Linker and Loaders

- The linker is a system program that combines the code of a target program with codes of other program & library routines.

- To linking, the language translator builds a object module for a program which contains both target code of program & information about other program [it needs during its execution].

- The linker extracts this information from the object module, locates the needed program & combine with target code to produce program in machine language.

- so that can execute without requiring the assistance of any other program.

- Such program is called a binary program.
- The loader is a system program that loads a binary program in memory for execution

Introduction :-

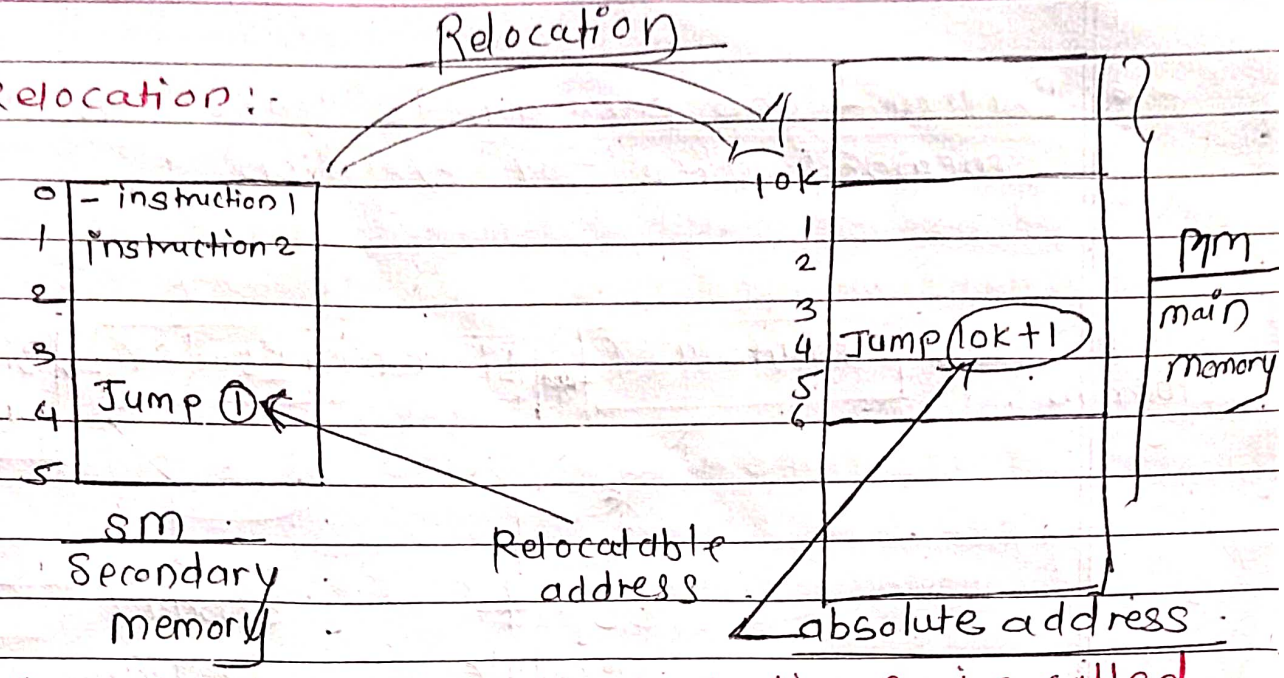
Execution of a program written in programming language is achieved in the following four step:

1) • Translation - A program is translated into a target program.

2) Linking :- The code of a target program is combined with codes of those program and library routines that is calls.

PAGE No.
DATE: / /

3) Relocation:-



Jump① means go to the instruction ① i.e called Relocatable address

When once done the Relocatable address of instruction then this address are assigning in main memory for execution.

When go to the main memory 10k+1 is known as absolute address.

- A program may have been coded or translated with the idea of executing in a specific area of memory
- The operating system may have used that memory area for another purpose.
- so it may allocate a different memory area for the program execution.
- Relocation is the action of changing the memory addresses used in the code of the program so that it can execute correctly in the allocated memory area.

PAGE No.
DATE: / /

Loading - The program is loaded in a specific memory area for execution.

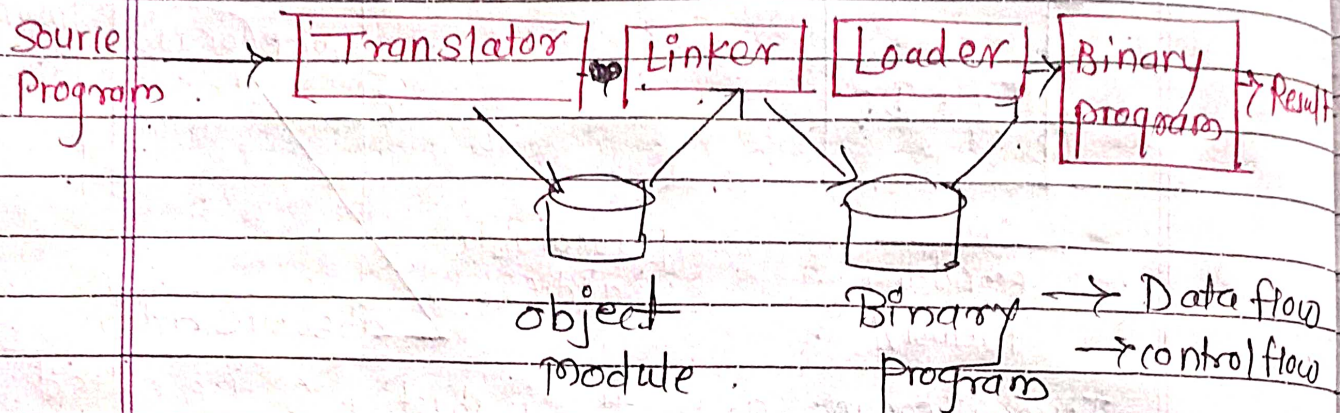


diagram Program execution :

1) Translated time - (translated address) - Address assigned by the translator.

2) Linked address - Address assigned by the linker

3) Load time (load address) :- Address assigned by the loader.

- origin & execution start address -

- Translated origin - Address of the origin used by the translator. It is either the address specified by the programmer in an ORIGIN or START statement, or a default value.

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 in memory for execution.

PAGE No.
DATE: / /

Relocation and Linking concepts

- **Program Relocation –**

– Let AA be the set of absolute addresses - instruction or data addresses - used in the instructions of a Program P. AA would be a nonempty set.

– if the program P expects some of its instructions and data to occupy memory words with specific addresses such a program is called address sensitive Program.

It contain.

– 1)- An address sensitive instruction. An instruction that uses an address a_i included in set AA.
[AA - means · Absolute address].

2)- An address constant :- A data word that contains an address a_i included in set AA.

– Program relocation is the action 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 ≠ translated origin, relocation must be performed by the [linker].

– If load origin ≠ linked origin, relocation must be performed by the loader.

- loaders do not perform relocation - that is, load origin = linked origin, such loaders are called absolute loaders.

PAGE No.
DATE: / /

- Performing relocation:

Let the translated and linked origins of program p be t_origin_p & l_origin_p, respectively
- consider a symbol symb in P.

Let its translation time address be tsymb and linked address be lsymb.
- The relocation factor of P is defined.

relocation_factor_p = l_origin_p - t_origin_p

↓

linked origin of program p − translated origin of program p.

- Note that relocation_factor_p can be positive, negative or zero.

- consider a statement which uses symb as an operand. The translator puts the address tsymb in the instruction generated for it.

Now,

~~t sy~~. tsymb = t_origin_p + dsymb

where dsymb - is the offset of symb in p.

After program P has been relocated to the linked origin, i.e, l_origin_p, we have.

lsymb = l_origin_p + dsymb.

lsymb = t_origin_p + relocation_factor_p + dsymb
= t_origin_p + dsymb + relocation_factor_p
= tsymb + relocation_factor_p.

PAGE No.
DATE: / /

- **Linking -**

⇒ A program unit is any program or routine that is to be linked with another program.
- We assume that each program unit has been assembled separately to produce an object module.
- Let an application consist of a set of program unit $Sp = \{Pi\}$.
- Now consider a program unit P_i that requires the use of another program unit P_j.

- during its execution - either it uses the address of some instruction in P_j in one of its instructions, possibly in a subroutine call instruction.

- **public defination** - A symbol defined in a program unit that may be referenced in other program units.

- **External reference** - A reference to a symbol that is not defined in the program unit containing the reference.

* **EXTRN and ENTRY statements**

- An ENTRY statement in a program unit lists the public definations of the program unit.

- An EXTRN statement lists the symbols to which external references are made in the program unit.

~~Mem~~

* Relocation and linking concepts

public & external References

- A program unit P_1 interacts with program unit P_2 by using addresses As P_2's instruction & data in its own instructions.
- for this P_1 & P_2 must contain public defination & external references as follows:

i) Public defination - A symbol defined in program unit may be referenced in other program unit.

ii) External Reference - A reference to symbol which is not declared in the program containing the reference.

- The ENTRY statement lists the public definatio of a program unit.

- The EXTRN statement lists the symbol to which external references are made in program unit.

e.g - The program - ENTRY statement indicates that a public defination Total exists in program.

DATE: / /

Program P

	statement	Address	code
	ORIGIN 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		

Program Q

statement	Address	code
START 200		
ENTRY ALPHA		
- - -		
- - -		
- - -		
ALPHA DS 25	231)	00 0 025
END		

PAGE No.
DATE: / /

let If the program p link with program q.

- In the program p's-
IF the link origin 'p' is 500, & size is 42 words & ~~then link 0~~
why 42. because in p program Address are 500 to 541 so total 0 to 41 is 42 size
- so the link origin of q is
500 + 42 = 542
and link time.
542 + 31 = (573)
↑
this 31 in the program q address added (31) is extra.
so address of ALPHA is 573. assign in program p

Imp step Step II - You guys write down this step.
Linking is the process of binding an external reference to the correct link time address.

- Linking is performed by putting the link time address of ALPHA in the instruction of p using ALPHA

linked origin = starting execution address of Program p + size of program p
500 + 42 = 542

relocation-factor$_p$ = L-origin$_p$ - t-origin$_p$
- linked origin - translated origin.
= 542 - (200) in the program p starting exe. addr.
= 342

l_symb linked address = relocation_factor + d_symb

= 231 + 342

= 573

↑

ALPHA Address.

- **Binary Program** - A binary program is a machine language program comprising a set of program units $\{p\}$ such for every p_i in $\{p\}$

1. p_i has been relocated to the memory area whose starting address matches its linked origin, and.

2. Each external reference in p_i has been resolved.

- To peform a binary program from a set of object module,

The programmer invokes the linker by using the command.

linker <link origin>, <object module names> [, <execution start address>]

Where <link origin> - specifies the memory address to be given to the first word of the binary program.

<execution start address> - is usually a pair (program unit name, offset in program unit).

PAGE No.
DATE: / /

- The linker converts it into the linked start address, and stores it along with the binary program for use when the program is to be executed.

- If specification of <execution start address> is omitted the execution start address is assumed to be the same linked origin.

- Therefore we have assumed the load origin of a ~~pro~~ program to be the same as its linked origin.

- The loader simply loads the binary program into the area of memory starting at its linked origin for execution.

• object Module •

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

- The object module of a program unit P consists of the following four components:

1. Header :- The header contains the 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 the instruction that require relocation, that is, it describes IRRp. Each RELOCTAB entry contains a single field:

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

4. Linking table (LINKTAB): This table contain information of public defination & external references.
contain three fields.

1) Symbol :- symbolic name.

2) Type :- The values PD & EXT indicate whether the entry pertains to a pubic defination or an external references. respectively.

3) Translated address :- for a public defination, it is the translated address of the first memory word allocated to the symbol.
- for an external reference, it is the address of the memory word that is required to contain the address of the symbol.

example- object module
refer the Program P (page 8).
- The header contains the information
translated origin = 500
size = 402
executable start address = 500.

2- The program component contains the machine language instruction.

PAGE No.
DATE: / /

relocation table.

translated address
500
538

4. The linking table.

symbol	type	translated address
ALPHA	EXT	518
MAX	EXT	519
A	PD	540

other symbols defined in program P do not appear in the linking table because they are not declared as public def^n in ENTRY statements. e.g., the symbol LOOP.

• Linking of overlay structured program.

- some parts of a program may be executed only briefly, or not at all, during execution.
- memory requirement of the program can be reduced by not keeping these parts in memory at all times.

- some parts of a program are given the same load address during linking.
- This way, only one of these parts can be in memory at any time because loading of another part that has the same load address.

PAGE No.
DATE: / /

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.

1) — A permanently resident part, called the root.
root is compulsory executed.

2) — A set of overlays that would be loaded in memory when needed.

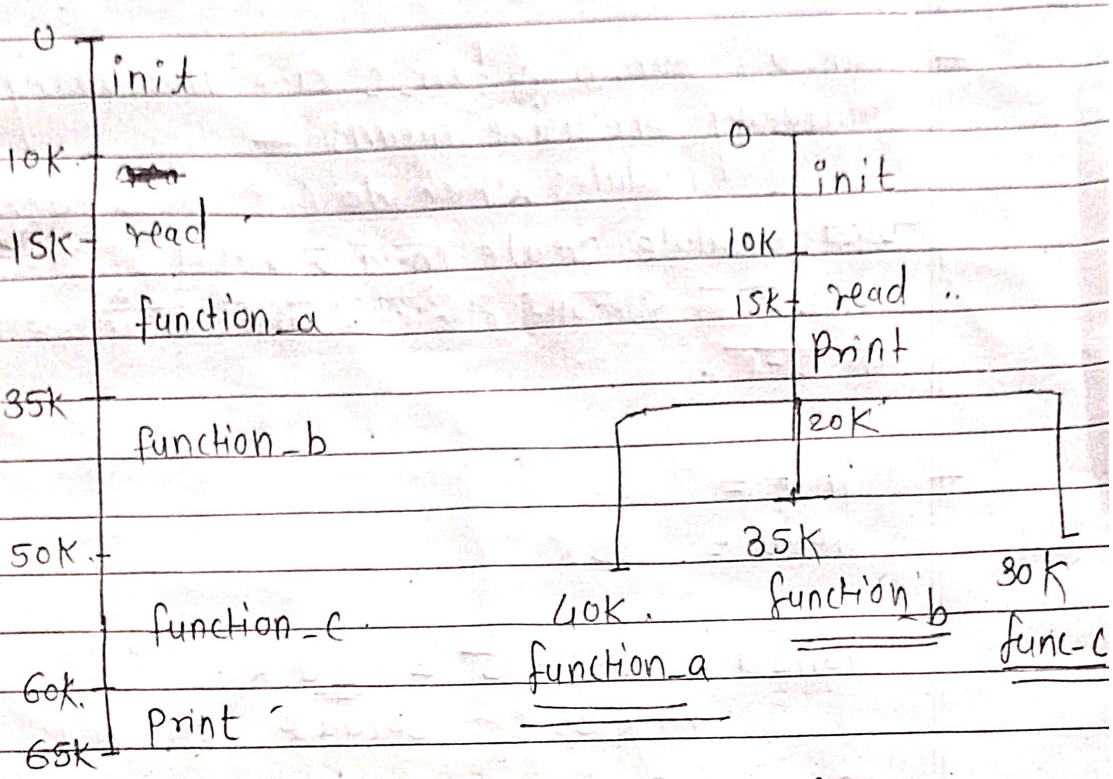
- An overlay manager is linked with the root.
- To start with, the root is loaded in memory & given control of execution.
- It invokes the overlay manager organizes loading of the required overlay which previously loaded overlay with same load origin.

Example of overlay structure is assembler.

- the passes of the assembler would form different overlays, the data structures shared by them would exist in the root.

• Design of an overlay structured program

15



In (b) diagram 20K. print value. How it in (a) program print is middle in 60k & 65k

so 65 - 60 = 5 - value of p.

after the transfered root area they will added read value so (15 + 5 = 20).

so print value is 20

- consider a program with 6 sections named init, read, function_a, function b, function_c & print.

- init performs some initialization & passes control to read.

- read reads one set of data & invokes one of function_a, function_b or function_c depending on the values of the data.

- print is called to print the result.

- fun_a. fun_b & fun_c are mutually exclusive

Mutually exclusive means :-

Modules which do not calls each other

असे module create करावे जामतात जे एकमेकांना call करू शकणार नाहीत : दोन्ही एकमेकांची separate करायचे.

- init = 10K
  read = 15K

  fun_a = 35 − 15 = 20 K.

  Why 35 & 15 because fun_a is middle in 35 & 15.

  fun_b = 50 − 35
  = 15K.

  fun_c = 60 − 50
  = 10K.

  Print - 65K.

  all transfer in root section.

  show in fig (b)

  so. ~~the fun~~ in the section b.

  value of init = 10K

  read = 15K as it is transfer

  change value p = 20

  so the fun_a will be 20 + 20 = 40K.

  ↑ is print value

  ↑ fun-a value in program (a)

PAGE No.
DATE: / /

funb = 20 + 15
= 35K

func = 20 + 10
= 30K

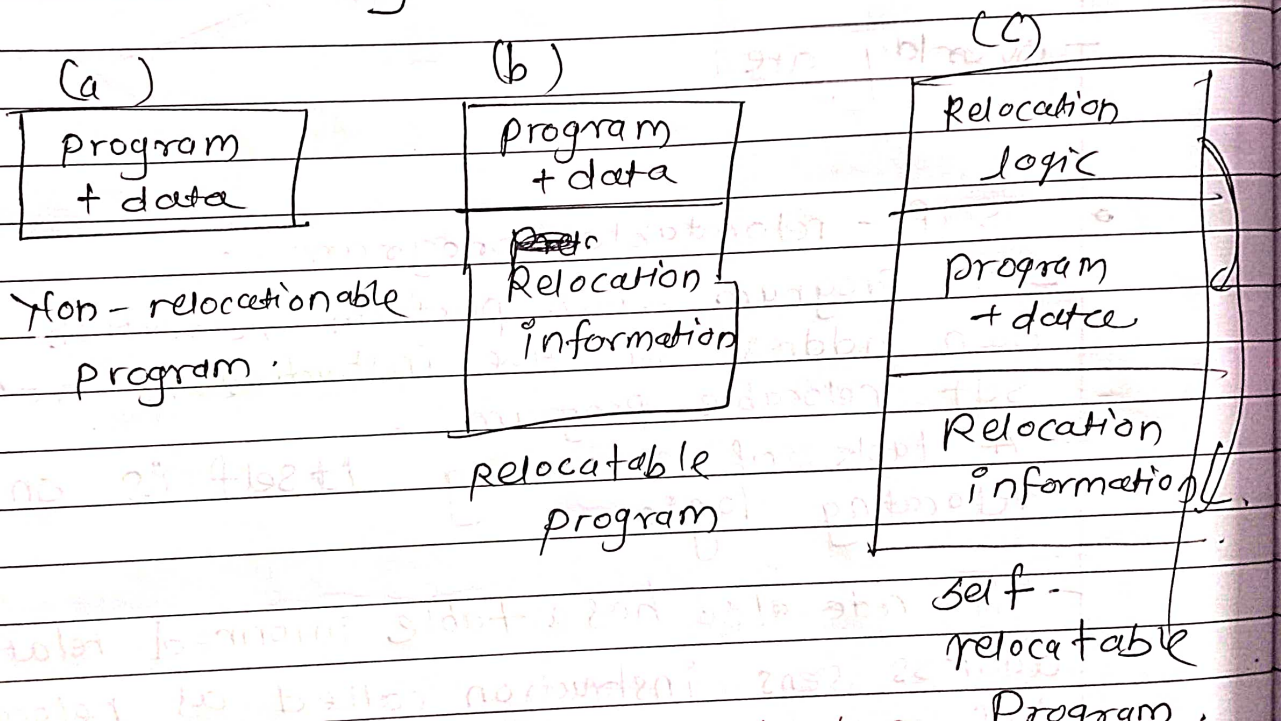
above program (a) same program execution need 65K memory and in program (b) same program execution need 40K memory.

means funa, fun-b & func the same memory the overall memory requirement is reduced to 40K.

- Self - relocatable program.
- A Program which perform relocation & it's own address sensitive instruction is named as self relocable program.
- A task of relocating itself is one the relocating logic
- This code also has a table informed related to address sens. instruction called as relocation information.
- A start relocating logic address is always specified as execution start addres of any program.
- By making use of relocating logic & call information related to address sensitive instruction is performs relocation by itself.

Date : / /
Page No.:

- After relocation & program execution control is transferred.
- commonly used compilers like TC, FORTRAN, COBOL in multiuser system contain code for self - relocation.
- self - relocatable programs are useful for operating system pointof view in multiuser or time sharing environment.
- There program can execute in any area of memory.



Algorithm 5.1 (Program relocation)

1. *program_linked_origin* := <*link origin*> from the `linker` command;
2. For each object module mentioned in the `linker` command
   (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 contained in the *program* component of the object module into the *work_area*.
   (d) Read RELOCTAB of the object module.
   (e) For each entry in RELOCTAB
      (i) *translated_address* := address found 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 found in the word that has the address *address_in_work_area*.
   (f) *program_linked_origin* := *program_linked_origin* + *OM_size*;

The computations performed in the algorithm are along the lines described in Section 5.2.1. *program_linked_origin* contains the linked address that should be assigned to an object module. It is initialized to <*link origin*> from the `linker` command and after processing an object module it is incremented by the size of the object module in Step 2(f) so that the next object module would be granted the next available linked address. For each entry in the RELOCTAB, Step 2(e)(ii) computes the address of the word in the *work area* that contains the address sensitive instruction. It is computed by calculating the offset of this instruction within the program of the object module and adding it to the start address of the *work area*.

Algorithm 5.3 (First pass of the LINKER program)

1. *program_linked_origin* := <*load origin*>; (Use a default value if <*load origin*> is not specified.)
2. Repeat Step 3 for each object module to be linked.
3. (a) If an LNAMES record, copy the names found in the *name list* field into the NAMELIST array.
   (b) If a SEGDEF record
      (i) *i* := *name index*; *segment_name* := NAMELIST [*i*];
      segment_addr := origin specification in the *attributes* field (if any);
      (ii) If an absolute segment, enter (*segment_name*, *segment_addr*) in NTAB.
      (iii) If a relocatable segment that cannot be combined with other segments
         A. Align the address contained in *program_linked_origin* on the next word or paragraph as indicated in the *attributes* field.
         B. Enter (*segment_name*, *program_linked_origin*) in NTAB.
         C. *program_linked_origin* := *program linked origin* + *segment length*;
   (c) For each PUBDEF record
      (i) *i* := *base* field of the PUBDEF record;
      segment_name := NAMELIST [*i*];
      symbol := *name*;
      (ii) *segment_addr* := linked address of *segment_name* in NTAB;
      (iii) *sym_addr* := *segment_addr* + *offset*;
      (iv) Enter (*symbol*, *sym_addr*) in NTAB.

Example 5.13 (First pass of the LINKER program)

...resumed.

Algorithm 5.4 (Second pass of the LINKER program)

1. *list_of_object_modules* := Object modules named in the LINKER command;
2. Repeat Step 3 until *list_of_object_modules* is empty.
3. Select an object module and process its object records.
   (a) If an LNAMES record
       Enter the names in NAMELIST.
   (b) If a SEGDEF record
       (i) *i* := *name index*;
       (ii) *segment_name* := NAMELIST [*i*];
       (iii) Enter (*segment_name*, *linked address* from NTAB) in the i^{th} entry of the SEGTAB.
   (c) If an EXTDEF record
       (i) *external_name* := name from EXTDEF record;
       (ii) If *external_name* is not found in NTAB, then
            A. Locate an object module in the library which contains *external_name* as a segment name or a public definition.
            B. Add name of the object module to *list_of_object_modules*.
            C. Add the symbols defined as public definitions in the new module to the NTAB by performing the first pass, i.e., Algorithm 5.3, for the new object module.
       (iii) Enter (*external_name*, *linked address* from NTAB) in EXTTAB.
   (d) If an LEDATA record
       (i) *i* := *segment index*; *d* := *data offset*;
       (ii) *program_linked_origin* := SEGTAB [i].*linked address*;
       (iii) *address_in_work_area* := address of *work_area* + *program_linked_origin* − <*load origin*> + *d*;
       (iv) Move data from LEDATA into the memory area starting at the address *address_in_work_area*.
   (e) If a FIXUPP record, for each FIXUPP specification

(i) i := *frame datum*; f := *offset* from *locat* field;

(ii) *program_linked_origin* := SEGTAB [i].*linked address*;

(iii) *fix_up_address_in_work_area* := address of *work_area* +
program_linked_origin − <*load origin*> + f;

(iv) Perform the required fixup using the linked address from the SEGTAB or the EXTTAB according to codes in the *locat* and *fix data* fields.

(f) If a MODEND record

If a start address is specified, compute the corresponding linked address (analogous to the computation while processing an LEDATA record) and record it in the executable file being generated.

Example 5.14 (Second pass of the linker program) Figure 5.8(a) illustrates contents of the various data structures before the object module SECOND is processed by the second pass of the LINKER program. While linking the object modules of Example 5.11 autolinking would be performed for the external name GAMMA of object module SECOND. Let GAMMA be a public definition in object module THIRD. The first pass of LINKER is performed for THIRD, so the segments and public symbols defined in THIRD are added to NTAB. It includes PRECISE, a segment name and EMP_DATA, another public definition (see Figure 5.8(b)). The name THIRD is also added to the list of object modules. It ensures that the second pass would be performed for the object module THIRD.