SYSTEM SOFTWARE
LINKER

Nguyen Huu Duc

School of Information and Communication Technology
Hanoi University of Science and Technology

What is a linker?

- A linker is a system software that plays a role of combining multiple objects into a single executable program.
  - Input are object files being linkable
  - Output is a new object file

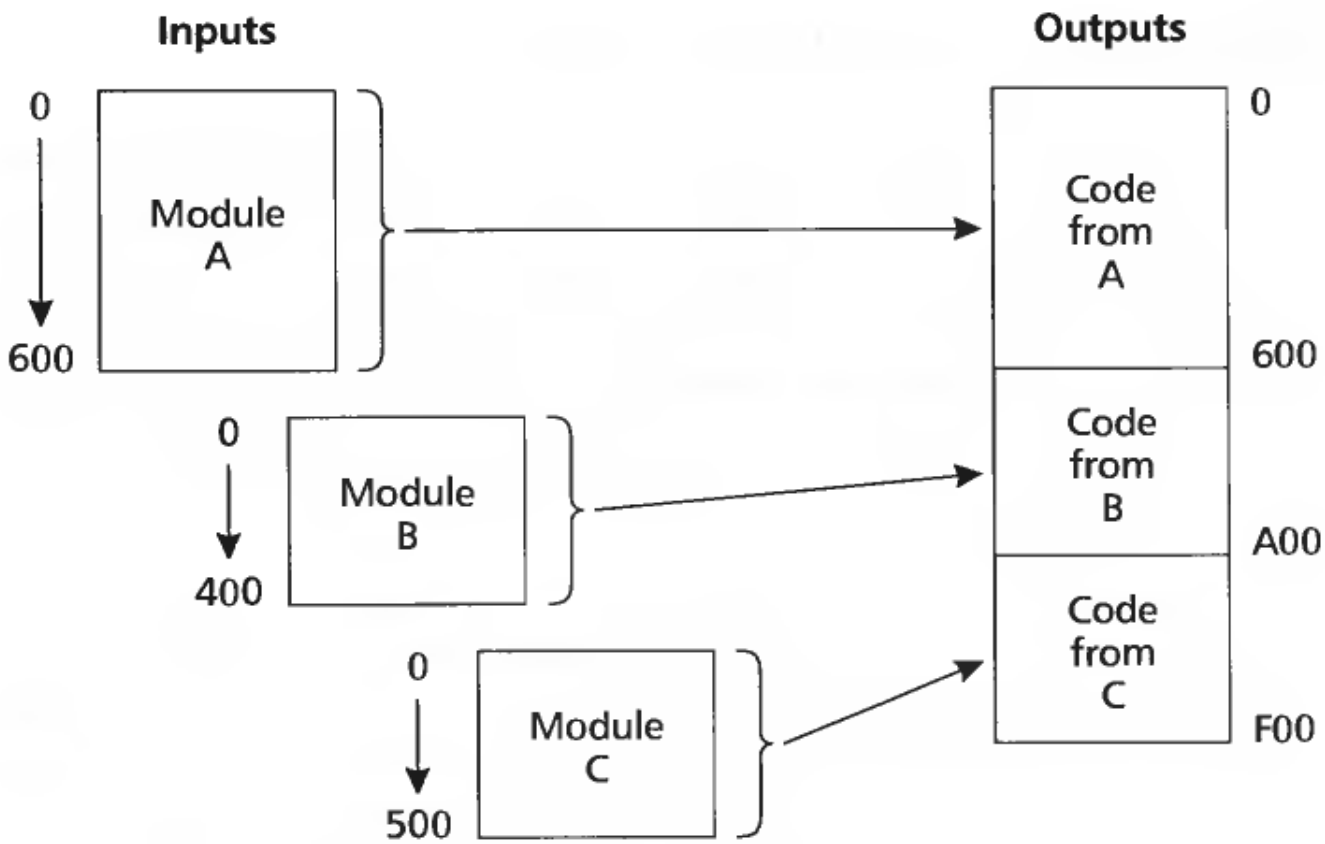
Activities of a linker

- Plan memory for the output object file.
  - Calculate size of memory for each segment (program/data) of the input object files
  - Pair segments being the similar role on the input object files
  - Assign address to each segments
- Symbol management
  - Manage function, variable: name, data type, address
  - Manage the range of symbols
  - Assign address to each of symbol
- Relocate variables and data based on the above information

Memory Planning

- Object file is organized into segment for executable code and data.
  - Executable code (text)
  - Data
    - Initialized data (data)
    - Un-initialized data (bbs)
- Each segment is assigned a base address matching with memory model
  - Objects defined in segment is relatively referred to this base address
- Segments are not allowed to overlap
- Linker calculates size of each segment of the input object file, pairs segments having the similar role into a single segment, and plans memory for each of these segments on the output object file.
  - The base address for segment is assigned exactly
  - Avoid redundancy

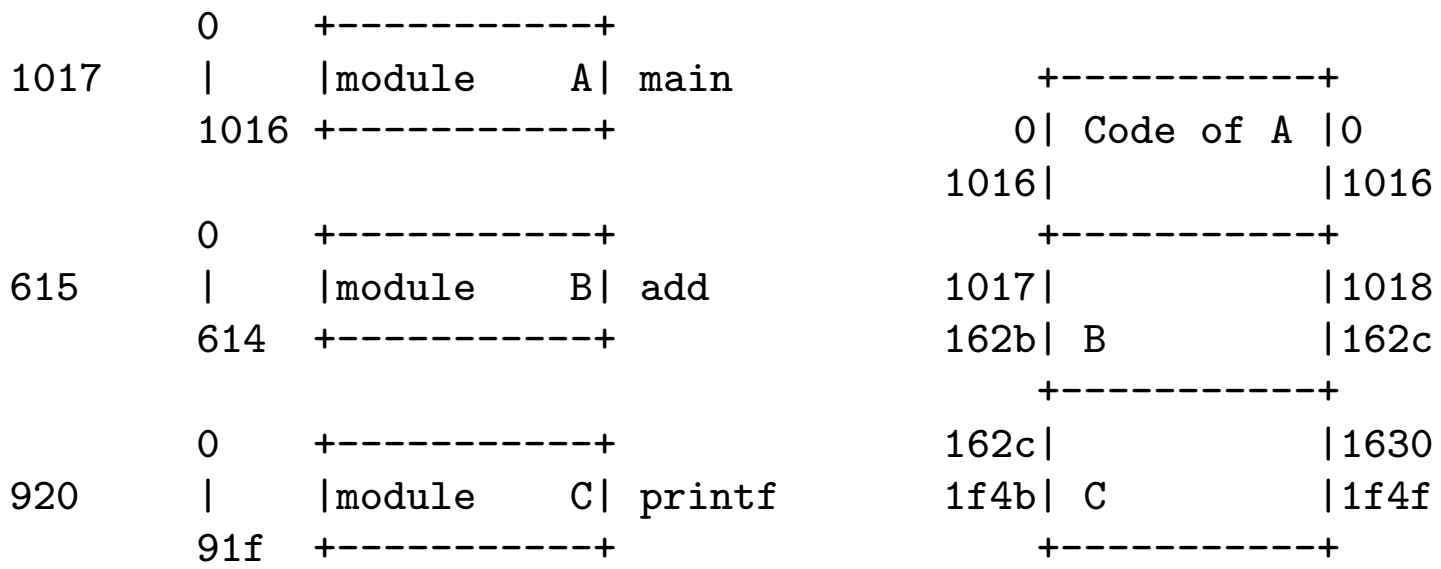
Ví dụ


Inputs
Outputs
0
Module A
600
0
Code from A
600
0
Module B
400
Code from B
A00
0
Module C
500
Code from C
F00

Segment alignment

- On some processors, base address of segment needs to be located appropriately
  - PowerPC: 8 byte alignment - base address must be divisible by 8
  - Intel32: Not need to be aligned, however, performance will be improved if segment address is divisible by 4 (4byte alignment)

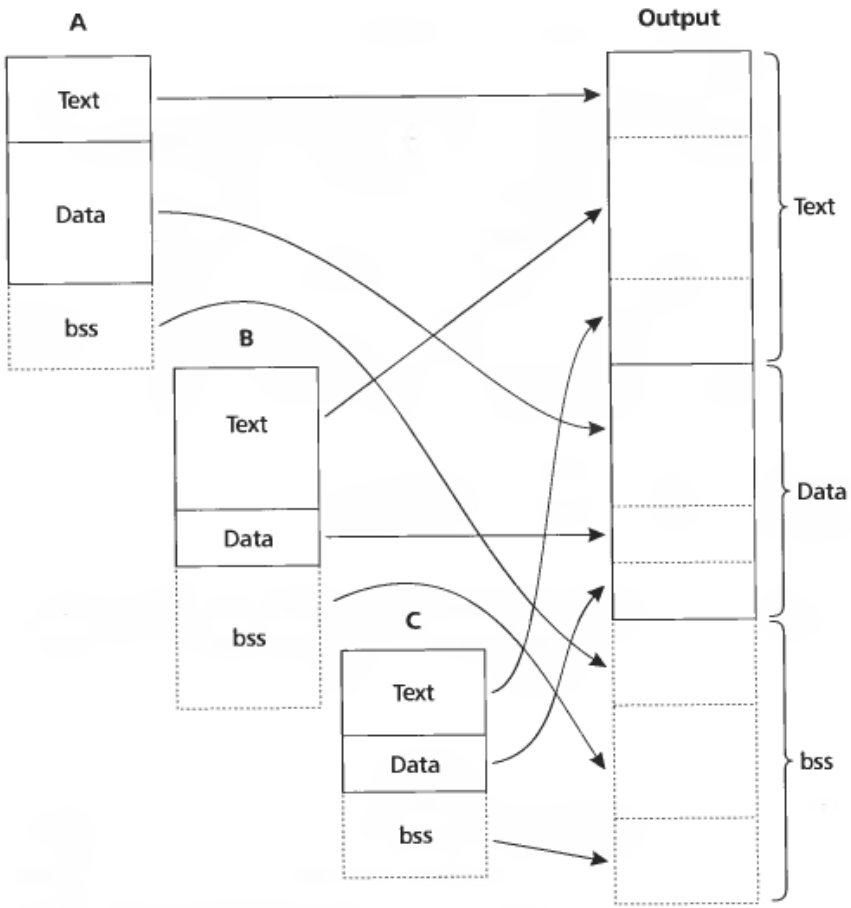
Example


0
1017
1016
0
615
614
0
920
91f
module A
main
module B
add
module C
printf
0
Code of A
1016
0
1016
1017
1018
162b
B
162c
162c
1630
1f4b
C
1f4f

Memory planning when linking multiple-segment object file

- Most object file formats contain many segments
  - E.g. a.out
    - text
    - data
    - bbs
- Linking object files which has many segments requires a combination of correspondent segments of each object file into a single segment in the output object file, and plans a base address suitable for these segments.

Example


A
Text
Data
bss
B
Text
Data
bss
C
Text
Data
bss
Output
Text
Data
bss

Memory planning when linking multiple-segment object file

- Two phase of linking multiple-segment object files
  1. Read object files as input, determine size of segments
  2. Relocate segments in the output file
- Memory alignment
  - For the input segments which belongs to the same output segment, using the method of word alignment
  - For the different output segments, using the method of page alignment (Page size depends on hardware requirements, E.g. 4KB)
  - Data and BBS, using the word alignment because actually they have the same feature

Ví dụ

	text	data	bss
main	1017	320	50
add	920	127	100
minus	615	300	840
printf	1390	1213	1400

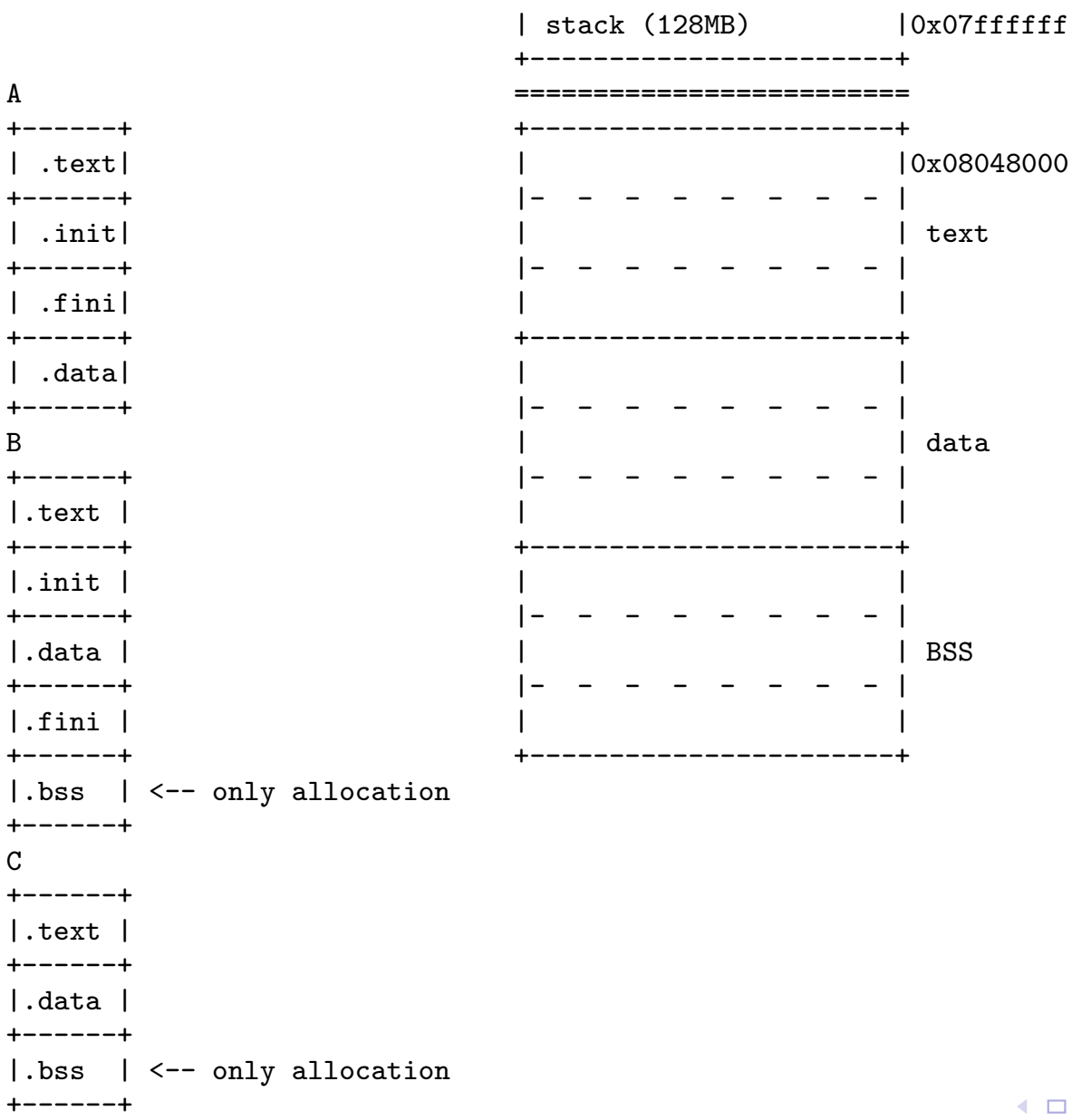
Suppose that page size is 1000h:

	text	data	bss
main	1000-2016	5000-531f	695c-69ab
add	2018-2937	5320-5446	69ac-6aab
minus	2938-2f4c	5448-5747	6aac-72eb
printf	2f50-42df	5748-695a	72ec-86eb

Memory planning for ELF

- Structure of the object file ELF is more complex than that of a.out (sections)
  - .text
  - .data
  - .bbs
  - .init
  - .fini
  - .rodata
  - .data1
  - Dynamic linking supported sections
- Segments are not loaded from the address 0
  - Stack is loaded above the program code section
  - Heap is loaded below the program code section
- Memory planning for ELF
  - Similar to a.out, but combining sections which have the same feature

Example


A
.text
.init
.fini
.data
B
.text
.init
.data
.fini
.bss
<-- only allocation
C
.text
.data
.bss
<-- only allocation
stack (128MB)
0x07fffffff
0x08048000
text
data
BSS

Symbol management

- Symbol plays an important role in references between object files
- Information about symbols are stored in the symbol table (and the name table) of object file
- Besides, symbols are used as debug information or for management of sections/memory segments

Information about symbols in ELF

```
int name;       // name (index in the name table)
int value;      // Value (Relative for relocation file
                          Absolute for executable file)
int size;       // size of object or function
char type:4;    // Type (object, function, section, or special symbol)
char bind:4;    // scope (local, global, weak)
char other;     // reserve
short sect;     // number of segment, ABS, COMMON, or UNDEF
```

Types of symbol

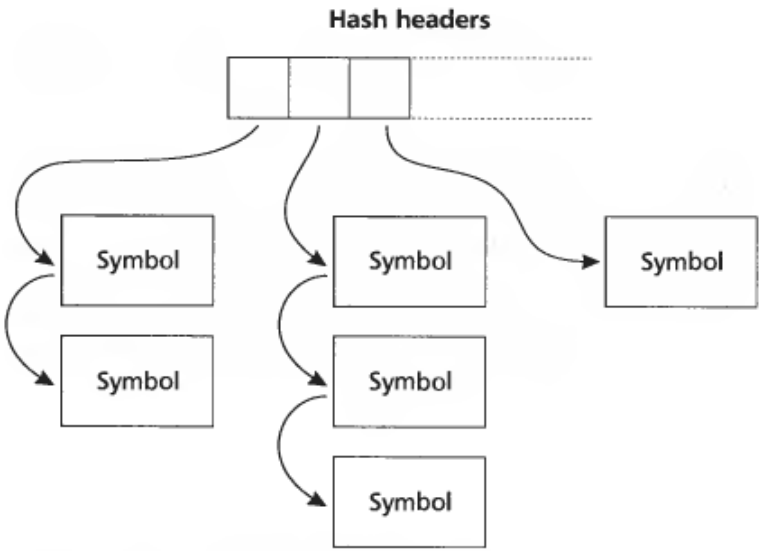
- Global symbol is defined in object file (is referred in the range of module or from other module)
- Global symbol is referred in an object file, but is defined in an other object file
- Name of sections
- Local symbol (For debug)
- Row index of source program (For debug)

Manage the symbol table in the linker

- Read the symbol table from object files
- Build a integrate symbol table
- Link the reference section with the definition section of symbol
- Extract a part of symbol information if necessary

Manage the symbol table when linking

- When linking, in order to access fast, symbols in the symbol table is managed differently with symbols in files
  - Hash array



Manage the symbol table when linking

```
struct sym symhash[N];          ... hash header

struct sym {
    struct sym *next;
    int fullhash;               ... hash value
    char *symname;              ... symbol
    long value;
    int size;
    char defined;               ... times defined
    char referenced;            ... times referred
};
```

Manage the symbol table when linking

- Symbols are classified, stored in N batches
- Each symbol s will be stored at the batch i which is calculated by the formular (h % N) where h is hash value of s
- Whenever s is defined, value of "defined" will increase by 1
- Whenever s is referred, value of "referenced" will increase by 1

Name mangling

- Symbol's name in the symbol table is usually not identical to the name defined in source program
  - Avoid conflict
  - Overriding function
  - Type checking support
- E.g. function in C++
  - Function `func(float, int, unsigned char)`
  - Adjust: `func_FfiUc`

Some other types of symbol

- Symbol 'weak': not necessary to be linked to the definition
- Support debugging
  - Row index
  - Name, type, and location of varialbe (local)

Relocation

- Relocation is the process of assigning load addresses to various parts of [a] program and adjusting the code and data in the program to reflect the assigned addresses
- Two kind of relocation mechanism
  - Hardware relocation
    - Physical address of functions and variables is usually not determinable, however, the address in their virtual address space is determinable. Hardware (MMU) undertakes the conversion from virtual address to physical address.
  - Software relocation
    - The linker or the loader performs a relocation for functions/variables to create a loadable file suitable with the virtual address space and can be loaded by hardware (EXE starts from the address 0, a.out starts from the address 0x1000, ELF starts from the address 0x08048000)

Relocation when loading

- In most cases, relocation when loading (by hardware) is not necessary because the loading program was relocated by the linker
- For a dynamic linking library DLL or a shared library, relocation when loading is necessary to ensure that they are in the same location in the virtual address space
- Relocation when loading is usually much simpler than relocation when linking

Relocation when linking

- Adjust address for segments and symbols
- Relocate reference to function and variable in the code

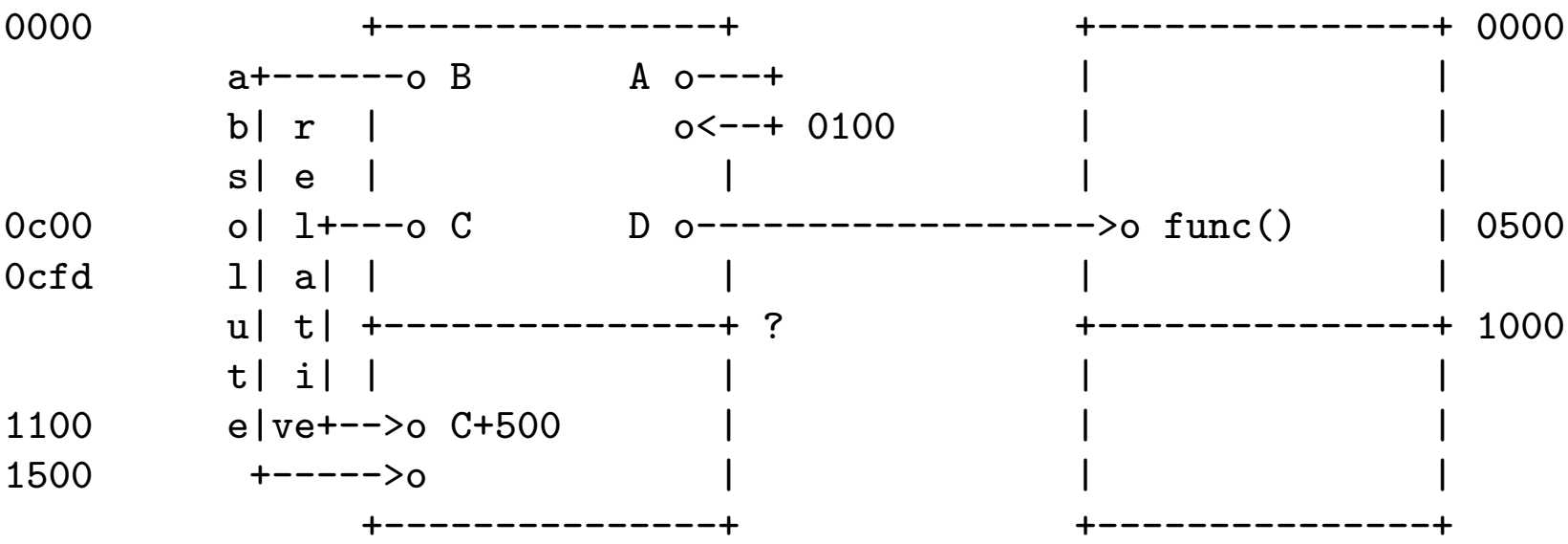
Adjust address of segments and symbols

- Plan memory for segments
- Collect information about symbols (function, data) which is defined in segments
- Adjust information about address of segments and symbols in the symbols table

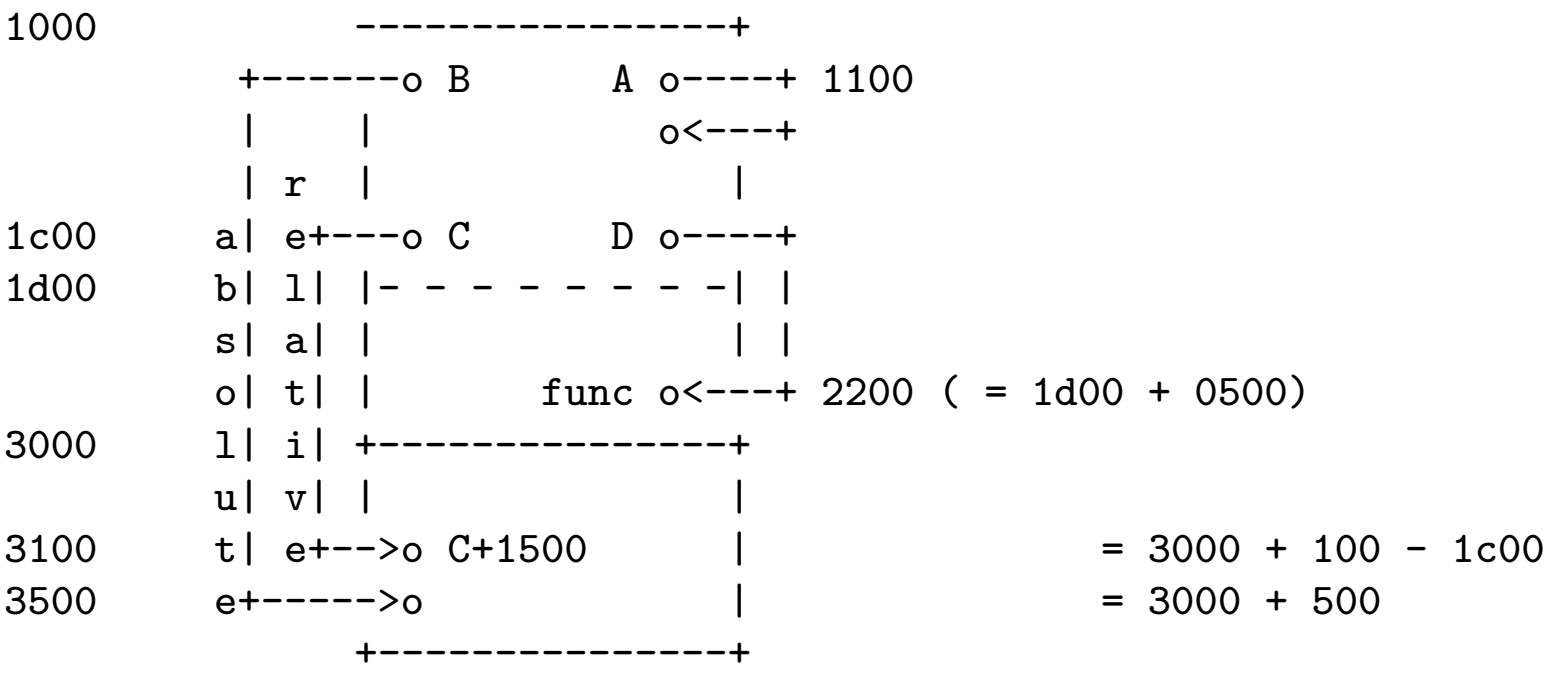
Relocate reference to function and variable

- Local absolute reference in the same segment
- Local relative reference between segments
- reference to global symbols

Example - before linking



Example - after linking



Relocation technique

- Simple case: The relocation table stores a list of segment addresses necessary to relocate, the linker just adds the base segment address to the value stored at that address.
- With the system of virtual address, the relocation table is more complex
  - Relocation table for data segment
  - Relocation table for executable code

A relocation item for a.out

```
struct relocation_info {
int address;    /* offset */
unsigned int    r_symbolnum:24,
                r_pcrel:1,
                r_length:2,
                r_extern:1,
                r_baserel:1,
                r_jmptable:1,
                r_relative:1,
                r_copy:1;
};
```