Dynamic memory allocation

C programming · CM10 · 90 minutes

ISTIC · University of Rennes 2026–2027

Louis Ledoux

Start with a small program

Université de Rennes
Louis Ledoux

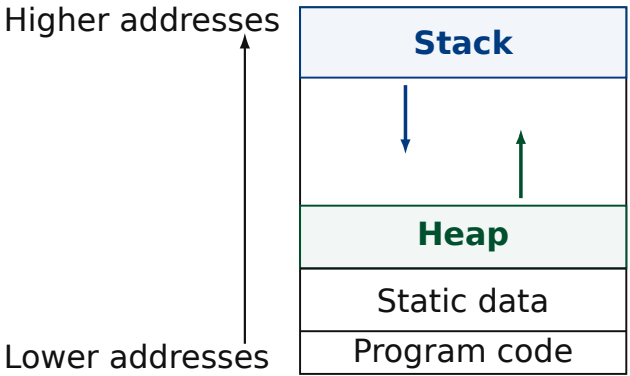
```
int total = 0;
void visit(void) {
    int values[4] = {10, 20, 30, 40};
    total += values[0];
}
```

Call `visit()` twice. What survives between the calls?

Live example: `demos/14-memory-layout/main.c`

Where does the program keep its data?

Université de Rennes
Louis Ledoux



Function calls
Automatic locals

Room to grow
Opposing directions

Dynamic allocations
Managed by the program

Globals and `static`

A typical layout, not a C language guarantee. Not to scale.

Who controls an object’s lifetime?

Université de Rennes
Louis Ledoux

Static storage

Globals and `static` objects exist for the **whole program**. Storage is reserved before execution.

Automatic storage

An ordinary local exists during its **block execution**. Each recursive call gets its own locals.

The implementation manages their storage. Deep recursion can **exhaust the stack**.

What if the size comes from the user?

Université de Rennes
Louis Ledoux

```
size_t count;
if (scanf("%zu", &count) != 1) return EXIT_FAILURE;
if (count == 0 || count > 1000) return EXIT_FAILURE;
int *p = malloc(count * sizeof *p);
if (p == NULL) return EXIT_FAILURE;
```

The size is chosen **during execution**. The allocation can outlive the function that creates it.

Live example: `demos/15-runtime-size/main.c`

Request storage for 100 integers

Université de Rennes
Louis Ledoux

```
size_t count = 100;
int *p = malloc(count * sizeof *p);
if (p == NULL) {
    fputs("Allocation failed\n", stderr);
    return EXIT_FAILURE;
}
```

`malloc` returns the block's address, or **`NULL`** on failure. The contents are **uninitialised**.

Count elements, request bytes

Université de Rennes
Louis Ledoux

```
void *malloc(size_t size);   // From <stdlib.h>
int *p = malloc(100 * sizeof *p);
```

- `size_t` is unsigned, with at least **16 value bits**.
- `sizeof` is an **operator** giving a size in bytes.
- `sizeof *p` is the size of one `int`, not the pointer.

Sizes depend on the platform. In C, the `void *` result needs no cast.

Initialise, then use the block

Université de Rennes
Louis Ledoux

```
for (size_t i = 0; i < count; ++i)
    p[i] = 0;
printf("first=%d, last=%d\n", p[0], p[count - 1]);
```

One contiguous allocation

Valid indices are **0 ≤ i < count**. Write each value before reading it.

Live example: `demos/01-malloc/main.c`

Finish the job with `free`

Université de Rennes
Louis Ledoux

```
void free(void *pointer);   // From <stdlib.h>
free(p);                    // After the last use.
p = NULL;
```

End the allocation's lifetime

Pass the start address of a live allocation from `malloc`, `calloc` or `realloc`. `free` returns no value.

`free(NULL)` has no effect. Free each allocation **once**.

Invalid or repeated `free` is **undefined behaviour**.

Four functions in <stdlib.h>

Université de Rennes
Louis Ledoux

Function	Purpose
`malloc(bytes)`	Allocate uninitialised storage
`calloc(count, size)`	Allocate storage with all bits zero
`realloc(p, bytes)`	Resize an allocation; it may move
`free(p)`	Release an allocation

Compile, run, then inspect

Université de Rennes
Louis Ledoux

```
gcc -std=c17 -Wall -Wextra -g -O0 main.c -o program
./program
valgrind --leak-check=full ./program
```

Read the **error summary**, then the **heap summary**. A program can print the expected result and still misuse memory.

Live example: `demos/04-array/main.c`

One character creates an overrun

Université de Rennes
Louis Ledoux

```
int *a = malloc(4 * sizeof *a);
if (!a) return EXIT_FAILURE;
for (size_t i = 0; i <= 4; ++i)
    a[i] = (int)(10 * (i + 1));
```

Which iteration writes outside the allocation?

Live example: `demos/16-overrun/main.c`

Allocated does not mean initialised

Université de Rennes
Louis Ledoux

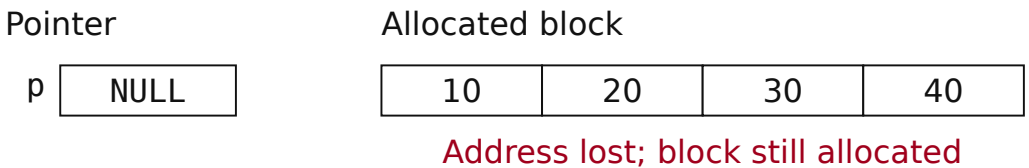
```
int *a = malloc(4 * sizeof *a);
if (!a) return EXIT_FAILURE;
printf("a[0] = %d\n", a[0]);

valgrind --track-origins=yes ./program
```

Live example: `demos/17-uninitialised/main.c`

Memory leak: lose the address

Université de Rennes
Louis Ledoux

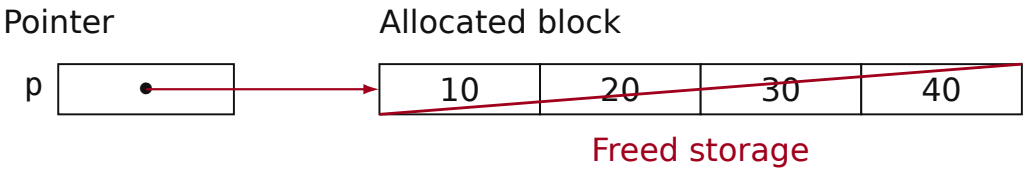
```
p = NULL;  // No free before the only address is lost.
```



Live example: `demos/18-lost-allocation/main.c`

Dangling pointer: the object is gone

Université de Rennes
Louis Ledoux




```
free(p);
printf("%d\n", *p);  // Use after free: undefined behaviour.
```

Live example: `demos/02-dangling/main.c`

Local objects can expire too

Université de Rennes
Louis Ledoux

```
int *bad_result(void) {
    int value = 42;
    return &value;  // The local dies when this call ends.
}
```

A dangling pointer need not come from free

Returning an address does not extend an automatic object’s lifetime.

Who is responsible for reclamation?

Université de Rennes
Louis Ledoux

C: explicit responsibility

For every successful allocation, decide who will release it and when. Repeated leaks consume more memory.

Java: garbage collection

The runtime can reclaim unreachable objects automatically.

Checkpoint · questions 1–5

Université de Rennes
Louis Ledoux

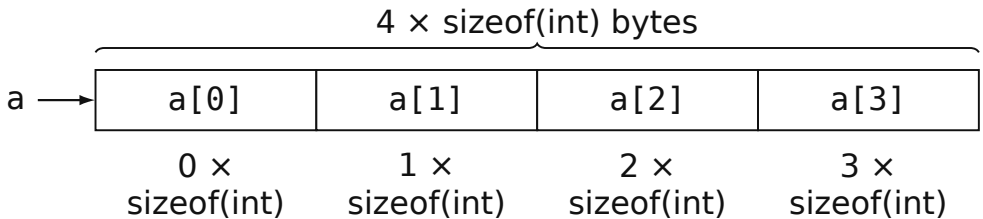
Answer questions **1–5**, then discuss your reasoning.

Open the allocation quiz.

A one-dimensional integer array

Université de Rennes
Louis Ledoux

```
int *a = malloc(4 * sizeof *a);
```

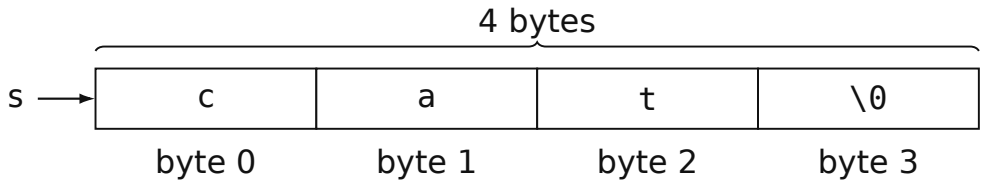


Use `a[i]`, equivalently `*(a + i)`, for **0 ≤ i < 4**.

A character array can hold a string

Université de Rennes
Louis Ledoux

```
char *s = malloc(4 * sizeof *s);
```



Three letters need **four bytes**, including **`'\0'`**.

Live example: `demos/05-string/main.c`

Assigning a pointer is not copying text

Université de Rennes
Louis Ledoux

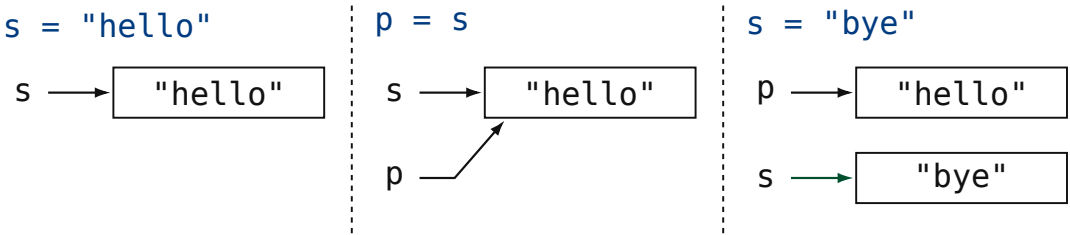
```
char *s = malloc(100);
if (!s) return EXIT_FAILURE;
s = "hello";
char *p = s;
s = "bye";
```

Predict the values of p and s, and the fate of the allocation.

Live example: `demos/03-leak/main.c`

Follow the three assignments

Université de Rennes
Louis Ledoux



Original heap block: 100 bytes, now unreachable

The literals are separate from the lost heap block. Neither pointer can now be passed to `free`.

Copy characters into owned storage

Université de Rennes
Louis Ledoux

```
char *s = malloc(6);
if (!s) return EXIT_FAILURE;
strcpy(s, "hello");  // Include <string.h>.
printf("%s\n", s);
free(s);
```

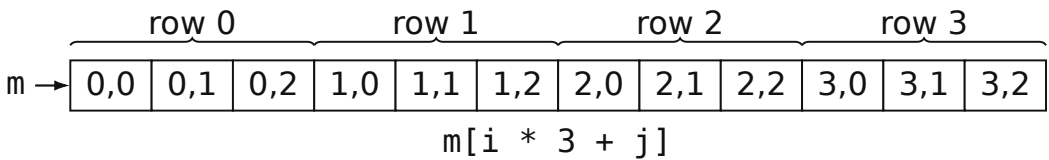
Alternatively, just reference a literal

Use `const char *s = "hello";` when no writable copy is needed.

A 2D array in one block

Université de Rennes
Louis Ledoux

```
int *m = malloc(4 * 3 * sizeof *m);
/* After checking m: element (i, j) is m[i * 3 + j]. */
```



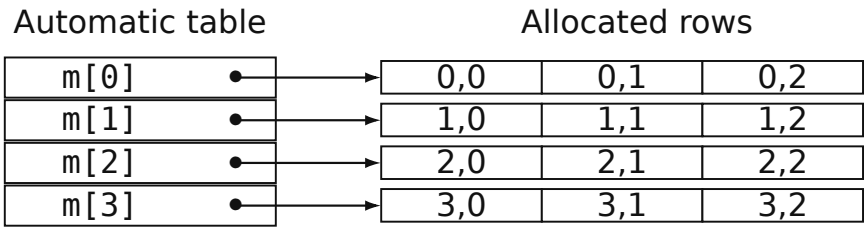
One allocation, one **`free(m)`**.

Live example: `demos/06-matrix-flat/main.c`

An array of row pointers

Université de Rennes
Louis Ledoux

```
int *m[4];  // Automatic table of four pointers.
for (size_t i = 0; i < 4; ++i)
    m[i] = malloc(3 * sizeof *m[i]);
```



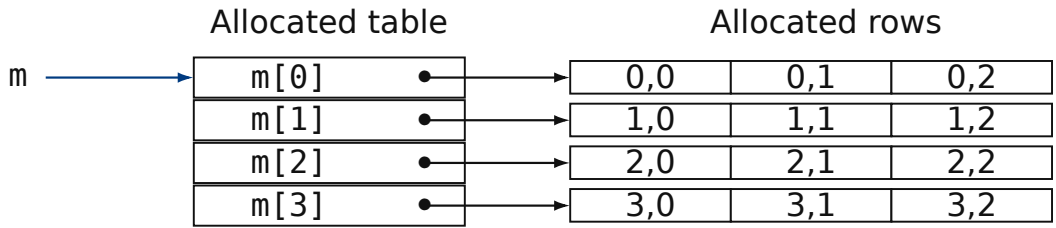
Check each row. Use `m[i][j]`; free each row, **not** `m`.

Live example: `demos/07-matrix-rows/main.c`

Allocate the pointer table too

Université de Rennes
Louis Ledoux

```
int **m = malloc(4 * sizeof *m);
for (size_t i = 0; i < 4; ++i)
    m[i] = malloc(3 * sizeof *m[i]);
```



Check `m` first, then each row. Free **rows first**, then `free(m)`.

Live example: `demos/08-matrix-dynamic/main.c`

Checkpoint · questions 6–8

Université de Rennes
Louis Ledoux

Answer questions **6–8**, then discuss your reasoning.

Open the allocation quiz.

User-defined types

Université de Rennes
Louis Ledoux

Three tools

`enum` names integer constants. `struct` groups members. `typedef` introduces a name for a type.

Use them to describe more complex data, including list nodes.

Enumerations

Université de Rennes
Louis Ledoux

```
enum day { mon, tue, wed, thu, fri, sat, sun };
enum day today;
today = wed;
```

By default, values start at **0** and increase by **1**.

Named integer constants

`mon` is 0; `wed` is 2; `sun` is 6.

A structure groups several members

Université de Rennes
Louis Ledoux

```
struct address {
    char street[100];
    int number;
};
```

A type definition

Members may be integers, floating-point values, arrays, pointers or other structures. No address object exists yet.

Define a type, then create objects

Université de Rennes
Louis Ledoux

```
struct address home;
struct address office;
```

These definitions allocate **two distinct objects** of the same type.

Local objects need initialisation

Ordinary local members have indeterminate values until initialised or assigned. The type definition alone sets no values.

Initialising a structure

Université de Rennes
Louis Ledoux

```
struct address home = { "Paul Bert", 12 };
```

Or combine the type and object definitions:

```
struct address {
    char street[100];
    int number;
} home = { "Paul Bert", 12 };
```

Structure layout and alignment

Université de Rennes
Louis Ledoux

Padding

The implementation may insert unused bytes between members and at the end to satisfy alignment requirements.

- Members appear in declaration order.
- `sizeof(struct T)` includes padding.
- Layout and alignment depend on the implementation.

Same members, different order

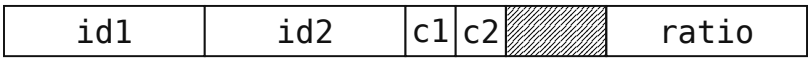
Université de Rennes
Louis Ledoux

```
struct grouped {
    int id1, id2;
    char c1, c2;
    float ratio;
};
struct interleaved {
    int id1; char c1;
    int id2; char c2;
    float ratio;
};
```

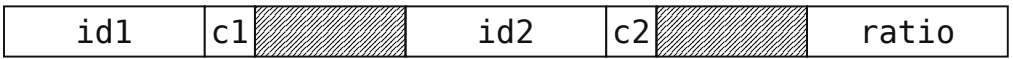
See where the padding goes

Université de Rennes
Louis Ledoux

Grouped: 16 bytes



Interleaved: 20 bytes



Here: 4-byte `int`/`float`, alignment 4; 1-byte `char`.

16 vs 20 bytes in this layout. Measure with `sizeof` and `offsetof`.

Live example: `demos/09-struct-padding/main.c`

Access members through an object

Université de Rennes
Louis Ledoux

```
struct address home = { "Paul Bert", 12 };
strcpy(home.street, "Rue de Paris");
home.number = 14;
```

The dot operator

Use `.` with a structure object.

`strcpy` needs <`string.h`> and enough space for the terminator.

Access members through a pointer

Université de Rennes
Louis Ledoux

```
struct address *p = &home;
strcpy(p->street, "Rue de Paris");
p->number = 16;
```

The arrow operator

Use **->** with a pointer. `p->number` is the same member as `(*p).number`.

Both forms above modify **home**.

typedef: a name for a type

Université de Rennes
Louis Ledoux

```
typedef int LENGTH;
typedef enum { FALSE, TRUE } BOOLEAN;
typedef struct date DATE;
```

Alias

A `typedef` introduces a type name, not an object or a distinct underlying type. Then write `LENGTH width;`.

Combine a date and a day of the week

Université de Rennes
Louis Ledoux

```
struct date { short day, month; int year; };
typedef enum { mon, tue, wed, thu, fri, sat, sun } DAY;
typedef struct { DAY day; struct date date; } A_DATE;
```

A nested structure

A_DATE contains a DAY and a complete `struct date`.

Access nested members

Université de Rennes
Louis Ledoux

```
A_DATE today = { wed, { 3, 9, 2014 } };
today.day = wed;
today.date.year = 2014;
A_DATE *p = &today;
p->day = today.day;
p->date.year = 2026;
```

`p->date.year` reaches the nested member of **the same object**.

Live example: `demos/10-nested-struct/main.c`

Checkpoint · questions 9–12

Université de Rennes
Louis Ledoux

Answer questions **9–12**, then discuss your reasoning.

Open the allocation quiz.