

Introduction to C

Programming Workshop in C (67316)

Fall 2018

Lecture 4

25.10.2018

Pointers are variables that store the address of other variables

- **Declaration**

<type> *p; (e.g. `int *p;`)
p points to object of type <type>

- **Pointer → value (de-reference)**

*p refers to the object p points to
(e.g. `*p = x; y = *p;`)

- **Value → pointer**

&x - the address of x (e.g. `p = &y;`)

Passing Pointers & Arrays to functions

```
int foo( int *p );
```

```
int foo( int a[] );
```

```
int foo( int a[NUM] );
```

Are declaring the same interface:

In all cases, a ***pointer to int*** is being passed to the function foo

void *

`void *p` defines a pointer to undetermined type

```
int j;  
int *p = &j;  
void* q = p; // no cast needed  
p = (int*)q ; // cast is needed
```

All pointers can be casted one to the other, it may be useful sometimes, but beware...

- No pointer arithmetic is defined for void*
- We cannot access the content of the pointer – dereferencing is not allowed

Memory Management

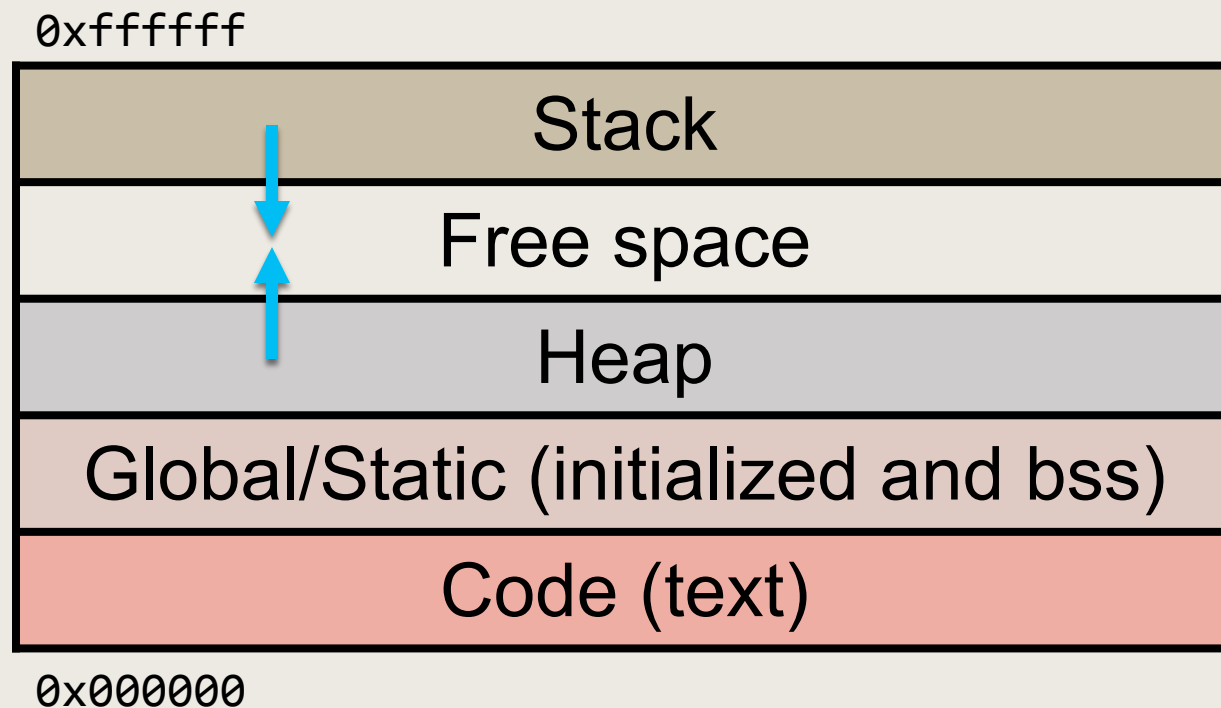
Finally!

Memory Organization (very briefly)

- Code segment
- Data Segment
 1. Stack
 2. Global / static area
 3. Dynamic heap

Memory Organization (very briefly)

- ❑ Allocated and initialized when loading and executing the program
- ❑ Memory access in **user mode** is restricted to this address space



Stack

Maintains memory during function calls:

- Arguments of the function
- Local variables
- Call Frame

Variables on the stack have **limited lifetime** and their size is defined during compilation

Stack - Example

```
int foo( int a, double f )
```



```
{
```

```
    int b;
```

```
    ...
```

```
    {
```

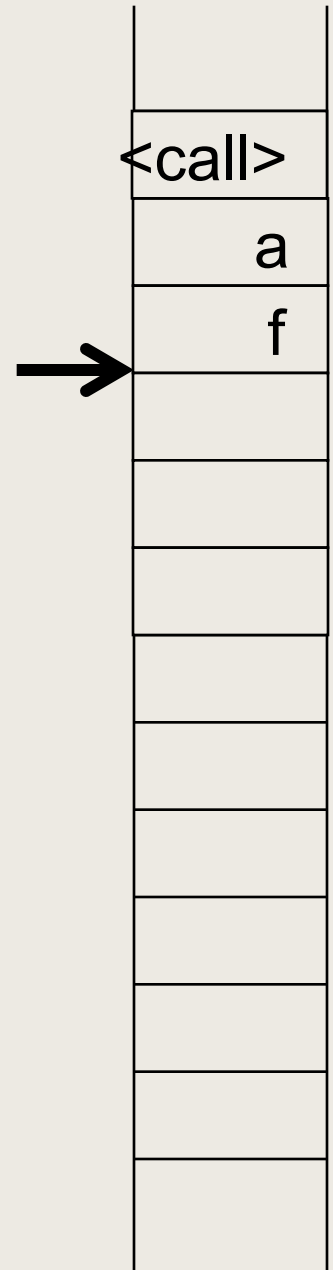
```
        int c;
```

```
        ...
```

```
    }
```

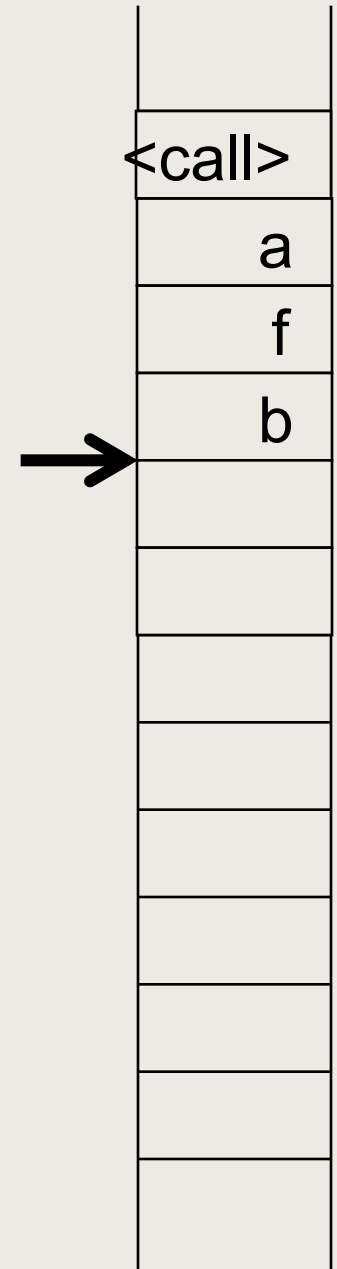

```
    ...
```

```
}
```



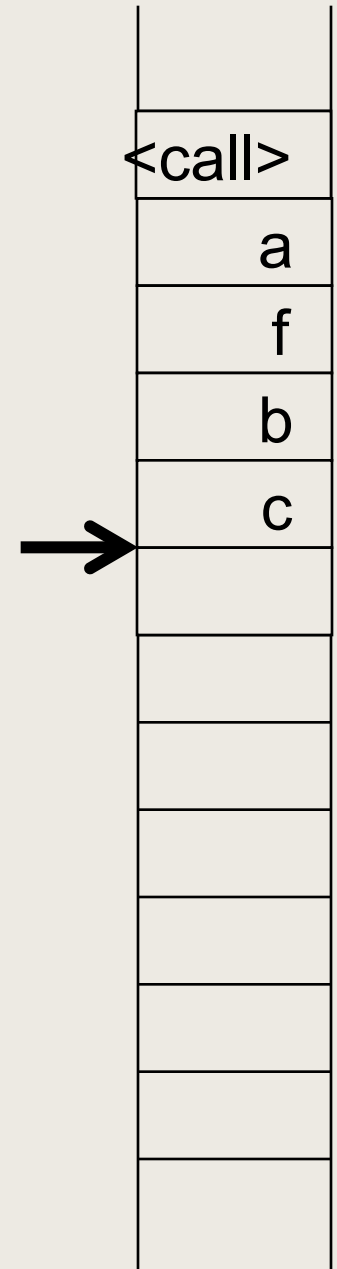
Stack - Example

```
int foo( int a, double f )  
{  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
    ...  
}
```



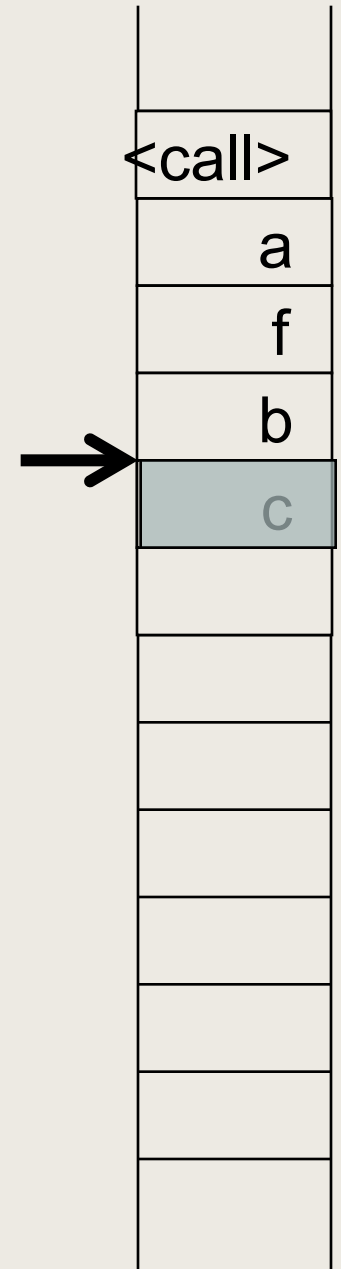
Stack - Example

```
int foo( int a, double f )  
{  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
    ...  
}
```



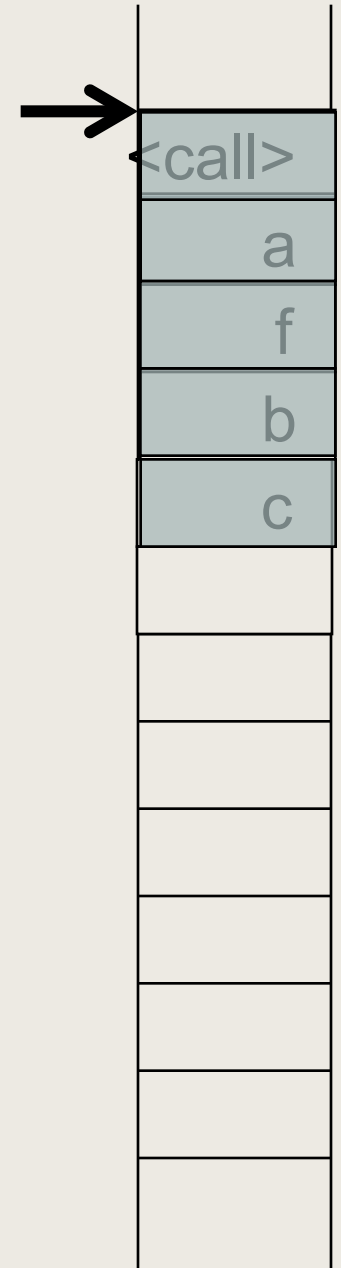
Stack - Example

```
int foo( int a, double f )  
{  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
    ...  
}
```



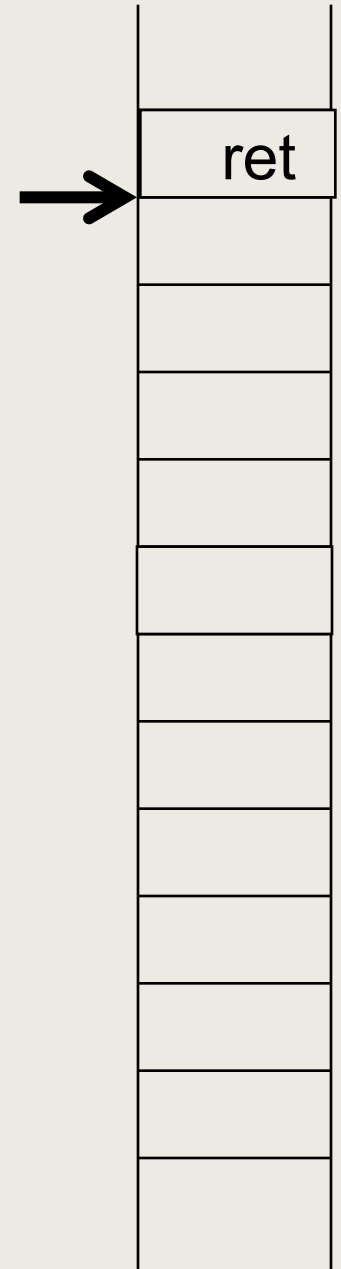
Stack - Example

```
int foo( int a, double f )  
{  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
    ...  
}
```



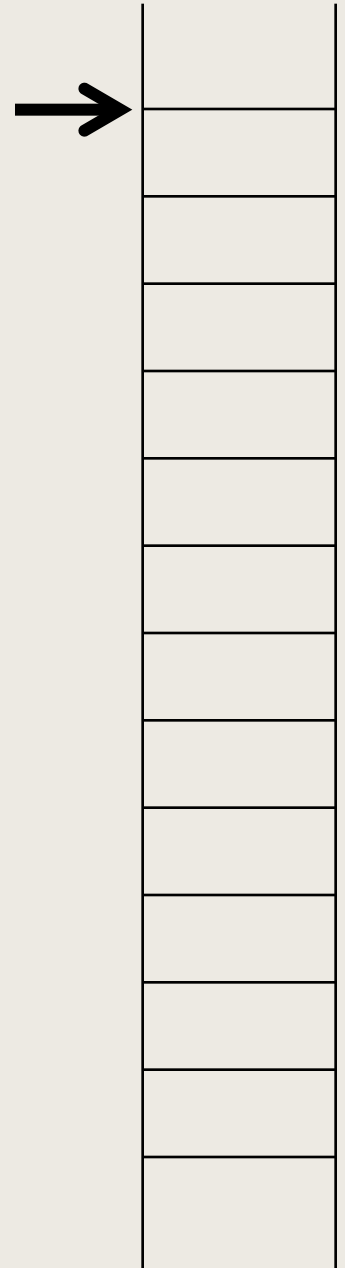
Stack - Example

```
int foo( int a, double f )  
{  
    int b;  
    ...  
    {  
        int c;  
        ...  
    }  
    ...  
}
```



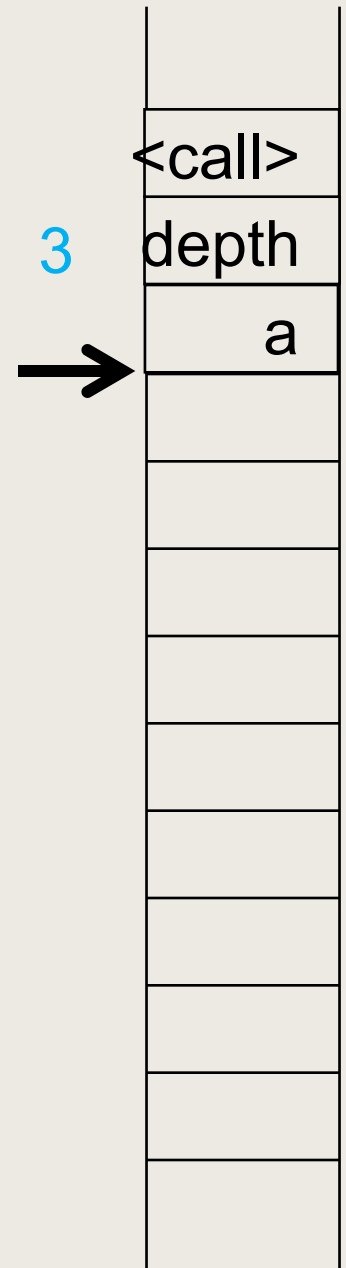
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



Stack – recursive example

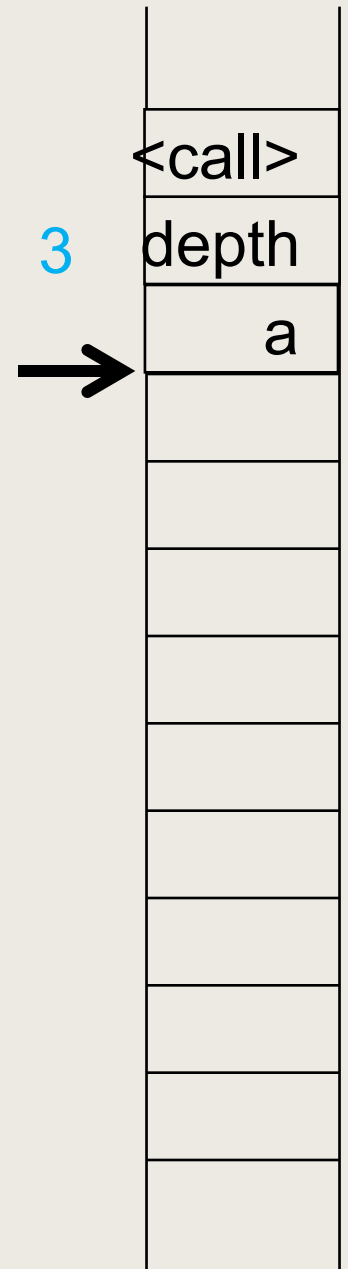
```
void foo( int depth )  
{  
    → int a;  
    if( depth > 1 )  
    {  
        foo( depth-1 );  
    }  
}  
int main()  
{  
    foo(3);  
    ...  
}
```



Stack – recursive example

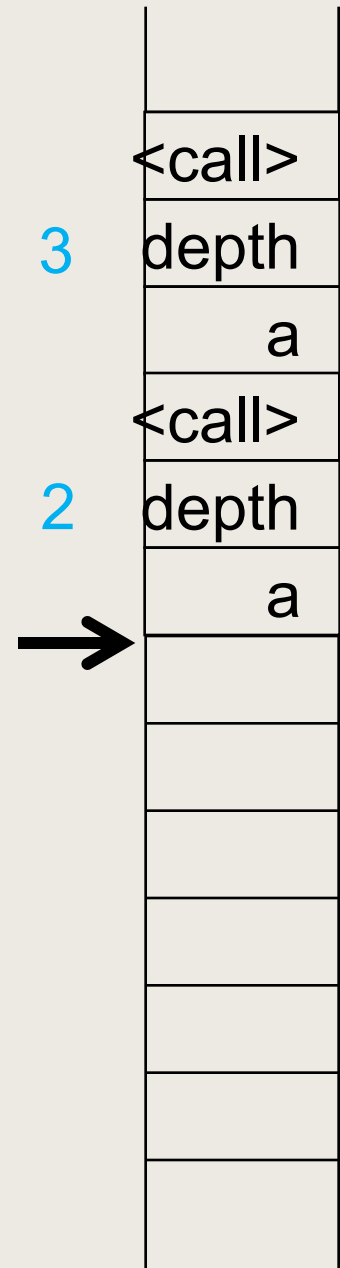
```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}

int main()
{
    foo(3);
    ...
}
```



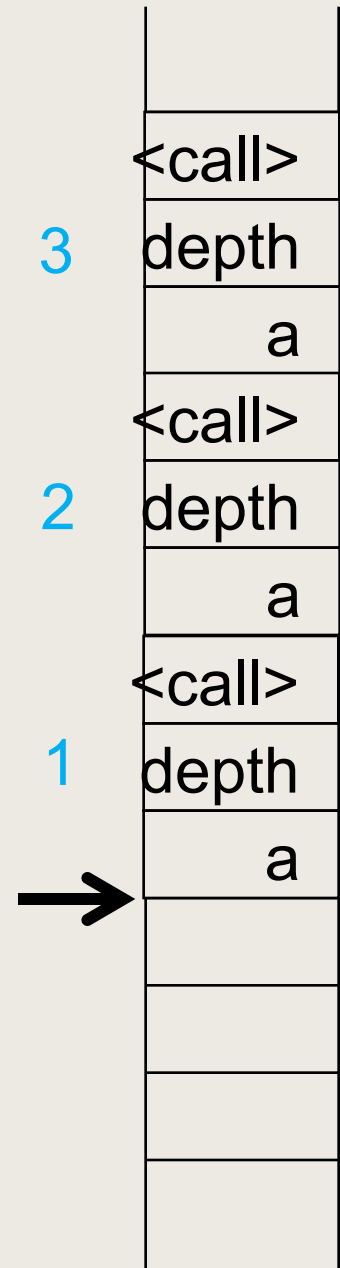
Stack – recursive example

```
void foo( int depth )
{
    → int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



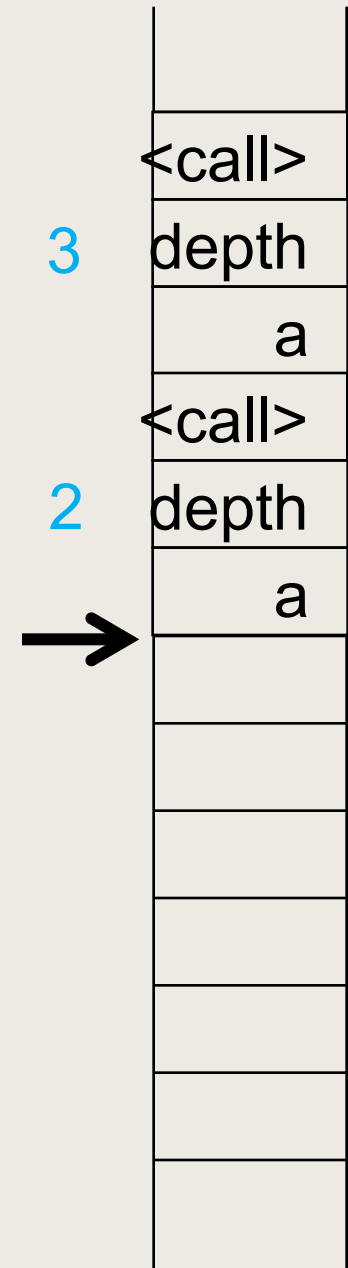
Stack – recursive example

```
void foo( int depth )
{
    → int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



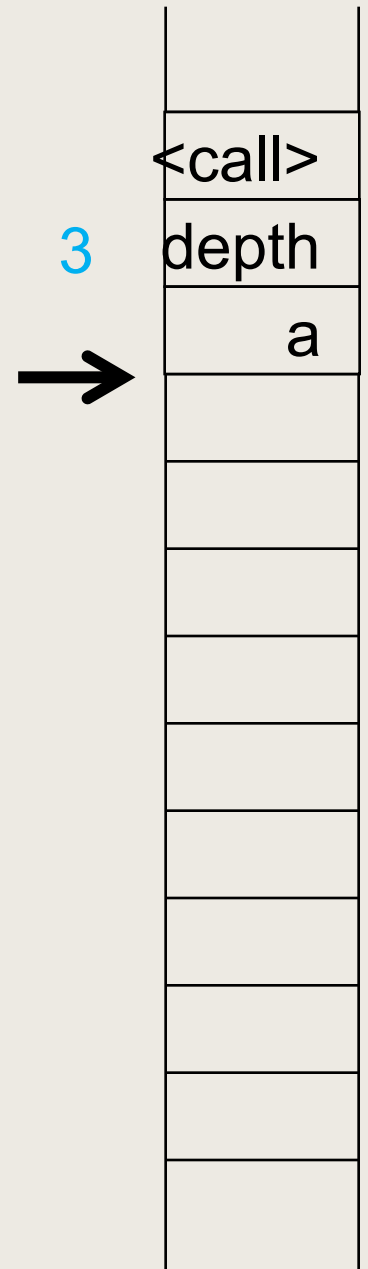
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
    → }
int main()
{
    foo(3);
    ...
}
```



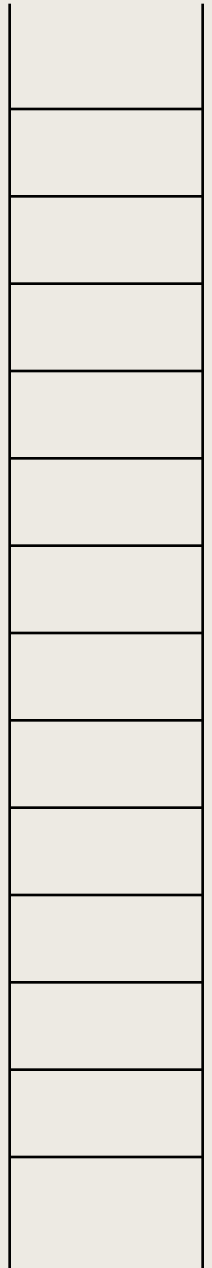
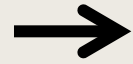
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



Stack – recursive example

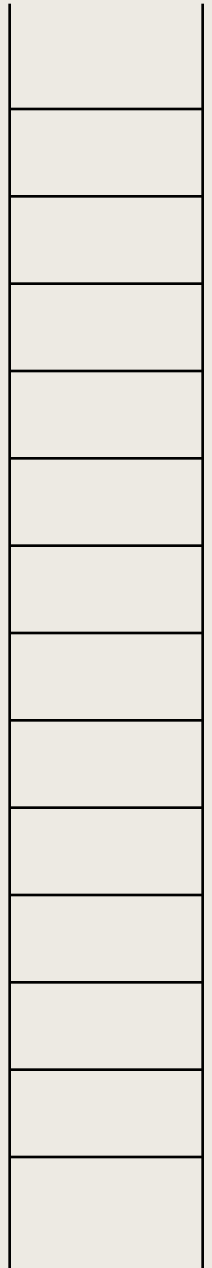
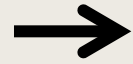
```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}

int main()
{
    foo(3);
    ...
}
```



Stack -- errors



```
void foo( int depth )  
{  
    int a;  
    if( depth > 1 )  
    {  
        foo( depth );  
    }  
}
```

What is the error?

Stack -- errors

```
void foo( int depth )  
{  
    int a;  
    if( depth > 1 )  
    {  
        foo( depth );  
    }  
}
```

Will result in run time error,
out of stack space

Why do we need dynamic allocation?

Global / Local variables size –
must be defined in compile time (except VLA)

```
#define LIST_OF_NUMBER_SIZE 1000
int staticArray[LIST_OF_NUMBER_SIZE];
int main()
{
    int i = some_result(); // also, int i = 3;
    int VLA[i]; // illegal in C89, ok in C99
    int someArray[10*LIST_OF_NUMBER_SIZE];

    ...
}
```

Why do we need dynamic allocation?

Example: program to reverse the order of lines of a file

To this task, we need to

- Read the lines into memory
- Print lines in reverse

How do we store the lines in memory?

Global area: reverse example

```
#define LINE_LENGTH 100
#define NUMBER_OF_LINES 10000
char g_lines[NUMBER_OF_LINES][LINE_LENGTH];

...

int main()
{
    int n = readLines();
    for( n-- ; n >= 0; n-- )
        printf("%s\n", g_lines[n]);
}
```

Compile time size limit is problematic

- ❑ The program cannot handle files **larger** than these specified by the **compile time choices**
- ❑ If we set `NUMBER_OF_LINES` to be very large, then the program requires this amount of memory even if we are reversing a short file

➔ Want to **use memory on “as needed” basis**

Dynamic Heap

1. Memory that can be allocated and freed during run time
2. The programmer controls how much is allocated and when
3. Can adjust to limitations based on run-time situation (available memory on the computer)

Allocating Memory from Heap

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

typedef of unsigned
integral type

```
void *malloc( size_t Size );
```

Returns a pointer to a new memory block of size **Size**, or NULL if it cannot allocate memory of this size.

Always check allocation success

```
char *str = (char*)malloc(5*sizeof(char));  
if (str == NULL)  
{  
    // print error message or perform  
    // other relevant operation  
    exit(1); // we don't have to exit  
}
```


How do we use it?

```
void *malloc( size_t Size );
```

```
int* iptr =  
    (int*)malloc(sizeof(int));
```

```
double *d_ptr =  
    (double*)malloc(n*sizeof(double));
```

De-allocating memory

```
void free( void *p );
```

Returns the memory block pointed by p to the pool of unused memory

No error checking!

- If p was not allocated by `malloc`, or if it was free-ed before, **undefined behavior**
- *free(NULL)* suppose to do nothing

Example

```
void *malloc( size_t Size );  
int* iptr =  
    (int*) malloc(sizeof(int));
```

...

```
free(iptr);  
iptr=NULL;
```

How much memory to free?

`free` gets `void*` -

how does it know how many bytes
should be freed at this address?

How much memory is de-allocated?

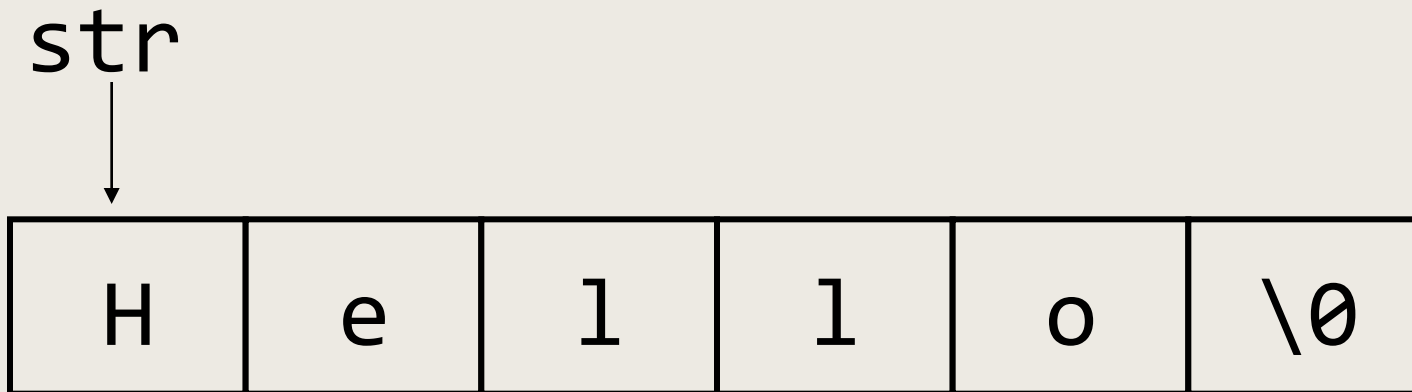
The system keeps track of how much memory was allocated, (e.g., by putting some information right before the allocated address) and thus knows how much memory to free

So when calling **free** – it only needs to get the same address returned by **malloc**.

C strings - revisited

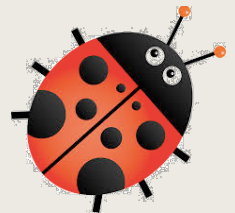
```
const char *str = "hello";
```

str is the address where the string begins – it can be considered as a read-only global variable



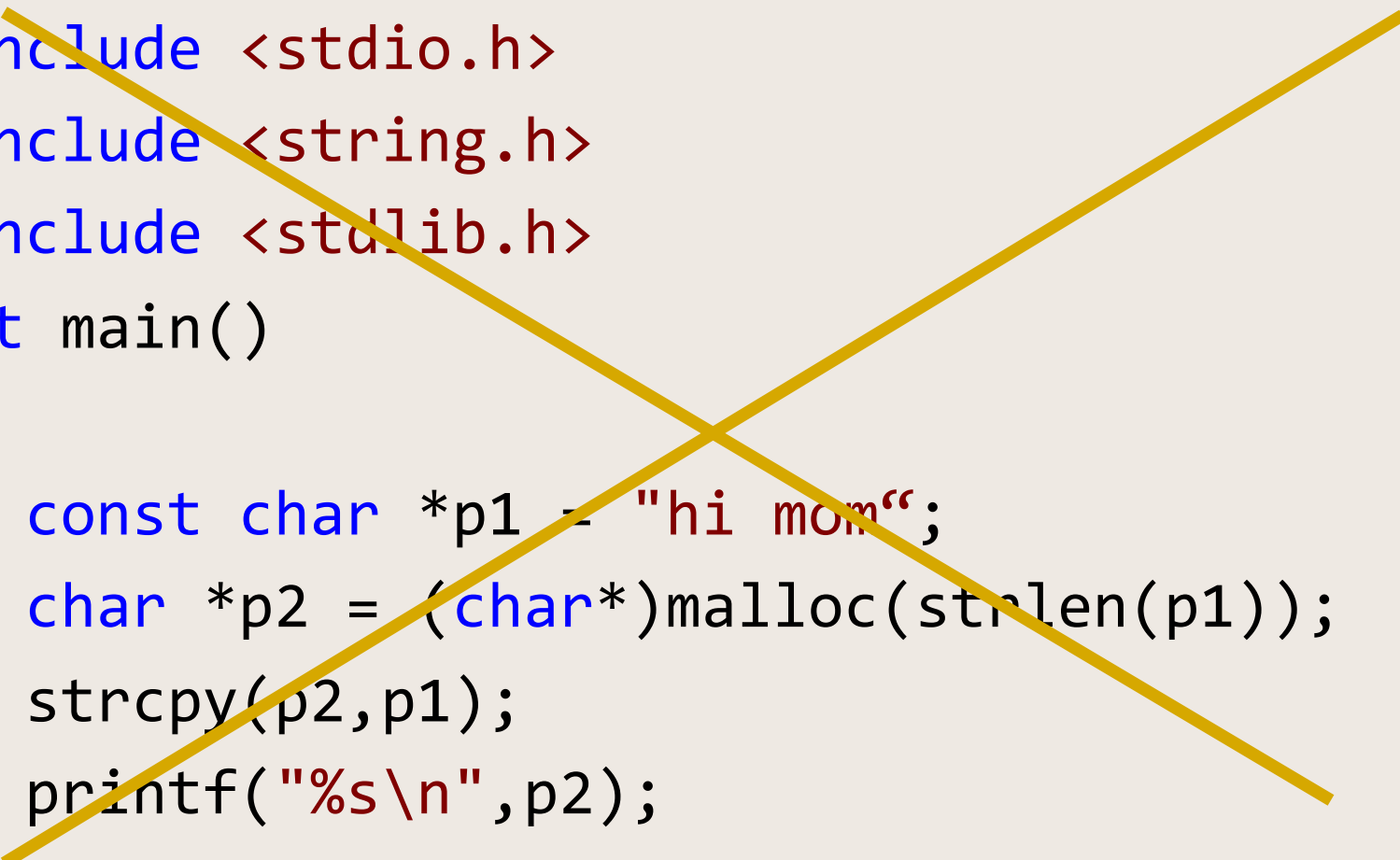
How to copy a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    const char *p1 = "hi mom";
    char *p2 = (char*)malloc(strlen(p1));
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```




How to copy a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    const char *p1 = "hi mom";
    char *p2 = (char*)malloc(strlen(p1));
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```



How to copy a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    const char *p1 = "hi mom";
    char *p2 = (char*)malloc(strlen(p1) + 1);
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```



Example: strdup

```
#include <string.h>
char *strdup (const char *s)
{
    // Reserve space for length plus \0 char
    char *d = (char *)malloc(strlen(s) + 1);
    if (d == NULL)
        return NULL; // No memory
    strcpy(d,s); // Copy the characters
    return d; // Return the new string
}
```

Memory leaks



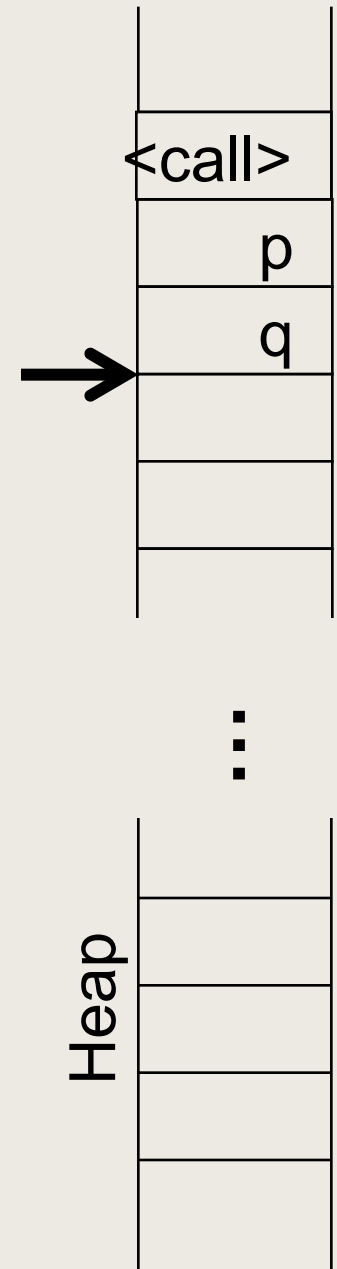
```
void foo( char const* p )
```

```
{
```

```
    char *q = strdup( p );
```

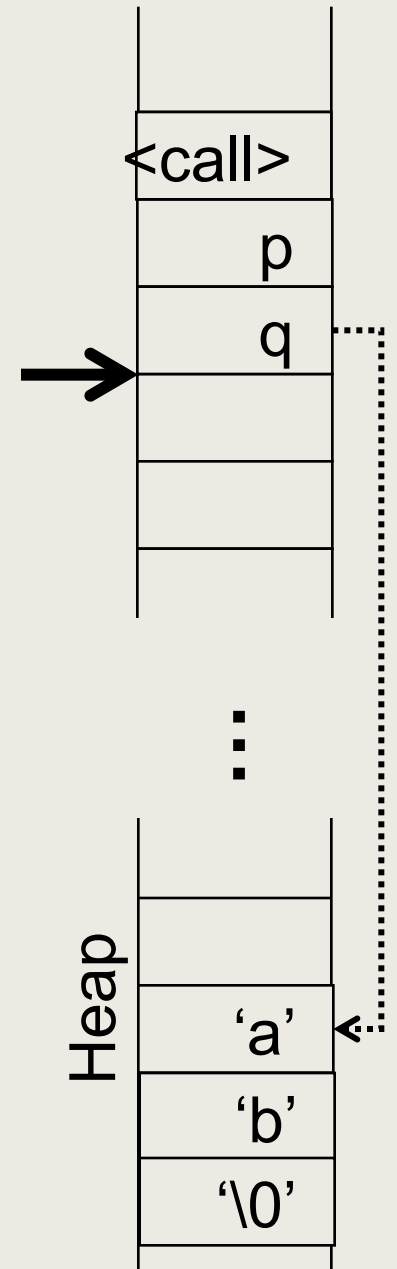
```
    // do something with q
```

```
}
```



Memory leaks

```
void foo( char const* p )  
{  
    char *q = strdup( p );  
    // do something with q  
}
```

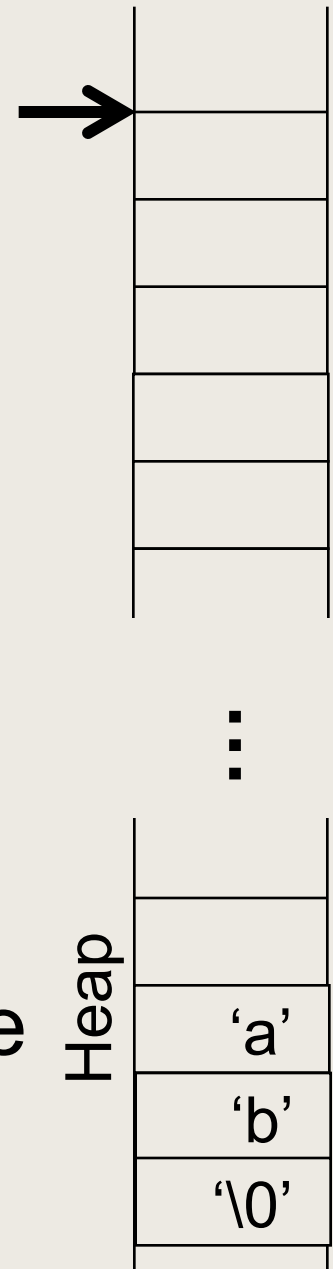


Memory leaks

```
void foo( char const* p )  
{  
    char *q = strdup( p );  
    // do something with q  
}
```



The allocated memory remains in use
and cannot be reused later on!



Further Knowledge - check man pages

`void* malloc(size_t n)`

- `malloc()` allocates blocks of memory
- returns a pointer to uninitialized block of memory on success (NULL on failure)
- the returned value should be cast to appropriate type:

```
int *p = (int*) malloc (sizeof(int)*length);
```

`void* calloc(size_t n, size_t n)`

- `calloc()` allocates an array of n elements each of 'size' bytes
- initializes memory to 0

```
int *p = (int*) calloc (length, sizeof(int));
```

`realloc` - attempts to resize the memory block pointed to by **ptr** that was previously allocated with a call to **malloc** or **calloc**.

`free` - deallocate the memory previously allocated by `malloc`, `calloc`, or `realloc`