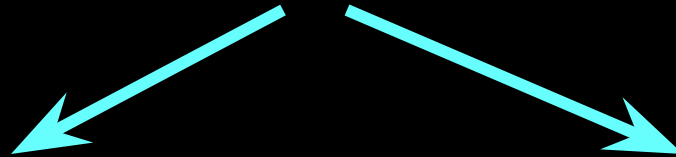
1. The same set of translated addresses may have been used in different object module

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.

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.

Example – translated origin

	Statement		Address	Code
	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			
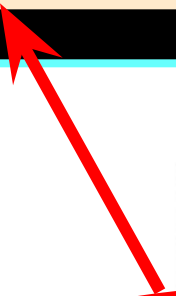
Example

Statement		Address	Code
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

Statement		Address	Code
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			

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

Statement		Address	Code
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…

1. Address Sensitive instruction

READ   A

2. Address constant (absolute Address)

ORIGIN 500
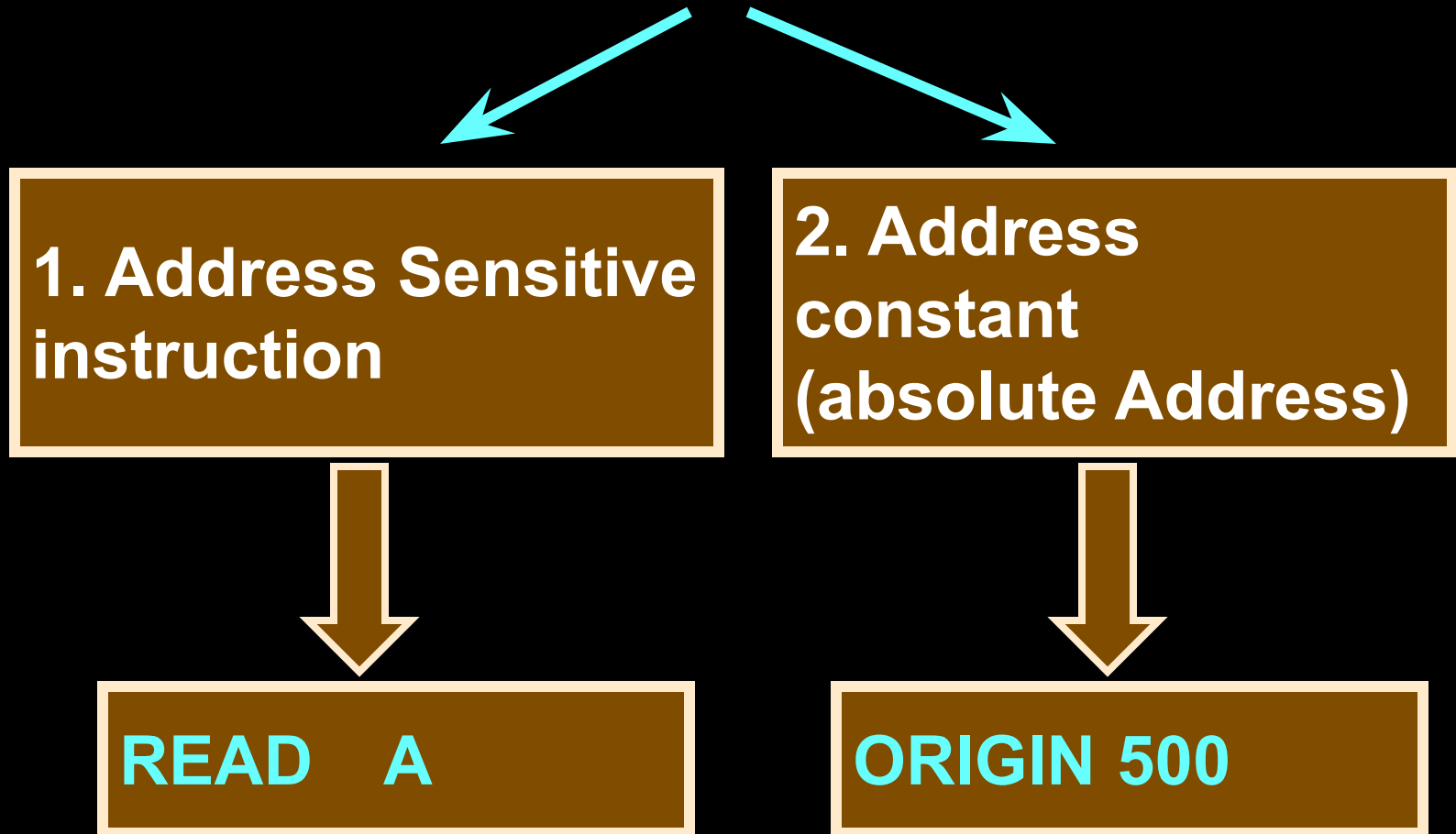
- An address sensitive program can only execute correctly if the start address of the memory area allocated to it is same as the translated origin

- To execute correctly from any other memory area, all the address sensitive instructions need to be **corrected**

- 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
- *l_origin* = linked origin
- *Symb* = any symbol in program P
- *T_symb* = translation time address
- *L_symb* = link time address
- Relocation_Factor = *l_origin - t_origin*

$$relocation\_factor_{\mathbf{p}} = l\_origin_{\mathbf{p}} - t\_origin_{\mathbf{p}}$$

$$t_{symb} = t\_origin_{\mathbf{p}} + d_{symb}$$

$$l_{symb} = l\_origin_{\mathbf{p}} + d_{symb}$$

$$l_{symb} = t\_origin_{\mathbf{p}} + relocation\_factor_{\mathbf{p}} + d_{symb}$$

Statement			Address	Code
	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			

- 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.
- 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.

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 Pi and Pj,
- It is achieved by following linking concepts.

- 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

Statement			Address	Code
	START	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
	READ	A	500)	+ 09 0 540
LOOP			501)	
		$\vdots$		
	MOVER	AREG, ALPHA	518)	+ 04 1 000
	BC	ANY, MAX	519)	+ 06 6 000
		$\vdots$		
	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

```
Statement              Address        Code

START   200
ENTRY   ALPHA

- -

- -

ALPHA   DS      25       231)      + 00 0 025
END
```

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

	Statement		Address	Code
	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			

27

EXTERN statement

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

```
        Statement           Address      Code

        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

- 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
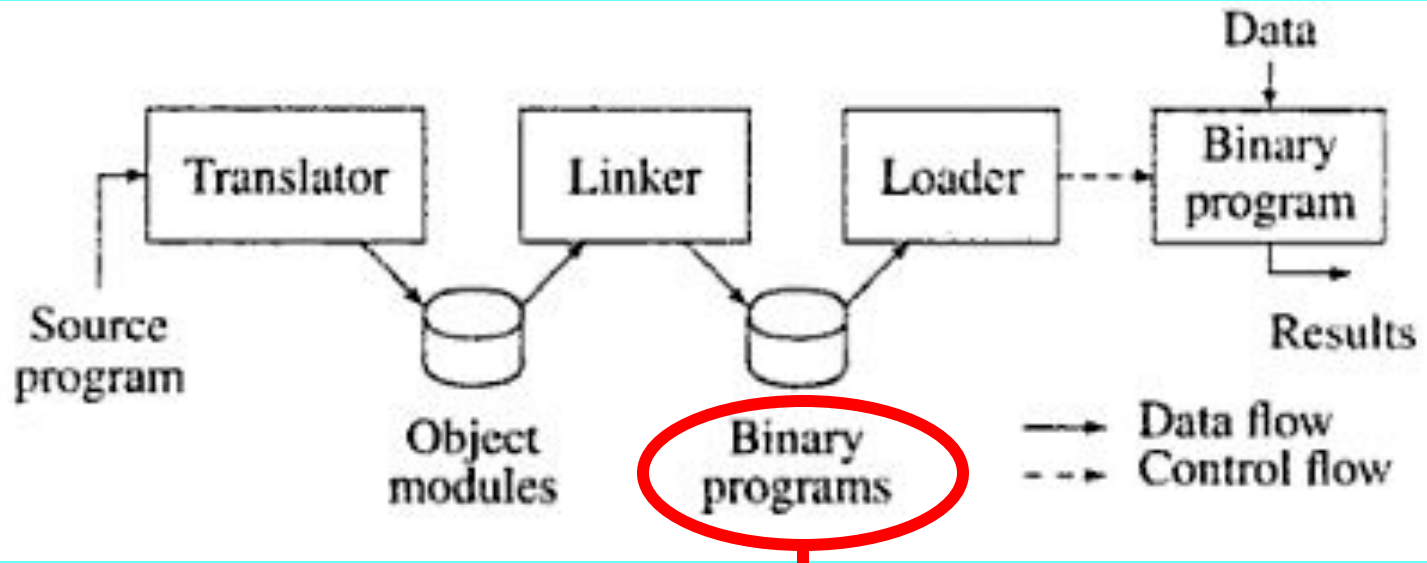
Ready to execute form of program

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.

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

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

  [,<execution start address>]

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

- <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
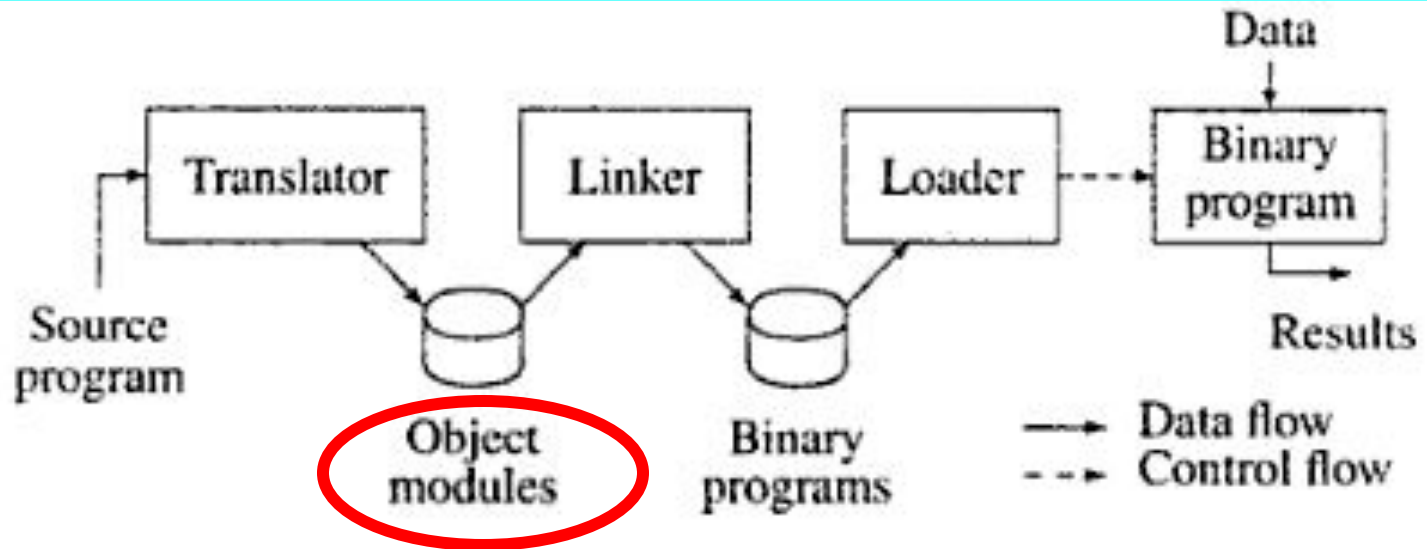
Object module of program P -four components

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 RELOCTAE 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* := *<link origin>* 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

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.

symbol	linked address
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

Statement		Address	Code
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			

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

45

Program

Non Relocatable

Relocatable

Self Relocating

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.

- 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.
  - 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.

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, 12H[BX]

- 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

    PURGE ABC – ABC no longer represents xyz
  - ABC   EQU 25 – now represents 25

SEGMENT, ENDS

- Start and End of a segment

- ASSUME   :
- Informs assembler that the address of segment would be in segment register

- ASSUME DS:SAMPLE_DATA

- 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,DATAA_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

65

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, LNAMES and SEGDEF records

- 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.

- LNAMES 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.

- 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.

- 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 with the segment.

A0H	length	segment index (1-2)	data offset (2)	data	check-sum

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.

8AH	length	type (1)	start addr (5)	check-sum

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:

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.

Dynamic Linking

- It is performed during execution of binary program
- 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

Benefits of Dynamic linking

- 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.

- 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

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.

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

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

- 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

83

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
  - Function_a
  - Function_b
  - Function_c
  - Print
  - 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
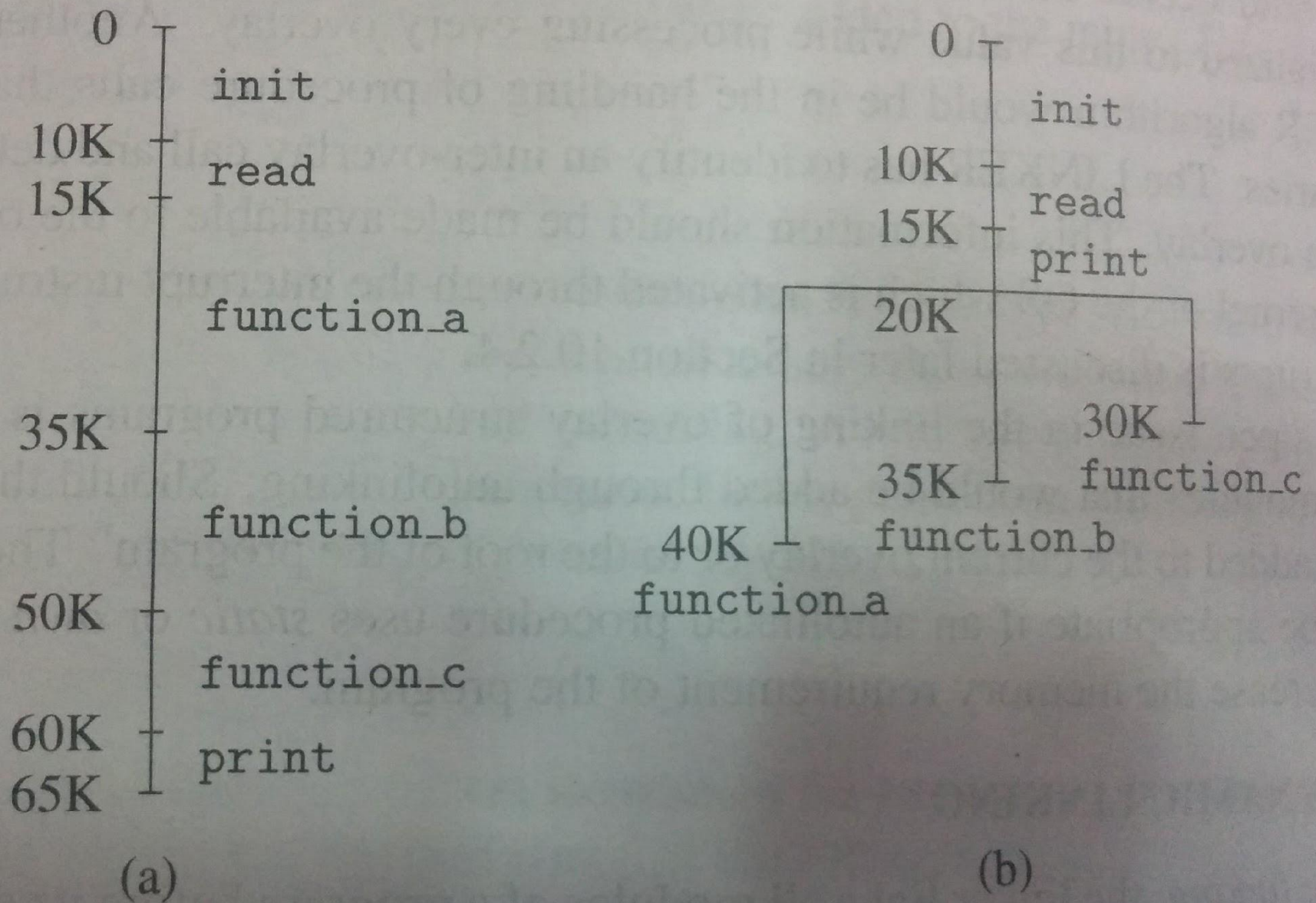
Figure 5.9 An overlay tree

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 places 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.
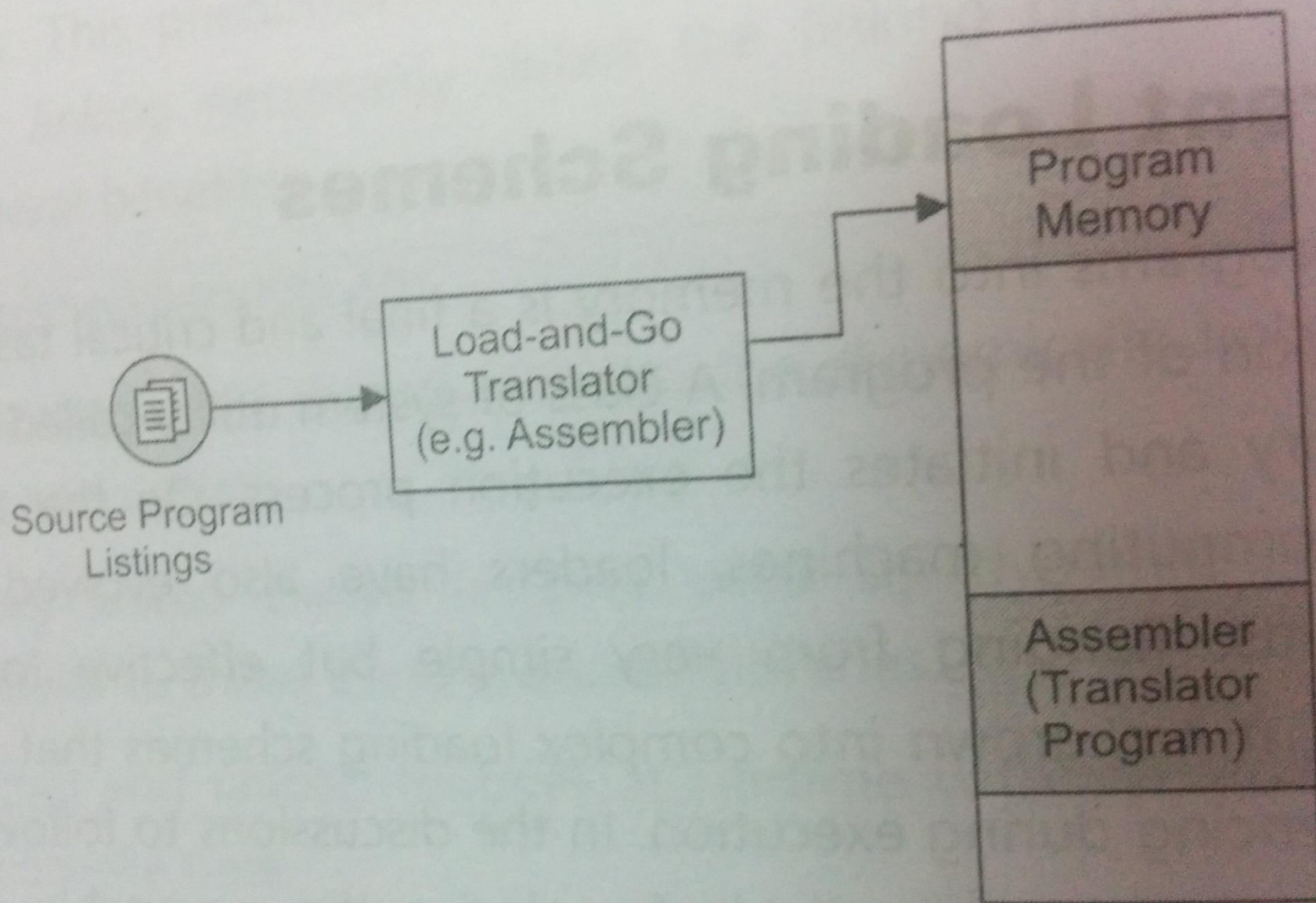
Figure 6: Compile-and-Go Loading Scheme

- General loading scheme:
  - It improves the compile and go by allowing different source programs to be translated separately into their respective object programs
  - The object code(modules) is stored in secondary storage area and then they are loaded
  - The loaders combines modules and loads in memory
  - Rather than entire assembler , a small loader does the job.
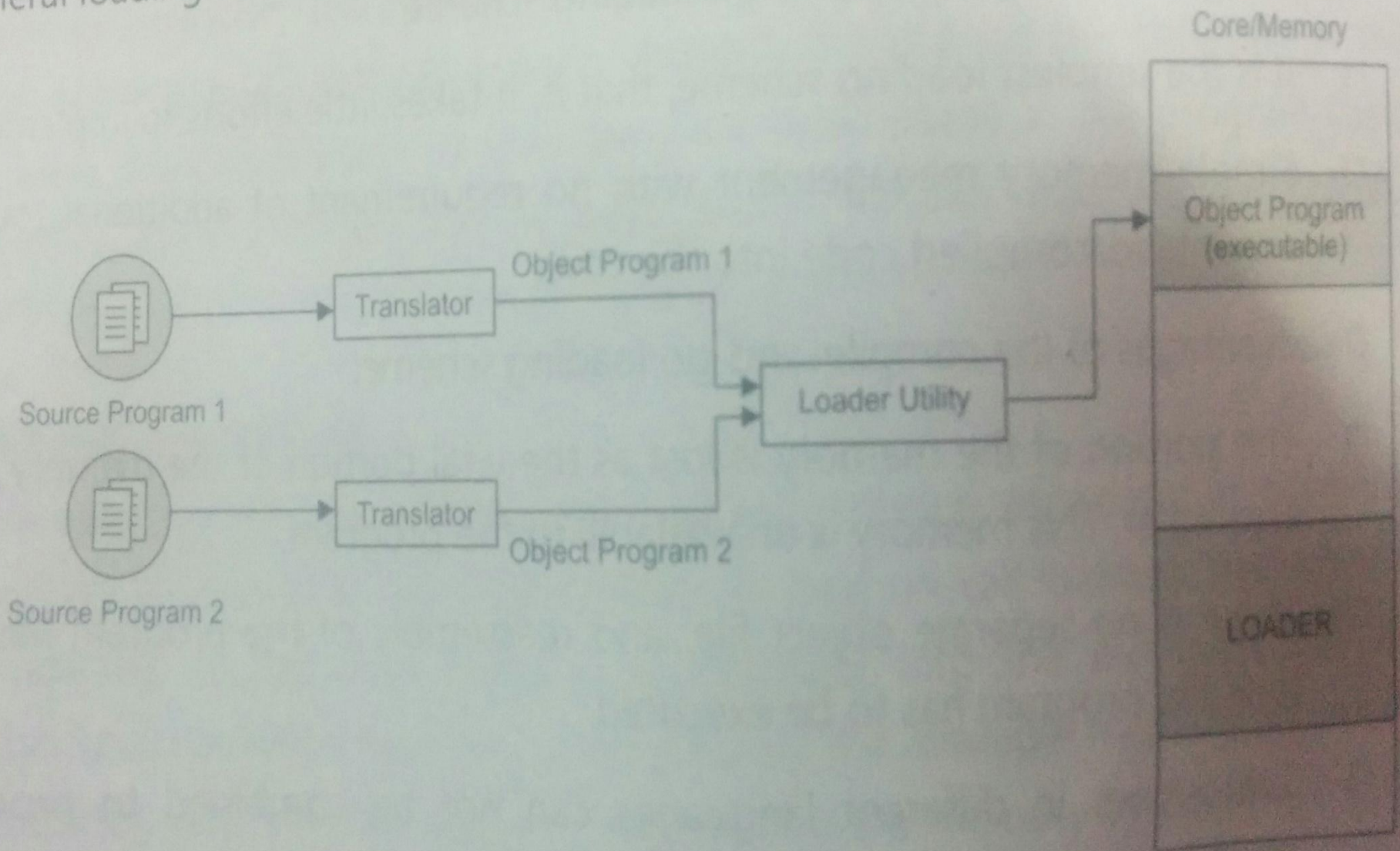
▲ Figure 7: General Loading Scheme

- Relocating loaders:
  - Allow program relocation or relocation of various modules for the programmer
  - 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
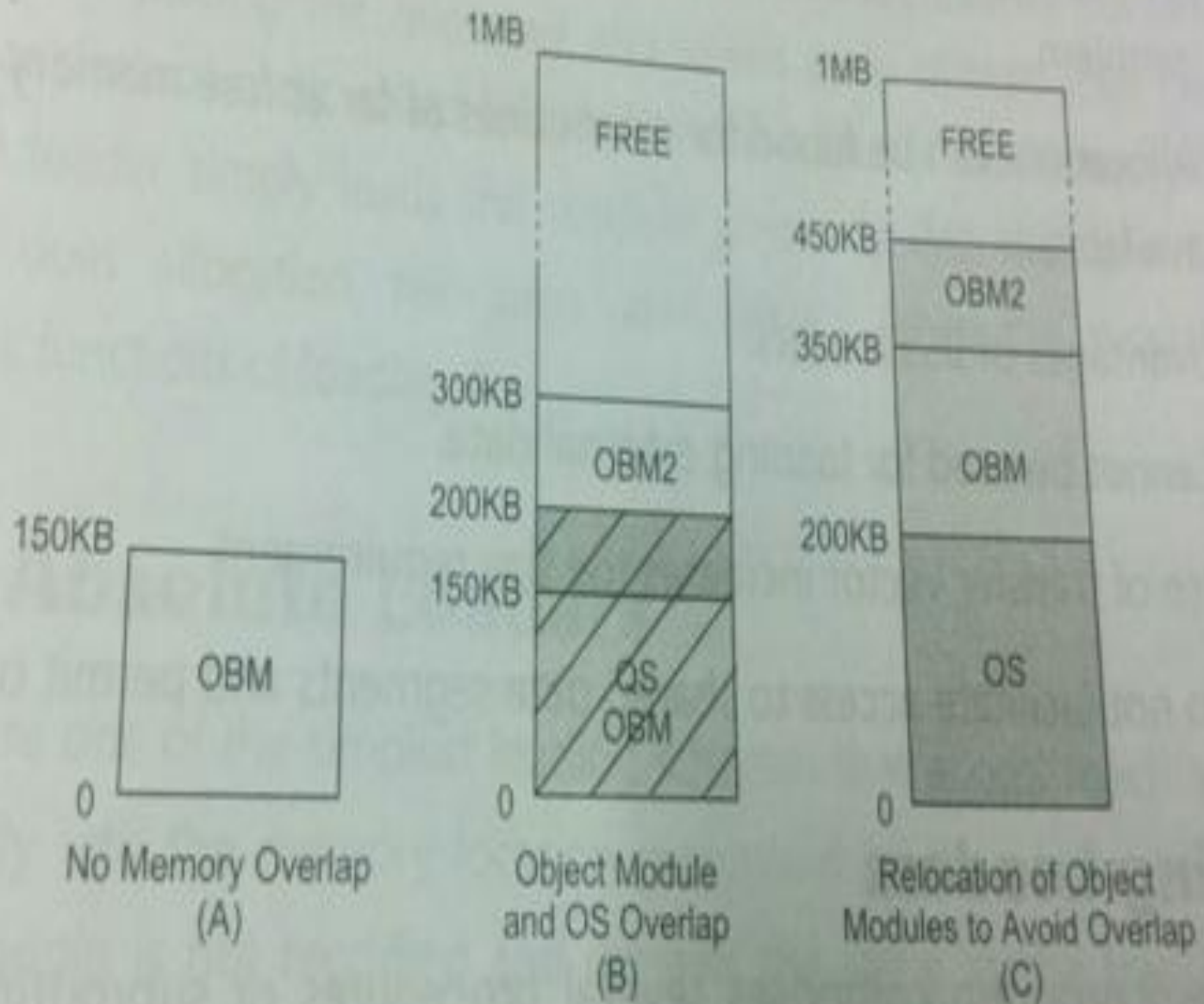
▲ Figure 9: Concept of Relocation (Performed by Relocating Loaders)

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

- 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

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

- 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

- Relocation
- Loading

- Advantage:
  - simple to implement and efficient in execution
  - Saves the memory
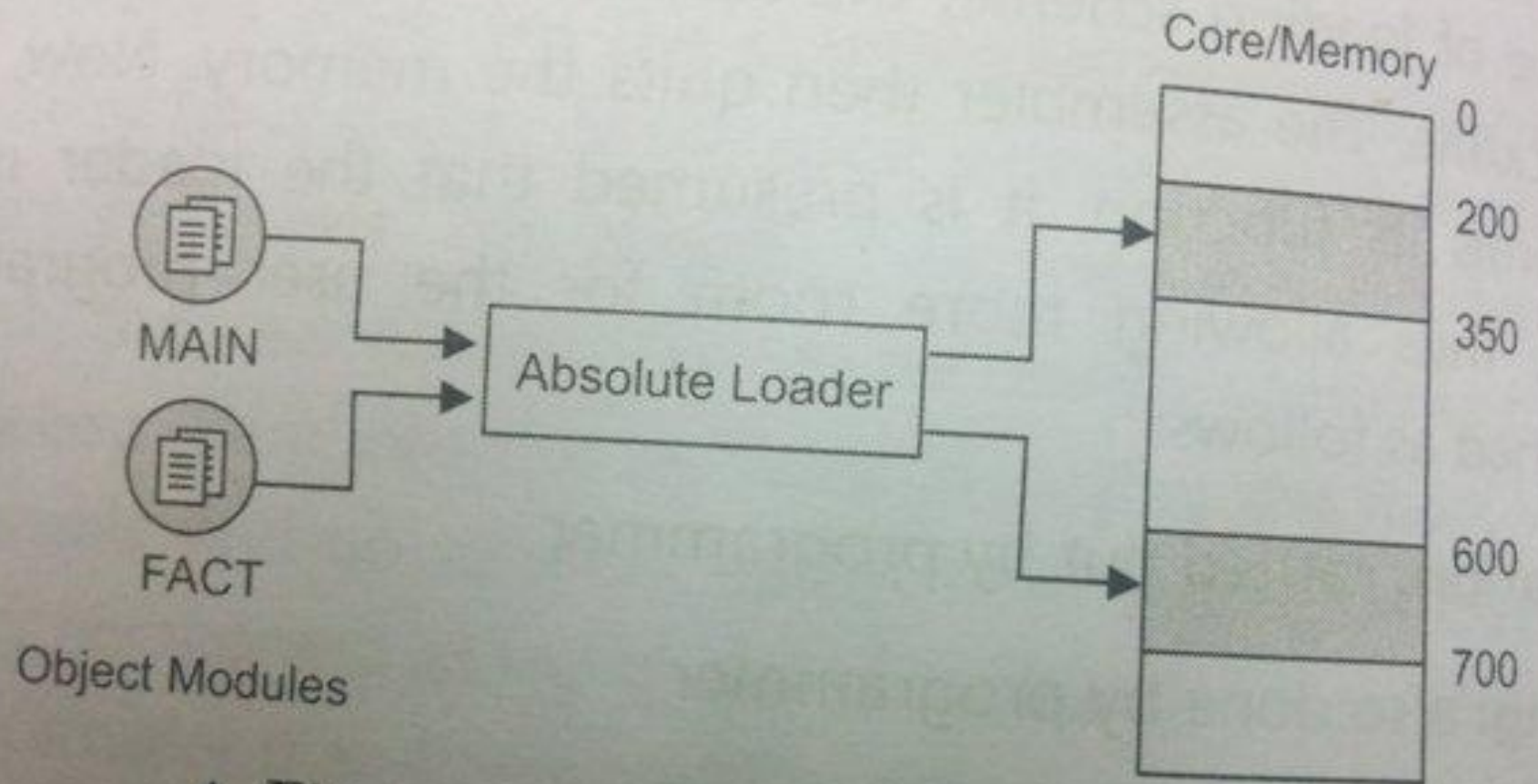
▲ **Figure 8: Absolute Loading Scheme**

- 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

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