Imperative programming

Dynamic memory handling

Tamás Kozsik et al

Eötvös Loránd University

October 25, 2022

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

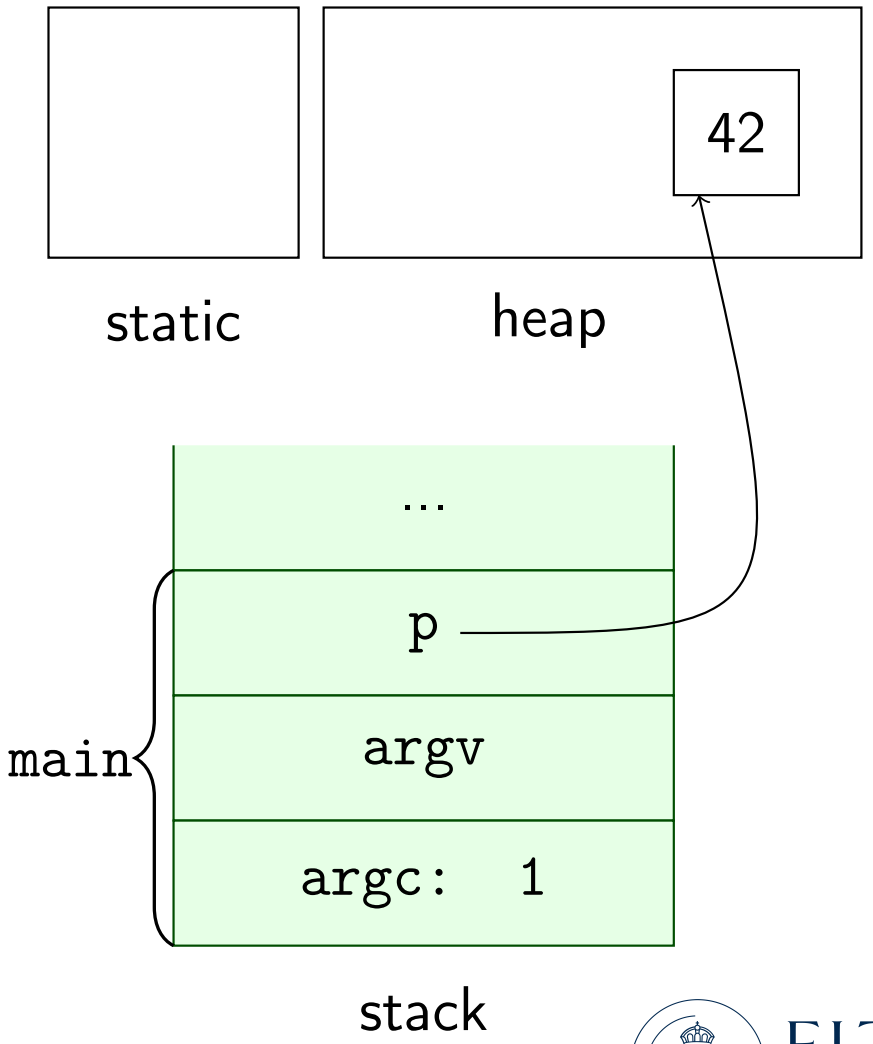
Table of contents

Dynamic memory handling

- Dynamic storage variables
  - Heap (dynamic storage)
- Lifetime: programmable
  - Creation: with allocation statement
  - Deletion
    - Deallocation statement (C)
    - Garbage collection (Haskell, Python, Java)
- Usage: indirection
  - Pointer (C)
  - Reference (Python, Java)

Pointers in C

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

int main()
{
  int* p;
  p = (int*)malloc(sizeof(int));
  if (p != NULL)
  {
    *p = 42;
    printf("%d\n", *p);
    free(p);
    return 0;
  }
  return 1;
}
```



Ingredients

- Pointer variable: `int* p;`
  - Warning: `int* p, v;`
  - Similarly: `int v, t[10];`
- Dereferencing (where does it point?): `*p`
- "Points nowhere": `NULL`
- Allocation and deallocation: `malloc()` and `free()` (`stdlib.h`)
- Type cast: `void*` $\rightarrow$ e.g. `int*`

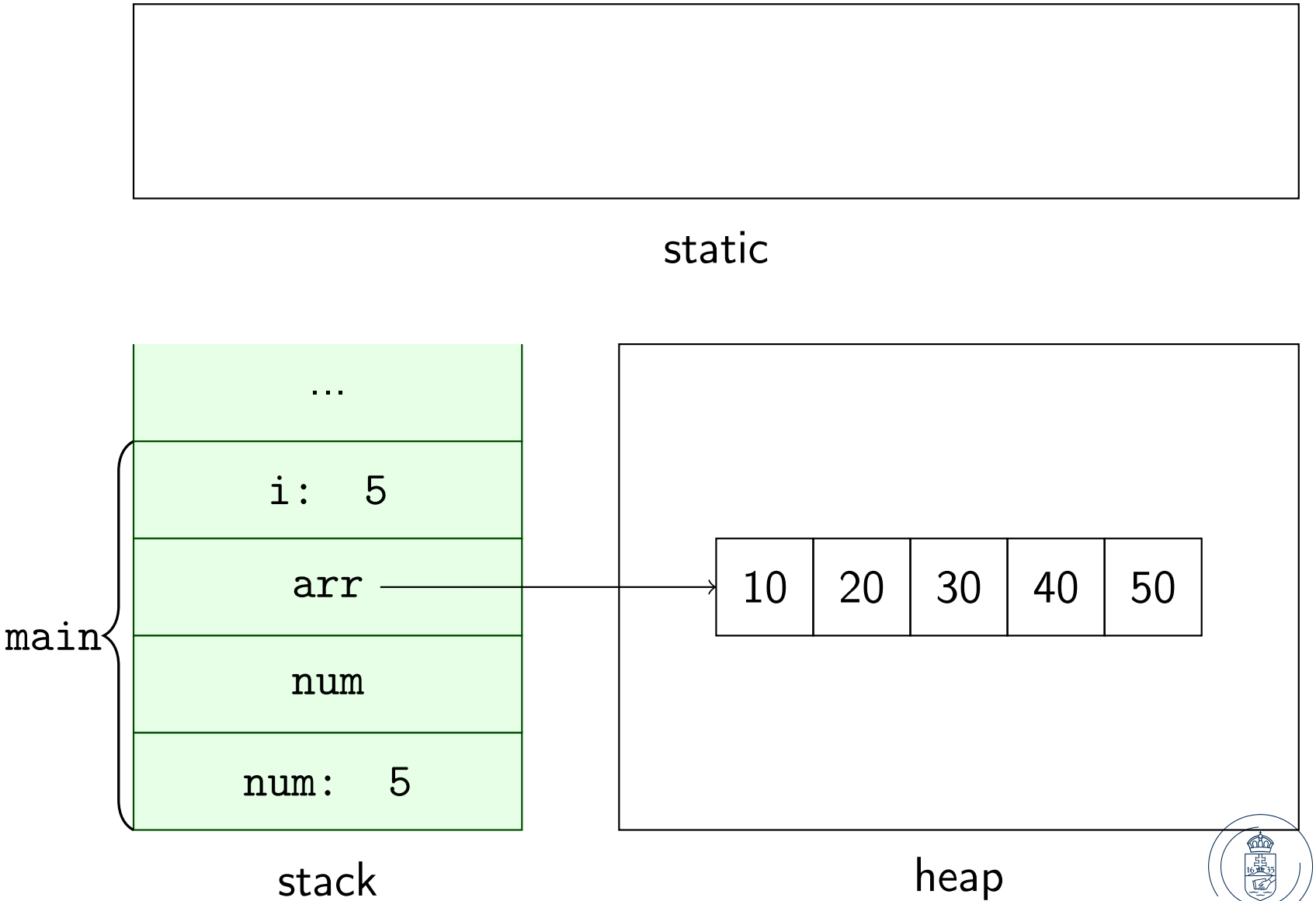
What is it good for?

- Dynamic size data(-structure)
- Linked data structure
- Output function parameters
- ...

Dynamic size data structures

```c
#include <stdlib.h>

int getNum() { ... }

int main()
{
  int num = getNum();
  int* arr = (int*)malloc(num * sizeof(int));

  if (arr == NULL)
    return 1;

  for (int i = 0; i < num; ++i)
    arr[i] = (i + 1) * 10;

  free(arr);
}
```

Dynamic size data structures

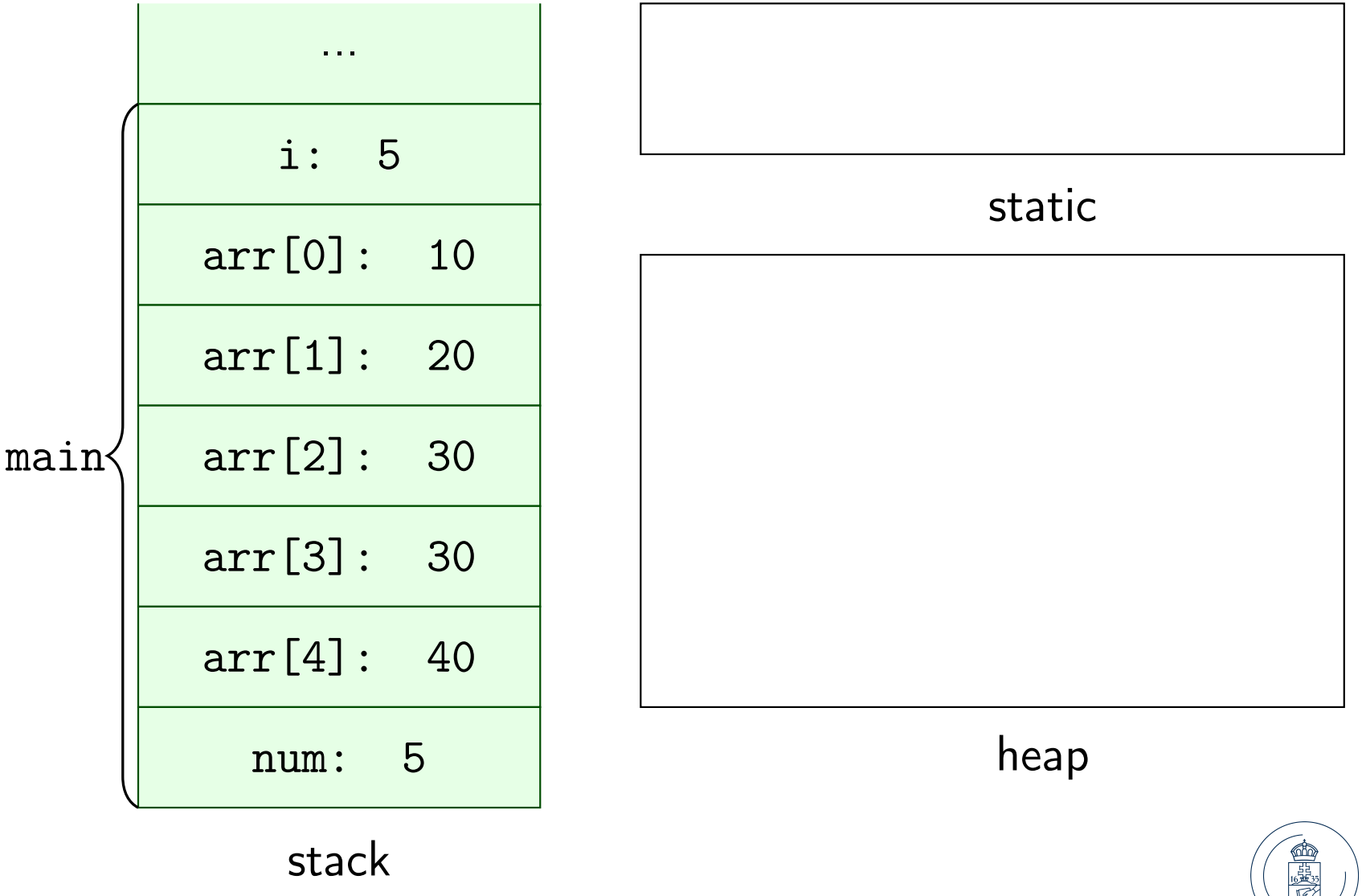


Avoid this solution

```
#include <stdlib.h>

int getNum() { ... }

int main()
{
  int num = getNum();
  int arr[num];

  if (arr == NULL)
    return 1;

  for (int i = 0; i < num; ++i)
    arr[i] = (i + 1) * 10;
}
```

- C99: Variable Length Array (VLA)
- Not available in ANSI C and C++ standards

Solution to avoid: VLA

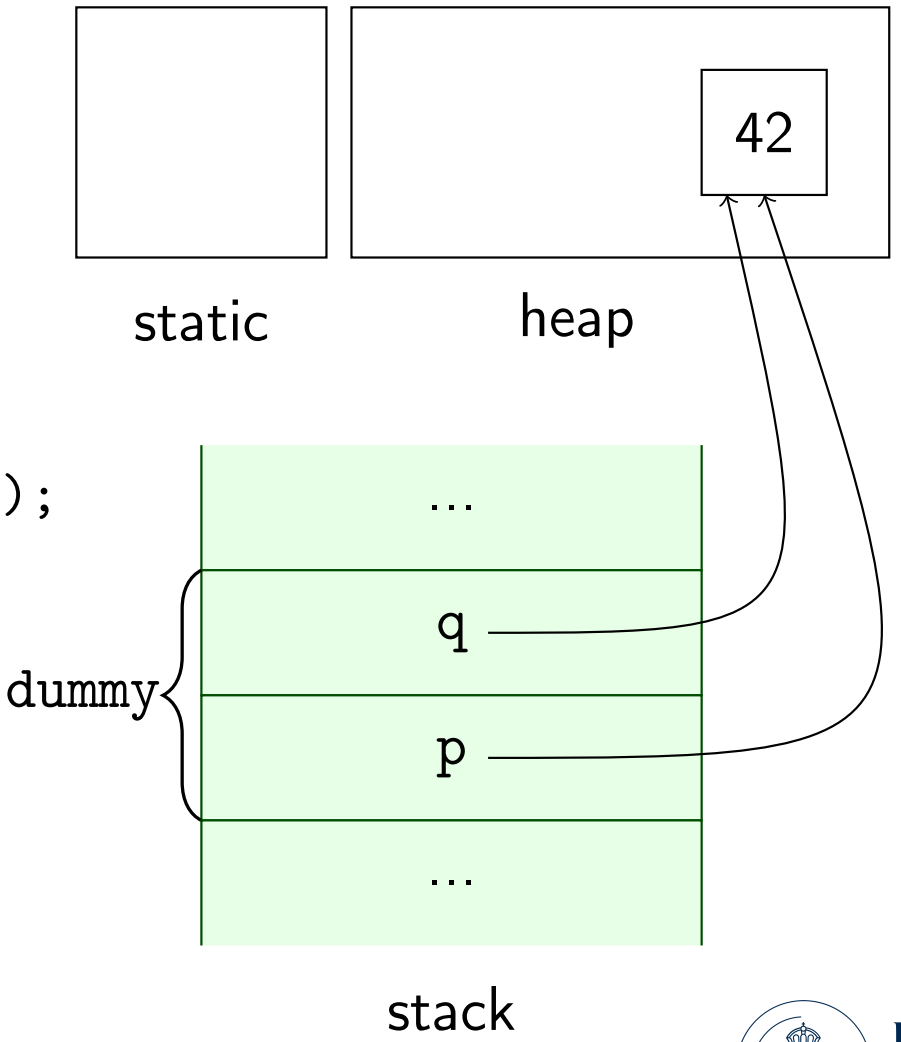


Linked data structure

- Sequence type
- Binary tree type
- Graph type
- ...

Aliasing

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

void dummy()
{
  int *p, *q;
  p = (int*)malloc(sizeof(int));
  if (p != NULL)
  {
    q = p;
    *p = 42;
    printf("%d\n", *q);
    free(p);
  }
}
```



Deallocation

Every dynamically created variable must be deallocated exactly once!

- Referencing a deallocated object
- Multiple deallocations
- No deallocation: memory leak

Referencing a deallocated object

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

void dummy()
{
  int *p, *q;
  p = (int*)malloc(sizeof(int));
  if (p != NULL)
  {
    q = p;
    *p = 42;
    free(p);
    printf("%d\n", *q);    /* hiba */
  }
}
```

Multiple deallocations

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

void dummy()
{
  int *p, *q;
  p = (int*)malloc(sizeof(int));
  if (p != NULL)
  {
    q = p;
    *p = 42;
    printf("%d\n", *q);
    free(p);
    free(q);    /* hiba */
  }
}
```

No deallocation

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

void dummy()
{
  int *p, *q;
  p = (int*)malloc(sizeof(int));
  if (p != NULL)
  {
    q = p;
    *p = 42;
    printf("%d\n", *q);
  }   /* hiba */
}
```

Owner?

```
int* make_ptr() {
  int* p = (int*)malloc(sizeof(int));
  *p = 42;
  return p;
}
```

```
int global = 42;
int* make_ptr() {
  return &global;
}
```

```
int main()
{
  int* ptr = make_ptr();
  printf("%d\n", *ptr);
  free(ptr);   /* should we deallocate? */
}
```