# CS100 Lecture 9

`struct`, Recursion

## Contents

`struct`

# Define a `struct`

A `struct` is a **type** consisting of a sequence of **members** whose storage is allocated in an ordered sequence.

Simply put, place several things together to form a new type.

```c
struct Student {
  const char *name;
  const char *id;
  int entrance_year;
  int dorm;
};
```

```c
struct Point3d {
  double x, y, z;
};
struct Line3d {
  // P(t) = p0 + tv
  struct Point3d p0, v;
};
```

## `struct` type

The name of the type defined by a `struct` is `struct Name`.

- Unlike C++, the keyword `struct` here is necessary.

```c
struct Student stu; // `stu` is an object of type `struct Student`
struct Point3d polygon[1000]; // `polygon` is an array of 1000 objects,
                              // each being of type `struct Point3d`.
struct TreeNode *pNode; // `pNode` is a pointer to `struct TreeNode`.
```

*** The term "*object*" is used interchangeably with "*variable*".**

- *Objects* often refer to variables of `struct` (or `class` in C++) types.
- But in fact, there's nothing wrong to say "an `int` object".

# Members of a `struct`

Use `obj.mem`, the **member-access operator** `.` to access a member.

```c
struct Student stu;
stu.name = "Alice";
stu.id = "2024533000";
stu.entrance_year = 2024;
stu.dorm = 8;
printf("%d\n", student.dorm);
++student.entrance_year;
puts(student.name);
```

# Dynamic allocation

Create an object of `struct` type dynamically: Just allocate `sizeof(struct Student)` bytes of memory.

```
struct Student *pStu = malloc(sizeof(struct Student));
```

Member access through a pointer: `ptr->mem`, or `(*ptr).mem` **(not `*ptr.mem`!)**.

```
pStu->name = "Alice";
pStu->id = "2024533000";
(*pStu).entrance_year = 2024; // equivalent to pStu->entrance_year = 2024;
printf("%d\n", pStu->entrance_year);
puts(pStu->name);
```

As usual, don't forget to `free` after use.

```
free(pStu);
```

# Size of a `struct`

```c
struct Student {
  const char *name;
  const char *id;
  int entrance_year;
  int dorm;
};
```

```c
struct Student *pStu = malloc(sizeof(struct Student));
```

What is the value of `sizeof(struct Student)`?

# Size of a `struct`

Try these:

```c
struct A {
  int x;
  char y;
  double z;
};
```

```c
struct B {
  char x;
  double y;
  int z;
};
```

```c
printf("%zu\n", sizeof(struct A));
```

```c
printf("%zu\n", sizeof(struct B));
```

Possible result: `sizeof(struct A)` is `16`, but `sizeof(struct B)` is `24` (on Ubuntu 22.04, GCC 13).

# Size of `struct`

```c
struct A {
  int x;    // 4 bytes
  char y;   // 1 byte
  // 3 bytes padding
  double z; // 8 bytes
};
```

- `sizeof(struct A) == 16`

```c
struct B {
  char x;   // 1 byte
  // 7 bytes padding
  double y; // 8 bytes
  int z;    // 4 bytes
  // 4 bytes padding
};
```

- `sizeof(struct B) == 24`

It is guaranteed that

$$\texttt{sizeof(struct X)} \geqslant \sum_{\texttt{member}\in\texttt{X}} \texttt{sizeof(member)}.$$

The inequality is due to **memory alignment requirements**, which is beyond the scope of CS100.

## Implicit initialization

What happens if an object of `struct` type is not explicitly initialized?

```c
struct Student gStu;

int main(void) {
  struct Student stu;
}
```

## Implicit initialization

What happens if an object of `struct` type is not explicitly initialized?

```
struct Student gStu;

int main(void) {
  struct Student stu;
}
```

- Global or local `static`: "empty-initialization", which performs **member-wise** empty-initialization.
- Local non-`static`: every member is initialized to indeterminate values (in other words, uninitialized).

# Explicit initialization

Use an initializer list:

```c
struct Student stu = {"Alice", "2024533000", 2024, 8};
```

**Use C99 designators:** (highly recommended)

```c
struct Student stu = {.name = "Alice", .id = "2024533000",
                      .entrance_year = 2024, .dorm = 8};
```

The designators greatly improve the readability.

**[Best practice]** Use designators, especially for `struct` types with lots of members.

# Compound literals

```c
struct Student *student_list = malloc(sizeof(struct Student) * n);
for (int i = 0; i != n; ++i) {
  student_list[i].name = A(i); // A, B, C and D are some functions
  student_list[i].id = B(i);
  student_list[i].entrance_year = C(i);
  student_list[i].dorm = D(i);
}
```

Use a **compound literal** to make it clear and simple:

```c
struct Student *student_list = malloc(sizeof(struct Student) * n);
for (int i = 0; i != n; ++i) {
  student_list[i] = (struct Student){.name = A(i), .id = B(i),
                                     .entrance_year = C(i), .dorm = D(i)};
}
```

# `struct`-typed parameters

The semantic of argument passing is **copy**:

```c
void print_student(struct Student s) {
  printf("Name: %s, ID: %s, dorm: %d\n", s.name, s.id, s.dorm);
}

print_student(student_list[i]);
```

In a call `print_student(student_list[i])`, the parameter `s` of `print_student` is initialized as follows:

```c
struct Student s = student_list[i];
```

The copy of a `struct`-typed object: **Member-wise copy.**

## `struct`-typed parameters

In a call `print_student(student_list[i])`, the parameter `s` of `print_student` is initialized as follows:

```c
struct Student s = student_list[i];
```

The copy of a `struct`-typed object: **Member-wise copy.** It is performed as if

```c
s.name = student_list[i].name;
s.id = student_list[i].id;
s.entrance_year = student_list[i].entrance_year;
s.dorm = student_list[i].dorm;
```

# Return a `struct`-typed object

Strictly speaking, returning is also a **copy**:

```c
struct Student fun(void) {
  struct Student s = something();
  some_operations(s);
  return s;
}
student_list[i] = fun();
```

The object `s` is returned as if

```c
struct Student tmp = s; // 1st copy
student_list[i] = tmp;  // 2nd copy
```

**But in fact, the compiler is more than willing to optimize this process.** We will talk more about this in C++.

## Array member

```c
struct A {
  int array[10];
  // ...
};
```

Although an array cannot be copied, **an array member can be copied**.

The copy of an array is **element-wise copy**.

```c
int a[10];
int b[10] = a; // Error!
```

```c
struct A a;
struct A b = a; // OK
```

# Summary

A `struct` is a type consisting of a sequence of members.

- Member access: `obj.mem`, `ptr->mem` (equivalent to `(*ptr).mem`, but better)
- `sizeof(struct A)`, no less than the sum of size of every member.
  - But not necessarily equal, due to memory alignment requirements.
- Implicit initialization: recursively performed on every member.
- Initializer-lists, designators, compound literals.
- Copy of a `struct`: member-wise copy.
- Argument passing and returning: copy.

## Exercise

Consider a 3-d coordinate point $(x, y, z)$ and a line $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{v}$. Define some `struct`s to represent these concepts.

Write some functions to calculate the distance between points and lines, to calculate the point $\mathbf{P}\left(t_0\right)$ for a given $t_0$, and to print some information.

Learn and try to use **initializer-lists**, **designators** and **compound literals**.

```c
double dist(struct Point3d p, struct Line3d line);
struct Point3d line_at(struct Line3d line, double t);
```

# Recursion

## Problem 1. Calculate $n!$

A piece of cake!

```
int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; ++i)
    result *= i;
  return result;
}
```

**You should be able to write this in 30 seconds.**

# Calculate $n!$

Consider the **recurrence relation**:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n > 0. \end{cases}$$

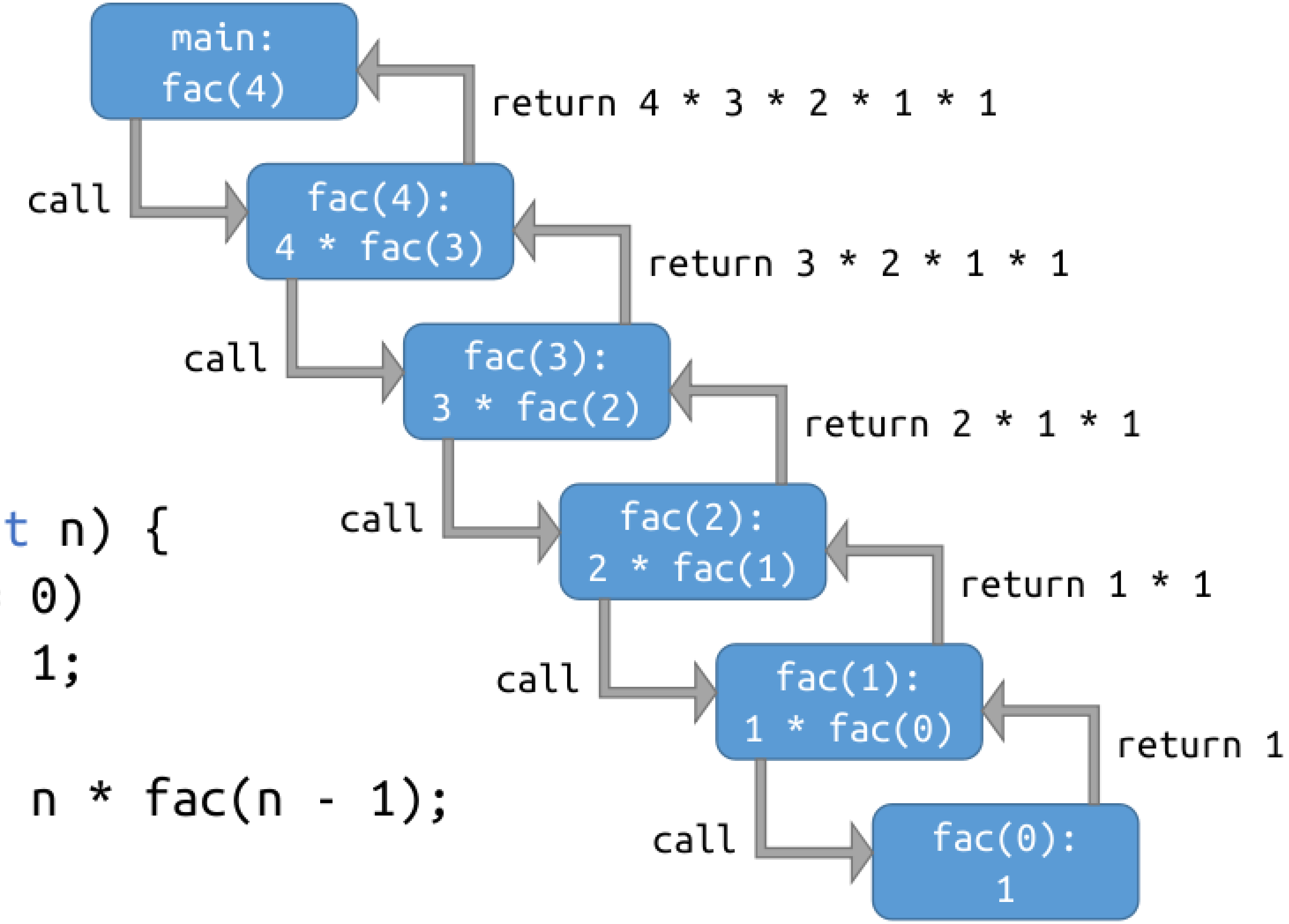
Translate it directly into C:

```c
int factorial(int n) {
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

**This is perfectly valid and reasonable C code!**

- The function `factorial` **recursively** calls itself.

# Calculate $n!$




```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n - 1);
}
```

## Problem 2. Print a non-negative integer

If we only have `getchar`, how can we read an integer?

- We have solved this in recitations.

If we only have `putchar`, how can we print an integer?

- Declared in `<stdio.h>`.
- `putchar(c)` prints a character `c`. That's it.

For convenience, suppose the integer is non-negative (unsigned).

## Print a non-negative integer

To print $x$:

- If $x < 10$, just print the digit and we are done.
- Otherwise ($x \geqslant 10$), we first print $\left\lfloor \dfrac{x}{10} \right\rfloor$, and then print the digit on the last place.

```c
void print(unsigned x) {
  if (x < 10)
    putchar(x + '0'); // Remember ASCII?
  else {
    print(x / 10);
    putchar(x % 10 + '0');
  }
}
```

## Simplify the code

To print $x$:

1. If $x \geqslant 10$, we first print $\left\lfloor \dfrac{x}{10} \right\rfloor$. Otherwise, do nothing.
2. Print $x \bmod 10$.

```
void print(unsigned x) {
  if (x >= 10)
    print(x / 10);
  putchar(x % 10 + '0');
}
```

## Print a non-negative integer

To print $x$:

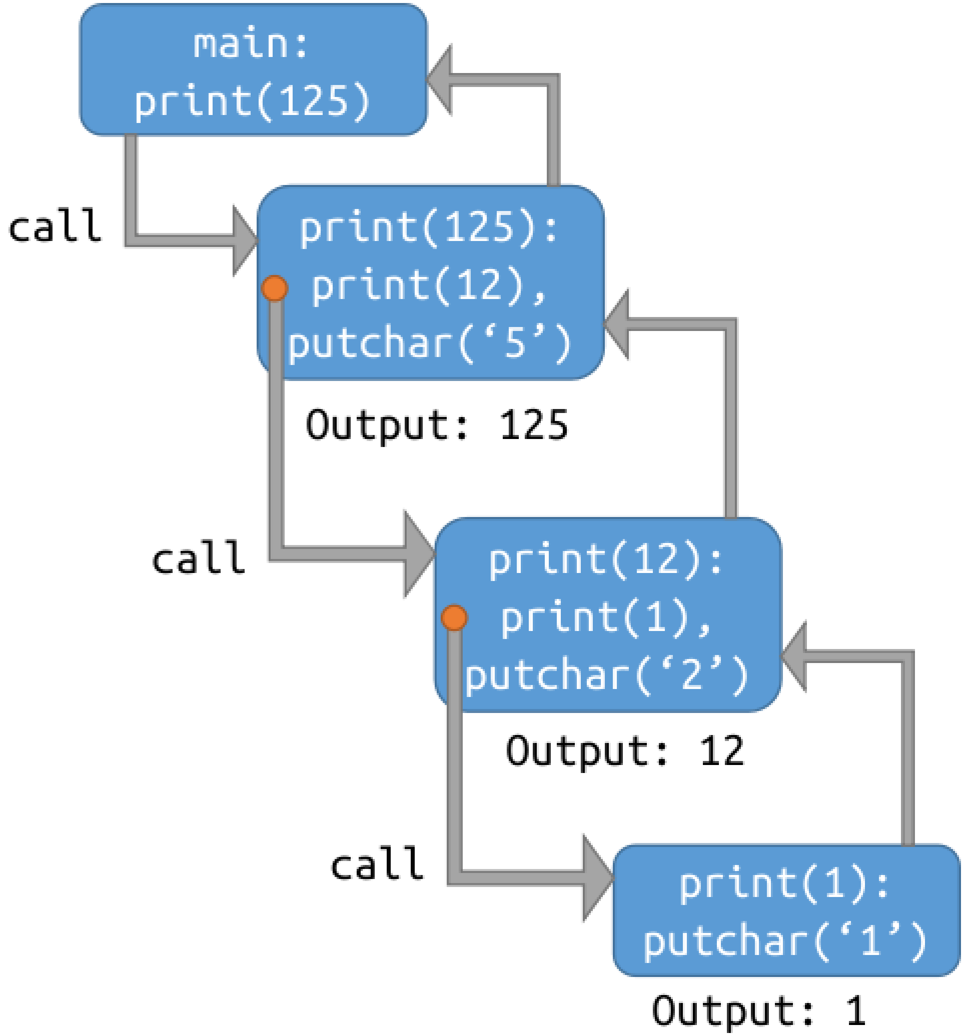
1. If $x \geqslant 10$, we first print $\left\lfloor \dfrac{x}{10} \right\rfloor$.

   Otherwise, do nothing.

2. Print $x \bmod 10$.

```
void print(unsigned x) {
  if (x >= 10)
    print(x / 10);
  putchar(x % 10 + '0');
}
```

**Exercise**: Allow for negative numbers.



## Design a recursive algorithm

Suppose we are given a problem of scale $n$.

1. Divide the problem into one or more **subproblems**, which are of smaller scales.
2. Solve the subproblems **recursively** by calling the function itself.
3. Generate the answer to the big problem from the answers to the subproblems.

*** Feels like mathematical induction?**

## Exercise: Quick power

Calculate $x^n$ following the recurrence relation:

$$x^n = \begin{cases} 1, & n = 0, \\ x \cdot x^{n-1}, & n \text{ is odd}, \\ \left(x^{n/2}\right)^2, & n \text{ is even}. \end{cases}$$

# Exercise: Quick power

Calculate $x^n$ following the recurrence relation:

$$x^n = \begin{cases} 1, & n = 0, \\ x \cdot x^{n-1}, & n \text{ is odd}, \\ \left(x^{n/2}\right)^2, & n \text{ is even}. \end{cases}$$

```
unsigned long long quick_power(unsigned long long x, int n) {
  if (n == 0)
    return 1;
  if (n % 2 == 1)
    return x * quick_power(x, n - 1);
  else {
    unsigned long long t = quick_power(x, n / 2);
    return t * t;
  }
}
```

## Problem 3. Selection-sort

How do you sort a sequence of $n$ numbers? (In ascending order)

Do it **recursively**.

## Problem 3. Selection-sort

How do you sort a sequence of $n$ numbers $\langle a_0, \cdots, a_{n-1}\rangle$? (In ascending order)

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1}\rangle$, for some $k$.

- If $k = n - 1$, we are done.
- Otherwise ($k < n - 1$):
    i. Find the minimal number $a_m = \min\{a_k, a_{k+1}, \cdots, a_{n-1}\}$.
    ii. Put $a_m$ at the first place by swapping it with $a_k$.
    iii. Now $a_k$ is the smallest number in $\langle a_k, \cdots, a_{n-1}\rangle$. All we have to do is to sort the rest part $\langle a_{k+1}, \cdots, a_{n-1}\rangle$ **recursively**.

# Selection-sort

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1}\rangle$, for some $k$.

```
void sort_impl(int *a, int k, int n) { // "impl" stands for "implementation"

}
void sort(int *a, int n) {
  sort_impl(a, 0, n); // Why do we set k = 0 here?
}
```

# Selection-sort

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1} \rangle$, for some $k$.

- If $k = n - 1$, we are done.

```
void sort_impl(int *a, int k, int n) {
  if (k == n - 1)
    return;




}
```

## Selection-sort

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1} \rangle$, for some $k$.

- If $k < n - 1$:
   i. Find the minimal number $a_m = \min\left\{a_k, a_{k+1}, \cdots, a_{n-1}\right\}$.

```
void sort_impl(int *a, int k, int n) {
  if (k == n - 1)
    return;

  int m = k;
  for (int i = k + 1; i < n; ++i)
    if (a[i] < a[m])
      m = i;

}
```

## Selection-sort

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1} \rangle$, for some $k$.

- If $k < n - 1$:
  i. Find the minimal number $a_m = \min \{a_k, a_{k+1}, \cdots, a_{n-1}\}$.
  ii. Put $a_m$ at the first place by swapping it with $a_k$.

```c
void sort_impl(int *a, int k, int n) {
  if (k == n - 1) return;

  int m = k;
  for (int i = k + 1; i < n; ++i)
    if (a[i] < a[m]) m = i;

  swap(&a[m], &a[k]); // the "swap" function we defined in previous lectures

}
```

# Selection-sort

Do it **recursively**: Suppose we are going to sort $\langle a_k, a_{k+1}, \cdots, a_{n-1}\rangle$, for some $k$.

- If $k < n - 1$:
  i. Find the minimal number $a_m = \min\left\{a_k, a_{k+1}, \cdots, a_{n-1}\right\}$.
  ii. Put $a_m$ at the first place by swapping it with $a_k$.
  iii. Sort the rest part $\langle a_{k+1}, \cdots, a_{n-1}\rangle$ **recursively**.

```
void sort_impl(int *a, int k, int n) {
  if (k == n - 1) return;

  int m = k;
  for (int i = k + 1; i < n; ++i)
    if (a[i] < a[m]) m = i;

  swap(&a[m], &a[k]); // the "swap" function we defined in previous lectures

  sort_impl(a, k + 1, n); // sort the rest part recursively
}
```

# Selection-sort

```
void sort_impl(int *a, int k, int n) {
  if (k == n - 1)
    return;

  int m = k;
  for (int i = k + 1; i < n; ++i)
    if (a[i] < a[m])
      m = i;

  swap(&a[m], &a[k]); // the "swap" function we defined in previous lectures

  sort_impl(a, k + 1, n); // sort the rest part recursively
}

void sort(int *a, int n) {
  sort_impl(a, 0, n);
}
```