Lecture 1:

hello world.c structure:

```
#include <stdio.h>              // Contains printf declaration
int main()                      // Entry point of program
{
    printf("Hello, World!\n");  // Prints to the console
    return 0;                   // Exit with no error
}
```

gcc:

```
$ gcc hello_world.c -o hello_world    // output hello_world
$ ./hello_world
```

gcc function:

1. Compile your C code (hello.c) into machine code.
2. Link everything together (your code + lib, like printf from libc).
3. Produce an executable file called hello.

DATA TYPE & Size

char	1byte(8bits)	-128~127	Unsigned char	0~2^8-1
short	2byte (16bits)	-2^15 ~2^15-1	Unsign Short	0~2^16-1
int	4 byte (32bits)	-2^31 ~2^31-1	Unsign int	0~2^32-1
long	8 byte (64bits)	-2^63 ~2^63-1	Unsign long	~2^64-1
float	4 byte			
double	8 byte			

printf + format specifiers

%d	Integer in decimal
%u	Unsigned integer in decimal
%x	Unsigned integer in hexadecimal
%X	Unsigned integer in uppercase hexadecimal
%c	ASCII character
%f	Float or double
%s	String
%zu	size_t len=strlen(s); printf("length=%zu\n", len);

long size_of_float = sizeof(float); -> **4**
long size_of_long = sizeof(size_of_float);
->return **long** is 8

Arrays & memory layout

arrays sit in contiguous memory

```
int a[4];
int a[4] = {1, 2, 3, 4}
int a[] = {1, 2, 3, 4}

int a[4] = {1,2};          // 后面自动补 0
char s[] = "hi!";          // {'h','i','!','\0'}
```

```
int numbers[3]={42,69,420};
char letters[]={'o','m','g','!'};
int *p = numbers;
int *p = &numbers[0];
printf("%d\n", *p);      //42
printf("%d\n", numbers[0]); // 42
*p = 100;
printf("%d\n", numbers[0]);  // 100

printf("%p\n", (void *)p);
```

numbers会decay成指向第一个元素的指针，类型是int*

Pointers & pointer arithmetic

int *p	声明指针，变量名前
*p	解引用指针 dereference, the object pointed to by indirect
&value	address-of operator 取地址运算符

```
// We can take the address of some data using &
int value = 50;        // 50 is stored somewhere in memory
int *pointer = &value; // pointer holds the address where it is stored
*pointer = 20;         // By dereferencing the pointer,
```

Poiner size: 取决于机器是 32 位还是 64 位

```
int value = 50;
Int *pointer = &values;
pointer       // assume address is 1000
pointer + 1   // address is 1004,(int is 4byte)
```

Arrays and pointers:

```
int array[4];
int *point = array; // array decay to &array[0]
array[0] = 96;
pointer[0] = 96;
```

Strings C string = char array ending with '\0'

```
char s[] = "Hello";             // 数组，内容可改
char s[6] = "Hello";            // 指定长度
const char s2[] = "Hello";      // 数组，内容不该被改
char *p = "Hello";              // 指向字面量，不要改内容
const char *str = "Hello";      // 推荐：指向字面量，只读
```

Sample

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr + 2;                 // *ptr (dereference) is 30
```

Q: Value?

- *ptr=30//ptr points to arr[2], ptr is a pointer(address)
- *(ptr+1) = 40 // point to arr[2+1] = arr[3]
- *(ptr-1) = 20 // point to arr[2-1] = arr[1]
- ptr[1] = *(ptr+1) = 40

Memory regions

- ***Static***(globals, string literals)
  - Space is reserved in the executable and loaded when the program starts.
- ***Stack***(local variables) When a **function** is called, space for its locals is pushed;
  - when it returns, that space is popped.
- ***Heap***(dynamically allocated)
  - Data is accessed via pointers, and you must free it manually.
  - malloc/calloc/realloc手动申请的内存

Q: What is the **Difference** between **stack** and **heap** memory?
A: Stack: automatic storage for local variables; managed by function calls/returns; fast; lifetime = scope.
Heap: manual allocation via malloc/free; flexible size and lifetime; slower; can leak or dangle.
Q. Why is returning the address of a **local variable wrong**?
A: Local variables live on the **stack**. Once the function returns, its stack frame is invalid, so the pointer becomes a **dangling pointer**, leading to undefined behavior.

Struct

```
struct Person{
  const char *name;
  int id;};
struct Person person = {
    .name = "Lothar",
    .id   = 1729,};
person.name = "Bob";                  // Setting a field
struct Person *indirect = &person;    // Pointer to an object
indirect->id = 1337;                  // Access through pointer
(*indirect).id = 1337;                // Same as above
```

Sample Cal Struct Memory

```
struct Point {
    char c;      // 1 byte ----> size 1, 3 bytes padding
    int x;       // 4 bytes----> size 4, no padding
    char d;      // 1 byte ----> size 1, 3 bytes padding
    int y; };    // 4 bytes----> size 4, no padding
```

*Q: What is **sizeof**(struct Point) and why? 16 bytes*
Best is 12 bytes-int, int, char, char = 4+4+1+1 = 10 < **12**
long, int, char, char = 8+4+1+1 < 16
Place larger-alignment members, and smaller ones later to reduce padding.

Enumerations

Named integer constants can be declared using an enum

```
enum Token_Kind
{
  TOKEN_NONE,                    // 0
  TOKEN_IDENTIFIER,              // 1
  TOKEN_INT_LITERAL,             // 2
  TOKEN_STRING_LITERAL,          // 3
};
```

Union

```
union Token_Data
{
    int int_value;
    const char *str_value;
};
```

Discriminated (tagged) union = enum tag + union data inside a struct

```
struct Token
{
    enum Token_Kind kind;
    union Token_Data data;
};
```

struct: All members occupy their own separate space in memory and exist at the same time.
union: All members share the same memory space, and only one member is meaningful at any given time.

File I/O basics

```
# Read entire file
with open("input.txt", "r") as f:
    content = f.read()
```

```
# Read line by line
with open("input.txt", "r") as f:
    for line in f:
        print(line)
```

```
# Write to file
with open("output.txt", "w") as f:
    f.write("Hello\n")
```

```
FILE *in_file = fopen("src.txt", "rb");
char buffer[1024];
int bytes_read = fread(buffer, 1, sizeof(buffer), in_file);
fclose(in_file);

FILE *out_file = fopen("dest.txt", "wb");
fwrite(buffer, 1, bytes_read, out_file);
fclose(out_file);
```

Lecture 2:

CPU executes instructions; RAM holds instructions and data.
Fetch-Decode-Execute(ALU)
CPU check Program Counter -> Decode(IR) -> ALU

3 Types of Operands 操作数(CPU)

Register: stored locally on cpu(rax, rbx) rax is 64 bits wide
Memory: in memory at a given address([rbp-8])
Immediate: baked into the instruction itself(5)

Register

- General Purpose Registers/Floating Point(SIMD) Registers
- Each 64-bit register (like rax) has smaller sub-register names (eax, ax, ah, al) that access the low 32/16/8 bits of the same physical register.

Cache is a small, fast memory located close CPU, used to keep recently or frequently used data. (RAM copies)
Pipelining: overlapping fetch/decode/execute of different instructions.
Virtual Memory
Each process has its own virtual address space; Varies from run to run

Virtual memory and an address space?
A:Virtual memory: each process sees a private, contiguous address space, mapped by the OS to physical memory (and disk).

Address space: the range of virtual addresses a process can use; includes code, heap, stack, etc.
Syscalls: To interact with the operating system, a special syscall instruction is used
Application Binary Interface (ABI) is a set of conventions that functions must follow so that separately compiled code and libraries can work together at the binary level.

Lecture3:Assembly

0	read	ssize_t read(int fd, void *buf, size_t count);
1	write	write(int fd, const void *buf, size_t count);
2	open	int open(const char *pathname, int flags, mode_t mode);
60	exit	: exit is 60 | mov rax, 60

rax	Accumulator(syscal number) Before each syscall, we put the system call number into rax. When we enter the kernel, it reads rax right then to decide which system call to execute. Retrun **value** stored in rax syscall 编号 & 返回值 在 rax mul / div / imul reg / idiv reg 默认都用 rax 做其中一个操作数
rdi	First parameter
2~6 parameter : rsi, rdx, rcx, r8, r9	
rsp	Stack Pointer 栈顶指针 rsp 永远指向当前栈顶 (stack top) 控制 push/pop 和返回地址
rbp	Base/Frame Pointer 栈帧基址寄存器
edi	rdi的低32位叫 EDI

```
global _start            ; Export this symbol
_start:                  ; label
  mov rax, 60            ; rax holds the syscall number (60 = exit)
  mov rdi, 42
  syscall
```

syscall:switches the CPU from user mode to kernel.

syscall执行以后，CPU 进内核，内核首先看 rax 是什么号来决定调用哪个系统调用.

Types of Memory Operands

```
[number]                   ; displacement only
[reg]                      ; base register only
[reg + number]             ; base + displacement
[reg + reg*scale]          ; base + index*scale (1, 2, 4, or 8)
[reg + reg*scale + number] ; base + index*scale + displacement
```

Mov instruction

```
mov reg, reg    ; Copy from a register to a register
mov reg, mem    ; Load memory into a register
mov reg, imm    ; Put an immediate in a register
mov mem, reg    ; Store a register value in memory
mov mem, imm    ; Store an immediate value in memory

mov dest, src   ; Think "dest = src"
```

Arithmetic and logical ops

```
add reg, reg    ; Add a register to a register
add reg, mem    ; Add a value from memory to a register
add reg, imm    ; Add an immediate to a register
add mem, reg    ; Add a register to a value in memory
add mem, imm    ; Add an immediate to a value in memory

add dest, src   ; Think "dest += src"
```

Instructions

mov rax, 3	assigning to a variable
add rax, 4	Arithmetic and logical operations Computes a result and stores it somewhere Also sets some status flags
jmp start	Flow control Changes the instruction pointer Used for loops, conditionals, and function calls

and d, s	mov rax, 0b1101 ; 13
	mov rbx, 0b1010 ; 10
	and rax, rbx ; rax = 0b1000 = 8

or d, s	mov rax, 0b0101 ; 5
	mov rbx, 0b1010 ; 10
	or rax, rbx ; rax = 0b1111 = 15

xor d, s	mov rax, 0b0110 ; 6
	mov rbx, 0b1010 ; 10
	xor rax, rbx ; rax = 0b1100 = 12

mul reg; imul reg;	rdx:rax = rax * reg rdx:rax = rax * reg
div reg; idiv reg;	rax = rdx:rax / reg; rdx = rdx:rax % reg (unsigned) reg ; rax = rdx:rax / reg; rdx = rdx:rax % reg
cqo	Sign extend rax into rdx This is needed to prepare for idiv

Flag

Zf	ZERO Flag
Sf	Sign Flag 结果的最高位是 1 → SF = 1 (as negative) 结果最高位是 0 → SF = 0, 用在有符号比较 (jg/jl/jge/jle) 里。

JUMP - signed

jmp label	cmp rax, rbx; 相当于做 (rax - rbx), 但不保存结果, 只改标志位 jz equal; 如果 rax == rbx, 就跳到 equal
jz label	If ZF = 1, jump. cmp rax, rbx; if rax==rbx, ZF=1
jnz label	*ZF = 0, jump*
jl label	*; jump if less (signed <)*
jle label	*; jump if less or equal (signed <=)*
jg lable	*; jump if greater (signed >)*
jge label	*Jump if >= (signed >=)*

JUMP - unsigned

jb label	; jump if below (unsigned <)
jbe label	; jump if below or equal (unsigned <=)
ja label	; jump if above (unsigned >)
jae label	; jump if above or equal (unsigned >=)

Call ret

call label	; Pushes the return address before jumping
ret	; Pops the return address and jumps there

Other Operations

inc dest	; dest += 1
dec dest	; dest -= 1

Load effective address

lea 不是真的"读内存"，只是把中括号里的地址公式算出来，存进寄存器。

lea reg, mem;	Compute the address for mem, store that in reg

Two's complement rule: negative numbers are represented as ~a + 1.

Assembly code Sample 1:

```
fn main() -> int
{
  return 42
}
```

```
global _start
_start:
  call main
  mov rdi, rax      ; return code
  mov rax, 60       ; exit syscall
main:
  mov rax, 42
  ret
```

Assembly code Sample 2:

```
fn main() -> int {
  let n: int = 0
  while n < 10 {
    call print_int(n)
    call print_nl()
    set n = n + 1
  }
  return n - 10
}
```

```
global _start
_start:
  call main
  mov rdi, rax ; return code
  mov rax, 60 ; exit syscall
  syscall
main:
  mov rdi, 0        ; n = 0, rdi = n
.while:
  cmp rdi, 10       ; rdi - 10
  jge .end_while    ; rdi >= 10?
  call print_int    ; rdi = n already
  call print_nl
  inc rdi           ; rdi += 1
  jmp .while
.end_while:
  mov rax, rdi      ; rax = rdi
  sub rax, 10       ; rax = n - 10

  ret        ; 从栈里弹出刚才那个地址，跳回来
```

Lecture4:

SAMPLE Register Operations

mov rax, 10
mov rcx, 3
imul rcx
add rax, 5
	1. mov rax, 10 → rax = 10
	2. mov rcx, 3 → rcx = 3
	3. imul rcx → rdx:rax = rax*rcx =10*3=30 (rax=30, rdx=0 for small result)
	低64位放在RAX|高64位放在 RDX
	4. add rax, 5 → rax = 30 + 5 = 35

mov rax, 100
mov rdx, 0
mov rcx, 3
div rcx
	1. 被除数低 64 位
	2. 被除数高 64 位 = 0, 所以 rdx:rax = 100
	3. 除数 = 3
	4. 做无符号除法: (rdx:rax) / rcx
	结果: rax = **商** = 33, rdx = **余数** = 1

mov rax, -100
cqo
mov rcx, 7
idiv rcx
	1. 被除数低 64 位 = -100 (有符号)
	2. 符号扩展到 rdx:rax, 如果 rax 是负数, 就让 rdx = -1
	所以现在 rdx:rax = -100 (128 位被除数)
	3. 除数 = 7 (有符号)
	4. 做有符号除法: (rdx:rax) / rcx
	结果: rax = 商 = -14, rdx = 余数 = -2

SAMPLE – Conditional Jumps

mov rax, -5
cmp rax, 0
jg .positive
	jg = jump if greater(signed)
	-5 < 0, so doesn't jump

mov rax, 5
mov rcx, 10
cmp rax, rcx
jl .less
mov rax, 0
jmp .end
.less:
mov rax, 1
.end:
ret
	1. mov rax,5→rax = 5
	2. mov rcx,10→rcx = 10
	3. cmp rax,rcx→Compute5-10=-5(negative), set flags (SF=1, ZF=0)
	4. jl .less → Jump if less (SF ≠ OF). Since 5 < 10, jump to .less
	5. mov rax, 1 → rax = 1
	6. .end: → Continue to ret
	7. ret → Return with rax = 1
	
	If rax was 15 (rax=15, rcx=10): -> cmp rax, rcx→15-10=5 (positive) jl .less → Don't jump (15 ≥ 10) mov rax, 0→rax=0
	jmp .end→ Skip_to_end | ret → Return 0
	Returns 1 if rax<rcx, otherwise returns 0

Assmebly code Sample 3:

```
int square(int x) {
return x * x; }

int main() {
    int a = 3;
    return square(a)
+ square(a + 1);
}
```

```
global square |  global main
section .text

square:
    mov eax, edi         ; eax = x
    imul eax, edi        ; eax = x * x
    ret                  ; return eax

main:
    push rbp
    mov  rbp, rsp

    mov  edi, 3          ; argument x = 3
    call square          ; eax = square(3)
    mov  esi, eax        ; save square(3) in esi

    mov  edi, 4          ; argument x = 4
    call square          ; eax = square(4)

    add  eax, esi        ; eax = square(3) + square(4)

    pop  rbp
    ret                  ; return eax as main's return value
```

EBNF = Extended Backus-Naur Form 语言的语法的格式

Helpful C Constructs

- Defining a "string type with an explicit length" 带长度的字符串

```
typedef struct Counted_String
{
    char *data;
    long count;
} Counted_String;
#define counted_str_lit(s) (Counted_String){(s), sizeof(s) - 1} 专门给"字符串字面量"用的
用来把普通 C 字符串 (char *, 以 '\0' 结尾) 包装成 Counted_String。
#define counted_cstr(s) (Counted_String){(s), strlen(s)}
```

- 比较两个 Counted_String 是否完全相等的函数

```
bool strings_match(Counted_String s1, Counted_String s2);
bool strings_match(Counted_String s1, Counted_String s2) {
    if (s1.count != s2.count) return false;            // 长度不一样一定不等
    for (long i = 0; i < s1.count; i++) {
        if (s1.data[i] != s2.data[i]) return false;    // 有任意一个字符不同就 false
    }
    return true;                                       // 全部一样
}
```

- 判断一个字符串是不是以某个前缀开头

```
bool starts_with(Counted_String str, Counted_String prefix) {
    if (prefix.count > str.count) return false;        // 前缀比整体还长，不可能
    for (long i = 0; i < prefix.count; i++) {
        if (str.data[i] != prefix.data[i]) return false;
    }
    return true;
}
```

- 判断 str 是不是以 suffix 结尾

```
bool ends_with(Counted_String str, Counted_String suffix) {
    if (suffix.count > str.count) return false;
```

```
    long start = str.count - suffix.count;            // 后缀在 str 中的起点
    for (long i = 0; i < suffix.count; i++) {
        if (str.data[start + i] != suffix.data[i]) return false;
    }
    return true;
}
```

- **strlen** is a C standard library function, declared in <string.h>.

```
#include <string.h>
const char *s = "hello";
size_t len = strlen(s);   // len = 5
```

Dynamic Arrays

A pointer plus a count and a capacity points at an expandable range of items.

Dynamic Array Example

```
typedef struct Token_Array
{
  Token *items;    // 指向 Token 的动态数组
  long count;     // 当前已经存了多少个 Token
  long capacity;   // 最多能存多少个 Token (容量)
} Token_Array;

void append_token(Token_Array *arr, Token token)
{
  if (arr->count >= arr->capacity) {        // count >= capacity
      arr->capacity *= 2;                   // capacity *= 2
      if (arr->capacity == 0) arr->capacity = 16;      // initial
      arr->items = realloc(arr->items,      // 扩到新内存, 并更新指针
                           arr->capacity*sizeof(*arr->items));
  }
  arr->items[arr->count++] = token; // 把新token 放到 items[count], 然后
count++}
```

Advantages: Size can be decided at runtime. Less waste, Better for building reusable data structures.

Dynamic Memory(The Heap)

```
void *malloc(size_t num_bytes);
  int *a = malloc(10 * sizeof(int));      // 要 10 个 int 的空间
void *calloc(size_t nums_items, size_t, bytes_per_item);
  int *a = calloc(10, sizeof(int));                // 10 个 int, 都初始化为 0
void *realloc(void *old_pointer, size_t new_num_bytes);
  a = realloc(a, 8 * sizeof(int));       // 扩成 8 个 int,可能会"换地址位置"

void free(void *pointer);
```

realloc: may change the address

```
Token *token = &arr->items[7];
append_token(arr, new_token);   // 可能触发扩容
printf("Kind: %d\n", token->kind);  // 这里出事
```

When a dynamic array resizes, it may move its data to a new buffer, so any previously saved pointer to an element becomes dangling, and dereferencing it means reading freed memory.

Switch Statements

```
char c = *ptr;
switch (c) {
case 'a': {
  // Handle c == 'a'
  } break; // Without break, fall through to cases below!
case 'b':
case 'c': {
    // Handle c == 'b' || c == 'c'
  } break;
default: // Handle all other cases
}
```

TWO main ways to combine:

- **Traditional**: compiler each file separately, combine during linking
  Slower full build, but good modularity, clear errors, fast incremental builds. [gcc -c a.c, gcc -c b.c, gcc -c main.c]
  [gcc a.o b.o main.o -o prog]
- **Unity**: #include all source
  Faster full build and better optimization, but more name/macro conflicts and worse incremental builds.

```
// foo.h Traditional header
#ifndef FOO_H
#define FOO_H

typedef struct Foo {
    int a, b;
} Foo;

// prototype
Foo make_foo(int a, int b);

#endif // FOO_H
```

```
// foo.c source
#include "foo.h"

// implementation
Foo make_foo(int a, int b) {
    return (Foo){
        .a = a,
        .b = b,
    };
}
```

Lecture 5:

Abstract Syntax Tree

```
fn main() -> int {
    let n: int = 0
    while n < 10 {
        call prt_int(n)
        call prt_nl()
        set n = n + 1
    }
    return n - 10
}
```

```
program
└── fn_definitions
    └── main
        ├── []        // 参数列表：空
        ├── int       // 返回类型：int
        └── body
            ├── let  // let n: int = 0
            │   ├── name: n
            │   ├── type: int
            │   └── value: 0
            ├── while // while n < 10
            │   ├── condition: <
            │   │   ├── identifier n
            │   │   └── integer 10
            │   └── body
            │       ├── call
            │       │   ├── name: prt_int
            │       │   └── arg: n
            │       ├── call // print_nl()
            │       │   └── name: prt_nl
            │       └── set
            │           ├── identifier n
            │           └── binary_op +
            │               ├── identifier n
            │               └── integer 1
            └── return // return n - 10
                └── binary_op -
                    ├── identifier n
                    └── integer 10
```

```
fn main() -> int {
return 5 }
```

```
PROGRAM
└── FN "main"
    ├── return_type: TYPE_INT
    └── body:
        └── RETURN
            └── INTEGER 5
```

Arrays - insert index

```
int insert_token_at(Token_Array *arr, long index, Token token)
{
    if (index < 0 || index > arr->count) return 0;

    if (arr->count >= arr->capacity) {
        long new_cap = arr->capacity ? arr->capacity * 2 : 16;
        Token *p = realloc(arr->items, new_cap * sizeof(*arr->items));
        if (!p) return 0;
        arr->items = p;
        arr->capacity = new_cap;
    }

    for (long i = arr->count; i > index; i--) {
        arr->items[i] = arr->items[i - 1];
    }

    arr->items[index] = token;
    arr->count++;
    return 1;
}
```

Linked List

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {                  // struct
    int value;
    struct Node *next;
} Node;
```

```
void push_front(Node **head, int value) {  // ll head insert new node O(1)
    Node *node = malloc(sizeof(Node));     // 分配新节点
    if (node == NULL) {
        perror("malloc failed");
        exit(1);
    }
    node->value = value;
    node->next  = *head;  // 新节点指向原来的头结点
    *head = node;   }      // head point to new node
```

```
void pop_front(Node **head) {         // O(1)
    if (!head || !*head) return;    // empty list: do nothing

    Node *old = *head;              // save old head
    *head = old->next;              // move head forward
    free(old);  }                    // free old head
```

```
void insert_after(Node *iter, int value) { // O(1)
    if (!iter) return;
    Node *node = malloc(sizeof(Node));
    if (!node) { perror("malloc failed"); exit(1); }
    node->value = value;
    node->next  = iter->next;
    iter->next  = node; }
```

```
void remove_node(Node **head, Node *prev, Node *iter) {   // O(1)
    if (!iter) return;
    if (!prev) *head = iter->next;    // iter is head
    else       prev->next = iter->next;
    free(iter); }
```

```
void remove_first(Node **head, int target) {               // O(n)
    Node *prev = NULL;
    Node *iter = *head;
    while (iter) {
        if (iter->value == target) {
            remove_node(head, prev, iter);
            return;
        }
        prev = iter;
        iter = iter->next; } }
```

```
void print_list(Node *head) {
    Node *p = head;
    while (p != NULL) {
        printf("%d -> ", p->value);
        p = p->next; }
    printf("NULL\n");  }
```

```
int main(void) {
    Node *head = NULL;
    push_front(&head, 1);
    push_front(&head, 2);
    push_front(&head, 3);
    print_list(head);

    insert_after(head, 99);
    print_list(head);
```

```
    remove_first(&head, 2); // del first 2
    print_list(head); // 3->99->1->Null
    pop_front(&head);  // del head 3
    print_list(head);  // 99 -> 1 -> NULL

    free_list(head);
    return 0; }
```

Doubly Linked List

```
typedef struct Node {
    int value;          // data
    struct Node *next;  // pointer to next node
    struct Node *prev;  // pointer to previous node
} Node;
```

+Can traverse forwards and reverse
-Takes up more memory
-More operations to insert/delete

Linked Lists	Arrays
• Fast insertion/deletion • Flexible memory allocation (nodes can be anywhere in memory) • Traversal depends on data order (must follow next pointers) • No random access • Extra memory for pointers in each node	• Compact, contiguous memory layout • Random access in O(1) by index (arr[i]) • Appends can be slow when resizing (need to copy to a bigger array) • waste space if capacity > size • Insert/delete near the start or middle is slow (need to shift many elements)

Hash Table

```
#define TABLE_SIZE 1024  // fixed size for exam

typedef struct Entry {
    const char *key;      // or char *key;
    int value;
    struct Entry *next;
} Entry;

typedef struct {
    Entry *buckets[TABLE_SIZE];   // array of bucket heads
} HashTable;
```

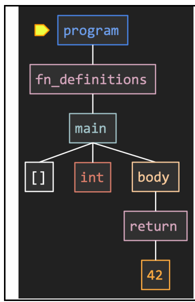
Binary Search

```
#include <stdio.h>

int binary_search(int *arr, int n, int target) {
    int left = 0;
    int right = n - 1;  // search in [left, right]

    while (left <= right) {
        int mid = left + (right - left) / 2;  // avoid overflow

        if (arr[mid] == target) {
            return mid;                       // found
        } else if (arr[mid] < target) {
            left = mid + 1;                   // search right half
        } else {
            right = mid - 1;                  // search left half
        }
    }
    return -1;  // not found
}
```




```
global _start

_start:
    call main
    mov rdi, rax   ; return code
    mov rax,60     ; exit syscall
    syscall

main:
    mov rax, 42
    ret
```

Lecture 6:

Sample 1

```
2 + 3 * 4 - 5
       (-)
      /   \
   (+)     5
   /  \
  2    (*)
       /  \
      3    4
```

```
IR
PUSH 2
PUSH 3
PUSH 4
MUL
ADD
PUSH 5
SUB
```

Sample 2

```
1 + 2 * 3 - 4 / 2
         -
       /   \
      +     /
     / \   / \
    1   *  4  2
       / \
       2  3
```

post-order:
1 2 3 * + 4 2 / -

```
PUSH 1
PUSH 2
PUSH 3
MUL     ; 2 * 3
ADD     ; 1 + (2 * 3)
PUSH 4
PUSH 2
DIV     ; 4 / 2
SUB
; (1 + 2*3) - (4 / 2)
```

Sample 3

```
PUSH 10
STORE a (slot 0)
PUSH 5
STORE b (slot 1)
LOAD a
LOAD b
MUL
RETURN
```

```
Store to a: a = 10
Store to b: b = 5
Compute 10 * 5 = 50
Final result: 50
Final memory:
a=10, b=5
```

```
let a: int = 10
let b: int = 5
return a * b
```

1.The Shunting Yard Algorithm

- Maintain a *stack of operators* and a *stack of operands*
- Read *tokens* from input expression
- If it's an operand (+ - * /) , push it onto operand stack (数字/子树)
- If it's an operator, compare *precedence* to top of operator stack
- While lower precedence than top of op stack: pop top and build trees

[When cur(-) ≤ stack_top (*+), pop stack_top, (*+)] 要入栈(*)的优先级高于栈顶(+)，就压入；要入栈(-)的优先级低(*)，就pop栈顶

- Push new operator onto operator stack
- Until operator stack is empty, pop and build trees

2.Fix it After

- First pass: build a **naive left-associative tree**, completely ignoring precedence.
- Second pass: traverse this tree and repair it using tree rotations so that higher-precedence operators (like *) move "down" into the correct place.

3. Including Precedence in Grammar (把优先级写进 Grammar 里)

- Encode precedence directly into the grammar / parser structure, so the parse tree is automatically correct.

4. Pratt Parsing

Pratt parsing recursively parses expressions with a minimum precedence

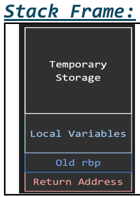
用一个函数 parse_expr(min_prec) 来递归，根据运算符优先级决定什么时候"停"

```
parse_expr(min_prec):
    lhs = parse_primary()  // 先读一个最基本的东西：数字/变量/括号表达式
    tok = next_token()
    while tok 是二元运算符 且 precedence(tok) > min_prec:
        rhs = parse_expr(precedence(tok))
        lhs = make_binary_node(lhs, tok, rhs)
        tok = next_token()
    return lhs
```

Lecture7:

Variable Storage:
Local variables live in the stack frame so each function call has its own copy.

Stack Frame:



function preamble and postamble

```
foo:
    push rbp         ; save old rbp
    mov  rbp, rsp    ; set new rbp
    sub  rsp, 24     ; make room for local variables
    ; function body goes here

    mov  rsp, rbp    ; restore rsp
    pop  rbp         ; restore old rbp
    ret
```

Top to bottom = low to high

The stack pointer **rsp** is not stable during evaluation of an expression. We reserve another register to keep a stable position within the stack frame.

Traditionally, the base pointer **rbp** is used for this duty.

```
call foo: saved the return address
  push rbp        ; save old rbp -> rsp = rsp - 8, *[rsp] = rbp | 8 byte
  mov rbp, rsp    ; set new rbp, 让 rbp 指向当前栈帧的基准位置
  sub rsp, 24     ; 给本函数的局部变量留空间
```

This code is the function prologue and epilogue that manages the stack frame.
push rbp saves the caller's frame pointer,
mov rbp, rsp creates a new frame pointer for this function, and
sub rsp, 24 allocates 24 bytes for local variables and temporaries.
At this point the stack (low → high) is: return address, saved old rbp, then local variables/temporary storage.
In the epilogue, **mov rsp, rbp** discards the locals, **pop rbp** restores the caller's frame pointer, and **ret** pops the return address and jumps back to the caller.

```
↑ 低地址
0x0FE0:  local/temporary slot (local3)  ← rsp
0x0FE8:  local/temporary slot (local2)
0x0FF0:  local/temporary slot (local1)
0x0FF8:  Old rbp                        ← rbp
0x1000:  Return Address
↓ 高地址
```

Sample – Stack Frame. Assembly code

```
void foo(int a, int b) {
    int x = 10;
    int y = a + b + x;
}

int main() {
    foo(5, 15);
    return 0;
}
```

64-bit，每个 slot 8 bytes

```
高地址-----------------------------
    [rbp + 8]   Return Address
    [rbp]       Old rbp
    [rbp - 8]   a  (slot0)
    [rbp -16]   b  (slot1)
    [rbp -24]   x  (slot2)
    [rbp -32]   y  (slot3)
低地址 Lower addresses----------
```

```
foo:
    push rbp                ; prologue
    mov  rbp, rsp
    sub  rsp, 32            ; 4 个 8-byte slot
    ; 把参数寄存器存到栈上的 slot 里
    mov  QWORD [rbp-8],  rdi     ; a
    mov  QWORD [rbp-16], rsi     ; b

    ; int x = 10;
    mov  QWORD [rbp-24], 10      ; x = 10

    ; int y = a + b + x;
    mov  rax, [rbp-8]            ; rax = a
    add  rax, [rbp-16]           ; rax = a + b
    add  rax, [rbp-24]           ; rax = a + b + x
    mov  [rbp-32], rax           ; y = rax

    ; epilogue
    mov  rsp, rbp
    pop  rbp
    ret
```

Sample2 – Stack Operations

```
push 10

push 20

pop rax

push 30
```

Initial rsp = 0x1000
```
Address   Value
0x1000    ??   ← rsp
```
After push 10 -=8
rsp=rsp-8
[rsp]=10
```
0x0FF8 10 ← rsp (0x1000 - 8) rsp = 0x1000
0x1000    ??
```
After push 20 -=8
```
Address   Value
0x0FF0    20     ← rsp (0x0FF8 - 8)
0x0FF8    10
0x1000    ??
```
After pop rax: +=8
rax = 20 (value at rsp)
rsp = 0x0FF8 (rsp + 8)
```
Address   Value
0x0FF8    10     ← rsp
```
After push 30:
```
Address   Value
0x0FF0    30     ← rsp (0x0FF8 - 8)
0x0FF8    10
```
Stack contains: [30 (top), 10]
rsp = 0x0FF0; rax = 20

Since local variables are in memory, we should use offsets from rbp to read and write them.
mov rax, [rbp - 8] ; read from local variable 1
mov [rbp - 16], rax ; write to local variable 2

```
long local_var1;  // 放在 [rbp - 8]
long local_var2;  // 放在 [rbp - 16]
```

Hash table

In a compiler, the **symbol table** is usually implemented as a hash table that maps identifier names to their type and storage information.

unsorted array	Lookup:O(n)insertion: O(1)
sorted array + binary search	Lookup:O(logn), insertion:O(n)
balanced search tree	Lookup:O(logn), insertion:O(logn)

Lecture8:

Handling Parameters/Functions as Symbols

Q: Why does the compiler store a function's full signature (parameter types, number of parameters, return type, etc.) in the symbol table instead of just storing its name?

A: The compiler need to perform static type checking: it compares the types of the arguments in a call against the parameter types stored in the symbol table.

Levenshtein Distance

The minimum number of steps needed to transform string A into string B by inserting, deleting, or replacing characters.

```
if s[i-1] == t[j-1]:
    dp[i][j] = dp[i-1][j-1]
else:
    dp[i][j] = 1 + min(
        dp[i-1][j],    # delete
        dp[i][j-1],    # insert
        dp[i-1][j-1]   # replace
    )
```

Dynamic Programming

Break the problem into overlapping subproblems. Each subproblem has optimal substructure (optimal solution made from optimal sub-solutions). Store and reuse sub-results in a table (memo) to avoid recomputation.

Top-down approach, we define a recursive function for the subproblems and use a memo (cache) to remember results, so each state is computed at most once.

- Allocate a table of results
- Pass it and the problem parameters to a recursive function
- The recursive function checks if result is stored in table
  a) If yes, return the cached result
  b) If not, compute recursively, passing the same table down

Bottom-up approach, we start from the base cases and iteratively fill in a DP table from smaller subproblems to larger ones until we reach the final state.

- Allocate a table of results
- Fill in trivial entries
- Progressively fill in entries based on the known entries
- Fill out enough of the table to compute the desired result, return it

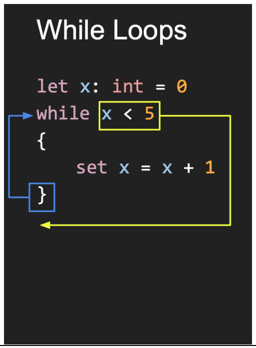
```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int coinChange(int *coins, int coinsSize, int amount) {
    if (amount == 0) return 0;
    if (coinsSize == 0) return -1;
    // dp[i] = minimum coins to make amount i
    int *dp = (int *)malloc((amount + 1) * sizeof(int));
    if (!dp) return -1;   // malloc failed
    int INF = amount + 1;
    for (int i = 0; i <= amount; i++) {
        dp[i] = INF;
    }
    dp[0] = 0;
    // bottom-up DP
    for (int i = 1; i <= amount; i++) {
        for (int j = 0; j < coinsSize; j++) {
            int coin = coins[j];
            if (i - coin >= 0 && dp[i - coin] != INF) {
                if (dp[i] > dp[i - coin] + 1) {
                    dp[i] = dp[i - coin] + 1;
                }
            }
        }
    }

    int result = (dp[amount] == INF) ? -1 : dp[amount];
    free(dp);
    return result;
}
```

Lecture8:

`if x < 3`
`{`
`  set y = 0`
`}`
`else`
`{`
`  set y = x`
`}`
	Intermediate Representation
	`LOAD x`
	`PUSH 3`
	`LESS_THAN           ; 现在栈顶是布尔值 (x < 3)`
	`JUMP_IF_FALSE else ; 如果条件是假，就跳到 else 那个 label`
	
	`PUSH 0`
	`STORE y             ; then 分支`
	`JUMP endif          ; 执行完 then，跳过 else`
	
	`LABEL else:`
	`LOAD x`
	`STORE y             ; else 分支`
	
	`LABEL endif:`




```
LABEL while        ; while:

  LOAD x           ; cond: x < 5 ?
  PUSH 5
  LESS_THAN
  JUMP_IF_FALSE endwhile

  LOAD x           ; body: x = x + 1
  PUSH 1
  ADD
  STORE x

  JUMP while       ; 回到 while 条件

LABEL endwhile     ; endwhile:
```

Handling Boolean Expressions

- Conditional jumps instructions
  - Use flags (ZF, SF, OF, …) + jXX to **immediately decide** where to jump.
- Conditional set instructions
  - Turn a condition into a **value** (0 or 1) in a register.

IR / stack machine:
Every boolean operator (<, ==, &&, ||, !) is a value-producing instruction.
Result is always 0 (false) or 1 (true), pushed onto the stack.
if / while:
• first compile_expr(cond) → stack top = 0/1
• then JUMP_IF_FALSE / JUMP_IF_TRUE based on stack top

Graphs

Degree	The number of neighbors of a vertex
Path	A sequence of distinct vertices where each is a neighbor of the previous
Cycle	A path ending in a neighbor of the first vertex
Arc	A directed edge
DAG	Directed acyclic graph

Graph Representations

Adjacency Matrix	**Neighbor List**
• Square matrix with one row and one column for each vertex • Entry in row i col j is 1 if vertex i is adjacent to vertex j and 0 otherwise • In a weighted graph, the entry is the weight of the edge	• Each vertex stores a list of its neighbors
Useful for dense graphs (lots of edges)	More efficient for sparse graphs
$O(n^2)$ node	$O(n + m)$ n+e

Graph Traversal

Depth First Search (DFS)	Breadth First Search (BFS)
Recursively visit a neighbor	Visit immediate neighbors before neighbors of neighbors
Depth-First Search explores a graph by always going **as deep as possible** along one path before backtracking and trying other paths.	Breadth-First Search explores a graph **level by level**, visiting all neighbors of the current frontier before moving on to the next level.

Lecture10:

C-style string	array of bytes + \0 strlen is O(n) `Hello, World!\n`
Pascal	A count followed by the array of bytes `13` | `Hello, World!`

String View	A count followed by a pointer to an array of bytes
	`13` | `0x1234`
	`0x1234` | `13 Hello, World!`

Outputting Strings printf

Use assembly code write own printing routines

```
section .data
data:
    dq 13              // store the 64-bit integer 13 at data
    db "Hello, World!"

section .text
global _start
_start:
    mov rax, 1            ; write
    mov rdi, 1            ; stdout

    ; rsi = 指向字符串的地址 = data 后面 8 个字节
    lea rsi, [rel data + 8] ; "H…"的首地址算出来，放进 rsi

    mov rdx, [rel data]     ; rdx = [data] 里的长度 = 13

    syscall

    ; 然后 exit(0)
    mov rax, 60
    xor rdi, rdi.           ; rdi = 0
    syscall
```

[...] means: "use this value as a memory address and access the memory at that address."

syscall:
write rax1 = write (rdi, rsi, rdx)->(1, buf, len)

```
mov rax, 1      ; 1 = write
mov rdi, 1      ; 第1个参数 fd = 1 -> stdout
mov rsi, buf    ; 第2个参数 buf = 缓冲区地址
mov rdx, len    ; 第3个参数 count = 长度
syscall         ; 相当于 write(1, buf, len);
```

Dynamic Memory

C standard library: malloc, free
Syscalls: mmap, munmap

```
#include <stdio.h>
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int a = 0;   // F(0)
    int b = 1;   // F(1)
    for (int i = 2; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;}
    return b;  } // F(n)
int main(void) {
    printf("%d\n", fib(10));
    return 0; }
```

Jive
```
let fibs: [int] = alloc(48)
set fibs[0] = 0
set fibs[1] = 1
let n: int = 2
while n < 48 {
  set fibs[n] = fibs[n - 1] +
fibs[n - 2]
  set n = n + 1
}
```

```
set fibs[n] = fibs[n - 1] + fibs[n -
2]

PEEK fibs // Load an element from
fibs onto the top of the stack.
```

stack machine IR
```
LOAD n
PUSH 1
SUB
PEEK fibs
LOAD n
PUSH 2
SUB
PEEK fibs
ADD
LOAD n
POKE fibs
```

Dijkstra's Algorithm

The goal of Dijkstra's algorithm is to compute the single-source shortest paths in a weighted graph with non-negative edge weights.

Initially:
- Distance to initial vertex is 0, all others ∞
- Previous vertices all none
- No shortest paths discovered

In a loop:
- Find unfinished vertex u with the shortest distance
- If distance is ∞, terminate: all remaining vertices are unreachable
- For each adjacent vertex v, if the path through u is shorter than recorded:
  - o Update distance to v and set previous vertex for v to u
- Mark u as finished: we know the shortest path to it

For each vertex, we have the minimum distance, and by tracing the previous vertices backwards, we have the shortest path to it

Sample
S → A (1)
S → B (4)
A → B (2)
Initialization:
  dist[S] = 0, all others = ∞
First, choose u = S:
  Relax its neighbors:
  dist[A] = 1, prev[A] = S
  dist[B] = 4, prev[B] = S >>> Set finished[S] = true.
Second, choose u = A, because among all unfinished vertices,
  A has dist = 1, which is smaller than B's 4.
  Use A to relax B:
  Path through A: 1 + 2 = 3 < 4
  → update dist[B]=3, prev[B]=A >>> Set finished[A] = true.
Third, choose u = B:
  dist[B] = 3, which is the correct answer.
  Relax its neighbors (none) >>> then set finished[B] = true.

When a node u is popped from the min-heap, dist[u] is guaranteed to be the true shortest distance from S to u.
Time Complexity: O((V + E) log V) Space Complexity: O(V + E)

```
typedef struct { s// 堆里的节点
    int node;
    int dist;
} HeapNode;

void swap(HeapNode *a, HeapNode *b) { // 交换堆元素
    HeapNode tmp = *a;
    *a = *b;
    *b = tmp;}

void heapify_up(HeapNode *heap, int idx) {// 向上调整 (bubble up)
    while (idx > 1) {                  // 根在 1
        int parent = idx / 2;
        if (heap[parent].dist <= heap[idx].dist) break;
        swap(&heap[parent], &heap[idx]);
        idx = parent; }}

void heapify_down(HeapNode *heap, int size, int idx) { // 向下调整
    while (1) {
        int left = idx * 2;
        int right = idx * 2 + 1;
        int smallest = idx;

        if (left <= size && heap[left].dist < heap[smallest].dist)
            smallest = left;
        if (right <= size && heap[right].dist < heap[smallest].dist)
            smallest = right;

        if (smallest == idx) break;

        swap(&heap[idx], &heap[smallest]);
        idx = smallest; } }

// 入堆
void heap_push(HeapNode *heap, int *size, int node, int dist) {
    (*size)++;
    heap[*size].node = node;
    heap[*size].dist = dist;
    heapify_up(heap, *size);
```

```c
}

// 出堆（返回最小的节点）
HeapNode heap_pop(HeapNode *heap, int *size) {
    HeapNode top = heap[1];        // 根是最小值
    heap[1] = heap[*size];
    (*size)--;
    if (*size > 0) {
        heapify_down(heap, *size, 1);
    }
    return top; }
```

```c
int networkDelayTime(int** times, int timesSize, int* timesColSize,
int n, int k) { //build adjacency list
    int m = timesSize;

    int *head = (int*)malloc((n + 1) * sizeof(int));
    for (int i = 1; i <= n; i++) head[i] = -1;

    int *to = (int*)malloc(m * sizeof(int));
    int *weight = (int*)malloc(m * sizeof(int));
    int *next = (int*)malloc(m * sizeof(int));

    for (int i = 0; i < m; i++) {
        int u = times[i][0];
        int v = times[i][1];
        int w = times[i][2];
        to[i] = v;
        weight[i] = w;
        next[i] = head[u];
        head[u] = i;   // 新边挂在链表头
    }
```

```c
    // ---------- Dijkstra initial ----------
    int *dist = (int*)malloc((n + 1) * sizeof(int));
    int *visited = (int*)malloc((n + 1) * sizeof(int));
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }
    dist[k] = 0;
// ---------- 最小堆 ----------
    // 最坏情况下可能 push 非常多，我们简单给一个较大的容量
    int heapCap = m * 3 + 5;
    if (heapCap < n * 3) heapCap = n * 3;
    HeapNode *heap = (HeapNode*)malloc(sizeof(HeapNode) * (heapCap));

    int heapSize = 0;
    // 把起点放进去
    heap_push(heap, &heapSize, k, 0);

    // ---------- Dijkstra 主循环 ----------
    while (heapSize > 0) {
        HeapNode cur = heap_pop(heap, &heapSize);
        int u = cur.node;
        int d = cur.dist;

        if (visited[u]) continue;   // 如果已经确定了最短路，跳过
        visited[u] = 1;

        // 小剪枝：如果堆里这个 dist 已经比数组里的大，跳过
        if (d > dist[u]) continue;

        // 松弛所有邻居
        for (int e = head[u]; e != -1; e = next[e]) {
            int v = to[e];
            int w = weight[e];
            if (!visited[v] && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                // 把更新后的 (v, dist[v]) 丢进堆
                if (heapSize + 1 < heapCap) {
                    heap_push(heap, &heapSize, v, dist[v]);
                }}}}
    // ---------- 计算答案 ----------
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        if (dist[i] == INF) {
            ans = -1;
            break;
        }
        if (dist[i] > ans) ans = dist[i];}
    // ---------- 释放内存 ----------
    free(head);
    free(to);
    free(weight);
    free(next);
    free(dist);
    free(visited);
```

```c
    free(heap);

    return ans;}
```

Heap

Heap: complete binary tree, a complete binary tree (shape property), and satisfies the heap order property (parent ≥ children for max-heap, ≤ for min-heap).

Peak: O(1)

Heap pop: O(logn)

1,把最后一个元素搬到根 2,heapify, if max heap, swap biger one

Heap push: O(logn)

1, 新元素永远先挂在"树的最底层最右边" 2, sift up

Step of Compiler:

1. read source code	
2. Lexer	Breaks source code into tokens
Purpose: Simplify parsing by grouping characters into meaningful units.	
Input: "fn main() -> int { return 42 }" Output: Tokens [fn, main, (,), ->, int, {, return, 42, }]	
fn add(a: int, b: int) -> int { return a + b } 1. fn (KEYWORD) 2. add (IDENTIFIER) 3. ((OPEN_PAREN) 4. a (IDENTIFIER) 5. : (COLON) 6. int (TYPE) 7. , (COMMA) 8. b (IDENTIFIER) 9. : (COLON) 10. int (TYPE) 11.) (CLOSE_PAREN) 12. -> (ARROW) 13. int (TYPE) 14. { (OPEN_BRACE) 15. return (KEYWORD) 16. a (IDENTIFIER) 17. + (PLUS) 18. b (IDENTIFIER) 19. } (CLOSE_BRACE) 20. EOF	
3. parsing	Builds Abstract Syntax Tree (AST) from tokens
Understand the grammatical structure of the program.	
Input: Token Output: Output: PROGRAM └── FN("main") └── RETURN └── INTEGER(42)	
4. stack machine IR	intermediate representation Converts AST to stack-based instructions
Purpose: Bridge between high-level AST and low-level assembly.	
Output: FN main | PUSH 42 | RETURN | END_FN	
5. code generation	Translates stack machine to x86-64 assembly
Translate stack machine to actual CPU instructions.	
Output: main: push 42 | pop rax | ret	
6. ASSEMBLING & LINKING (nasm + ld)	Creates executable binary
Output: ./main (executable file) echo $? # Prints: 42	

Build.sh

gcc	gcc **-c** hello_world.c -o hw.o will output hw.**o** (object file) Machine code(mov, add, syscall) data
gcc	.o → jive gcc lexer.o parser.o codegen.o sm.o s_.o main.o -o jive
foo.jive--(jive)--> foo.asm--(nasm)--> foo.o--(gcc/ld)--> foo (executable)	
jive	.jive → .asm ./jive "$PROGRAM_DIR/$source_file.jive" --run-sm -o "$asm_name.asm"
nasm	nasm -felf64 hello.**asm** -o hello.**o**
gcc	.o → an executable program

swap:

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;}

int main() {
    int x = 5, y = 10;
    swap(&x, &y);  // Pass addresses
    printf("%d %d\n", x, y);  // Prints: 10 5
    return 0;}
```

Point

```c
typedef struct {
    int x;
    int y;
} Point;

void move(Point p, int dx, int dy) { // void move(Point *p,
    p.x += dx;  // p->x += dx;
    p.y += dy;  // p->y += dy;
}

int main() {
    Point p = {10, 20};
    move(p, 5, 5); // move(&p, 5, 5);
    printf("(%d, %d)\n", p.x, p.y);  // What prints?
    return 0;}
```

strcpy = "string copy（拷贝字符串）"

Wrong	correct

char *create_greeting() {
    char msg[] = "Hello";
    return msg;}
 	
int main() {
	char *create_greeting() {
	    char *msg = **malloc**(6);
	    strcpy(msg, "Hello");
	    return msg;}

char *greeting =
create_greeting();
    printf("%s\n", greeting);
    return 0;}
	int main() {
	    char *greeting = create_greeting();
	    printf("%s\n", greeting);
	    free(greeting);
	    return 0; }

Traversal

```c
typedef struct Node {
    int value;
    struct Node *left;
    struct Node *right;
} Node;

void postorder(Node *root) {
    if (root == NULL) return;
    postorder(root->left);    // 左
    postorder(root->right);  // 右
    printf("%d ", root->value); // 根
}
```

Code -> IR(def variable)

fn main() -> int {
    let x: int = 5
    set x = x * 2
    return x
}
	LABEL main ; function main entry
	
	; let x: int = 5
	PUSH 5
	STORE x ; x = 5
	
	; set x = x * 2
	LOAD x ; stack: x
	PUSH 2 ; stack: x, 2
	MUL ; stack: x * 2
	STORE x ; x = x * 2
	
	; return x
	LOAD x ; stack: x
	RET ; return top of stack

Stack Machine

fn example() -> int {
    let a: int = 1 // slot 0
    let b: int = 2 // slot 1
    let c: int = 3 // slot 2
    let d: int = 4 // slot 3
    return 0
}
	Formula: [rbp - (slot + 1) * 8]
	Variable a (slot 0): [rbp - 8]
	Variable b (slot 1): [rbp - 16]
	Variable c (slot 2): [rbp - 24]
	Variable d (slot 3): [rbp - 32]
	Stack frame size: 32 bytes (4 variables × 8 bytes each)
	Preamble: sub rsp, 32

Endianness (big vs little)

Little-endian: Lowest-address byte stores the least significant byte (LSB).

Big-endian: Lowest-address byte stores the most significant byte (MSB).

```c
char *create_greeting() {
    char msg[] = "Hello"; // on stack
    return msg;           // returns pointer to dead stack
}
```

Wrong: msg is on the stack and becomes invalid after return → dangling pointer → undefined behavior.

How to correctly duplicate a string using malloc?

```c
char *duplicate_string(const char *s) {
    size_t len = strlen(s);
    char *copy = malloc(len + 1);
    if (!copy) return NULL;
    strcpy(copy, s);   // or memcpy(copy, s, len+1);
    return copy;}
```

Q: What is the difference between . and -> when accessing struct fields?（obj.x 和 ptr->x 区别？）

A: obj.x: obj is a struct value.

ptr->x is syntactic sugar for (*ptr).x where ptr is a pointer to a struct.

```c
// lexer. Token types
typedef enum Token_Kind {
    TOKEN_KEYWORD, TOKEN_IDENT, TOKEN_TYPE, TOKEN_INTEGER, TOKEN_EOF,
    TOKEN_PLUS, TOKEN_MINUS, TOKEN_STAR, TOKEN_SLASH, TOKEN_PERCENT,
    TOKEN_AMPERSAND, TOKEN_PIPE, TOKEN_CARET, TOKEN_TILDE,
    TOKEN_BANG, TOKEN_EQ, TOKEN_EQ_EQ, TOKEN_BANG_EQ,
    TOKEN_LESS, TOKEN_GREATER, TOKEN_LESS_EQ, TOKEN_GREATER_EQ,
    TOKEN_AND_AND, TOKEN_PIPE_PIPE,
    TOKEN_OPEN_PAREN, TOKEN_CLOSE_PAREN,
    TOKEN_OPEN_BRACE, TOKEN_CLOSE_BRACE,
    TOKEN_OPEN_BRACKET, TOKEN_CLOSE_BRACKET,
    TOKEN_ARROW, TOKEN_COMMA, TOKEN_COLON
} Token_Kind;

typedef enum Keyword {
    KEYWORD_FN, KEYWORD_LET, KEYWORD_SET, KEYWORD_IF, KEYWORD_WHILE,
    KEYWORD_CALL, KEYWORD_RETURN, KEYWORD_TRUE, KEYWORD_FALSE
} Keyword;
```

```c
// Parser.c
typedef enum Binary_Op
{ BINOP_ADD, BINOP_SUB, BINOP_MUL, BINOP_DIV, BINOP_MOD
} Binary_Op;

typedef enum AST_Kind
{ AST_NONE, AST_PROGRAM, AST_FN, AST_TYPE, AST_RETURN, AST_INTEGER,
AST_BINARY_OP, AST_LET, AST_SET, AST_IDENTIFIER,
AST_PARAMETER,  // Parameter in function definition
AST_CALL,       // Function call
} AST_Kind;

const char *ast_kind_as_cstr(AST_Kind kind)
{ switch (kind) {
case AST_NONE:    return "NONE (ERROR!)";
case AST_PROGRAM: return "PROGRAM";
case AST_FN:      return "FN";
case AST_TYPE:    return "TYPE";
case AST_RETURN:  return "RETURN";
case AST_INTEGER: return "INTEGER";
case AST_BINARY_OP: return "BINARY_OP";
case AST_LET:     return "LET";
case AST_SET:     return "SET";
case AST_IDENTIFIER: return "IDENTIFIER";
case AST_PARAMETER: return "PARAMETER";
case AST_CALL:    return "CALL";
default:          return "UNKNOWN (ERROR!)";          } }

typedef struct AST_Node AST_Node;

typedef struct AST_List // Doubly linked list of AST_Nodes
{
          long count;
          AST_Node *first;
          AST_Node *last;
} AST_List;
```