ESD Electronic Systems Design
Synthesis Verification Testing Power Communication

C Language Revise

EDALab Networked Embedded Systems

O.S. Laboratory

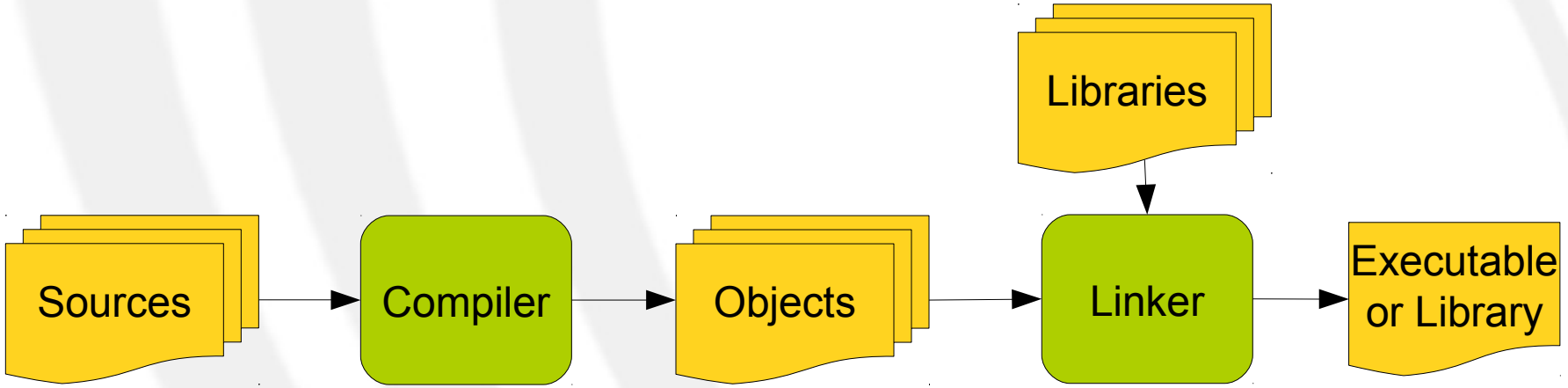
Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

C Programming Language

- C is a language:
  - Imperative.
  - Structured.
  - Weakly typed.
  - Minimalist.
  - With explicit memory management.
- C standard library:
  - Many functionalities.
  - No data structures.

C Programming Language



- GCC flags:
  - `I<includePath>`
  - `Wall`: all warnings
  - `c`: compile w/o linking
- LD/GCC flags:
  - `L<libPath>`
  - `l<lib>`

C Programming Language

```
/* File: manipulation.h */

/* Inclusion guard. */
#ifndef MANIPULATION_H
#define MANIPULATION_H

/* All the methods, e.g.:*/
int do_job( const int i );

#endif
```

```
/* File: manipulation.c */

/* Inclusion: */
#include "manipulation.h"

/* Implementation of all the
 * methods, e.g.: */
int do_job( const int i ){
    return ++i;
}
```

```
/* File: main.c */

#include "manipulation.h"

int main() {
   return do_job( 45 );
}
```

C Programming Language

- The first question:
  - Which C standard?
    - ***C89 (ANSI)***
    - C90 (ISO/IEC)
    - ***C99 (ISO/IEC)***
    - C11 (ISO/IEC)

C Programming Language

- Pros & cons:
  - ANSI: portable.
  - C99: some useful features.
    - Type modifiers (inline, restrict).
    - New types (long long).
    - Intermingled declarations and code.
    - One line comments (//).
    - Designated initializers.
    - Variadic macros.

Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

Input / Output

- Header stdio.h
  - int printf(const char * format, …)
    - Prints a formatted string, with optional parameters.
    - Returns the number of written characters, or a negative integer on error.
  - int scanf(const char * format, …)
    - Reads a formatted string, requiring pointers to optional arguments.
    - Returns the number of read arguments.

Input / Output

- Header stdio.h
  - int fprintf(FILE* stream, const char* format,…)
    - As printf, but on a stream.
  - int fscanf(FILE* stream, const char* format,…)
    - As scanf, but on a stream.
  - Default streams:
    - stdin: standard input.
    - stdout: standard output.
    - stderr: standard error.

Input / Output

- FILE* fopen(const char* fn, const char* mode)
  - Opens a file, or returns NULL on error, with mode:
    - “w”: write mode with creation of the file if it does not exist.
    - “r”: reading mode.
    - “a”: append mode with creation of the file if it does not exist.
    - “r+”: reading/writing.
    - “w+”: reading/writing with creation of the file if it does not exist.
    - “a+”: reading/append mode with creation of the file if it does not exist.
- int fclose(FILE * stream)
  - Closes a file, returning zero on success.

Input / Output

- Header string.h
  - size_t strlen(const char * s)
    - Returns the length of a string.
  - char* strcpy(char* dst, const char* src)
    - Copies string src in dst, returning dst.
  - int strcmp(const char * s1, const char * s2)
    - Compares two strings, returning:
      - Negative integer if s1 is less than s2.
      - Zero if s1 is equals to s2.
      - Positive integer if s1 greater than s2.

Input / Output

- Header string.h
  - void* memcpy(void*dst, const void* src, size_t s)
    - As strcpy, but on raw bytes. Returns dst.
    - Requires the number of bytes to copy.
  - int memcmp(const void* m1, const void* m2, size_t s)
    - As strcmp, but on raw bytes.
    - Requires the number of bytes to compare.

Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

C: Type Modifiers

- Const:
  - The value cannot be changed (it is a constant).
    - const int a = 5; /* a is the constant 5 */
    - int const a = 5; /* idem */
    - const int * b = NULL; /* b ***points to*** a constant int */
    - int const * b = NULL; /* idem */
    - int * const b = c; /* b is constant, and points to ***non constant integers*** */

C: Type Modifiers

- Memory-related modifiers:
  - Register: *suggest* to implement the variable into registers, for speed.

```
register unsigned int i;
for ( i = 0; i < 100; ++i ) { … }
```

  - Volatile: *forces* to implement the variable in memory. Useful with threads.
    - volatile int a = 0;
    - Thread 1:

```
while ( a < 100 );
```

Thread 2:

```
a = 342;
```

C: Type Modifiers

- Scope modifiers:
  - Extern: declaration has external linkage.
    - /* File inc.h */ extern int a; /* declaration */
    - /* File p1.c */ int a = 0; /* definition */
    - /* File p2.c */ a = 45; /* usage */
  - Static: declaration has internal linkage, and it is allocated in the static memory.
    - /* File p1.c */ static int a = 0; /* local declaration*/
    - /* File p2.c */ static int a = 0; /* local declaration */

C: Type Modifiers

- Restrict:
  - *Assume* no overlapping memories, to perform more aggressive optimizations.

```
void foo( char * restrict, char * restrict );

char a[ 10 ];
char * b = a + 3;
/* This ***may not*** work correctly: */
foo( a, b );
```

C: Type Modifiers

- Inline
  - *Suggest* to expand a function in the calling point, in order to improve performances.
  - The compiler must “see” the body to perform the expansion.

```
inline int foo(void)          /* If inline is applied */
{return 55;}

int a = foo();                int a = 55;
```

Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

C: Declarations & Definitions

- Declaration:
  - Warns the compiler of the existence of something.
    - E.g. the existence of a variable or method.
- Definition:
  - Tells to the compiler all the details about something.
    - E.g. allocates the memory for a variable, implements the body of a method.
- **Therefore each symbol can be declared multiple times, but must be defined only once.**
  - **Useful to split a program on multiple source files.**

C: Declarations & Definitions

	Declaration	Definition
Variable	extern int a;	int a; int b = 3; static int c = 45;
Method	void foo(void);	void foo(void) { }
Compound type (Struct / Union)	/* Forward declaration */ struct S;	/*Declaration & definition*/ struct S { };
New native type	/* Declaration & definition */ typedef int tab_size_t;	
New compound type	/* When S is a forward declaration */ typedef struct S S;	/*When S is declared & defined */ typedef struct S S;

C: Declarations & Definitions

- Forward declaration:
  - Hides all the details of a type.
  - A forwarded type can be used only as a pointer.
  - Useful to avoid illegal type recursion.
  - Used also as design pattern.

C: Declarations & Definitions

- Example of how to avoid type recursion:

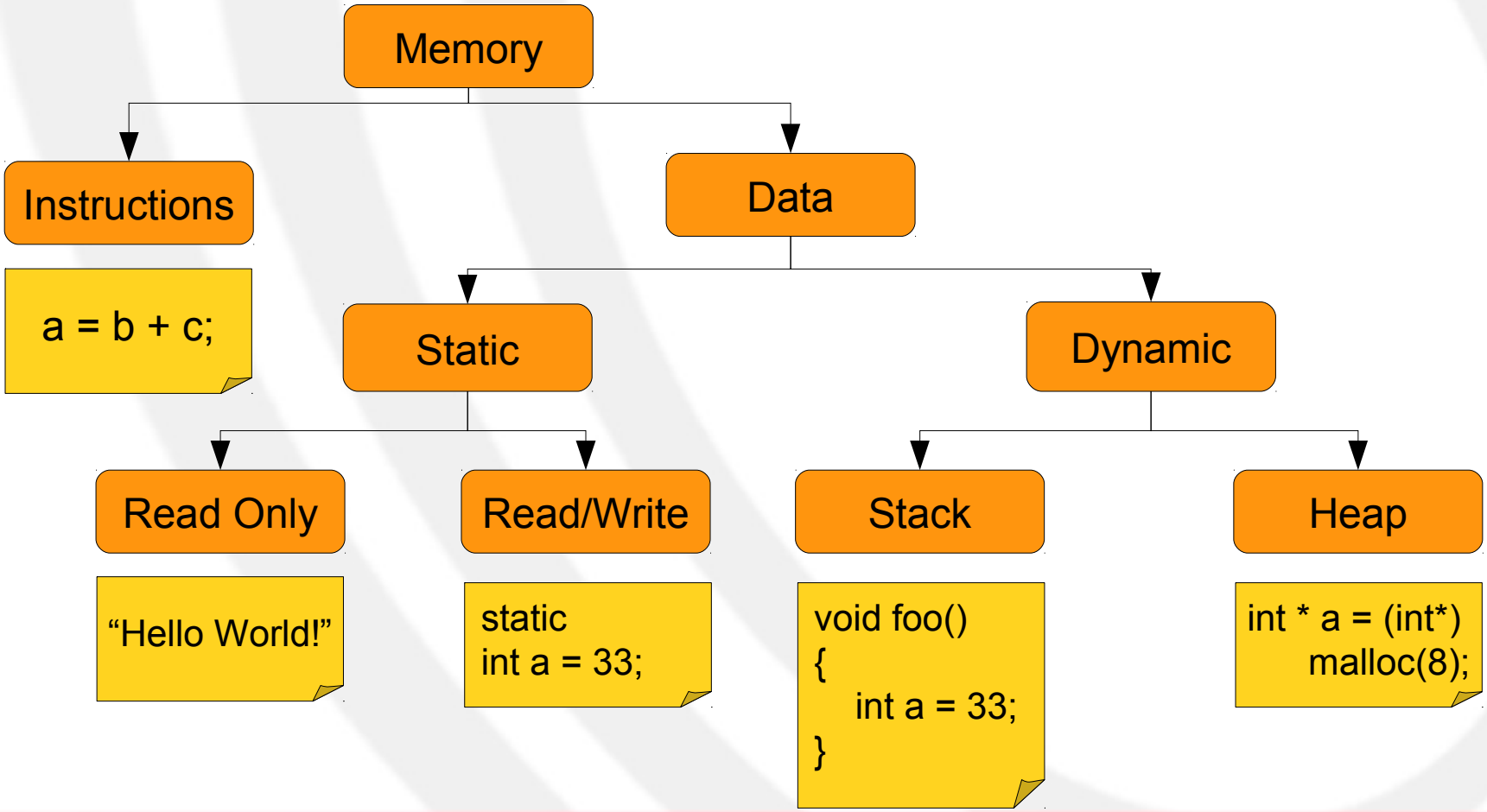
```
/* Illegal */




struct A {
    struct B b;
};

struct B {
   struct A a;
};
```

```
/* Legal */

/* Forward declaration: */
struct B;

struct A {
    struct B * b;
};

struct B {
   struct A a;
};
```

Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

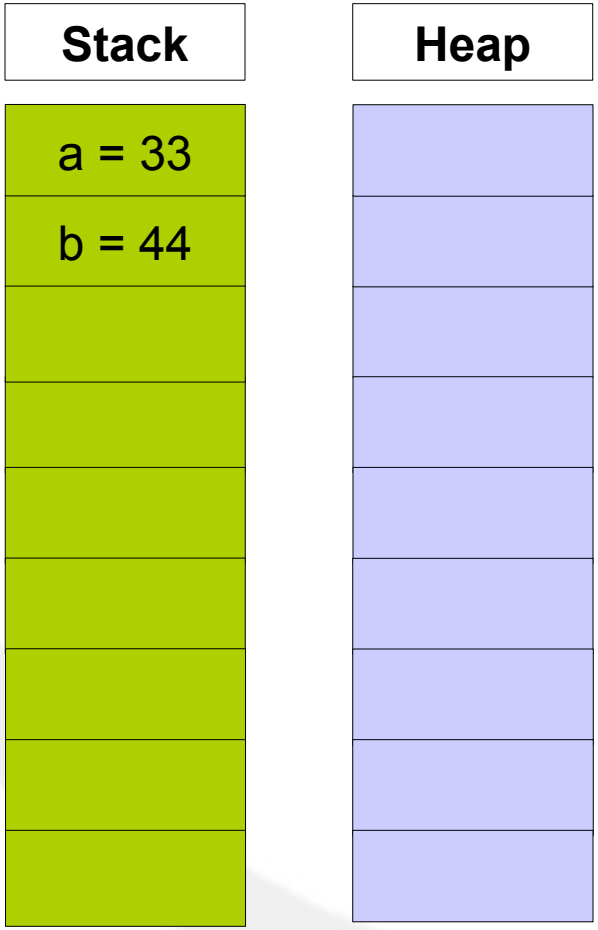
C: Memory


Memory
Instructions
Data
a = b + c;
Static
Dynamic
Read Only
Read/Write
Stack
Heap
“Hello World!”
static
int a = 33;
void foo()
{
int a = 33;
}
int * a = (int*)
malloc(8);

C: Memory

Variables on the stack.

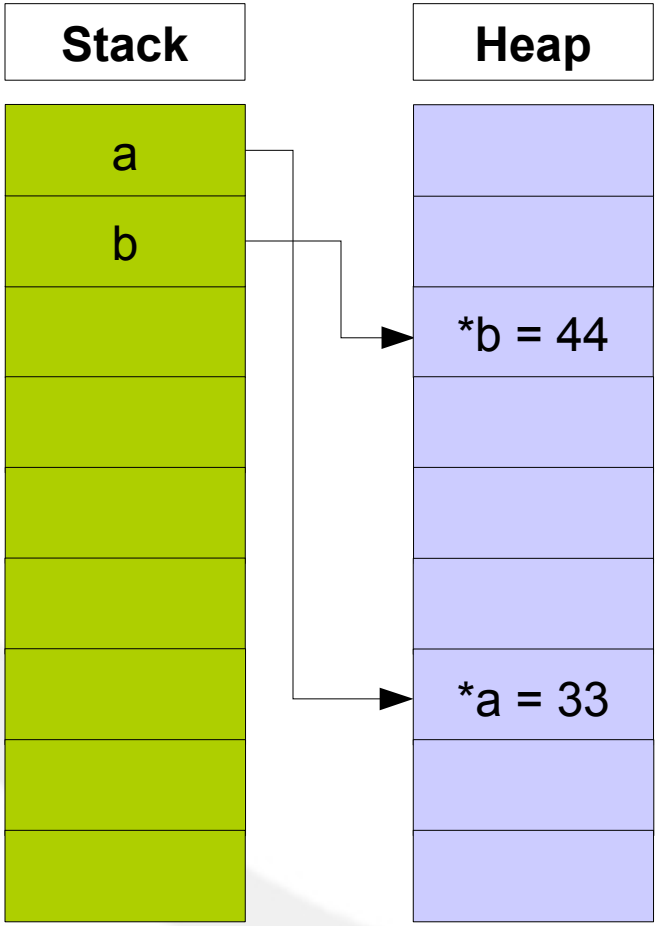
```
void foo()
{
    int a = 33;
    int b = 44;
}
```



C: Memory

Variables in the heap.

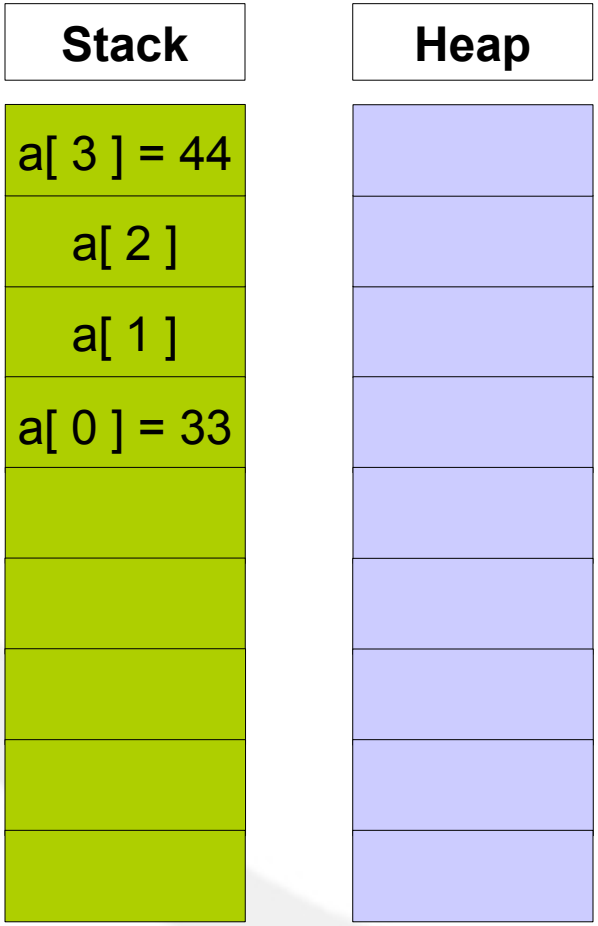
```
void foo()
{
    int *a = (int*) malloc(sizeof(int));
    int *b = (int*) malloc(sizeof(int));

    *a = 33;
    *b = 44;
}
```



C: Memory

Array on the stack.

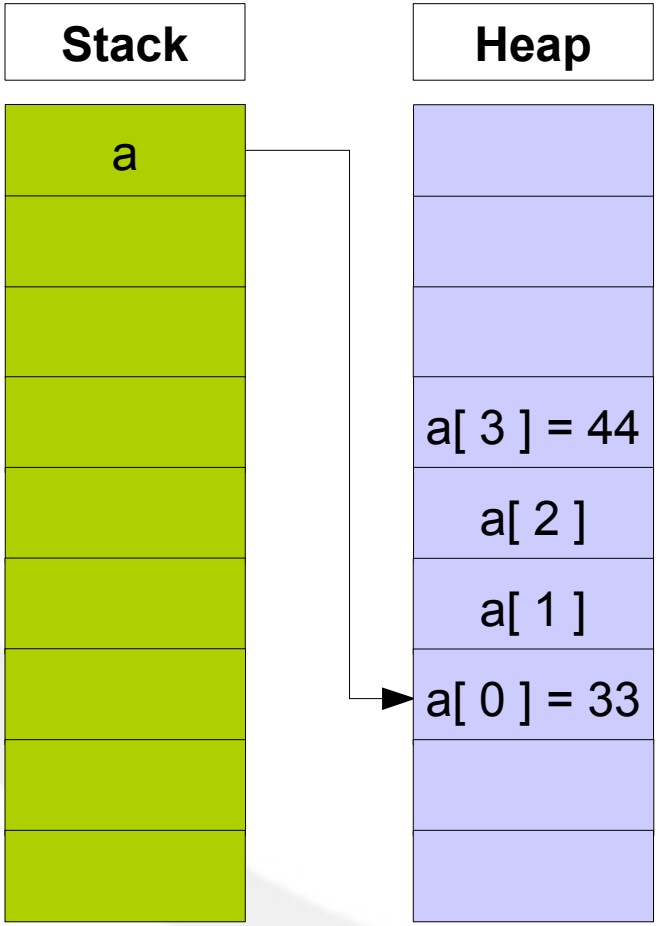
```
void foo()
{
    int a[ 4 ];

    a[ 0 ] = 33;
    a[ 3 ] = 44;
}
```



C: Memory

Array in the heap.

```
void foo()
{
    int *a = (int*) malloc(4*sizeof(int));

    a[ 0 ] = 33;
    a[ 3 ] = 44;
}
```

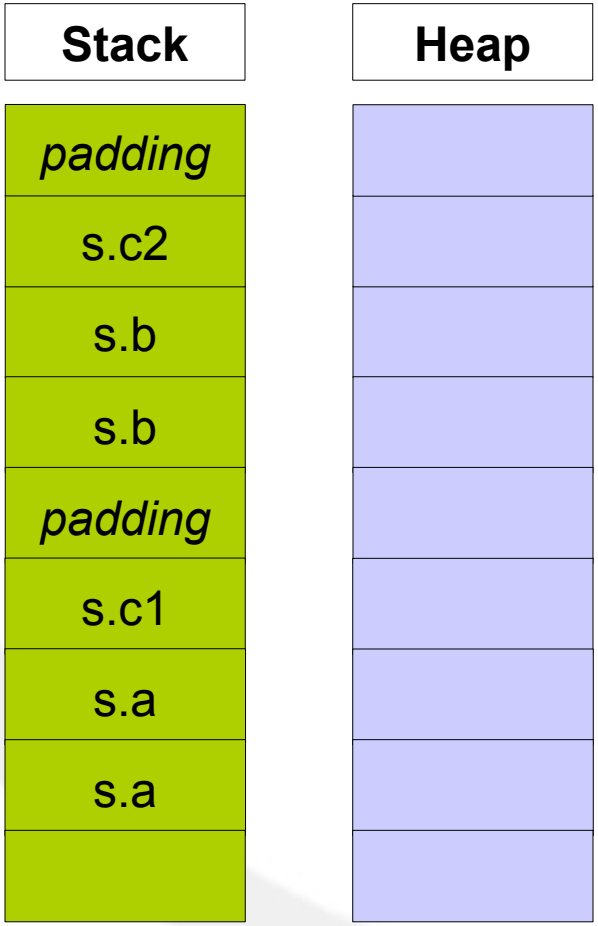


C: Memory

Struct with int size equals to 2 bytes, blocks of 1 byte, in the worst case.

```
struct S {
    int a;
    char c1;
    int b;
    char c2;
};
void foo(){
    struct S s;
}
```

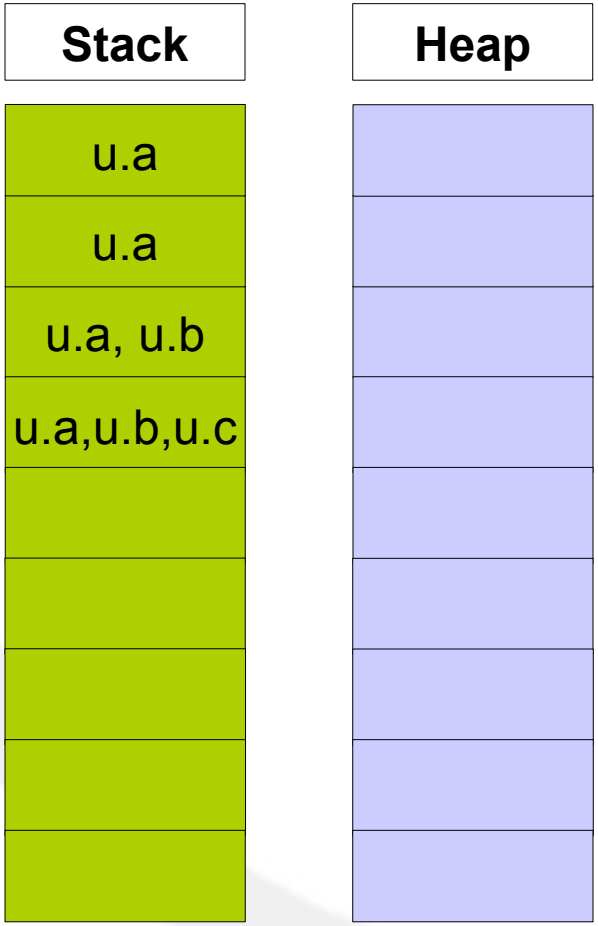
WARNING: in general sizeof(S) != 2*sizeof(int)+2*sizeof(char)



C: Memory

Union with int size equals to 4 bytes, and blocks of 1 byte.

```
union U {
    int a;
    char c;
    short b;
};

void foo(){
    union U u;
}
```



Agenda

- C Programming Language Overview.
- Main Library Methods.
- Type Modifiers.
- Declarations & Definitions.
- Memory.
- Arrays vs Pointers.

C: Arrays vs Pointers

- **Pointers are not arrays !!**
  - *Look back to previous slides!*
  - Pointers are **references** to memory locations.
  - Arrays are locations of **sequential memory**.

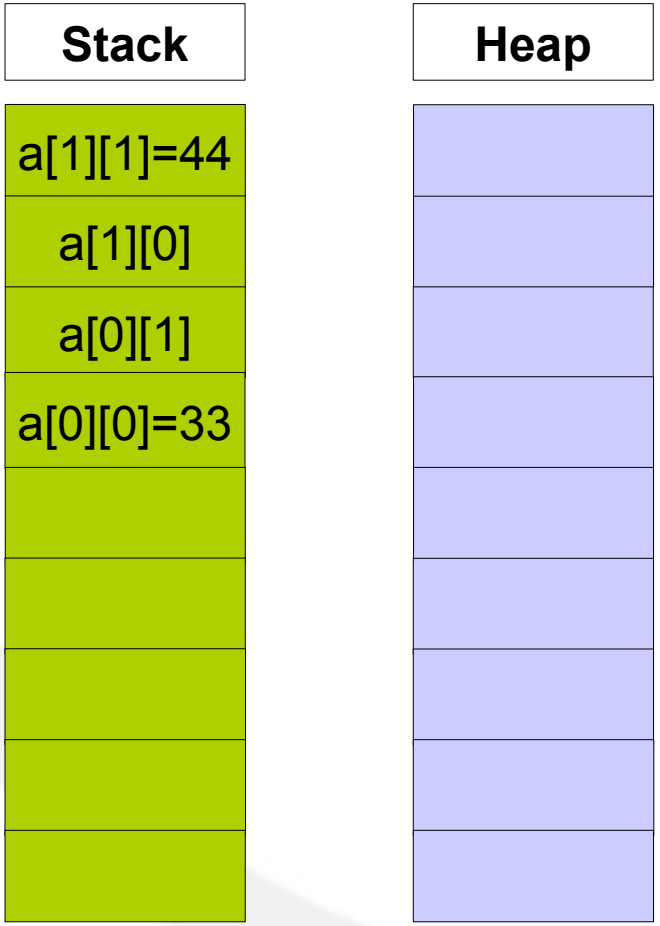
```
/* If a should be an array,
 * avoid this!! */

void foo( int * a, unsigned len );
```

```
/* Prefer this!! */


void foo( int a[], unsigned len );
```

C: Arrays vs Pointers

Matrix.

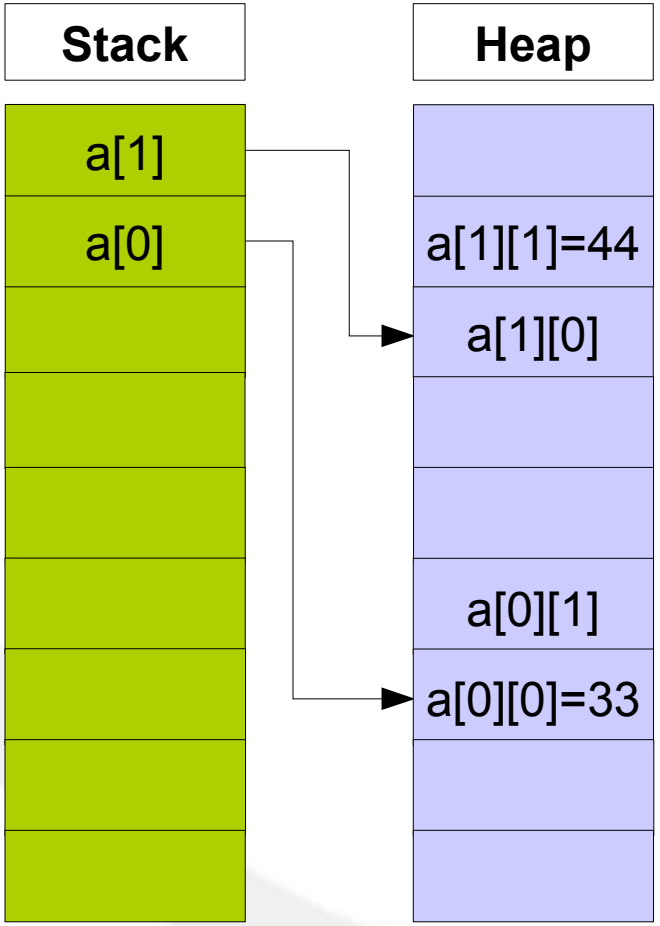
```
void foo()
{
    int a[ 2 ][ 2 ];

    a[ 0 ][ 0 ] = 33;
    a[ 1 ][ 1 ] = 44;
}
```



C: Arrays vs Pointers

Array of pointers.

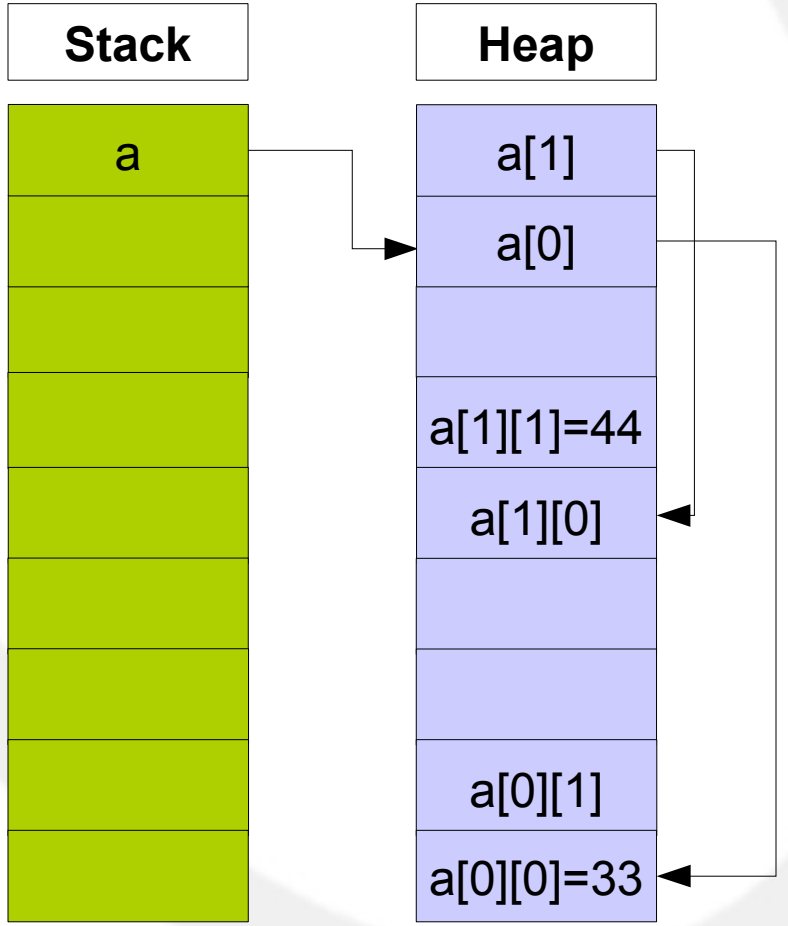
```
void foo()
{
    int *a[ 2 ];
    a[ 0 ] = (int*)malloc(2*sizeof(int));
    a[ 1 ] = (int*)malloc(2*sizeof(int));

    a[ 0 ][ 0 ] = 33;
    a[ 1 ][ 1 ] = 44;
}
```



C: Arrays vs Pointers

Pointer to pointers.

```
void foo()
{
    int **a;
    a = (int**)malloc(2*sizeof(int*));
    a[ 0 ] = (int*)malloc(2*sizeof(int));
    a[ 1 ] = (int*)malloc(2*sizeof(int));

    a[ 0 ][ 0 ] = 33;
    a[ 1 ][ 1 ] = 44;
}
```



C: Arrays vs Pointers

- C String:
  - An array of characters, terminated by '\0'
  - char a[] = “Hello”;
    - A string on the stack – 6 bytes.
  - char * a = “Hello”;
    - A string in read-only data memory – 6 bytes – and a pointer on the stack.
  - const char * a = “Hello”;
    - Idem, but better since allows compile time checks.

C: Arrays vs Pointers

Three string examples.

```
void foo()
{
    const char * a = “AAA”;
    char b [ ] = “BBB”;
    char * c = (char*)
            malloc(4*sizeof(char));
    strcpy(c, “CCC”);
}
```