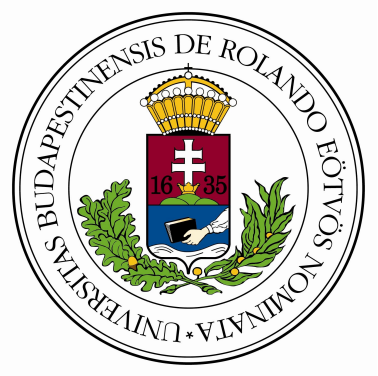
Imperative programming

8th Lecture



Kozsik Tamás

ELTE Eötvös Loránd Tudományegyetem

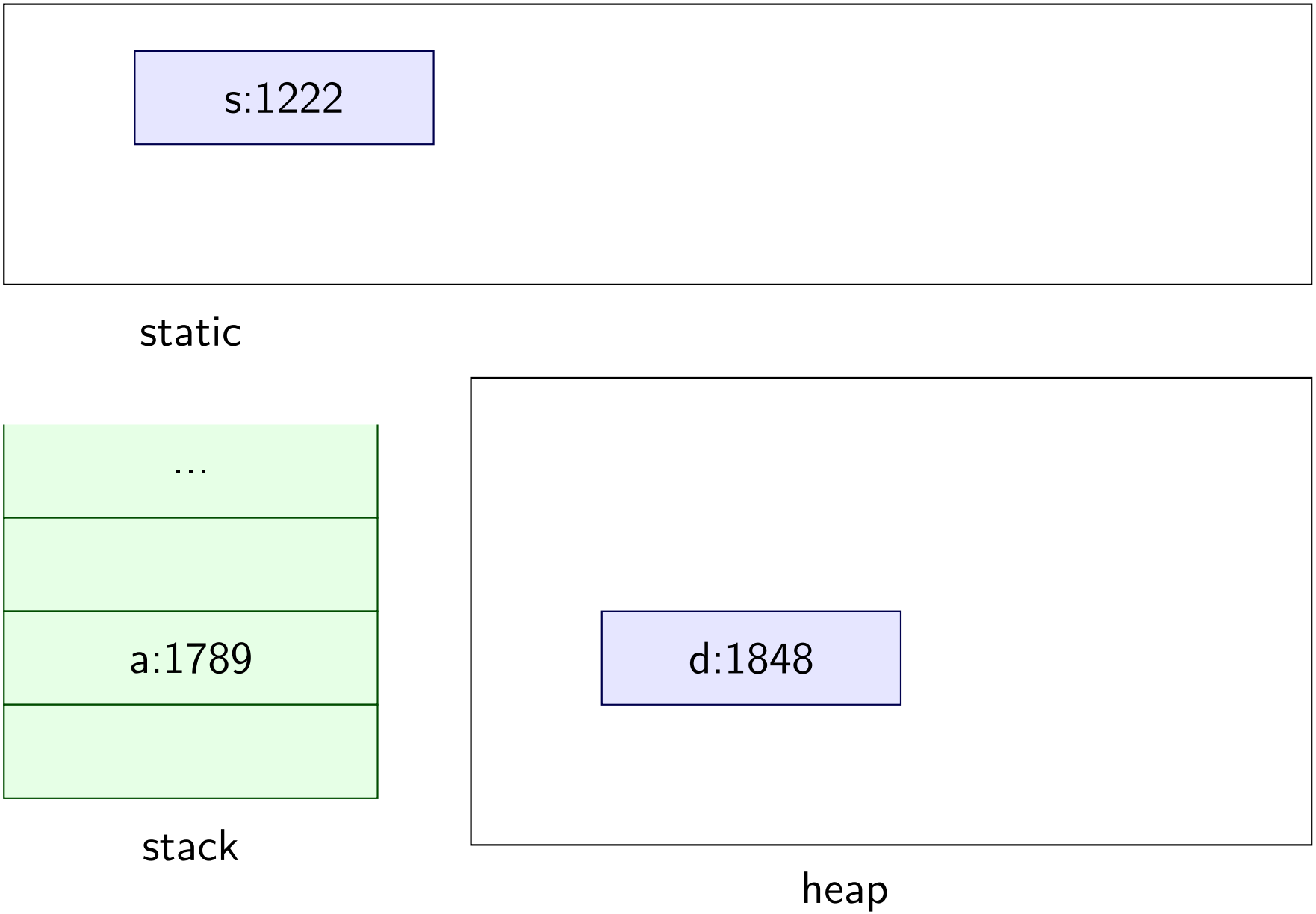
Outline

1. Dynamic Program Structure
   - (Execution Stack)
   - Lifetime and Storage of Variables
   - Parameter Passing
2. Dynamic Memory Handling

Memory Storage of „Variables’ ’

- static storage → static
- execution stack → automatic
- dynamic storage (heap) → dynamic

static - stack - heap



Static Storage Variable

- Static Storage
  - Static evaluation of declaration
  - Compiler knows how large memory is needed
- e.g. global variables
- Lifetime: from program start to program exit

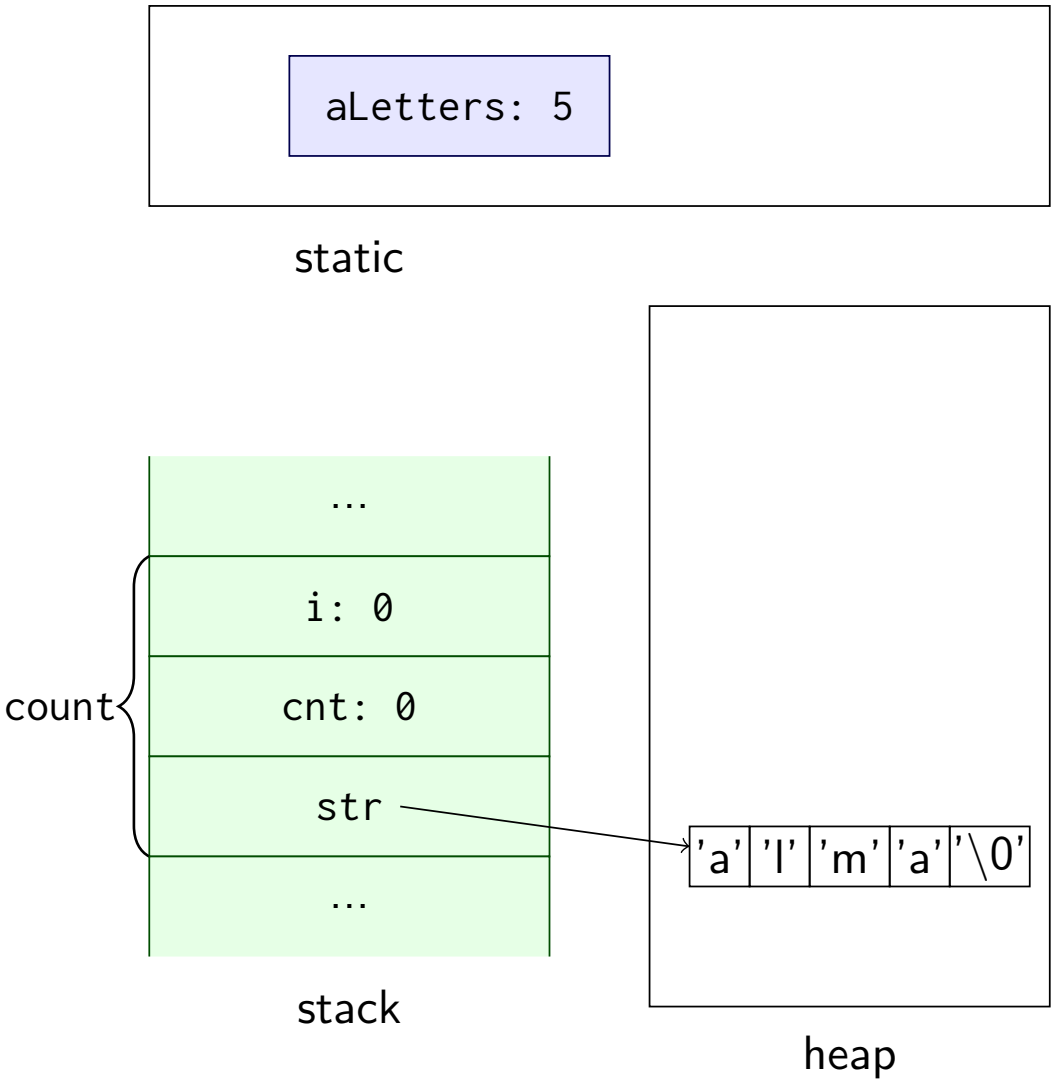
```
int counter = 0;
void signal()
{
    ++ counter;
}
```

Automatic Storage Variable

- Execution Stack
  - In activation records
- Local variables are *usually* like this
- Lifetime: execution of block
  - Automatically created and destroyed

```
int gcd( int a, int b ){
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

static - stack - heap




```
int aLetters = 0;
int count( char *str )
{
    int cnt=0, i=0;
    while (str[i]!='\0')
    {
        if (str[i]=='a')
            ++cnt;
        ++i;
    }
    aLetters += cnt;
    return cnt;
}
```

C - static local variables

- `static` keyword
- Scope: local variable
  - Information Hiding Principle
- Lifetime: like in case of global variables

```
int counter = 0;
void signal()
{
    ++ counter;

}
```

```
int signal()
{
    static int counter = 0;
    ++ counter;
    return counter;
}
```

Expressing Storage Class in C

- `static`
  - local
  - global
- `auto` (not used)
  - `auto` keyword has a different meaning in C++11
- `register` (not used)
  - optimization

```
int gcd( int a, int b ){
    auto int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

Optimization: memory operations on human scale

Source: David Jeppesen

clockrate	0.4 ns	1 sec
L1 cache	0.9 ns	2 sec
L2 cache	2.8 ns	7 sec
L3 cache	28 ns	1 min
DDR memory	~100 ns	4 min
SSD I/O	50-150 microsec	1,5-4 day
HDD I/O	1-10 ms	1-9 month
Internet	65 ms	5-10 year

C - static global declarations

- Not available in other compilation units
- „Internal Linkage''
- Part of the implementation
- Not part of the interface of the module
- Information hiding principle

```
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(){
    negative -= increment;
}

void signal(){
    positive += increment;
    compensate();
}
```

C program of multiple modules

```
gcc -c -W -Wall -pedantic -ansi main.c
gcc -c -W -Wall -pedantic -ansi positive.c
gcc -o main -W -Wall -pedantic -ansi positive.o main.o
```

positive.c

```c
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(){
    negative -= increment;
}
void signal(){
    positive += increment;
    compensate();
}
```

main.c

```c
#include <stdio.h>

int increment = 3;
extern int positive;
extern void signal(void);

int main(){
    signal();
    printf("%d\n", positive);
    return 0;
}
```

Usage of Global Variables

To be avoided!

Definition of Variables

With Declaration

- Statically and Automatically stored
  - Static Storage
  - Execution Stack
- Lifetime: from program structure
  - Scope
  - Except local static (C)

With Allocating Statement

- Dynamically stored
  - Heap (dynamic storage)
- Lifetime: programmable
- Deallocation
  - Deallocation Statement (C)
  - Garbage Collection (Python)

Block Statement

- New scope with local declarations
  - Contamination of namespace can be avoided
- Automatic storage variables
  - Lifetime can be shortened

Parameters of Subprogram

- In Definition: Formal Parameter List
- At Call: Actual Parameter List

Techniques of Parameter Passing

- Multiple parameter passing is present in various languages
  - pass-by-value, call-by-value
  - call-by-value-result
  - call-by-result
  - call-by-reference
  - call-by-sharing
  - call-by-need
  - call-by-name
- Execution stack!

Pass-by-value Parameter Passing

- Formal parameter: local variable with automatic storage class
- Actual parameter: initial value
- Call: value of actual parameter is copied into formal parameter
- Return: formal parameter lifetime is over

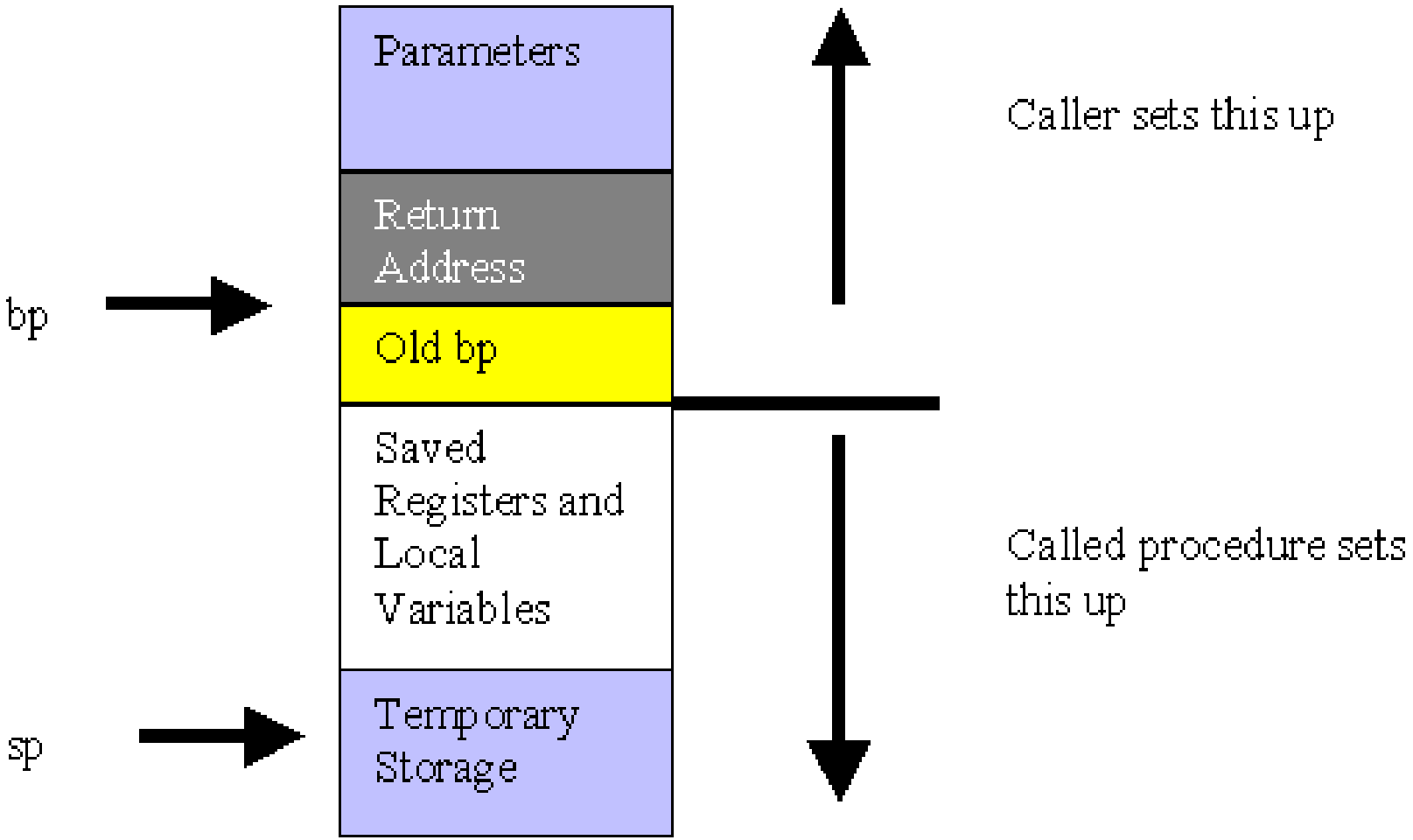
Pass-by-value Parameter Passing – Example

```
int gcd( int a, int b )
{
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
int main()
{
    int n = 1984, m = 356;
    int r = gcd(n,m);
    printf("%d %d %d\n",n,m,r);
}
```

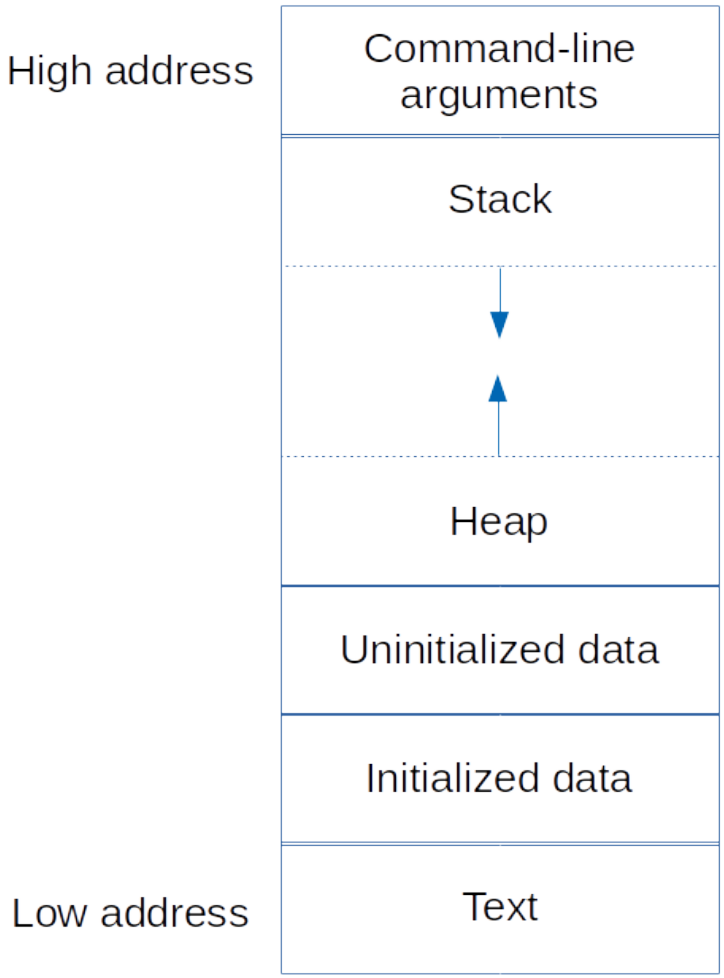
Activation Record

- All sorts of technical things
- Automatic storage variables of subprogram
  - e.g. formal parameters of subprogram
  - except parameters passed in registers

More precisely



Address space of C programs



Outline

1. Dynamic Program Structure
   - (Execution Stack)
   - Lifetime and Storage of Variables
   - Parameter Passing
2. Dynamic Memory Handling

Dynamic Memory Handling

- Dynamically stored „variables’ ’
  - Heap (dynamic storage)
- Lifetime: programmable
  - Creation: using allocation statement
  - Releasing
    - Deallocation statement (C)
    - Garbage collection (Python)
- Usage: indirection
  - Pointer (C)
  - Reference (Python)

Pointers in C

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

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

Ingredients

- Pointer (typed) variable: `int *p;`
  - Beware: `int* p, v;`
  - Likewise: `int v, t[10];`
- Dereference (where is it pointing to?): `*p`
- „It is not pointing to anywhere'': `NULL`
- Allocation and deallocation: `malloc` and `free` (`stdlib.h`)
  - Type forced: `void*` $\rightarrow$ pl. `int*`

What is it good for?

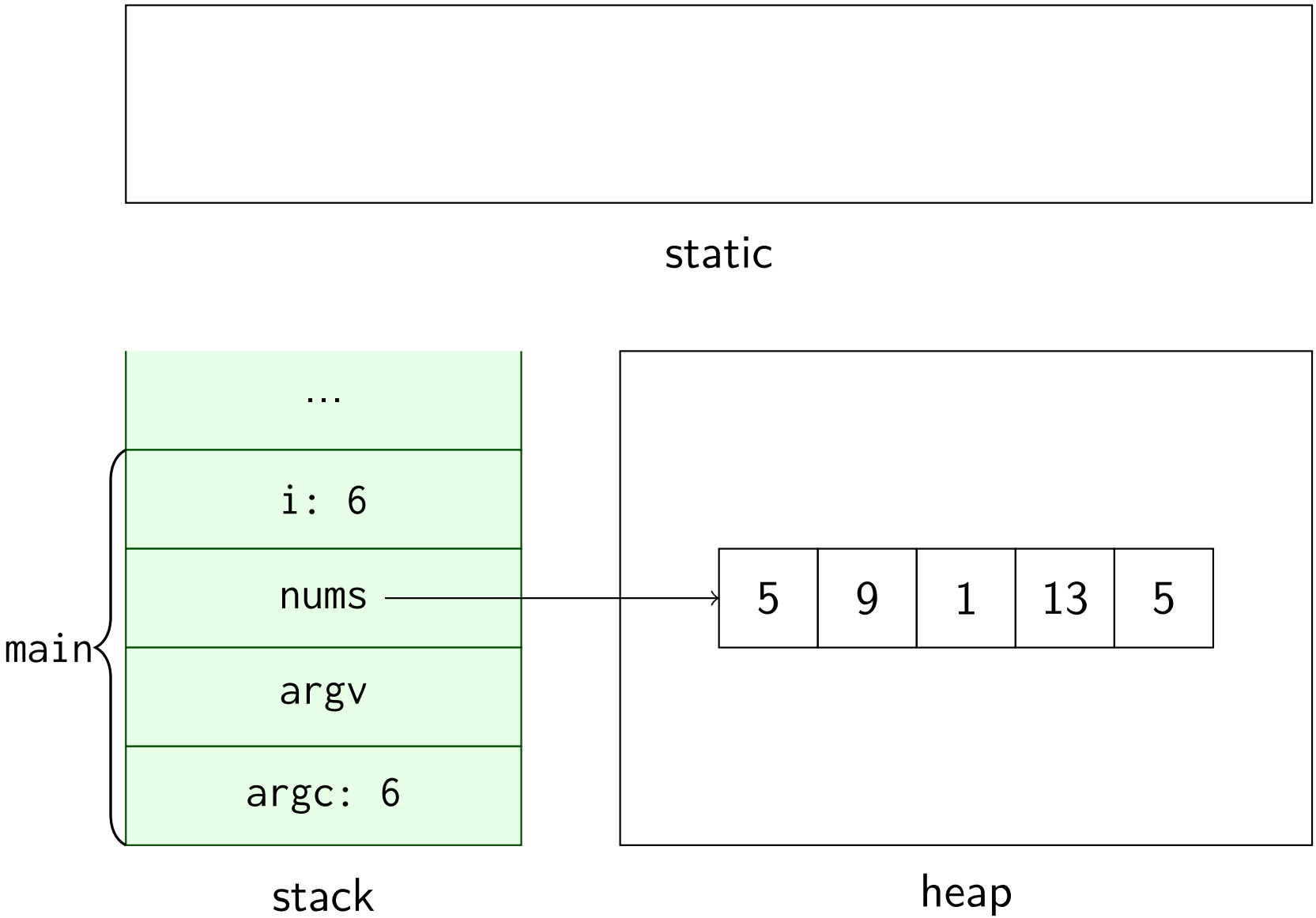
- Dynamic sized data(-structure)
- Chained data structure
- Parameter passing with output semantics
- ...

Dynamic Sized Data Structure

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

int main( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
    if( NULL != nums ){
        int i;
        for( i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
        /* TO DO: sort nums */
        for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
        free(nums);
        return 0;
    } else return 1;
}
```

Dynamic size data structures



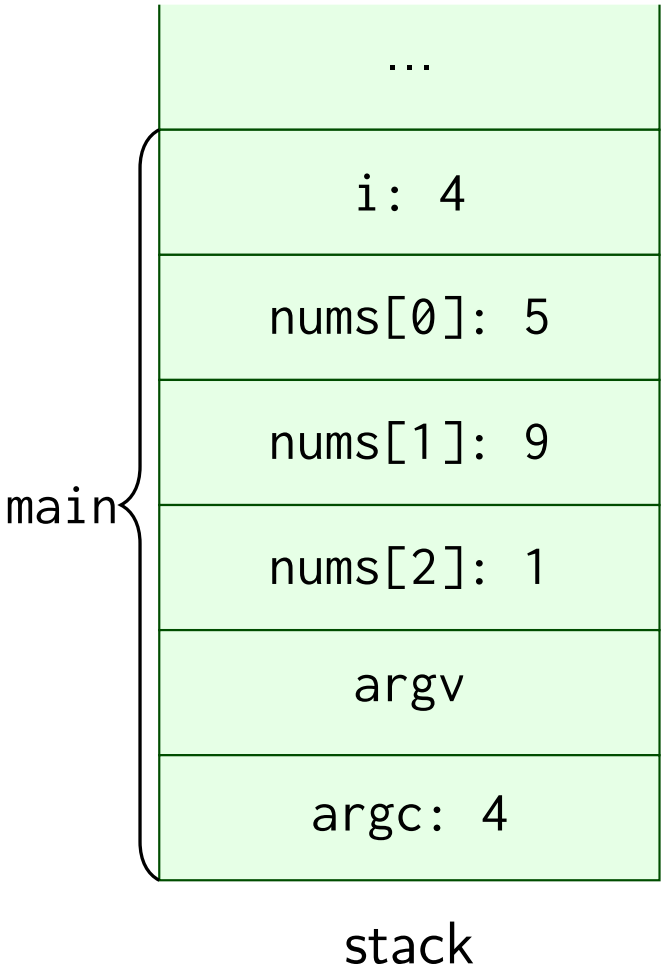
Solution to Avoid

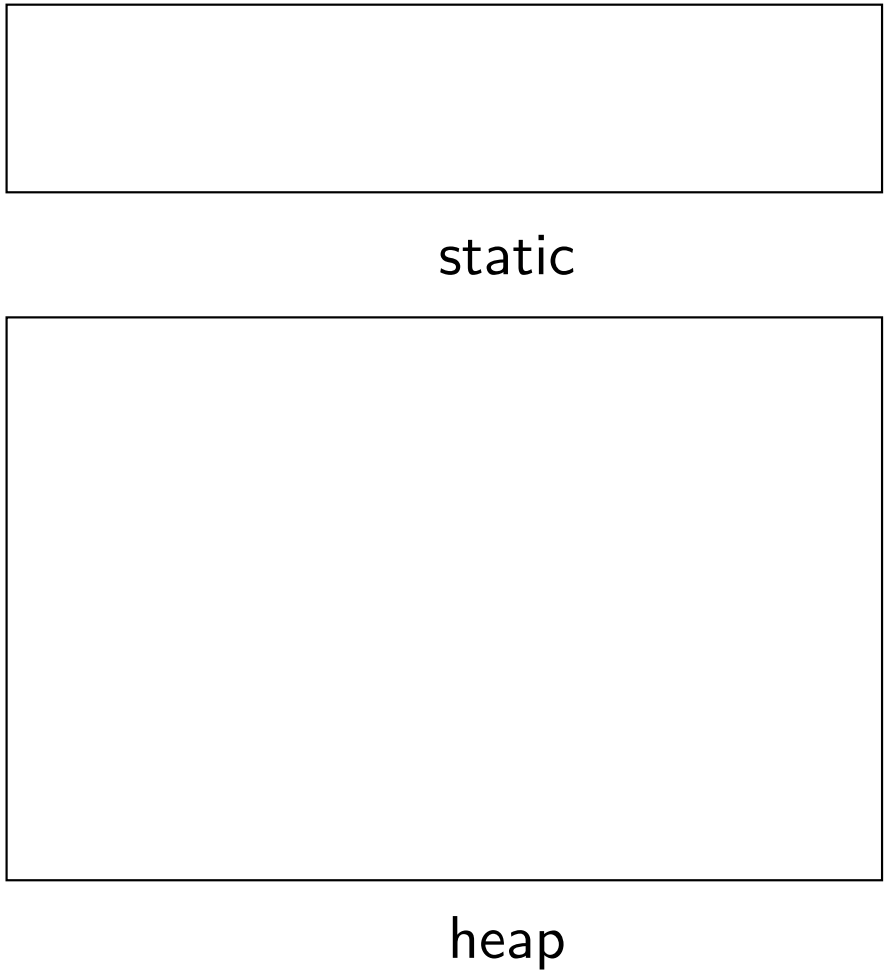
```c
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char* argv[] ){
    int nums[argc-1];
    int i;
    for( i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
    /* TO DO: sort nums */
    for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
    free(nums);
    return 0;
}
```

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

Avoid VLA





Chained Data Structure

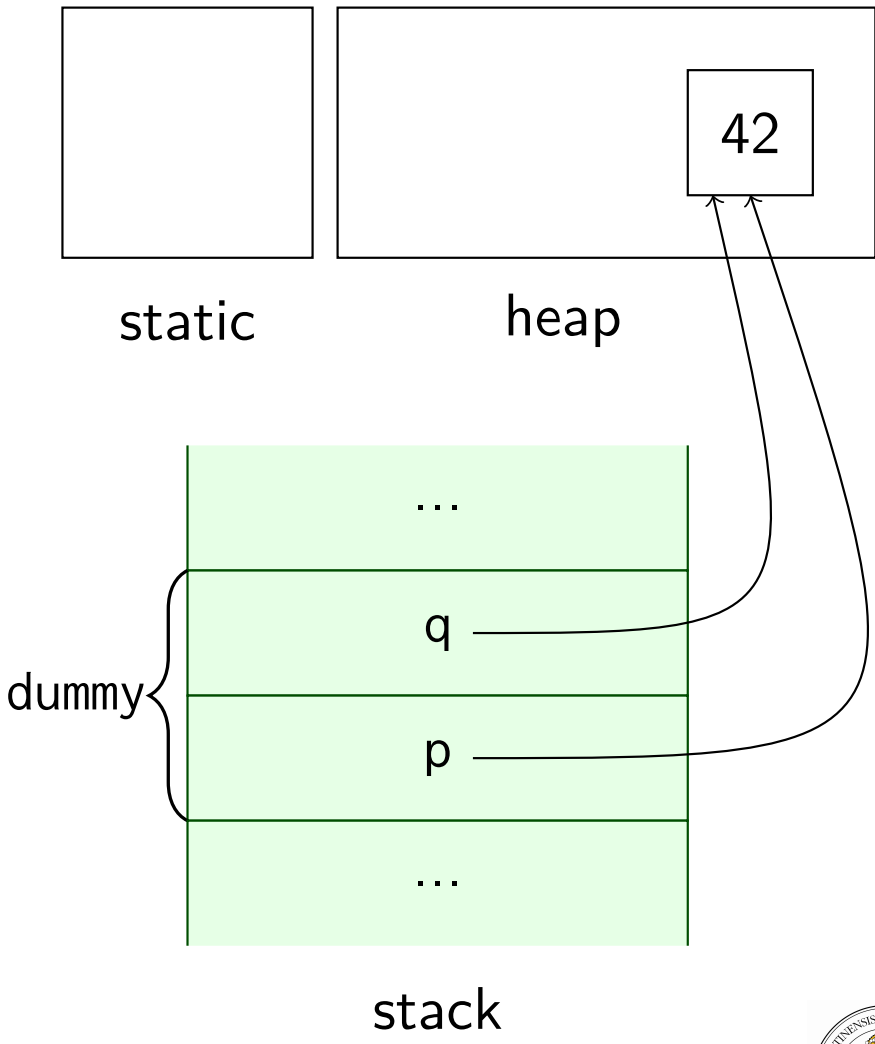
- Sequence type
- Binary tree type
- Graph type
- ...

Constant time insertion/removal during traversal

Aliasing

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

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



Deallocation (or Release)

Every dynamically created variable exactly once!

- If multiple times: error
- If not at all: „memory leaks''

Referring to a released (freed) variable is an error!

Referring to a Released Variable

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

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

Multiple Times Released Variable

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

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

Unreleased Variable

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

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

Owner?

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

Easy to Make Mistake!

```
int *produce( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
    if( NULL != nums ){
        for( int i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
    }
    return nums;
}
void *consume( int *nums ){
    for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
    free(nums);
}
int main( int argc, char* argv[] ){
    int *nums = produce(argc,argv);
    if( NULL != nums ){ /* TO DO: sort nums */ consume(nums); }
    return (NULL == nums);
}
```