$C \rightarrow$ Assembly and Assembly $\rightarrow$ C

Amir

November 17, 2025

- The -S flag tells GCC to **stop after producing assembly**.
- Useful for understanding how C code is lowered into machine-level instructions.
- Command format:

```
gcc -S filename.c
```

- Produces a file named `filename.s` containing assembly code.
- No object file (`.o`) or executable is generated at this stage.

What Appears in the .s Assembly File

- The generated .s file includes:
  - Function prologues/epilogues
  - Stack frame setup and teardown
  - Register usage
  - Instructions corresponding to expressions and control flow
- Example:

```c
int add(int x, int y) {
    return x + y;
}
```

```
add:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    -8(%rbp), %eax
    popq    %rbp
    ret
```

1. C: single variable - declaration and initialization

```c
// single.c
#include <stdio.h>
int main() {
  int x = 42;
  printf("x = %d\n", x);
  return 0;
}
```

2. Assembly (gcc -S): single variable

```
# single.s  (x86-64, AT&T syntax produced by gcc -S)
.globl main
main:
  pushq %rbp
  movq %rsp, %rbp
  movl $42, -4(%rbp)
  movl -4(%rbp), %edi
  leaq .LC0(%rip), %rsi
  xorl %eax, %eax
  call printf@PLT
  movl $0, %eax
  popq %rbp
  ret
.LC0:
  .string "x = %d\n"
```

3. Mapping: C variables to stack slots / registers

```
// Conceptual mapping:
// int x --> allocated at -4(%rbp)
// printf call uses calling convention: first arg in %
   edi (format) or %rdi for x86-64 System V
// Note: compiler optimizations can put 'x' in a
   register instead of stack
```

4. C: printf with expressions

```c
#include <stdio.h>
int main() {
  int a = 5, b = 7;
  printf("sum = %d\n", a + b);
  return 0;
}
```

5. Assembly: printf with expression

```
# sum.s
main:
  pushq %rbp
  movq %rsp, %rbp
  movl $5, -4(%rbp)
  movl $7, -8(%rbp)
  movl -4(%rbp), %edx   # a
  movl -8(%rbp), %eax   # b
  addl %edx, %eax       # eax = a + b
  movl %eax, %esi       # second arg (sum)
  leaq .LC0(%rip), %rdi # format string
  xorl %eax, %eax
  call printf@PLT
  ...
```

6. C: scanf example

```
#include <stdio.h>
int main() {
  int x;
  scanf("%d", &x);
  printf("you entered %d\n", x);
  return 0;
}
```

7. Assembly: scanf usage (stack & addresses)

```
# scanf.s
# &x is passed to scanf as pointer
leaq -4(%rbp), %rsi    # address of x -> second
    argument for scanf
leaq .LC0(%rip), %rdi  # format string
xorl %eax, %eax
call scanf@PLT
```

8. C: multiple variables

```c
#include <stdio.h>
int main(){
  int x=1,y=2,z=3;
  printf("%d %d %d\n", x,y,z);
  return 0;
}
```

9. Assembly: multiple variables

```
# multiple.s
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl $3, -12(%rbp)
# load for printf: push/pop or move to registers
movl -12(%rbp), %edx # z
movl -8(%rbp), %esi  # y
movl -4(%rbp), %edi  # x
leaq .LC0(%rip), %rax
# call printf
```

10. C: if condition

```c
#include <stdio.h>
int main(){
  int n = 5;
  if(n > 0) printf("positive\n");
  else printf("non-positive\n");
}
```

11. Assembly: conditional branch

```
# if.s
movl $5, -4(%rbp)
movl -4(%rbp), %eax
cmpl $0, %eax
jg .Lpositive
# else
leaq .Lelse(%rip), %rdi
call puts@PLT
jmp .Lend
.Lpositive:
leaq .Lpos(%rip), %rdi
call puts@PLT
.Lend:
```

12. C: ternary operator

```c
int a=10,b=20;
int m = (a>b) ? a : b;
// m holds max
```

13. Assembly: ternary -> conditional move / branches

```
movl a(%rip), %eax
cmpl b(%rip), %eax
cmovle %ebx, %eax  # conditional move (if available)
# else use branch/jump
```

14. C: for loop

```c
for(int i=0;i<5;i++){
  printf("%d\n", i);
}
```

```
movl $0, -4(%rbp)     # i = 0
.Lloop:
movl -4(%rbp), %eax
cmpl $5, %eax
jge .Lend
# body: call printf
incl -4(%rbp)
jmp .Lloop
.Lend:
```

16. C: while loop

```c
int i=0;
while(i<10){
  i+=2;
}
```

17. Assembly: while loop

```
movl $0, -4(%rbp)
.Lwstart:
movl -4(%rbp), %eax
cmpl $10, %eax
jge .Lwend
addl $2, -4(%rbp)
jmp .Lwstart
.Lwend:
```

18. C: switch statement

```c
switch(x){
 case 0: puts("zero"); break;
 case 1: puts("one"); break;
 default: puts("other");
}
```

19. Assembly: switch -> jump table

```
# compiler may produce jump table:
cmpl $1, %eax
ja .Ldefault
jmp *(.LJTI8_. + %rax*8)
```

20. C: function call

```c
int add(int a,int b){
  return a + b;
}
int main(){
  printf("%d\n", add(2,3));
}
```

```
# add.s
add:
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -4(%rbp) # a
  movl %esi, -8(%rbp) # b
  movl -4(%rbp), %edx
  addl -8(%rbp), %edx
  movl %edx, %eax
  popq %rbp
  ret
```

22. Returning values: registers

```
// On x86-64 System V: return value in %eax (or %rax
   for 64-bit)
int f(){ return 123; }
// assembly: movl $123, %eax ; ret
```

23. C: local array on stack

```c
int main(){
  int a[3] = {1,2,3};
  printf("%d\n", a[1]);
}
```

24. Assembly: array addressing

```
# a is at -12(%rbp)
movl -8(%rbp), %eax   # load a[1]
# or compute address: leaq -12(%rbp), %rax ; movl 4(%
    rax), %eax
```

25. C: pointers example

```
int main(){
  int x=10;
  int *p = &x;
  *p = 20;
  printf("x=%d\n", x);
}
```

26. Assembly: pointers and dereference

```
leaq -4(%rbp), %rax  # address of x
movq %rax, -16(%rbp) # store pointer p
movq -16(%rbp), %rax
movl $20, (%rax)     # *p = 20
```

27. C: malloc example

```
#include <stdlib.h>
int *p = malloc(sizeof(int));
*p = 5;
free(p);
```

```
movl $4, %edi     # size argument to malloc
call malloc@PLT
# returned pointer in %rax
movq %rax, -8(%rbp)
```

29. C: struct usage

```
struct Point { int x; int y; };
struct Point p = {1,2};
printf("%d,%d\n", p.x, p.y);
```

30. Assembly: struct layout and access

```
# struct p at -8(%rbp)
movl -8(%rbp), %eax    # p.x
movl -4(%rbp), %edx    # p.y  (offsets depend on
    layout)
```

31. C: recursion example (factorial)

```
int fact(int n){
 if(n<=1) return 1;
 return n * fact(n-1);
}
```

32. Assembly: recursion - call stack

```
# Each call pushes return address and local frame;
    args in %edi, return in %eax
call fact
# compiler may optimize tail recursion (not in this
    example)
```

33. C: volatile and compiler optimizations

```c
volatile int flag = 0;
// prevents compiler from optimizing away reads/writes
```

34. C: inline assembly (GCC)

```
int x=1;
asm ("incl %0" : "+r" (x)); // increments x
```

35. Assembly: calling a C function from asm

```
# extern printf
leaq .LC0(%rip), %rdi
call printf@PLT
```

36. Assembling and linking

```
# assemble: gcc -c single.s -o single.o
# link: gcc single.o -o single
# or compile C and view assembly: gcc -S single.c -o
   single.s
```

37. Example: -O0 vs -O2

```
// -O0: many stack slots, obvious moves
// -O2: registers, inlined functions
// Always compare gcc -S -O0 and gcc -S -O2 to learn
   differences
```

38. C: getchar/putchar

```c
int c = getchar();
putchar(c);
```

39. Assembly: getchar -> call libc

```
call getchar@PLT
# return in %eax
```

```
// printf("%d %d", a, b):
// first arg (format) in %rdi, then %rsi, %rdx, %rcx,
    ...
```

41. Mixed project: example

```
// myfunc.s defines _myfunc
// main.c declares: extern int myfunc(int);
// compile: gcc -c myfunc.s ; gcc -c main.c ; gcc
    myfunc.o main.o -o app
```

42. Assembly add called from C

```
# add.s
.globl add
add:
  movl %edi, %eax
  addl %esi, %eax
  ret
```

43. main.c that calls add

```c
#include <stdio.h>
extern int add(int,int);
int main(){
  printf("%d\n", add(4,5));
}
```

44. Debugging: objdump and gdb

```
objdump -d a.out    # disassemble
gdb a.out
(gdb) disassemble main
(gdb) break main
```

45. Assembly: direct syscall (write)

```
movq $1,%rax    # syscall write
movq $1,%rdi    # fd=1 stdout
leaq msg(%rip),%rsi
movq $len,%rdx
syscall
```

46. Calling conventions

```
// System V (Linux x86-64): rdi, rsi, rdx, rcx, r8, r9
// Windows x64: rcx, rdx, r8, r9
```

47. Inline asm portability

```
// Inline asm is compiler-specific and fragile across
   architectures
// Prefer separate asm files for portability and
   clarity
```

48. Exercises

```
1) Convert simple loop C->asm
2) Write assembly that calls printf
3) Translate recursion sample both ways
4) Observe -O0 vs -O2 assembly differences
```

```
Registers: %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp
Common opcodes: mov, add, sub, imul, idiv, call, ret, cmp, jmp
```

```
// Summary:
// - Use gcc -S to view generated assembly
// - Use objdump/gdb to inspect binaries
// - Practice writing small .s routines and link with
   C
// Next: cover x86 calling ABI in depth and
   optimization effects
```