

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 4

Macros and Loaders

text substitution before assembly; loading and relocating after it

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

CLASS

4 / 10

19 slides, taken from the full course:

Lecture 4, Lecture 5.1

A macro processor runs before the assembler



C, Lecture 1:

SAME AS C

You have used one already

In Lecture 1, `gcc -E` turned 7 lines of `hello.c` into 820: that was the C preprocessor pasting `#include` files and expanding macros. It is a macro processor.

DIFFERENT FROM C

It only copies text

A macro processor knows nothing about instructions or addresses. It replaces names by lines of text. The assembler (or compiler) sees only the result.

Book: Beck §4.1

Defining and calling a macro

the two definitions (maccnt.asm)

```
CHKCH    MACRO    &CH,&HIT
          LDA      ZERO
          LDCH     TEXT,X
          COMP     &CH
          JEQ      &HIT
          MEND
INCR      MACRO    &VAR
          LDA      &VAR
          ADD      ONE
          STA      &VAR
          MEND
```

two calls

```
LOOP     CHKCH    LETTERA,FOUND
FOUND    INCR      TOTAL
```

Prototype: CHKCH MACRO &CH,&HIT: the macro's name and its parameters. Parameter names start with &.

Body: The lines between MACRO and MEND. They are **stored, not assembled**.

MEND: Marks the end of the definition.

Call (macro instruction): The macro's name used like an opcode, with arguments: CHKCH LETTERA, FOUND.

DIFFERENT FROM C

Not C syntax

& marks a parameter; C uses plain names.

the C version

```
#define CHKCH(ch, hit) \
    if (text[i] == (ch)) \
        goto hit;

CHKCH('A', found);
```

SAME AS C

Same parts as C

#define NAME(params) is the prototype; the rest of the line is the body; the end of the line (or \) is MEND.

Book: Beck §4.1.1 (macro definition and expansion)

Expansion: the calls are replaced by the body

● ● ● source (after the definitions)

```
FIRST  LDX  ZERO
LOOP   CHKCH LETTERA, FOUND
NEXT   TIX  LENGTH
        JLT  LOOP
DONE   J    DONE
FOUND  INCR  TOTAL
        J    NEXT
```

Assembled and run: TOTAL = 3, the same as COUNT in Lecture 3.1.

● ● ● expanded source (real output)

```
FIRST  LDX  ZERO
.LOOP  CHKCH LETTERA, FOUND
LOOP   LDA  ZERO
        LDCH TEXT, X
        COMP LETTERA
        JEQ  FOUND
NEXT   TIX  LENGTH
        JLT  LOOP
DONE   J    DONE
.FOUND INCR  TOTAL
FOUND  LDA  TOTAL
        ADD  ONE
        STA  TOTAL
        J    NEXT
```

The call becomes a comment
.LOOP CHKCH LETTERA, FOUND
stays in the listing so you can see where the lines came from.

Arguments replace parameters
&CH → LETTERA, &HIT → FOUND.
Pure text: the processor does not check that LETTERA is a word.

The label moves to line 1
LOOP was on the call; now it is on the first generated line, so JLT LOOP still works.

Macro or subroutine? Real numbers

Count A's and N's in BANANA	Macros (CHKCH, INCR)	Subroutines (JSUB ... RSUB)
Instructions executed	81	139
Bytes of code	60	75
Bytes each extra use of CHKCH adds	12 (the whole body)	9 (LDA, JSUB, JEQ)
Passing arguments	free: text is copied in	extra code: put the argument in A, save it
Smaller from ...	1 to 6 uses	the 7th use on

Rule of thumb: a short piece used often and needing speed → macro. A long piece used in many places → subroutine.

```
gcc -S sumrun.c (real)
```

```
movl $7, %edi
call square
movl %eax, %edx
movl $49, %esi
# the macro became 49:
# no call at all
```

SAME AS C

C: macro vs function

A macro is copied in (fast, grows the code). A function is called (one copy, but call + return each time). C's `inline` tries to get both.

Book: Beck §4.1.1

Expanding LOOP CHKCH LETTERA,FOUND step by step

ARGTAB	value
?1	LETTERA
?2	FOUND

Before line 1: write `.LOOP CHKCH LETTERA,FOUND` as a comment.

Line 1: the call had the label LOOP and this line has none, so LOOP goes here.

After line 4: reaching MEND (DEFTAB[5]) ends the expansion; back to the source.

DEFTAB	stored line	what happens	output line
1	LDA ZERO	no ?: copy	LOOP LDA ZERO
2	LDCH TEXT,X	no ?: copy	LDCH TEXT,X
3	COMP ?1	?1 → LETTERA	COMP LETTERA
4	JEQ ?2	?2 → FOUND	JEQ FOUND

Substitution order: replace ?10 before ?1, or ?10 would become ARGTAB[1] followed by "0". The C lab code counts down from nargs for this reason.

Watch the macro processor work

The macro processor: define, then expand

DEFINEEXPAND

source: maccnt.asm

MACCNT START 2000
CHKCH MACRO &CH,&HIT
LDA ZERO
LDCH TEXT,X
COMP &CH
JEQ &HIT
MEND
INCR MACRO &VAR
LDA &VAR
ADD ONE
STA &VAR
MEND
FIRST LDX ZERO
LOOP CHKCH LETTERA,FOUND
NEXT TIX LENGTH
JLT LOOP
DONE J DONE
FOUND INCR TOTAL
J NEXT
TEXT BYTE C'BANANA'
LENGTH WORD 6
LETTERA WORD 65
ZERO WORD 0
TOTAL WORD 0
ONE WORD 1
END FIRST

DEFTAB

0 CHKCH MACRO &CH,&HIT
1 LDA ZERO
2 LDCH TEXT,X
3 COMP ?1
4 JEQ ?2
5 MEND
6 INCR MACRO &VAR
7 LDA ?1
8 ADD ONE
9 STA ?1
10 MEND

NAMTAB

CHKCH start 0 end 5
INCR start 6 end 10

ARGTAB

?1 = LETTERA ?2 = FOUND

expanded output

MACCNT START 2000
FIRST LDX ZERO
.LOOP CHKCH LETTERA,FOUND
LOOP LDA ZERO
LDCH TEXT,X

DEFTAB[2] has no parameter: copy it as it is.

Watch for

1. In **DEFINE**, DEFTAB fills but the output stays empty.
2. Parameter names turn into ?1, ?2.
3. **MEND** adds a line to NAMTAB.
4. In **EXPAND**, ARGTAB fills at each call.
5. The orange output line is the DEFTAB line with ?n replaced.

Click the video to play or pause. About 1½ minutes, no sound.

Three jobs: loading, relocation, linking



Loading

Copy the bytes of the T records into memory, then jump to the address in the E record.

absolute loader



Relocation

The program was assembled for address 0 but goes to 5000: fix every field that holds an address.

relocating loader



Linking

MAIN uses COUNTA from another file: find where COUNTA ended up and put that address into MAIN.

linking loader

C link: what happens when you type `./count`

ld (the linker) already did the linking when you ran `gcc`: it joined `main.o`, `counta.o` and `data.o` into one file. When you run it, the **kernel** loads the program's pieces into memory and **ld.so** loads and links the shared libraries (`printf` is in `libc.so.6`). Same three jobs, split between three programs.

Book: Beck §3.1 (basic loader functions)

The absolute loader: about ten lines of logic

count.obj (Lab 3 assembler, START 2000)

```
HCOUNT 0020000000A0
T0020001E04203300203350A0272820303020182C202D3820033...
T00201E1B0C20363C200F3C202442414E414E410000060000410...
T00209D03000001
E002000
```

1 Read **H**: name COUNT, start 2000, length A0. Check it fits in memory.

2 For each **T**: two hex characters → one byte; store at the address in columns 2–7.

3 At **E**: jump to 002000. The program runs.

the heart of it in C (sicload.c)

```
int start = hexn(r + 1, 6);
int len   = hexn(r + 7, 2);
for (int i = 0; i < len; i++)
    mem[start + i] = hexn(r + 9 + 2*i, 2);
```

SAME AS C

Like memcpy in a loop

A T record is `memcpy(mem + start, bytes, len)`.
The loader does not understand the program; it only copies bytes. Code and data look the same to it.

Book: Beck §3.1.1 (design of an absolute loader)

Who loads the loader? The bootstrap loader

boot.asm (our version of Beck's idea)

```
FIRST  CLEAR  A
      LDX    #128      . X = 80 hex
LOOP   JSUB   GETC      . 1st hex digit
      RMO    A,S
      SHIFTL S,4        . S = digit*16
      JSUB   GETC      . 2nd hex digit
      ADDR   S,A        . A = one byte
      STCH   0,X        . store at X
      TIXR   X,X        . X = X + 1
      J      LOOP
GETC   TD     INPUT      . wait for device
      JEQ    GETC
      RD     INPUT
      COMP   #4         . end of input?
      JEQ    128        . yes: go to 80
      ... '0'...'9','A'...'F' → 0..15 ...
```

The chicken-and-egg problem

A loader is a program, so something must load it. The bootstrap is so small that hardware (or a ROM) puts it at a **fixed** address and starts it.

It is an absolute loader

No H, T or E records: just a stream of hex characters from device F1, stored from address 80 upward.

End of input = go

Character X'04' (end of transmission) makes it jump to 80. Whatever was loaded there now runs.

Book: Beck §3.1.2 (a simple bootstrap loader), §3.4.3

The problem: addresses inside the code

Loc	Source	Object code	Address inside
0000	FIRST LDX ZERO	040033	0033
0003	LOOP LDA ZERO	000033	0033
0006	LDCH TEXT,X	508027	8027
0009	COMP LETTERA	280030	0030
000C	JEQ FOUND	300018	0018

count0.asm is assembled with **START 0**. LDA ZERO became 000033: "read the word at 0033".

Load it at 5000 without changes: ZERO is now at 5033, but the instruction still reads 0033: someone else's memory. Every address field must get **+5000**; the opcodes and the data must not.

```
● ● ● where.c: PIE (default), run twice (real)
```

```
$ ./where
main 0x55f030bf3199
global 0x55f030bf6010
$ ./where
main 0x55f50b9af199
global 0x55f50b9b2010
```

```
● ● ● gcc -no-pie, run twice (real)
```

```
main 0x401186 (both runs)
global 0x404028 (both runs)
```

Only the base moves (...3199 and ...6010 stay): a relocating loader at work. -no-pie is an absolute program.

Book: Beck §3.2.1 (relocation)

Method 1: modification records

```
count0.obj (sicas -m, Lab 3)
```

```
HCOUNT 00000000003C
```

```
T0000001E0400330000335080272...
```

```
T00001E1E0C00363C000F3C00244...
```

```
M00000104
```

```
M00000404
```

```
M00000704
```

```
M00000A04
```

```
M00000D04
```

```
M00001004
```

```
... 13 M records in all
```

```
E000000
```

1 **delta** = PROGADDR - start in H = 5000 - 0 = 5000

2 T: load the bytes at their address + **delta**

3 M M00000104: the 4 half-bytes at 0001 + **delta** → add delta to them

4 E: jump to entry + **delta** = 5000

```
./sicload -v -r 5000 count0.obj (real, first 3 M records)
```

```
M record: at 005001 0033 -> 5033
```

```
M record: at 005004 0033 -> 5033
```

```
M record: at 005007 8027 -> D027
```

Look at 8027 → D027: LDCH TEXT, X has the **x bit** set (8000). Adding 5000 leaves it set: 8027 + 5000 = D027.

Book: Beck §3.2.1 (relocation by modification records)

Loaded at 5000, and it still counts the A's

```
./sicload -x -r 5000 count0.obj (real)

Memory after loading:
005000 04503300 503350D0 27285030 3050182C
005010 502D3850 033C5024 00503618 50390C50
005020 363C500F 3C502442 414E414E 41000006
005030 00004100 00000000 00000001 xxxxxxxx
Ran 51 instructions, stopped at 005024: A=000003 X=000006
L=000000
Memory after running:
005030 00004100 00000000 03000001 xxxxxxxx
```

Instruction	assembled	at 5000
LDX ZERO	040033	045033
LDCH TEXT,X	508027	50D027
J DONE	3C0024	3C5024
WORD 65	000041	000041

Opcodes and constants did **not** change.

SAME AS C

Same as C pointers

An address is "base + offset". Relocation keeps every offset and changes the base, like moving an array: `&a[i]` changes, `i` does not.

DIFFERENT FROM C

Different in a PIE

x86-64 code mostly uses **RIP-relative** addresses (like SIC/XE's PC-relative), so the code needs almost no fixing at load time.

PART 4

The linking loader

Three files, assembled separately, joined in memory with one table: ESTAB.

Pass 1: give every section an address, fill ESTAB

1 CSADDR = PROGADDR = 4000

2 H record: enter the section name with CSADDR and CSLTH

3 D record: enter each name with CSADDR + its address

4 next section: CSADDR = CSADDR + CSLTH

5 Same name twice? Error: duplicate external symbol

Section	Symbol	Address	Length	How
MAIN		4000	11	PROGADDR
COUNTA		4011	28	4000 + 11
DATA		4039	C	4011 + 28
	TEXT	4039		4039 + 0
	LEN	403F		4039 + 6
	TOTAL	4042		4039 + 9

Pass 1 loads nothing. It reads only H and D records. R, T and M wait for Pass 2. The result is the **load map** that `./sicontrol -l` prints first; it matches this table exactly.

Book: Beck §3.2.3 (Pass 1)

Pass 2: load T records, apply M records with ESTAB

1 H: CSADDR = this section's address from ESTAB

2 T: store the bytes at CSADDR + address

3 M: look up the name in ESTAB; add or subtract its address

4 E of the main section: remember the entry point

Read row 1: at 4001 the instruction 4B100000 (+JSUB, address 0). ESTAB says COUNTA = 4011. Add it: **104011**. The flag half-byte 1 (e = 1) is outside the 5 half-bytes, so it stays.

M record says	at	before	after
+COUNTA	004001	100000	104011
+TOTAL	004005	100000	104042
+TEXT	00400B	000000	004039
+TOTAL	00400E	000000	004042
-TEXT	00400E	004042	000009
+LEN	004016	100000	10403F
+TEXT	00401C	900000	904039

-TEXT comes right after **+TOTAL** on the same word: $004042 - 4039 = 000009$, the distance from TEXT to TOTAL. It does not depend on PROGADDR.

Watch the linking loader work

Linking loader: Pass 1 builds ESTAB, Pass 2 loads and fixes

PASS 1PASS 2RUN

```
main.obj
HMAIN 0000000011
RCOUNTATOTAL TEXT
T00000114B100000F100003F2FFD00000000000
M00000105+COUNTA
M00000505+TOTAL
M00000B06+TEXT
M00000E06+TOTAL
M00000E06-TEXT
E000000

counta.obj
HCOUNTA0000000028
RTEXT LEN
T0000001EB410B44077100000B4005390000029004133200AB8503B2FEFAC404F0000
T00001E0AAC40190001AC043F2FEC
M00000505+LEN
M00000B05+TEXT
E

data.obj
HDATA 0000000000C
DTEXT 00000LEN 000006TOTAL 000009
T0000000C42414E414E41000006000000
E
```

ESTAB (external symbol table)

section	symbol	address	length
MAIN		004000	000011
COUNTA		004011	000028
DATA		004039	00000C
	TEXT	004039	
	LEN	00403F	
	TOTAL	004042	

CSADDR = 4000

memory															
004000	4B	10	40	11	0F	10	40	42	3F	2F	FD	00	00	00	00
004010	00	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004020	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004030	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004040	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

M00000505+TOTAL: at 4005, the address field of a format 4 instruction (5 half-bytes). Look up TOTAL in ESTAB: 4042. Add it: 100000 becomes 104042.

Watch for

1. In **PASS 1** memory stays empty: only ESTAB grows.
2. Each **CSADDR** = previous one + previous length.
3. In **PASS 2** each T record fills orange bytes.
4. Each M record changes only a few bytes (teal).
5. **RUN**: TOTAL becomes 03.

Click the video to play or pause. About 1¾ minutes, no sound.

Key ideas from Class 4

- 1 A macro processor replaces each call by the macro body with the arguments filled in, before the assembler runs.
- 2 A macro costs no call at run time but makes the program bigger; a subroutine is the opposite.
- 3 A loader copies T records into memory (loading), fixes address fields (relocation) and joins sections (linking).
- 4 A bootstrap loader is a tiny absolute loader that loads the real loader.
- 5 The linking loader's Pass 1 builds ESTAB (external symbol table); Pass 2 loads and applies M records.

Homework and practice

Homework 4 (bring it to the next class)

1. Expand a call of the INCR macro from the slides by hand. What does the assembler then see?
2. count0.obj was assembled at 0000 and is loaded at 5000. What happens to the field 0033? To the opcode?
3. Two errors appear in every linker: undefined symbol and duplicate symbol. In which pass is each found?

Try it in the browser

sicload: run the loader on real object files

[labs/lab05.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: Linking and the ELF Format