

Memory Management

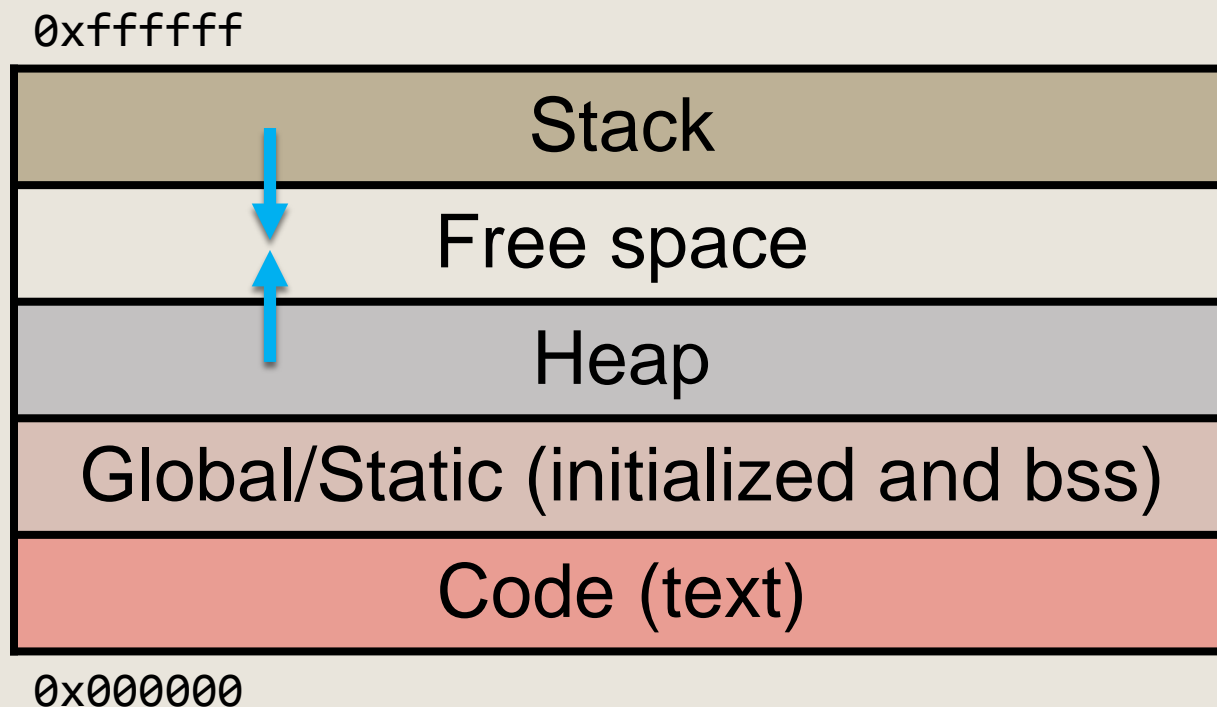
Finally!

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:

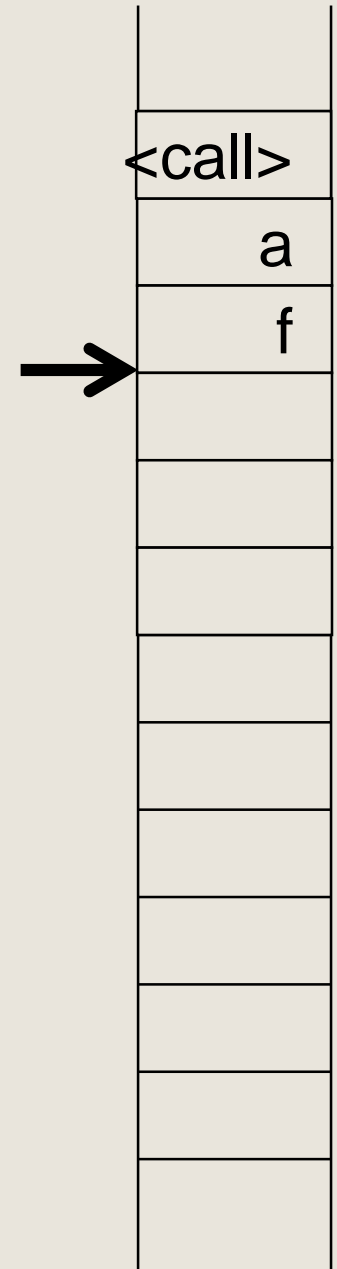
- Arguments of the function
- Local variables
- Call Frame

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

Stack - Example

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

```
→ {  
    int b;  
    ...  
}
```



Stack - Example

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

```
{
```

```
    int b;
```

```
    ...
```

```
    {
```

```
        int c;
```

```
        ...
```

```
    }
```

```
    ...
```

```
}
```

<call>

a

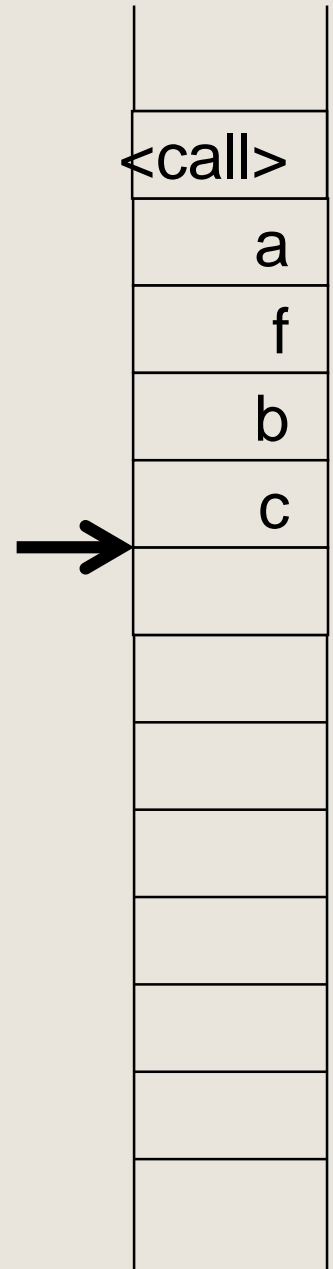
f

b



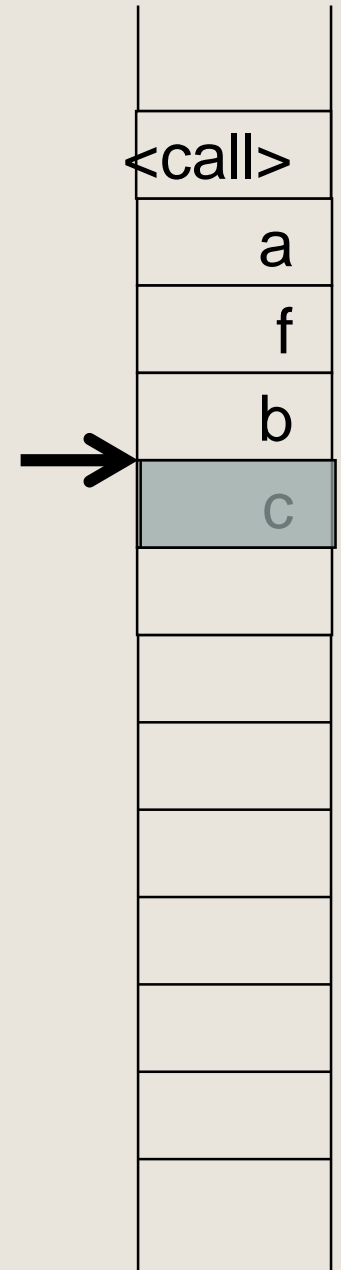

Stack - Example

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



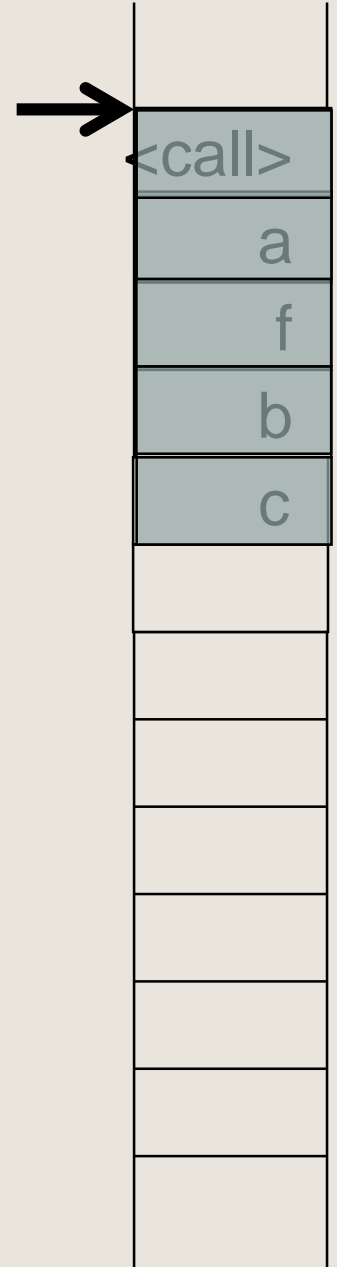
Stack - Example

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



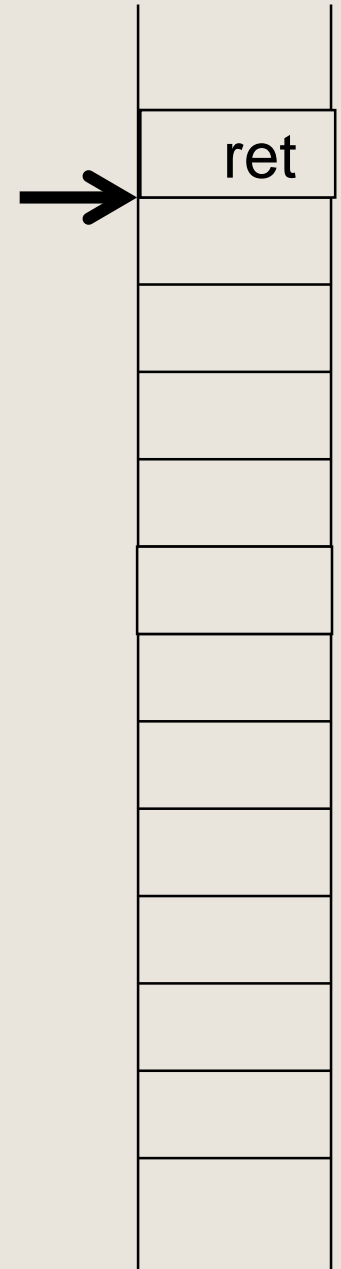
Stack - Example

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



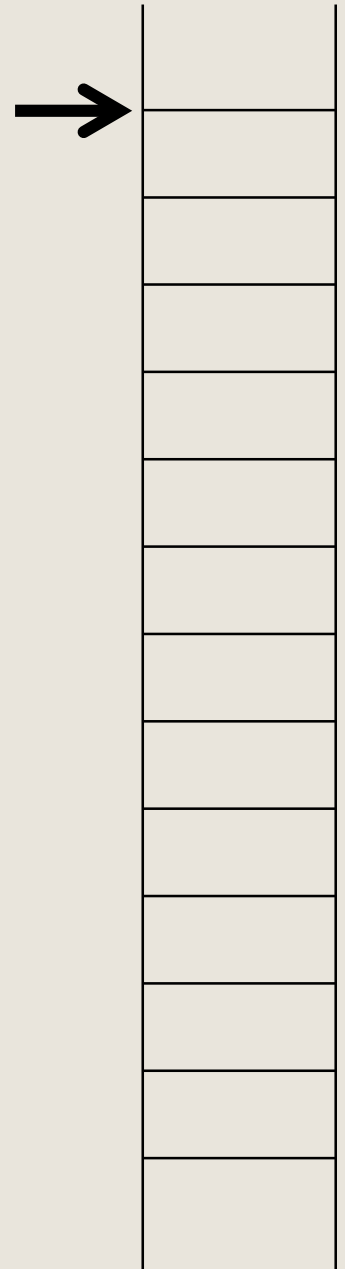
Stack - Example

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



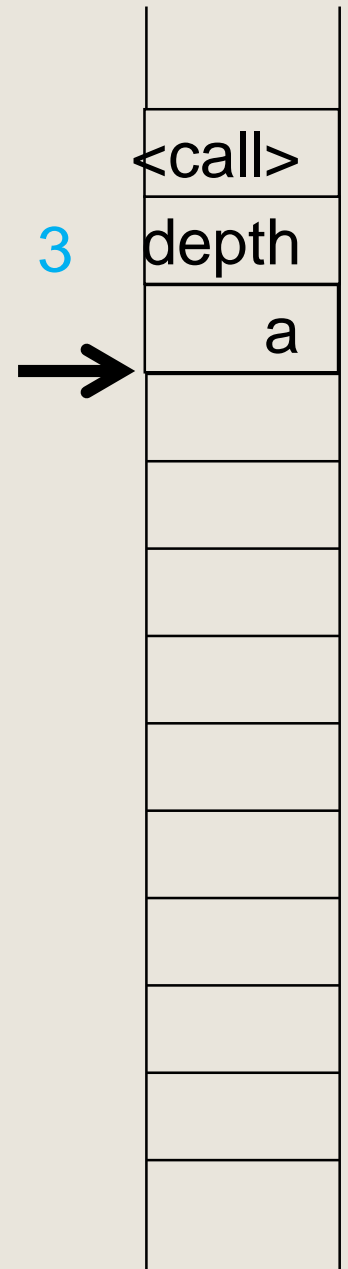
Stack – recursive example

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



Stack – recursive example

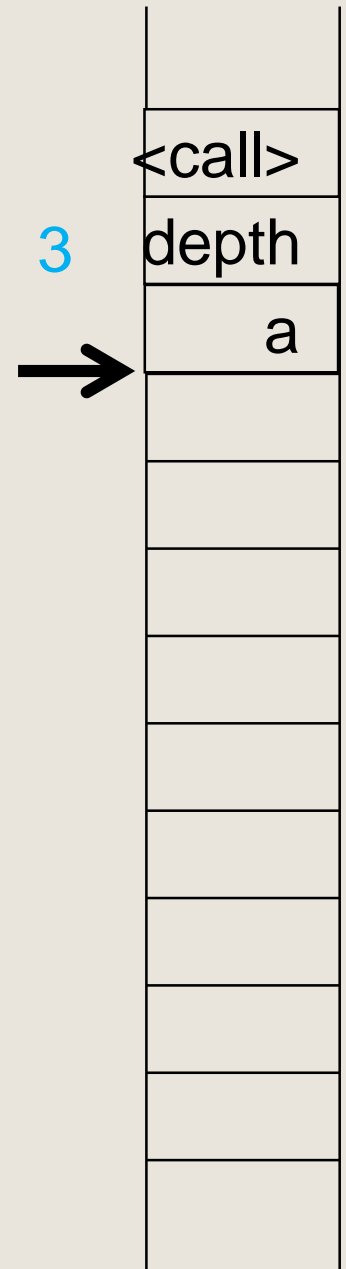
```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    ...
}
```



Stack – recursive example

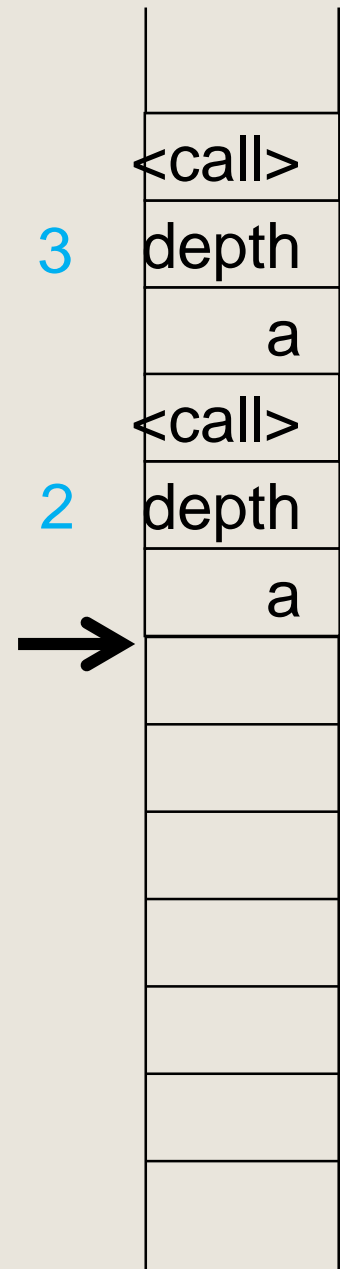
```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}

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



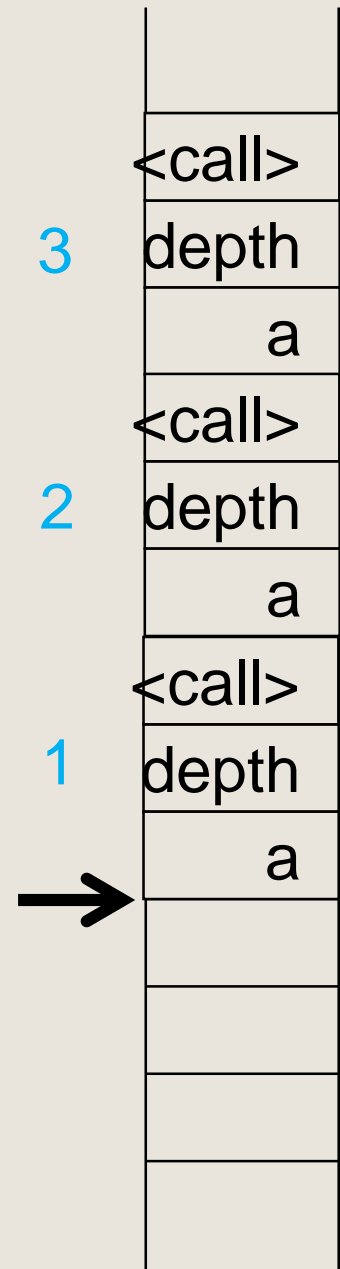
Stack – recursive example

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



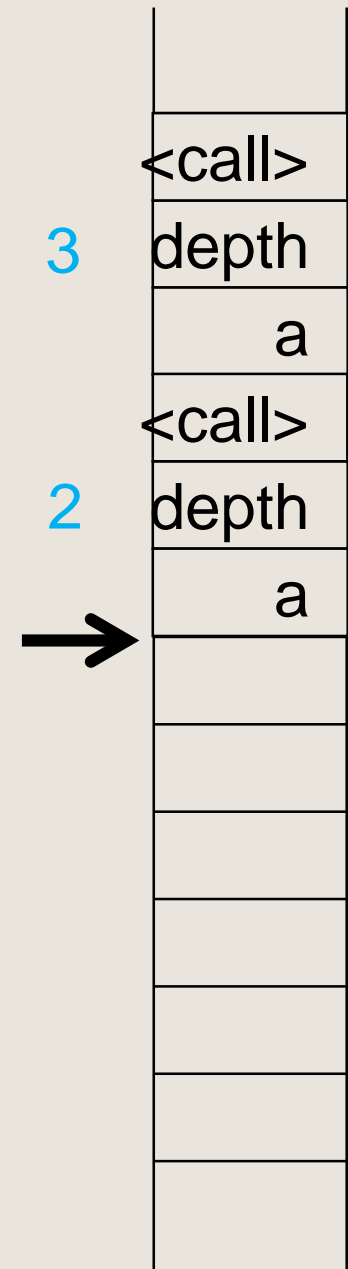
Stack – recursive example

```
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);
    ...
}
```



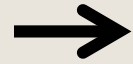
Stack – recursive example

```
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 );  
    }  
}
```

Will result in run time error,
out of stack space.

Global/Static area

```
int globalVariable = 1;  
int main()  
{  
    int localVariable = 3;  
    static int staticVariable = 4;  
    ...  
}
```

Why do we need dynamic allocation?

Global / Local variables size –
must be defined in compile time

(except VLA that will be discussed later in the course)

```
#define LIST_OF_NUMBER_SIZE = 1000;
int staticArray[LIST_OF_NUMBER_SIZE];
int main()
{
    int i = 3;
    int VLA[i]; //illegal in C89
    ...
}
```

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]);  
}
```


Why do we need dynamic allocation?

This solution 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. Limitations based on run-time situation - available memory on the computer

Allocating Memory from Heap

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

typedef of unsigned
integral type
(minimum 2 bytes)

```
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.

How do we use it?

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

```
struct Complex* complex_ptr =  
    (struct Complex*)  
    malloc(sizeof(struct Complex));
```

Always check allocation success

```
char *str = (char *)malloc(5*sizeof(char));  
if (str==NULL)  
{  
    //print error message or perform relevant  
    operation  
    exit(1);  
}
```

initialization function

```
int* iptr = (int*) malloc(sizeof(int));  
struct Complex* complex_ptr =  
    (struct Complex*)malloc(sizeof(struct Complex));
```

*iptr = not initialized

*complex_ptr = not initialized

Good design – an initialization function for a struct (poor constructor):

```
struct Complex *complex_ptr =  
    newComplex (1.0, 2.1);
```

initialization function

```
struct Complex
    *newComplex(double r, double i)
{
    struct Complex *p =
        (struct Complex*)
        malloc (sizeof (Complex));
    p->_real = r;
    p->_imag = c;
    return p;
}
```

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 free-ed before, undefined behavior
- *free(NULL)* supposed to do nothing

Example

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

...

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

And free the memory allocated to the struct?

```
void foo( double r, double i )  
{  
    struct Complex* p_c =  
        newComplex (r,i);  
    // do something with p_c  
    free(p_c);  
}
```

This version frees all allocated memory (good)

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

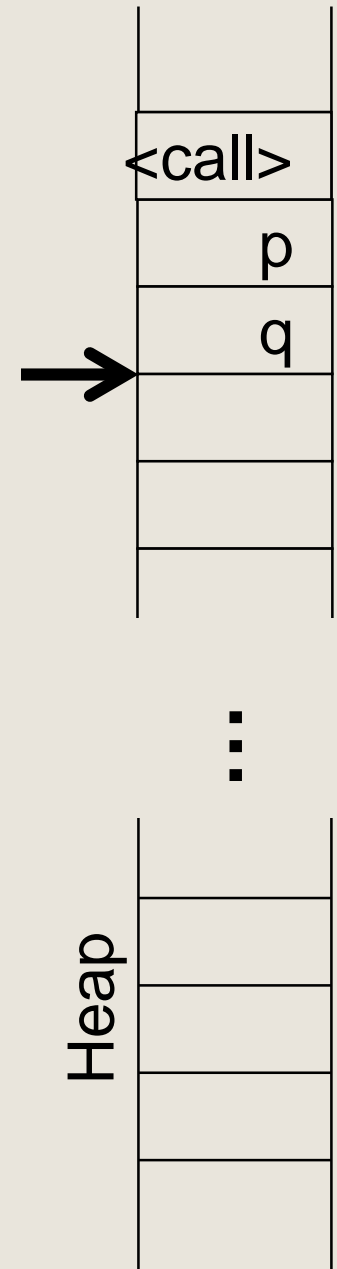
So when calling **free** – it only needs to get the same address returned by **malloc**.

Memory leaks

```
#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
}
```

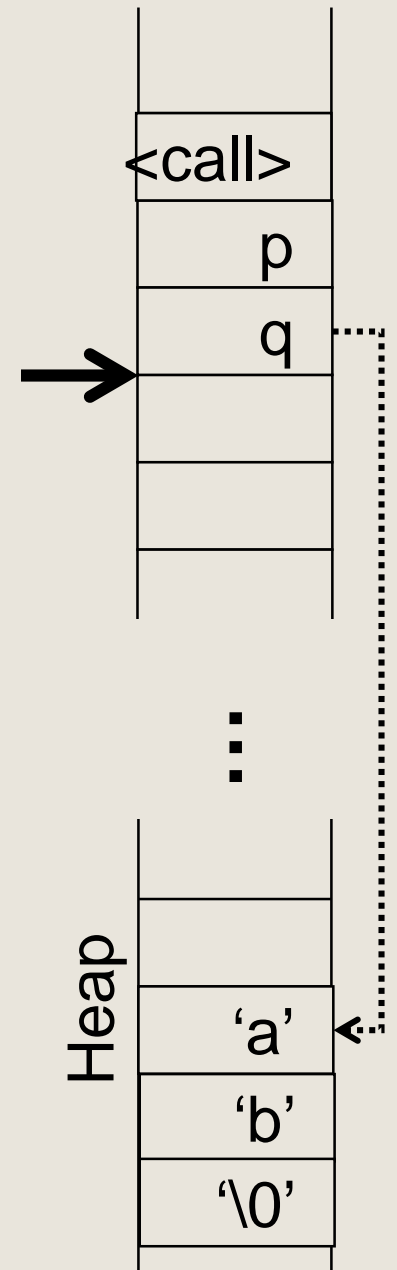

Memory leaks

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



Memory leaks

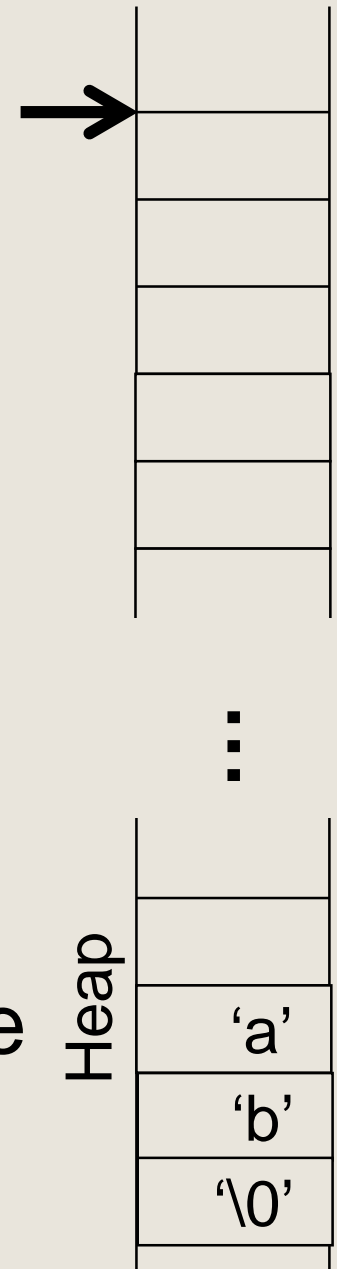
```
void foo( char const* p )  
{  
    char *q = strdup( p );  
    // do something with q  
}
```



Memory leaks

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

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



What about structs that allocate memory for internal fields?

```
struct Vec
{
    int *_arr;
};

struct Vec *newVec (int length)
{
    struct Vec *p = (struct Vec*) malloc (sizeof (struct Vec));
    if(p!=NULL)
        p->_arr = (int*) malloc (sizeof(int)*length);
    return p;
}

int main ()
{
    struct Vec *v = newVec(5);
    // do something
    free (v);
}
```


Memory leak – a bug!

```
struct Vec
{
    int *_arr;
};

struct Vec *newVec (int length)
{
    struct Vec *p = (struct Vec*) malloc (sizeof (struct Vec));
    if(p!=NULL)
        p->_arr = (int*) malloc (sizeof(int)*length);
    return p;
}

int main ()
{
    struct Vec *v = newVec(5);
    // do something
    free (v);
}
```

Poor destructors – better, but still bad

```
struct Vec
{
    int *_arr;
};
```

```
void freeVec (struct Vec *v)
{
    free (v->_arr);
    free (v);
}
```

```
int main ()
{
    struct Vec *v = newVec(5);
    // do something
    freeVec (v);
}
```

Poor destructors - good

```
struct Vec
{
    int *_arr;
},

void freeVec (struct Vec *v)
{
    if (v!=NULL)
    {
        free (v->_arr);
        free (v);
    }
}

int main ()
{
    struct Vec *v = newVec(5);
    // do something
    freeVec (v);
}
```

Further Knowledge

Read manual page of

malloc

calloc

realloc

free

Pointers to pointers & multi-dimensional arrays

Pointers to pointers

```
int i=3;
```

```
int j=4;
```

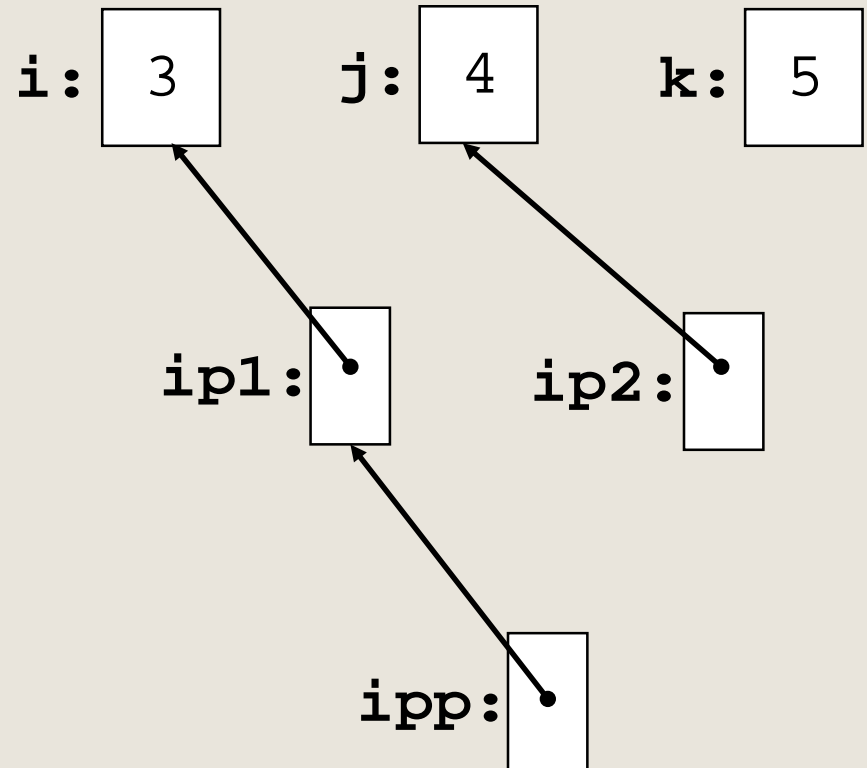
```
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
int **ipp = &ip1;
```

```
ipp = &ip2;
```



Pointers to pointers

```
int i=3;
```

```
int j=4;
```

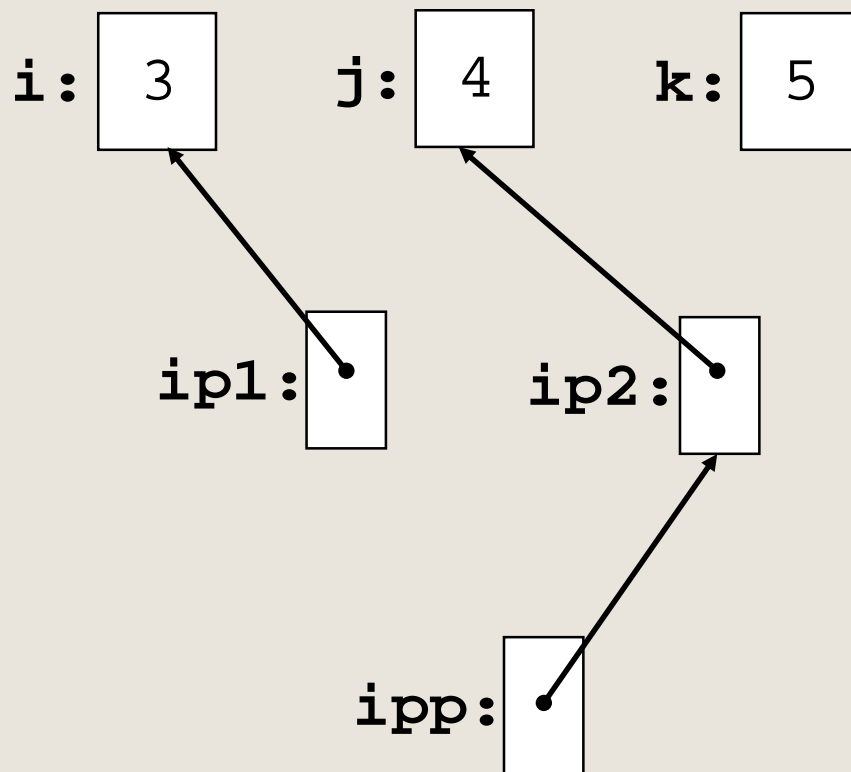
```
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
int **ipp = &ip1;
```

```
ipp = &ip2;
```



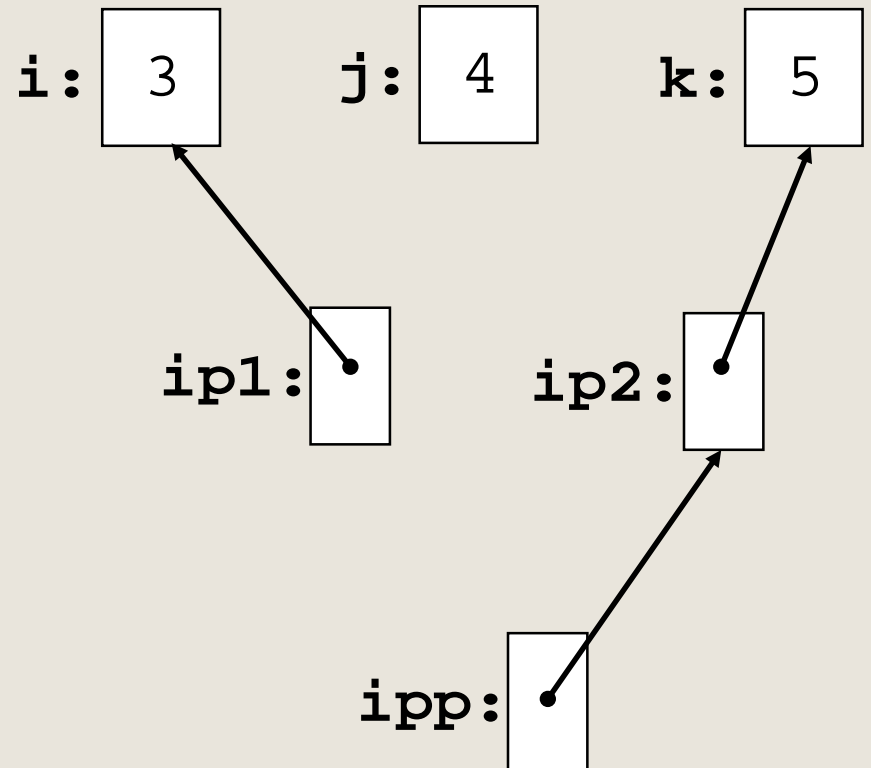
Pointers to pointers

```
int i=3;
```

```
int j=4;
```

```
int k=5;
```

```
*ip2 = &k;
```



Reminder – the swap function

Does nothing

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x, y;
    x = 3; y = 7;
    swap(x, y);
    // now x==3, y==7
}
```

Works

```
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}

int main()
{
    int x, y;
    x = 3; y = 7;
    swap(&x, &y);
    // x == 7, y == 3
}
```

pointers to pointers: example

```
// copy a string using "allocString"
void main()
{
    const char *s = "example";
    char *copy = NULL;
    allocString(strlen(s), &copy );
    strcpy(copy, s);
    free(copy);
}
```

pointers to pointers: example

//put pointer to an allocated string in pp

```
int allocString( int len, char ** pp)
{
    char *str = (char*)malloc(len + 1);
    if (str==NULL)
    {
        return 0;
    }
    *pp = str;
    return 1;
}
```

Multi-dimensional arrays

Static:

```
int arr[5][7]; // 5 rows, 7 columns
```

- Continuous memory: “array of arrays” (divided to 5 blocks of 7 ints)
- Size must be known at compile time.
- Efficient: one memory access to reach an index.

Semi-dynamic:

```
int *arr[5]; // array of 5 pointers to int
```

- Each row is in a different location.
- Number of rows must be known at compile time.
- Less efficient: two memory access to reach an index.

Fully dynamic:

```
int **arr; // pointer to pointer to int
```

- Each row is in a different location.
- Size may be unknown at compile-time.
- Even less efficient: three memory access to reach an index.

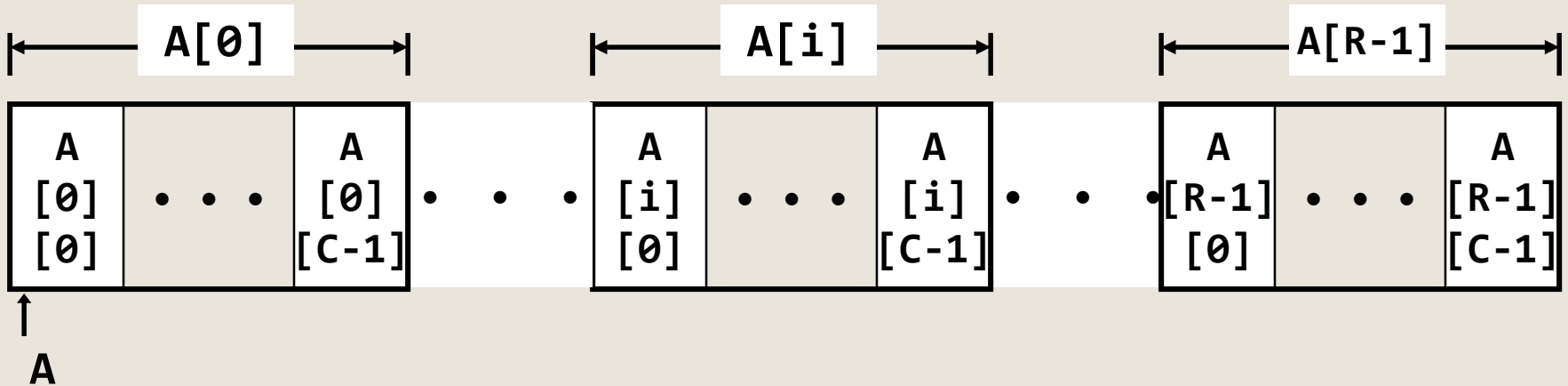
Static 2D array

```
int A[R][C];
```

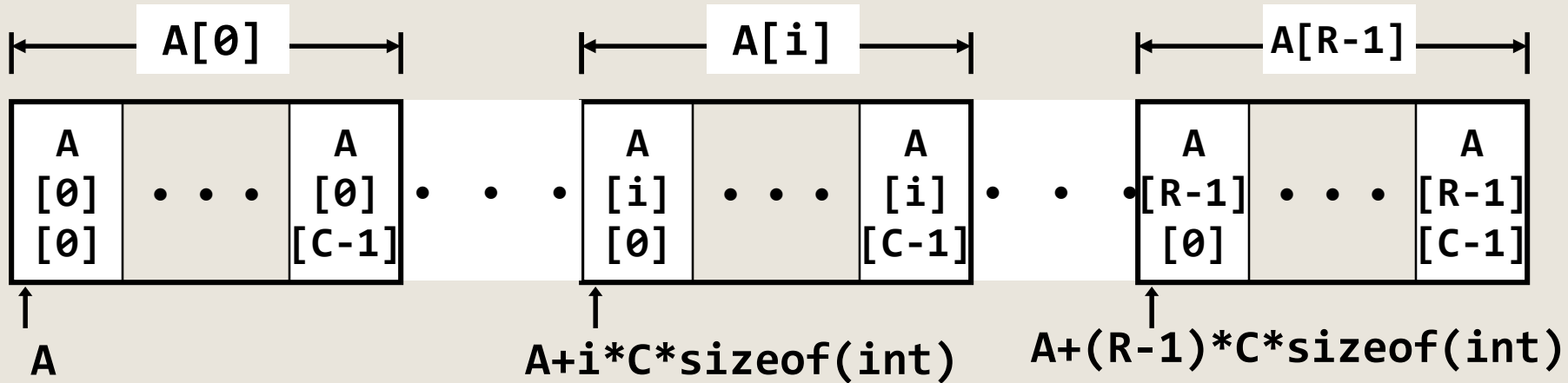
matrix representation

$A[0][0]$	$A[0][1]$...	$A[0][C-1]$
...
$A[i][0]$	$A[i][1]$...	$A[i][C-1]$
...
$A[R-1][0]$	$A[R-1][1]$...	$A[R-1][C-1]$

Row-major ordering →



Static 2D array



```
int A[R][C];
```

```
A[i][j]=5; // → put 5 in:  
           // A + (i*C + j)*sizeof(int)
```

```
// C is the size of each row
```

Semi-dynamic arrays – array of pointers

```
int *pa[5]; // allocates memory for 5 pointers
for (i=0; i<5; i++)
{
    pa[i] = (int*) malloc( 7*sizeof(int) );
    // pa[i] now points to a memory of 7 ints
    // note that we can allocated
    // different sizes for each row
}

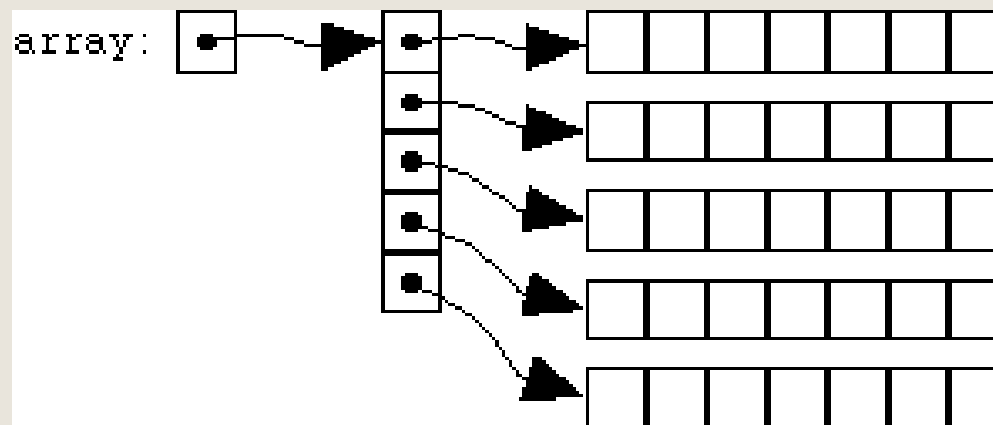
pa[i][j] = 5;
```

1. Go to $pa + i * \text{sizeof}(\text{int}^*)$ and take its value val – this is the i 'th row start address.
2. Put 5 in $val + j * \text{sizeof}(\text{int})$

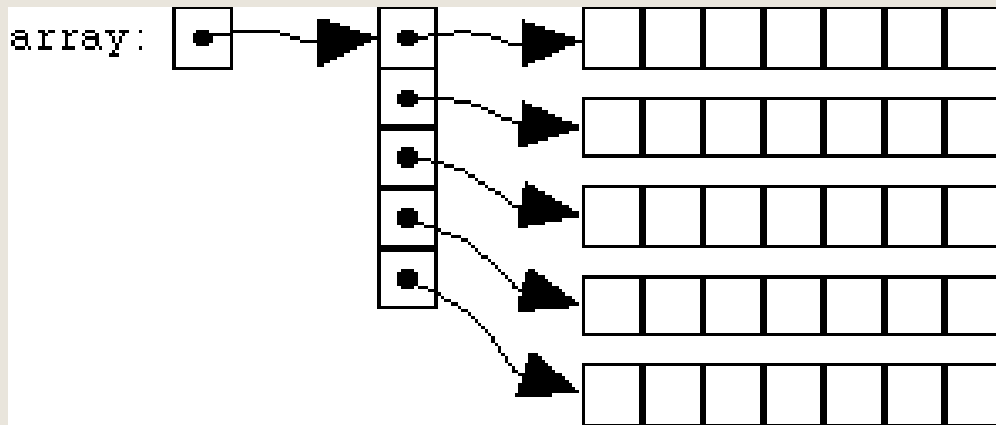
- Using pointer arithmetic : $*(pa+i)[j] = 5;$

Fully dynamically allocated arrays

```
int ** array;  
array = (int**)malloc(5*sizeof(int*));  
for (i=0; i<5; i++)  
{  
    array[i] = (int*)malloc(7*sizeof(int));  
}
```



Fully dynamically allocated arrays



```
array[i][j] = 5;
```

1. Go to array and take its value **v1**
2. Go to the address **v1 + i*sizeof(int*)** and take its value **v2**
3. Put 5 in **v2 + j*sizeof(int)**

With pointers arithmetic: **$*(*(array+i)+j) = 5$**

Dynamically Multi-dimensional arrays

Semi/Full dynamically allocated multi-dimensional array:

- Memory not continuous
- Each row can have different size
- **Access: `arr[i][j]`**

Dynamically Multi-dimensional arrays

- Don't forget to free all the memory – one `free` call for each one `malloc` call

e.g., for full dynamically allocated 2D array:

```
for (i=0; i<nrows; i++ )  
{  
    free( array[i] );  
    array[i] = NULL;  
}  
free( array );  
array = NULL;
```

More efficient memory arrangement

Instead of allocating `int **arr`,
allocate `int *arr`

```
int *arr =(int*)malloc(5*7*sizeof(int))
```

- **Access:** `arr[i][j] -> arr[i*ncols + j]`
 - 2 access to memory vs. 3 in 2-D (fully dynamic) representation.
 - Easier (and more efficient) implementation of iterators
- **But:**
 - Less readable code (can partially hide with macro)

pointers to pointers to ...

We also have pointers to pointers to pointers, etc.:

```
double ** mat1 = getMatrix();  
double ** mat2 = getMatrix();  
//allocate an array of matrices  
double *** matrices =  
(double ***) malloc(n*sizeof(double **));  
matrices[0] = mat1;  
matrices[1] = mat2;
```

Multi-dimensional arrays

Sending array to functions:

- `void func(int x[5][7]) //ok`
- `void func(int x[][7]) //ok`
- `void func(int (*x)[7]) //ok – we'll see next`
- `void func(int x[][]) //error`
- `void func(int * x[]) //something else`
- `void func(int ** x) //`

Pointers to arrays:

```
int foo (char arr_a[][20]);
```

```
int bar (char arr_b[20]);
```

arr_a is a pointer to an array of 20 chars

arr_b is a pointer to a char

Therefore:

```
sizeof (arr_a) = sizeof (void*);
```

```
sizeof (*arr_a) = 20*sizeof (char);
```

```
sizeof (arr_b) = sizeof (void*);
```

The "right-left" rule

1. Find the identifier
2. Look at the symbols on the right of the identifier
3. Look at the symbols to the left of the identifier
4. Goto step 2 until declaration is complete

The "right-left" rule

```
char (*arr)[5];
```

“arr is”

```
char *arr[5];
```

“arr is”

The "right-left" rule

```
char (*arr)[5];
```

“arr is”

“arr is pointer to”

```
char *arr[5];
```

“arr is”

“arr is array of”

The "right-left" rule

`char (*arr)[5];`

“arr is”

“arr is pointer to”

“arr is pointer to
array of”

`char *arr[5];`

“arr is”

“arr is array of”

“arr is array of
pointers to”

The "right-left" rule

`char (*arr)[5];`

“arr is”

“arr is pointer to”

“arr is pointer to
array of”

“arr is pointer to
array of char”

`char *arr[5];`

“arr is”

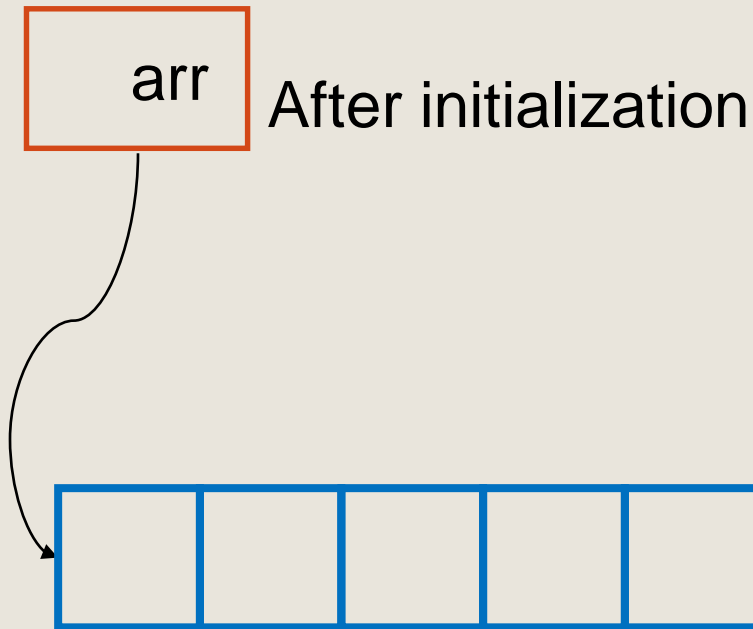
“arr is array of”

“arr is array of
pointers to”

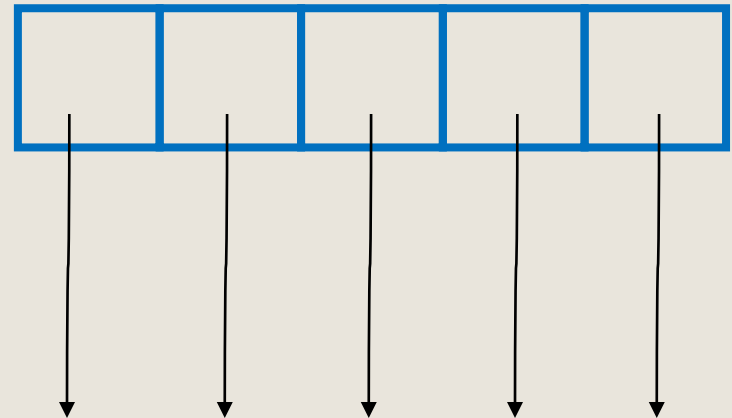
“arr is array of
pointers to char”

Pointers to arrays and arrays of pointers:

```
char (*arr)[5];
```



```
char *arr[5];
```



Pointers to arrays and arrays of pointers:

```
char (*arr)[5];
```

```
sizeof (arr) =  
    sizeof(void*)  
  
sizeof (*arr) =  
    5*sizeof(char)
```

```
char *arr[5];
```

```
sizeof (arr) =  
5*sizeof (char*) =  
5*sizeof (void*)  
  
sizeof (*arr) =  
    sizeof (char*)=  
    sizeof (void*)
```

Pointers to arrays:

Explicit declaration:

```
char (*arr_2d)[20];  
        // arr_2d is a pointer  
        // not initialized
```

Using typedef:

```
typedef char arr_2d_20[20];  
arr_2d_20 *p_arr_2d;
```

const

C's "const"

C's "const" is a qualifier that can be applied to the declaration of any variable to specify its value will not be changed.

C's "const"

Examples:

```
const double E = 2.71828;  
E = 3.14; // compile error!
```

```
const int arr[] = {1,2};  
arr[0] = 1; // compile error!
```

Helps to find errors during compilation time.

C's "const"

C's "const" can be **cast** away

```
const int arr[] = {1,2};  
int* arr_ptr = (int*)arr;  
  
arr_ptr[0] = 3;      // compile ok!  
                     // but might give a run-time error
```

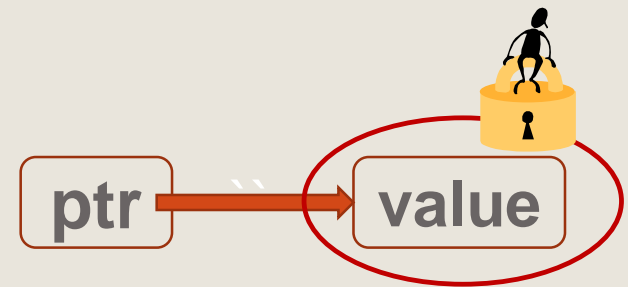
Don't protects from evil changes.

What the “const” declaration “protects”?

const **protects his left side**, unless there is nothing to his left, and only then it protects his right side.

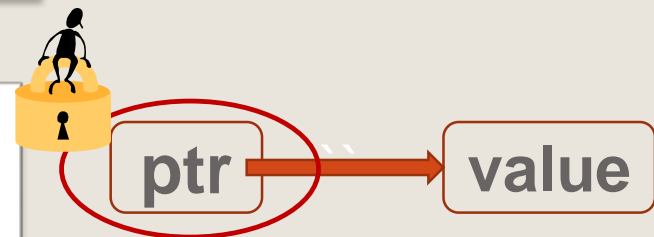
A pointer to a **const variable**:

```
int arr[] = {1,2,3};  
int const * p = arr;  
p[1] = 1;           // illegal!  
*(p+1) = 1;         // illegal!  
p = NULL;           // legal
```



A **const pointer** to a variable:

```
int arr[] = {1,2,3};  
int* const const_p = arr;  
const_p[1] = 0;      // legal!  
const_p = NULL;      // illegal!
```



C's “const”

How about *arr* itself?

```
int arr[] = {1,2,3};  
int* const const_p = arr;  
const_p[1] = 0;      // legal!  
const_p = NULL;      // illegal!  
  
int *p;  
arr = p;              // compile error!
```

Const and Pointer's Syntax

(2) and (3) are synonyms in C to a *pointer to a const int*

```
(1) int * const p = arr;  
(2) const int * p = arr;  
(3) int const * p = arr;
```

Const and User Defined Types

```
typedef struct Complex
{
    int _img;
    int _real;
} Complex;
```

```
Complex const COMP1 = comp2;    // ok, init. using comp2
Complex const COMP3 = {2,3};    // ok, copying values

COMP1._img = 3;                  // illegal!
COMP3._real = 4;                 // illegal!
```

All the members of a const variable are immutable!

Compare to Java's "final"

All (methods as well as data) are Class members.

```
final int LIMIT = 10;
LIMIT = 11;                                // illegal!

final MyObject obj1 = new MyObject();
MyObject obj2 = null;
MyObject obj3 = new MyObject();
obj2 = obj1; // fine, both point now to the same object
obj1 = obj3;                                // illegal!
obj1.setSomeValue(5);                       // legal!
```

"final" makes primitive types constants and references to objects constant.

The values inside the referred objects are not constant!

Because All are class members you would normally use them as class constants and declare them as "static final"

“Const” Usage

The const declaration can (and should!) be used in the definition of a function’s arguments, to indicate it would not change them:

```
int strlen(const char []);
```

Why use? (This is not a recommendation but a **must**)

- clearer code
- avoids errors
- part of the interfaces you define!
- can be used with const objects

We will see more of “const” meaning and usage when we get to C++

Strings, again

C's string literals ("")

When working with `char*`, C's string literals ("") are written in the code part of the memory. Thus, you can't change them!

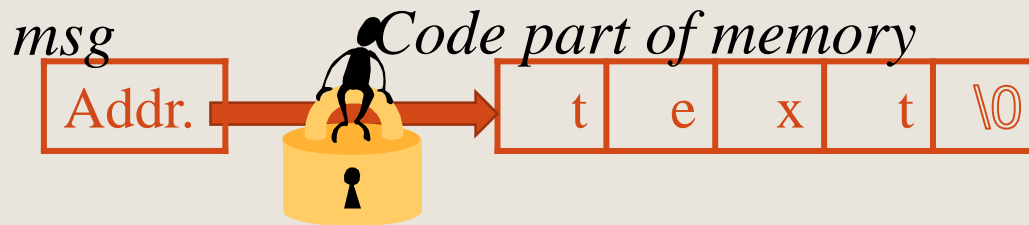
```
char* msg = "text";  
msg[0] = 'w';           // seg fault!
```



C's string literals ("") with const

So, what we do is:

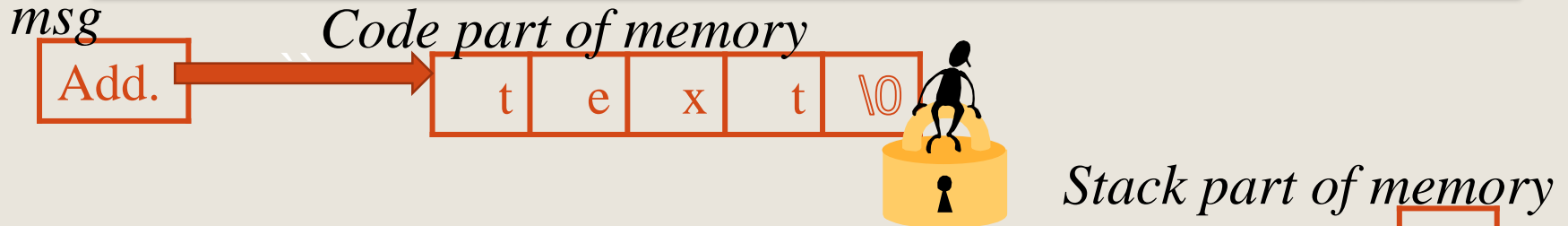
```
char const * msg = "text";  
msg[0] = 't';           // compile error!  
                        // better!
```



C's string literals ("")

Note the difference:

```
char* msg = "text";  
// msg is a pointer that points to a memory  
// that is in the code part
```



```
char msg2[] = "text";  
// msg2 is an array of chars that  
// are on the stack
```



C's string literals ("")

Now we understand why:

```
char* msg = "text";  
char msg2[] = "text";
```

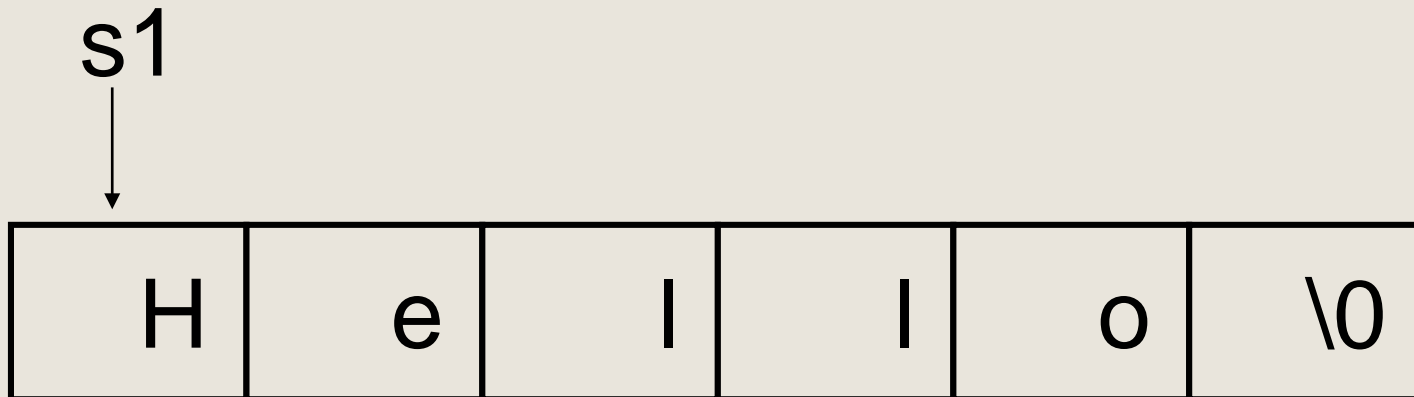
```
msg[0]= 'n';           // seg fault - trying to change
                        // what is written in the code
                        // part of the memory
```

```
msg2[0]= 'n';    // ok - changing what is written
                  in the stack part of the memory
```

C strings - revisited

```
const char *s1 = "hello";
```

s1 is the address of the code segment with the string
– it can be consider 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",
               *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",
               *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",
                *p2 = (char*)malloc(strlen(p1)+1);
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```

Example: strdup

Duplicate a string (part of the standard library):

```
char* strdup( char const *p )
{
    int n = strlen(p);
    char* q =(char*)malloc(sizeof(char)*(n+1));
    if( q != NULL )
    {
        strcpy( q, p );
    }
    return q;
}
```

Bitwise operators

Bitwise operators

- ❖ C allows to perform operations on the bit representation of a variable
- ❖ Done with “bitwise operators”

Operator symbol	Operation
&	and
	or
^	xor
~	One complement (not)
<<	Left shift
>>	Right shift

Bitwise operators examples

- *and* operation:

14 & 11 = 10 // decimal representation

1110 & 1011 = 1010 // binary representation

- *shift left* operation:

[variable] << [number of places]

001101 << 1 == 011010, 001101 << 2 == 110100

This operation actually computes: $var * 2^{(\text{number of places})}$

e.g.: $14 << 2 == 14 * (2^2) == 14 * 4 == 56$

Why?

What does *shift right* computes?

Bitwise operations to manipulate single bits

- Each char can represent 8 independent boolean variables
- For this we need to find a way to
 - Find the value of a single bit
 - Change the value of a single bit

Finding the value of a single bit

We can do that with the << and & operators:

```
int isZero (int variable, int position)
{
    return (variable & (1 << position)) == 0;
}
```

Example:

- We have a variable with the bit-pattern: **11001**
- We want to test the value of the 3rd bit (from the least-significant bit, starting from place 0, this is place 2):

Construct a mask: $0001 \ll 2 == 00100$

Apply: $11001 \& 00100 == 00000 == 0$

Toggling a single bit

Use the << and ^ operators:

```
int toggleBit (int variable, int position)
{
    return variable ^ (1 << position);
}
```

Example:

- We have a variable with the bit-pattern: **11001**
- We want to change the value of the 3rd bit (from the least-significant bit, starting from place 0, this is place 2):

Construct a mask: $0001 \ll 2 == 00100$

Apply: $11001 \wedge 00100 == 11101$

Using masks

flag1 = 00000001

flag2 = 00000010

flag3 = 00000100

mask = flag1 | flag2 | flag3

mask == 00000111

Using masks – low level file open

```
int open(const char *path, int oflag, ... );
```

- oflags is an options mask – how to open the file
- You can use several options together using the bitwise OR operator
- Read more at: <http://linux.die.net/man/3/open>

```
E.g.: fd = open(filename,  
                O_WRONLY | O_CREAT | O_TRUNC)
```

Open for
write only

Create if not
already exist

Clear file if
already exist

Using masks – hexadecimal notation

```
int x = 0x123; //  $3 \cdot 16^0 + 2 \cdot 16^1 + 1 \cdot 16^2 == 291$ 
```

```
int y = 0xFA; //  $10 \cdot 16^0 + 15 \cdot 16^1 == 250$ 
```

```
int mask = 0xFFFF0000;
```

```
//get val of the 16 most significant bits of x
```

```
y = (x & mask) >> 8;
```

```
//set to zero the 16 most significant bits of x
```

```
x &= ~mask;
```