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.

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

```
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.

```
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.

```
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`

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

- `sizeof(struct A) == 16`

```
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?

```
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:

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

Use C99 designators: (highly recommended)

```
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

```
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:

```
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**:

```
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:

```
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:

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

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

```
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**:

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

The object `s` is returned as if

```
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

```
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**.

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

```
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**.

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