Структура (**struct**) — композитный тип данных, инкапсулирующий без сокрытия набор значений различных типов.

Конструкция, которая бы могла агрегировать различные типы данных под одним именем.

#include <stdbool.h>

#include <stddef.h>

#include <stdio.h>

#include <stdlib.h>

#define heap\_blocks 16

#define block\_capacity 1024

struct heap {

struct block {

char contents[block\_capacity];

} blocks[heap\_blocks];

bool is\_occupied[heap\_blocks];

} global\_heap = {0};

struct block\_id {

size\_t value;

bool valid;

struct heap\* heap;

};

struct block\_id block\_id\_new(size\_t value, struct heap\* from) {

return (struct block\_id){.valid = true, .value = value, .heap = from};

}

struct block\_id block\_id\_invalid() { return (struct block\_id){.valid = false}; }

bool block\_id\_is\_valid(struct block\_id bid) {

return bid.valid && bid.value < heap\_blocks;

}

/\* find block \*/

bool block\_is\_free(struct block\_id bid) {

if(!block\_id\_is\_valid(bid))

return false;

return !bid.heap->is\_occupied[bid.value];

}

return block\_id\_invilid();

}

/\* allocate \*/

block allocate =true;

/\* find a free block, reserve it and return its id \*/

struct block\_id block\_allocate(struct heap\* heap) {

for(size\_t i=0;i<heap\_blocks;i++){

if(!heap->is\_occupied[i]){

heap->is\_occupied[i]=true;

return block\_id\_new(i,heap);

}

}

}

/\* mark block as 'free' \*/

void block\_free(struct block\_id b) {

b.heap->is\_occupied[b.value]=false;

}

/\* printer \*/

const char\* block\_repr(struct block\_id b) {

static const char\* const repr[] = {[false] = " .", [true] = " ="};

if (b.valid)

return repr[b.heap->is\_occupied[b.value]];

else

return "x";

}

void block\_debug\_info(struct block\_id b, FILE\* f) {

fprintf(f, "%s", block\_repr( b));

}

void block\_foreach\_printer(struct heap const\* h, size\_t count,

void printer(struct block\_id, FILE\* f),

FILE\* f) {

for (size\_t c = 0; c < count; c++)

printer(block\_id\_new(c,h), f);

}

void heap\_debug\_info(struct heap const\* h, FILE\* f) {

block\_foreach\_printer(h, heap\_blocks, block\_debug\_info, f);

fprintf(f, "\n");

}

/\* -------- \*/

int main() {

heap\_debug\_info(&global\_heap, stdout);

block\_allocate(&global\_heap);

struct block\_id bid = block\_allocate(&global\_heap);

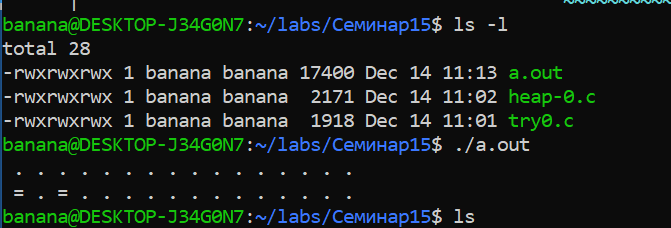
block\_allocate(&global\_heap);

block\_free(bid);

heap\_debug\_info(&global\_heap, stdout);

return 0;

}



#include <stdbool.h>

#include <stddef.h>

#include <stdio.h>

#include <stdlib.h>

#define HEAP\_BLOCKS 16

#define BLOCK\_CAPACITY 1024

enum block\_status { BLK\_FREE = 0, BLK\_ONE, BLK\_FIRST, BLK\_CONT, BLK\_LAST };

struct heap {

struct block {

char contents[BLOCK\_CAPACITY];

} blocks[HEAP\_BLOCKS];

enum block\_status status[HEAP\_BLOCKS];

} global\_heap = {0};

struct block\_id {

size\_t value;

bool valid;

struct heap\* heap;

};

struct block\_id block\_id\_new(size\_t value, struct heap\* from) {

return (struct block\_id){.valid = true, .value = value, .heap = from};

}

struct block\_id block\_id\_invalid() {

return (struct block\_id){.valid = false};

}

bool block\_id\_is\_valid(struct block\_id bid) {

return bid.valid && bid.value < HEAP\_BLOCKS;

}

/\* Find block \*/

bool block\_is\_free(struct block\_id bid) {

if (!block\_id\_is\_valid(bid))

return false;

return bid.heap->status[bid.value] == BLK\_FREE;

}

/\* Allocate \*/

struct block\_id block\_allocate(struct heap\* heap, size\_t size) {

size\_t blocks\_required = (size - 1) / BLOCK\_CAPACITY + 1;

if (blocks\_required > HEAP\_BLOCKS) return block\_id\_invalid();

bool can\_allocate = true;

for (size\_t i = 0; i < HEAP\_BLOCKS - blocks\_required + 1; i++) {

if (heap->status[i] == BLK\_FREE) {

if (blocks\_required == 1) {

heap->status[i] = BLK\_ONE;

return block\_id\_new(i, heap);

}

for (size\_t j = i; j < i + blocks\_required; j++) {

if (!heap->status[j] == BLK\_FREE) {

can\_allocate = false;

break;

}

}

if (can\_allocate) {

heap->status[i] = BLK\_FIRST;

heap->status[i+blocks\_required-1] = BLK\_LAST;

for (size\_t j = i + 1; j < i + blocks\_required - 1; j++) {

heap->status[j] = BLK\_CONT;

}

return block\_id\_new(i, heap);

}

}

}

return block\_id\_invalid();

}

/\* Free \*/

void block\_free(struct block\_id b) {

if (!b.valid) return;

if (b.heap->status[b.value] == BLK\_ONE) {

b.heap->status[b.value] = BLK\_FREE;

} else if (b.heap->status[b.value] == BLK\_FIRST) {

size\_t block\_id\_value = b.value;

while (b.heap->status[block\_id\_value] != BLK\_LAST) {

b.heap->status[block\_id\_value] = BLK\_FREE;

block\_id\_value++;

}

b.heap->status[block\_id\_value] = BLK\_FREE;

}

}

/\* Printer \*/

const char\* block\_repr(struct block\_id b) {

static const char\* const repr[] = {[BLK\_FREE] = " .",

[BLK\_ONE] = " \*",

[BLK\_FIRST] = "[=",

[BLK\_LAST] = "=]",

[BLK\_CONT] = " ="};

if (b.valid)

return repr[b.heap->status[b.value]];

else

return "INVALID";

}

void block\_debug\_info(struct block\_id b, FILE\* f) {

fprintf(f, "%s", block\_repr(b));

}

void block\_foreach\_printer(struct heap const\* h, size\_t count,

void printer(struct block\_id, FILE\* f), FILE\* f) {

for (size\_t c = 0; c < count; c++)

printer(block\_id\_new(c, h), f);

}

void heap\_debug\_info(struct heap const\* h, FILE\* f) {

block\_foreach\_printer(h, HEAP\_BLOCKS, block\_debug\_info, f);

fprintf(f, "\n");

}

/\* -------- \*/

int main() {

printf("test 1\n");

heap\_debug\_info(&global\_heap, stdout);

block\_allocate(&global\_heap, 1231);

struct block\_id bid = block\_allocate(&global\_heap, 5000);

block\_allocate(&global\_heap, 300);

block\_free(bid);

heap\_debug\_info(&global\_heap, stdout);

printf("test 2\n");

heap\_debug\_info(&global\_heap, stdout);

block\_allocate(&global\_heap, 20000);

heap\_debug\_info(&global\_heap, stdout);

return 0;

}

