

Chapter 2

From JAVA to C

#include - File Inclusion

- The C language has portability and reusability characteristics.
- Portability: the language is not tied down to a certain platform/environment.
- Reusability: the language enables reuse of code.
- The pre-processor directive “#include” enables including (standard) utility files and user defined files into the program during compilation.
- This way, the programmer can use utilities that were defined already by other programmers.

Input/Output (I/O)

- C supplies functions that are responsible for input/output.
- scanf is usually used for input and printf for output.
- Syntax:
 - scanf("% format string", &variable name);
 - printf("% format string", variable name);
- #include <stdio.h> library to use the I/O functions.
- Example:

```
#include <stdio.h>
int x;
float y;
char z;
scanf("%d%f%c", &x, &y, &z);
printf("x=%d y=%f z=%c\n", x, y, z);
```

if Statement

- “if” syntax:

 - if** (expression)
statement;

- If there are multiple statements, then add a block (using braces):

 - if** (expression)
{
 statements;
}

if-else Statement

- “if...else” Syntax:
if (expression)
 statement;
else
 statement;
- If there are multiple statements, then add block:
if (expression)
 {
 statements;
 }
else
 {
 statements;
 }

Example

- The program gets a character from user. If the character is a digit or 'y', the user gets a correct message and increments the character, else the user gets an error message:

```
char num;  
scanf("%c", &num);  
if (num >= '0' && num <= '9' || num == 'x'+1)  
{  
    printf("good choice\n");  
    num++;  
}  
else  
    printf("bad choice\n");
```

switch-case Statement

When few constant alternatives exist, we can select the correct one by using if-else, or even better by switch-case:

```
switch(expression)           // The value of the statement will be
{                             // looked for in the existing cases.
    case constant1:           // The matched case will be executed
        [ break; ]           // up to the end of the switch block
    case constant2:           // or a break.
        statement(s);
        [ break; ]
    case constant3:
        statement(s);
        [ break; ]

    default:                 // The default statement is optional.
                             // If no matching case was found, the
                             // statements in default will be executed.

}
```

switch-case Statement

- switch-case is useful for constructing menus.
- A menu is composed of two parts: 1. Menu 2. Selection
- Example:

Menu:

```
int selection;
printf("choose an option:\n"
      "1 - addition\n"
      "2 - subtraction\n"
      "3 - multiplication\n");
scanf("%d", &selection);
```

If we remove 'break' from case 1 and selection is 1, then both case1 and case2 will be executed!!

Selection:

```
switch(selection)
{
    case 1:
        printf("%d", x+y);
        break;
    case 2:
        printf("%d", x-y);
        break;
    case 3:
        printf("%d", x*y);
        break;
    default:
        printf("wrong");
        break;
}
```


for - Syntax

for(counter initialization; condition; counter increment)
statement;

for(initialization; condition; increment)
{
statements;
}



**For multiple
statements:
use a block**

- **Initialization:** initialization of the counter. Initialization is applied only once at the start.
- **Condition:** for each iteration, the condition is checked. If the condition is true, the statement(s) are executed, else the program continues after the 'for' block. Usually the condition checks the counter.
- **Increment:** Increment (or decrement) the counter. After incrementing the counter, the condition is checked.

for - Examples

- Printing numbered “hello world” 3 times:

```
int i;  
for(i=0; i<3; i++)  
    printf("no. %d hello world\n", i+1);
```

- Explanation:

Condition is true when $i=0,1,2$. In these cases, “hello world” will be printed. When $i=3$, the “for” loop will be exited.

- Calculating the sum of the teens group (10-19):

```
int i, sum=0;  
for(i=10; i<20; i++)  
    sum += i;
```

- Explanation:

In this case, i is initialized to 10, and the condition is $i<20$.

for - Comments

- Increment can be any mathematical operation:

```
int i;  
for (i = 20; i > 0; i /= 2)
```

- The counter can be any number type:

```
char c;  
for (c='a'; c<='z'; c++)
```

- Each of the 3 expressions can include more than one expression:

```
double i, j;  
for (i=11.1, j=22.2; i<j && j>=15; i*=2, --j)
```

- One or more expressions can be left blank:

```
int i = 10;                                //instead of initialization  
for (   ; i<20;   )  
{  
    printf("hello world\n");  
    i++;                                    //instead of increment  
}
```

while – Syntax

```
while (condition)
    statement;
while (condition)
{
    statements;
}
```



**For multiple
statements:
use a block**

Condition: In each iteration, the condition is checked. If the condition is true, the statement(s) are executed, else the program continues after the 'while' block. The value of the condition must be changed through the statement(s), otherwise it's an infinite loop.

while – Examples

- Printing numbered “hello world” 3 times:

```
int i=0;
while(i<3)
    printf("no. %d hello world\n", ++i);
```

- Explanation:

Condition is true when $i=0,1,2$. In these cases, “hello world” will be printed. When $i=3$, the “while” loop will be exited.
Attention: this loop is a counter loop, which is implemented within the while loop. Generally, prefer a “for loop” for a counter loop.

- Sum digits:

```
int num=345, sum=0;
while(num)    //exactly as: while(num != 0)
{
    sum += num%10;
    num /= 10;
}
```

while – Examples

- Getting a digit from the user:

```
char digit;  
while(digit < '0' || digit > '9')  
{  
    printf("enter a digit");  
    scanf("%c", &digit);  
}
```

- problem:

In this example, the initial digit's value is garbage – not good! Garbage could also be some character between '0' to '9'. In this case, the while loop will not be applied at all, and the user is never asked to enter a digit.

do-while Statement

- We can solve this problem by using a do-while loop:

```
char digit;  
do  
{  
    printf("enter a digit");  
    scanf("%c", &digit);  
} while(digit < '\0' || digit > '\9');
```

- In do-while loop, the statements are executed before the condition test.
- This kind of loop is used mainly in case the statement(s) need to be performed at least once.
- Syntax:
 - Parenthesis after the while are mandatory (even if there is only one statement).
 - Semicolon is mandatory at the end of do-while.

break Statement

- “break” command is intended to break out of a loop execution.
- Example: Getting a digit from the user for up to 3 times:

```
int flag = 0;
int i;
char digit;
for(i=0; i<3; i++)
{
    printf("enter a digit");
    scanf("%c", &digit);
    if(digit >='0' && digit <='9')
    {
        flag = 1;
        break;
    }
}
```
- Flag is necessary to inform us if the digit has been gotten.

continue Statement

- “continue” command is used to skip over the rest of the statements in the body of the loop.
- Example: Adding a sequence of numbers, except those that are divisible by 4:

```
int i, sum=0;
for(i=0; i<20; i++)
{
    if(!(i%4))
        continue;
    sum += i;
}
```

- When i is divided by 4 without a remainder, the ‘continue’ command is applied, and the statement “sum += i;” is not executed but skipped (to proceed with increment of i).

Function - introduction

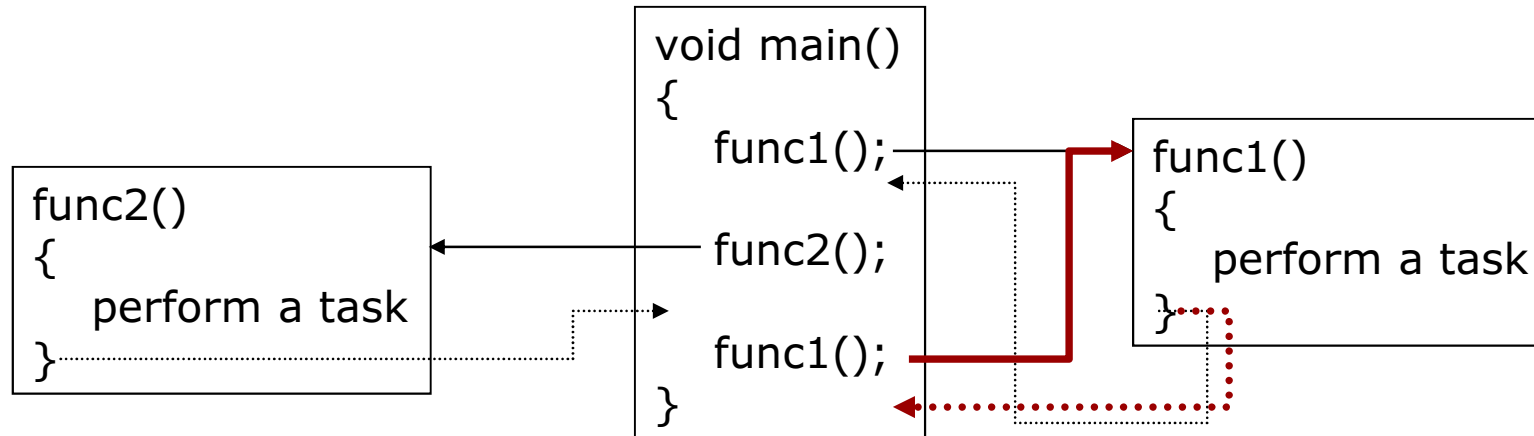
- C is a procedural language that is based on functions.
- A function is a self-contained unit of program code, designed to accomplish a particular task.
- Every C program starts by executing a function named main, which in turn may call other functions.
- A function can perform various actions and then return a value.

Introduction - Purpose

- We use functions for several purposes:
 - To save repetitions.
 - To make programs more modular (easier to change and maintain).
 - Save work by using functions that have already been written and tested in other programs.
 - Readability (avoid 'spaghetti' code).
- A function has 3 parts:
 1. Declaration
 2. Definition
 3. Invocation

Function Invocation

- To call a function we invoke its name attached to parentheses ().
- Example:



Declaration

- Function declaration is the prototype of a function.
- Syntax:

```
return_type Function_name(parameters);
```

- **return_type:**
 - The type of the value that the function returns.
 - If no return type is specified, `int` is the default.
 - If the function only performs some task and has no value to return, one should specify **void** as the return type.

Declaration

■ Parameters:

- The parameters are local variables that are initialized with values passed to the function.
- A function may have no parameters. In this case the parentheses are empty.
- The declaration role is to prevent errors in compilation. Without the declaration, the compiler cannot recognize the invoked function characteristics. Therefore, the declaration should always be before main.

Declaration

```
void print_header();
```

Return
type

Function
name

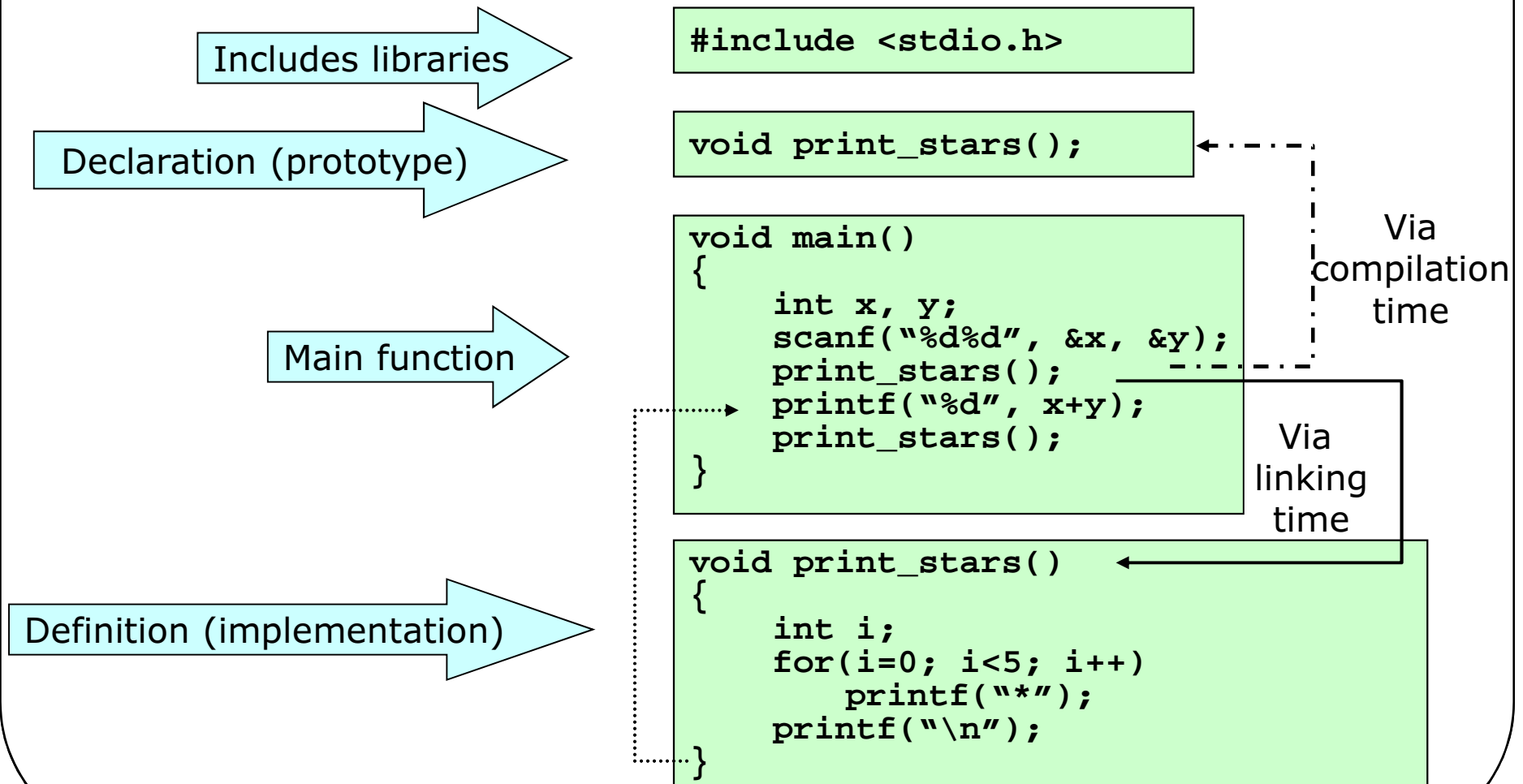
Parameters

```
void main()  
{  
    print_header();  
}
```

Definition

- The function body contains:
 - Definitions of local variables which are optional but usually exist.
 - Executable statements, which may be any statement in C.
- If the function returns a value, the *return* statement is mandatory. It may appear in a function more than once. Syntax: **return** (**expression**);
 - The parentheses are optional.
 - If the function returns no value (void), it can still be stopped before its end. Syntax: **return**;
- Usually the function appears below the main.

Definition



Parameters

- Function invocation is enabled with parameters.
- The parameters are sent to the function by being placed in the parentheses.
- We should define, both in the prototype and in the definition, the parameter's types, in the same order they are sent in the invocation.
- Invoking a function with inappropriate parameters causes a compilation error.
- Let's rewrite the `print_stars` function, so that it will print out a varying number of whichever char is requested by the calling function.
For example, `print_stars(6, '$');` will result in 6 dollar signs in a row.

Parameters

Includes libraries

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

Declaration (prototype)

```
void print_stars(int num, char sign);
```

Main function

```
void main()  
{  
    int x, y;  
    scanf("%d%d", &x, &y);  
    print_stars(6, '$');  
    printf("%d", x+y);  
    print_stars(5, '*');  
}
```

num=6
sign='\$'

Definition (implementation)

```
void print_stars(int num, char sign)  
{  
    int i;  
    for(i=0; i<num; i++)  
        printf("%c", sign);  
    printf("\n");  
}
```

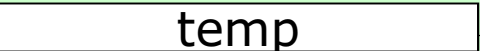
Returning a Value

- A function may return a value. The value is returned to the calling code.
- The returned value has:
 - Value
 - Type
 - Address
- The returned type is written at the start of the function prototype and definition.
- We can exploit the returned value of the invocation by assigning it to a variable.

Returning a Value

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

```
void print_stars();  
int sum_numbers(int x, int y);
```

```
void main()  
{  
    int x, y, sum;  
    scanf("%d%d", &x, &y);  
    sum =  temp  
    print_stars();  
    printf("%d", sum);  
    print_stars();  
}
```

```
void print_stars()  
{  
    printf("*****");  
}  
int sum_numbers(int x, int y)  
{  
    int sum;  
    sum = x+y;  
    return sum;  
}
```

temp is created
automatically by
the compiler

temp=sum

Returning a Value

No compatibility
is needed
between
parameter's
names or
returned value's
name and
main's variables

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

```
void print_stars();  
int sum_numbers(int num1, int num2);
```

```
void main()  
{  
    int x, y, adding;  
    scanf("%d%d", &x, &y);  
    adding = sum_numbers(x, y);  
    print_stars();  
    printf("%d", adding);  
    print_stars();  
}
```

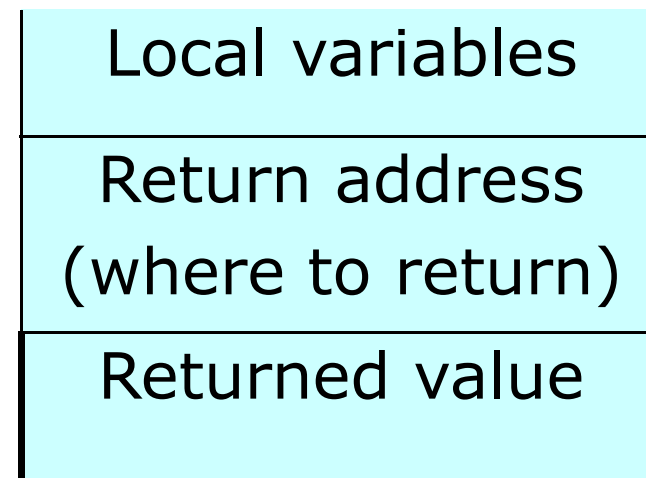
```
void print_stars()  
{  
    printf("*****");  
}  
int sum_numbers(int num1, int num2)  
{  
    int sum;  
    sum = num1+num2;  
    return sum;  
}
```

Scope and Lifetime

- The scope of a variable is the part of the program in which the variable is visible (i.e., may be accessed and used).
 - The scope of local variable is limited to a block. For instance, we can not access x (of main) from a function.
- The lifetime of a variable is the time between the 'birth' of the variable in the memory and the time it 'dies'.
 - The lifetime of a local variable is only throughout the block. For instance, x (of main) is born in the beginning of the main block and dies at the end of the main block.

Stack Segment

- Definition: storing local variables of a function in a memory area using a stack form (LIFO: Last In, First Out).
- Stack pointer: pointer to the active element in the stack.
- The stack is prepared during compiling time.



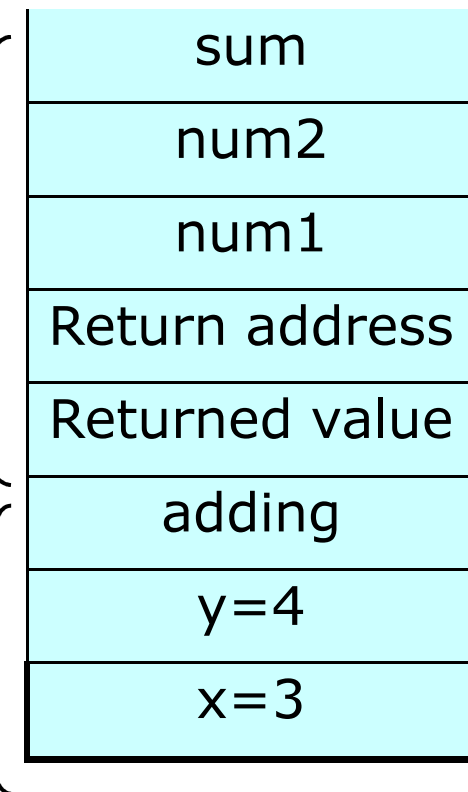
Stack Segment Example

```
void main()  
{  
    int x=3, y=4, adding;  
    adding = sum_numbers(x, y);  
}
```

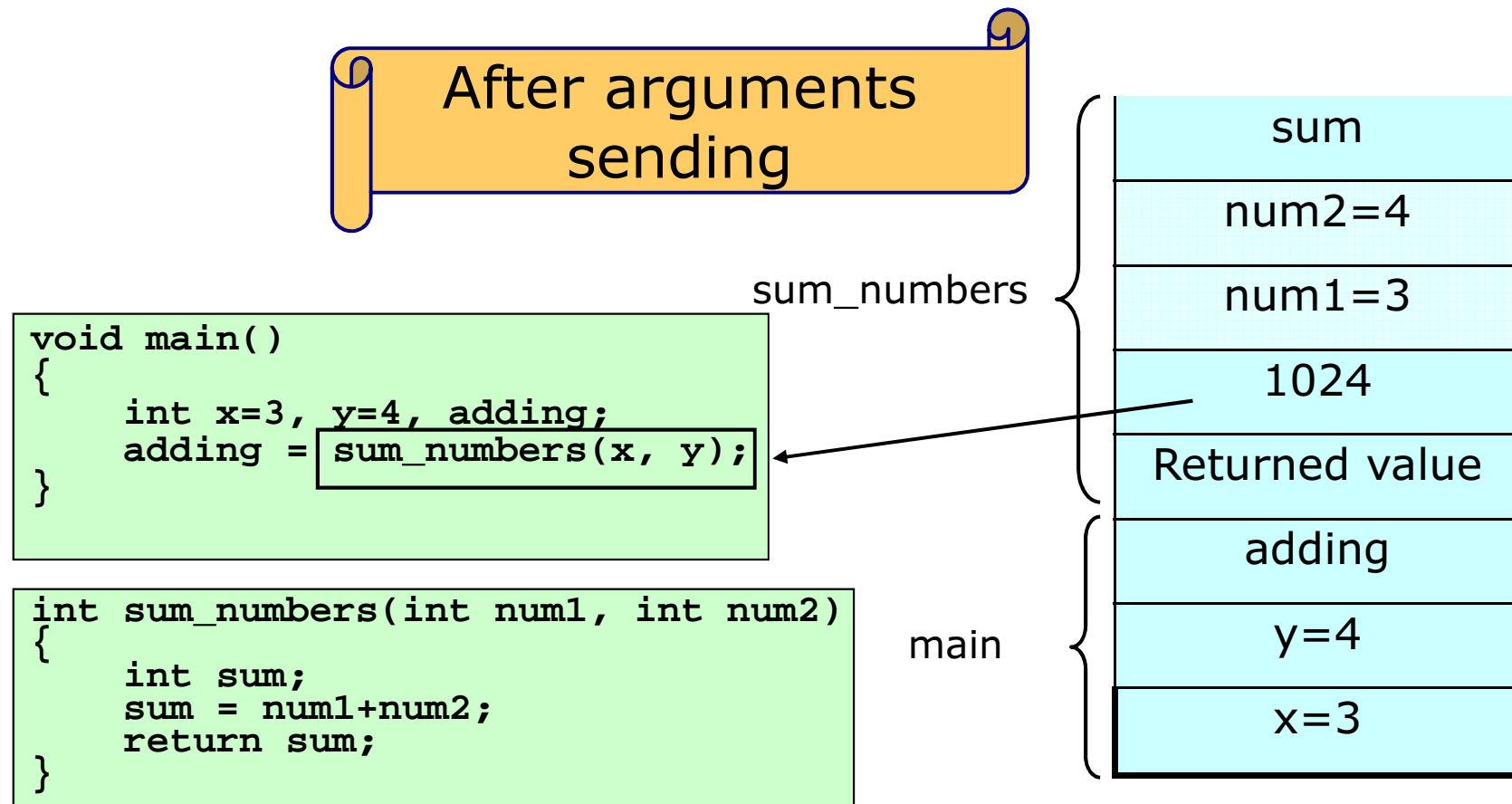
```
int sum_numbers(int num1, int num2)  
{  
    int sum;  
    sum = num1+num2;  
    return sum;  
}
```

sum_number
s

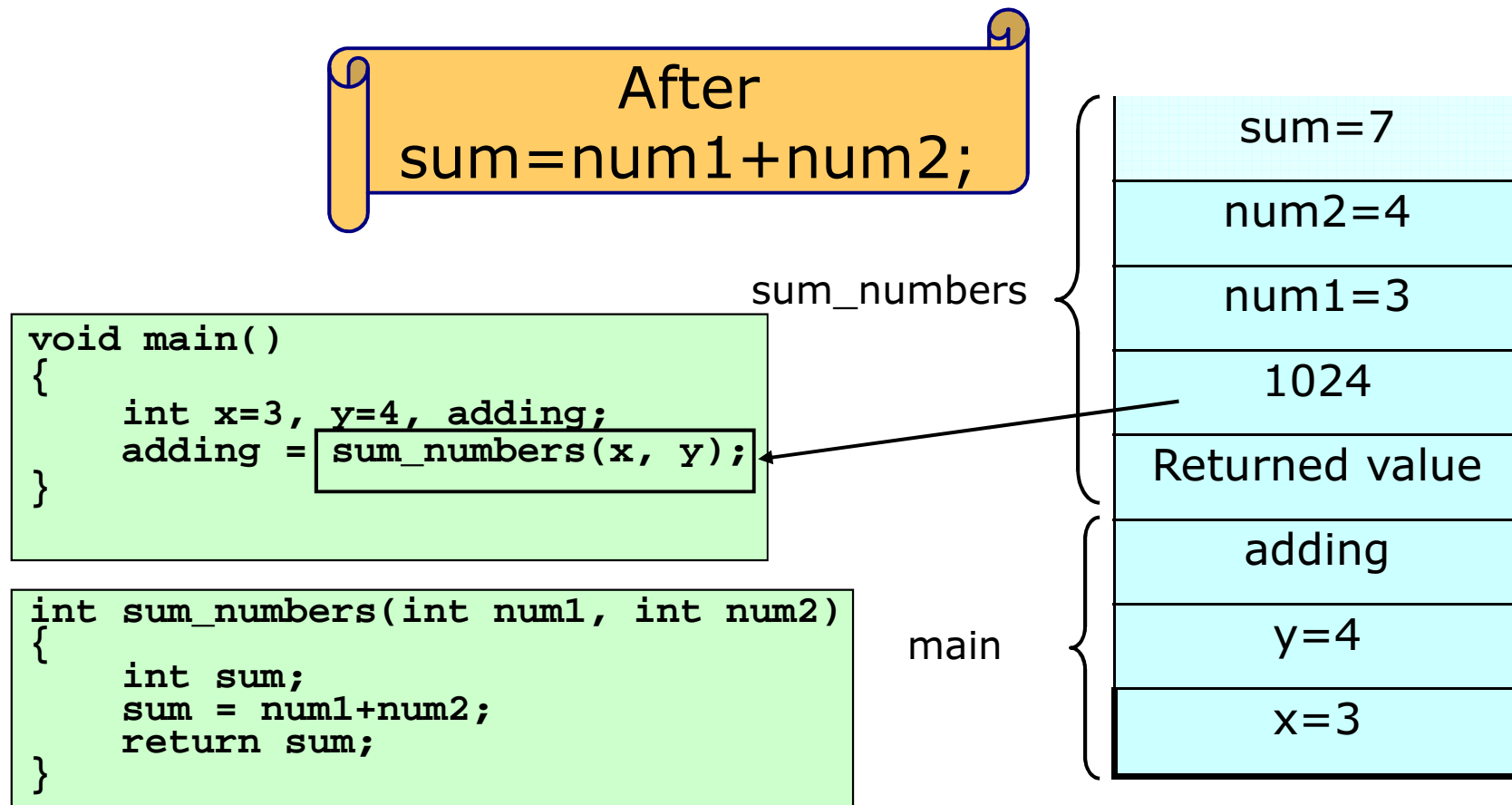
main



Stack Segment Example



Stack Segment Example



Stack Segment Example

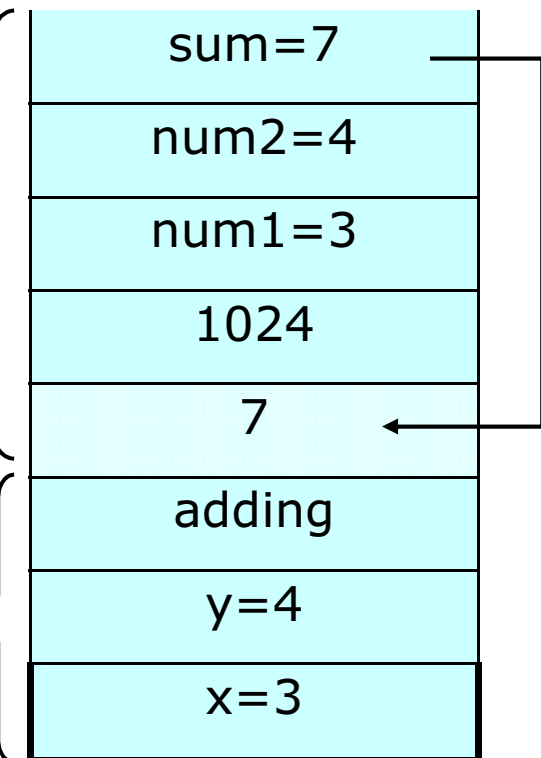
After update of
returned value

```
void main()  
{  
    int x=3, y=4, adding;  
    adding = sum_numbers(x, y);  
}
```

```
int sum_numbers(int num1, int num2)  
{  
    int sum;  
    sum = num1+num2;  
    return sum;  
}
```

sum_numbers

main



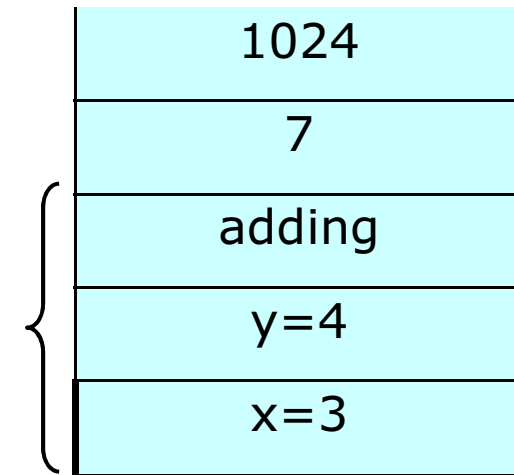
Stack Segment Example

After freeing of
local variables

```
void main()  
{  
    int x=3, y=4, adding;  
    adding = sum_numbers(x, y);  
}
```

```
int sum_numbers(int num1, int num2)  
{  
    int sum;  
    sum = num1+num2;  
    return sum;  
}
```

main



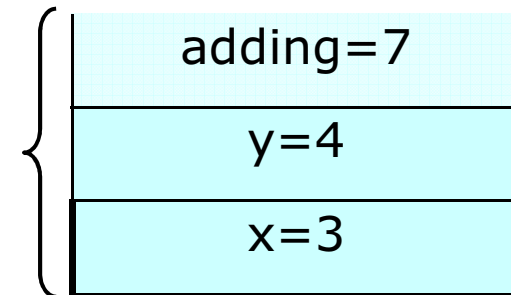
Stack Segment Example

After assigning
returned value to
adding

```
void main()  
{  
    int x=3, y=4, adding;  
    adding = sum_numbers(x, y);  
}
```

```
int sum_numbers(int num1, int num2)  
{  
    int sum;  
    sum = num1+num2;  
    return sum;  
}
```

main



Global

- Scope: from the line it is defined up to the end of the file.
- Lifetime: during program execution.
- Initialization: if the global variable is not initialized, its default initialization is 0.

■ Example:

```
void func();
int glob;      //0
void main()
{
    glob = 5;
    printf("%d", glob); //5
    func();
    printf("%d", glob); //8
}
void func()
{
    glob = 8;
}
```

Global variables are not created in the stack segment but in the data segment (heap)

Static

- Scope: from the line it is defined up to the end of its block.
- Lifetime: during program execution.
- Initialization: if the static variable is not initialized, its default initialization is 0.
- Example:

stat
0
address: 100


local
0
address: 200


```
void func();
void main()
{
    int i;
    for(i=0; i<3; i++)
        func();
}
void func()
{
    static int stat;
    int local = 0;
    printf("%d %d\n", stat++, local++);
}
```

Static variables are not created in the stack segment but in data segment (heap)

Static

- Example state after second call.
- Example: **void func();**

stat

 address: 100

local

 address: 340

```
void main()
{
    int i;
    for(i=0; i<3; i++)
        func();
}

void func()
{
    static int stat;
    int local = 0;
    printf("%d %d\n", stat++, local++);
}
```

Prepared by Meir Kalech

Static

- Example state after third call.

- Example: **void func();**

void main()

{

int i;

for(i=0; i<3; i++)

func();

}

void func()

{

static int stat;

int local = 0;

printf("%d %d\n", stat++, local++);

}

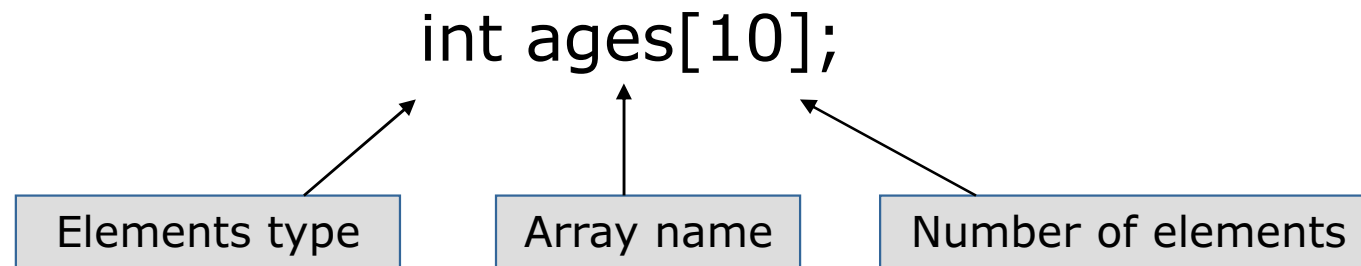
stat
2
address: 100

local
0
address: 280

Summary Table

	Global	Static	Automatic Local
declaration	outside any function	in a block { }	in a block { }
initialization	unless specified otherwise, automatically initialized to 0	unless specified otherwise, automatically initialized to 0	no automatic initialization – it should specifically be initialized after each 'birth'
scope	its file	its block	its block
'birth'	once, before <code>main()</code>	once, before <code>main()</code>	each time the block is entered
'death'	once, after <code>main()</code> ends	once, after <code>main()</code> ends	each time the block is exited
address	on the data segment	on the data segment	on the stack segment

Array Definition



Name of the array = First address of the array

address →	100	104	108	112	116	120	124	128	132	136
value →	?	?	?	?	?	?	?	?	?	?
index →	0	1	2	3	4	5	6	7	8	9

Array Definition

- sizeof() operator gets a type or variable and returns its size in bytes.

- For example:

```
int x;  
printf("%d", sizeof(int));    // print 4  
printf("%d", sizeof(x));     // print 4
```

- Array subscripting operator [] denotes an index in an array and enables access to the value of the addressed array index.

- For example:

```
int ages[10];  
ages[3] = 23;  
printf("%d", ages[3]);
```


$$4 * 10 = 40$$

- What is sizeof(ages)?

Notes on Arrays

- No boundary checks are performed by the computer.
- Size (number of elements) must be constant – requested amount of memory must be known at compilation time:

```
int i = 5;
```

```
int array[i];           // wrong, i is not a constant!
```

However:

```
#define SIZE 5
```

```
int array[SIZE];        // OK, SIZE is a symbolic constant!
```

- C does not have any operator to work with an array “as a whole”:
 - For instance: we can’t assign into an array multiple values at once; we can’t print all the array at once, etc...
 - question: what is the output of - `printf(“%p”, ages);` . 100

Array Initialization

- Within declaration (using an initialization list):

```
int array[5] = {5,8,2,7};
```

5	8	2	7	0
0	1	2	3	4

**Automatically
initialized to 0**

- If the size of array is not specified, size is determined by the number of initializing values provided:

```
int array[] = {5,8,2,7};
```

5	8	2	7
0	1	2	3

- Looping through the elements and assigning them a value:

```
int array[5], i;  
for (i = 0; i < 5; ++i)  
    array[i] = i; // or scanf("%d", &array[i]);
```

Prepared by Meir Kalech

Two-dimensional Array

- Two-dimensional (2D) array is an array of arrays.
- For instance:
 - the definition `int matrix[3][4]` means 3 arrays of 4 integers each.
- As for a one-dimensional (1D) array, the elements will be created sequentially in memory (in row-major order).

100	104	108	112	116	120	124	128	132	136	140	144
?	?	?	?	?	?	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- So what is the difference between the two???

String

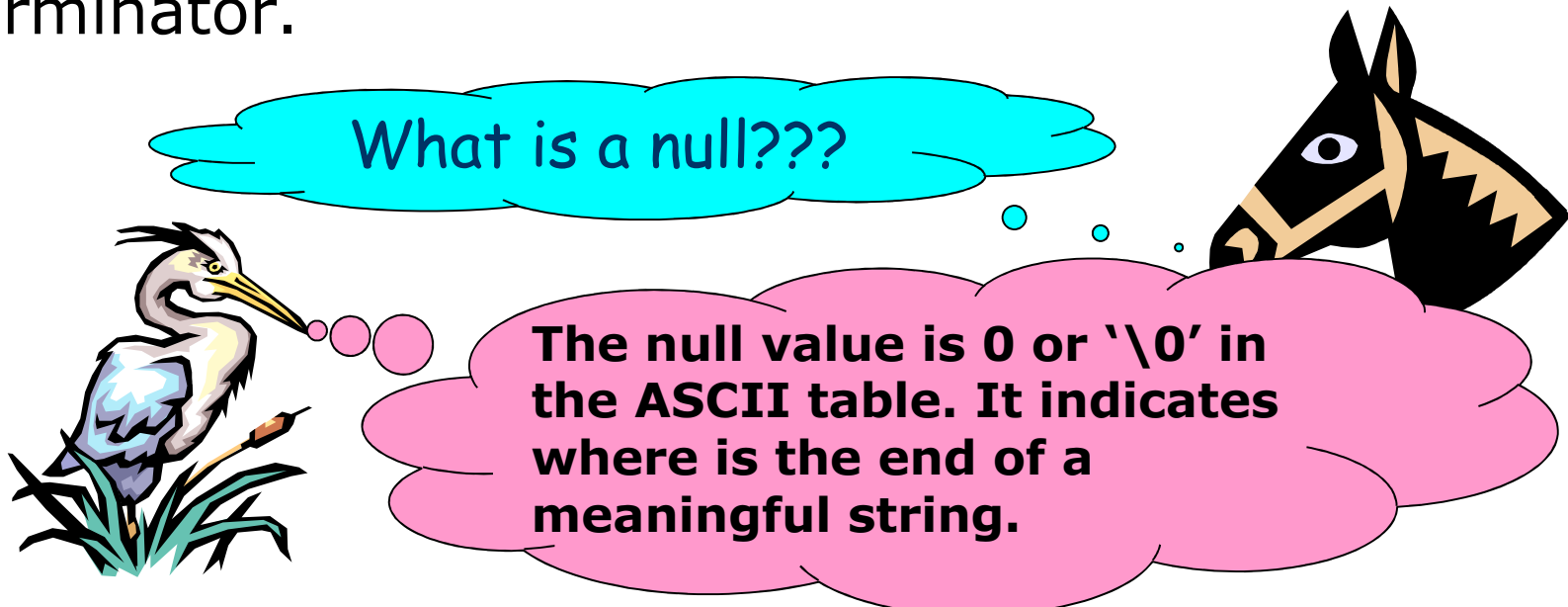
- String is an array of characters.
- It is treated separately because it is very common to define characters sequentially.
- When we store a word in an array (string) it is uncomfortable to take care of each character separately. For instance, to go over the characters to be inserted or to print a word.
- There are library functions which enable simple access to strings.

String Definition

- String definition:

```
char name[10];
```

- Name consists of 10 chars. But actually we can store only 9 chars, because one char is saved for the null terminator.



String Initialization

```
char name[13] = "Im a donkey";
```

I	m		a		d	o	n	k	e	y	\0	?
---	---	--	---	--	---	---	---	---	---	---	----	---

```
char name[] = "Im a donkey";
```

```
char name[] = {'I','m',' ','a',' ','d','o','n','k','e','y','\0'};
```

I	m		a		d	o	n	k	e	y	\0	
---	---	--	---	--	---	---	---	---	---	---	----	--

```
char name[13];  
scanf("%s", name); //Im a donkey
```

I	m	\0	?	?	?	?	?	?	?	?	?	?
---	---	----	---	---	---	---	---	---	---	---	---	---

```
gets(name); //Im a donkey
```

I	m		a		d	o	n	k	e	y	\0	?
---	---	--	---	--	---	---	---	---	---	---	----	---

String Initialization

```
char name[13] = "Im a donkey";
```

Comment:

The assignment is permitted only during declaration. There is no assignment operator between strings:

```
char name1[10],  
name2[10];  
name1 = name2;
```



Compilation
Error!!!

String Library

- String manipulations can be done using the `<string.h>` library (`#include <string.h>`).
- This library contains several functions that enable copy, comparison, search, etc... on strings.
- Comment: strings that are sent to functions as arguments, must be initialized (at least) with null.
- The following functions are defined for strings:

➤ **`int strlen(char str[])`**

Returns the length of string `str` (not including the null terminator).

```
int length;
```

```
char str1[10] = "Arie";
```

```
length = strlen(str1);    // length = 4
```

String Library

■ int strcmp(char str1[], char str2[])

Compares two strings (lexicographically); Returns:

- 0 - the strings are equal.
- >0 - the first string is greater.
- <0 - the second string is greater.

```
int compare;
```

```
char str1[10] = "Arie", str2[] = "arie";
```

```
compare = strcmp(str1, str2); // compare<0
```

■ char *strcpy(char str1[], char str2[])

Copies str2 to str1 (overwriting). (Used instead of: str1=str2;).

Comment: str1 does not have to be necessarily initialized.

```
char str1[10] = "Ariela", str2[] = "dana";
```

```
strcpy(str1, str2); // str1 must be longer than str2
```

str1	A	r	i	e	l	a	\0	?	?	?
------	---	---	---	---	---	---	----	---	---	---

String Library

■ int strcmp(char str1[], char str2[])

Compares two strings (lexicographically); Returns:

- 0 - the strings are equal.
- >0 - the first string is greater.
- <0 - the second string is greater.

```
int compare;
```

```
char str1[10] = "Arie", str2[] = "arie";
```

```
compare = strcmp(str1, str2); // compare<0
```

■ char *strcpy(char str1[], char str2[])

Copies str2 to str1 (overwriting). (Used instead of: str1=str2;).

Comment: str1 does not have to be necessarily initialized.

```
char str1[10] = "Ariela", str2[] = "dana";
```

```
strcpy(str1, str2); // str1 must be longer than str2
```

str1	d	a	n	a	\0	a	\0	?	?	?
------	---	---	---	---	----	---	----	---	---	---

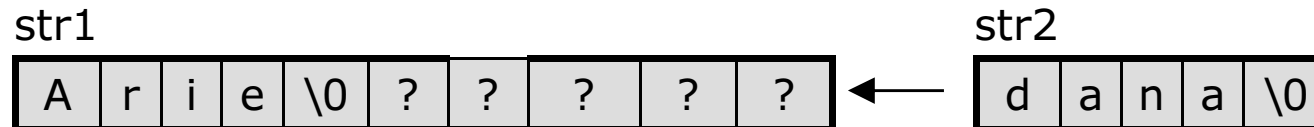
String Library

■ `char *strcat(char str1[], char str2[])`

Concatenating str2 to str1.

Comment: make sure that str1 is sufficiently long.

```
char str1[10] = "Arie", str2[] = "dana";  
strcat(str1, str2);
```



String Library

■ `char *strcat(char str1[], char str2[])`

Concatenating str2 to str1.

Comment: make sure that str1 is sufficiently long.

```
char str1[10] = "Arie", str2[] = "dana";
strcat(str1, str2);
```

str1

A	r	i	e	d	a	n	a	\0	?
---	---	---	---	---	---	---	---	----	---

str2

d	a	n	a	\0
---	---	---	---	----

```
char str1[10] = "Arie", str2[] = "daniela";
strcat(str1, str2);
```

str1

A	r	i	e	\0	?	?	?	?	?
---	---	---	---	----	---	---	---	---	---

str2

d	a	n	i	e	l	a	\0
---	---	---	---	---	---	---	----

String Library

■ `char *strcat(char str1[], char str2[])`

Concatenating str2 to str1.

Comment: make sure that str1 is sufficiently long.

```
char str1[10] = "Arie", str2[] = "dana";
strcat(str1, str2);
```

str1

A	r	i	e	d	a	n	a	\0	?
---	---	---	---	---	---	---	---	----	---

str2

d	a	n	a	\0
---	---	---	---	----

```
char str1[10] = "Arie", str2[] = "daniela";
strcat(str1, str2);
```

str1

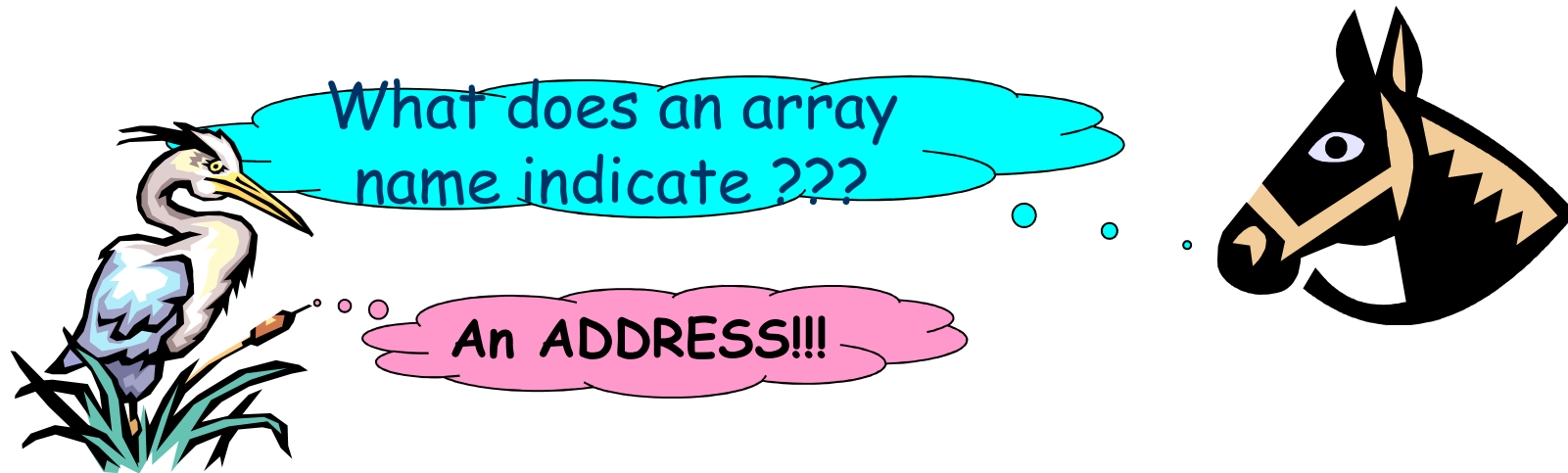
A	r	i	e	d	a	n	i	e	l
---	---	---	---	---	---	---	---	---	---

str2

d	a	n	i	e	l	a	\0
---	---	---	---	---	---	---	----

DANGEROUS!!!

Array as Parameter to Function



- When an array is passed as an argument to a function, **only its address** is sent.
- There are 2 ways to indicate types of (array) addresses:
 - `type[]` (examples: `char[]`, `int[]`, etc...)
 - `type *` (examples: `char *`, `int *`, etc...)

Array as Parameter to Function - Definition

sizeof(arr) is the number of bytes in arr (12)

BUT

sizeof(vector) is only 4 since only the address of arr is sent, and address is 4 bytes

```
void print_reverse(char *string);  
int sum_array(int vector[], int size);
```

```
void main()  
{  
    int sum, size, arr[]={1,2,3};  
    char str[]="sharon";  
    size = sizeof(arr)/sizeof(int);  
    sum = sum_array(arr, size);  
    print_reverse(str);  
}
```

```
void print_reverse(char *string)  
{  
    int i, size = strlen(string);  
    for(i=size-1; i>=0; i--)  
        printf("%c", string[i]);  
}  
int sum_array(int vector[], int size)  
{  
    int sum = 0, i;  
    for(i=0; i<size; i++)  
        sum += vector[i];  
    return sum;  
}
```

Array as Parameter - Stack

- An array address enables access to its elements using the operator [].
- With operator [] we can even change the values of the array, although the array is defined in main.
- Comment: It's not possible to change primitive types (char, int...) variables that are passed as arguments to a function.

Array as Parameter – Stack: Example

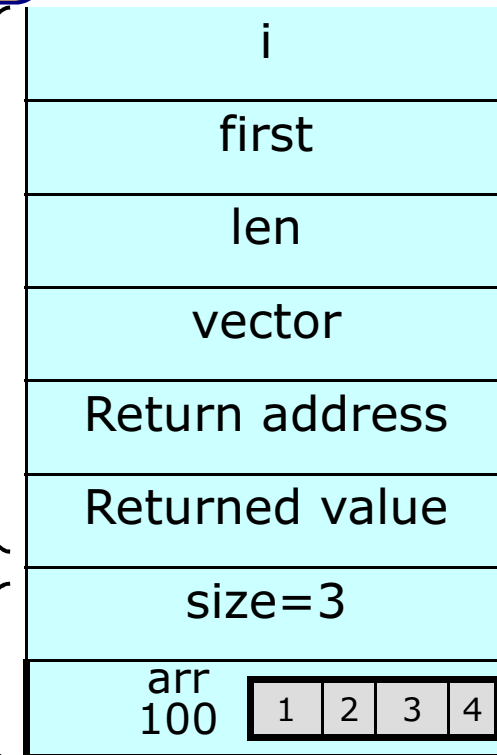
initialize gets arr, size and arr[0]. It assigns arr[0] to the first size elements of arr

```
void main()
{
    int arr[4]={1,2,3,4}, size=3;
    initialize(arr, size, arr[0]);
}
```

```
int initialize(int vector[], int len, int first)
{
    int i;
    for(i=1; i<len; i++)
        vector[i] = first;
    len = first = 8;
    return len;
}
```

initialize

main



Array as Parameter – Stack: Example

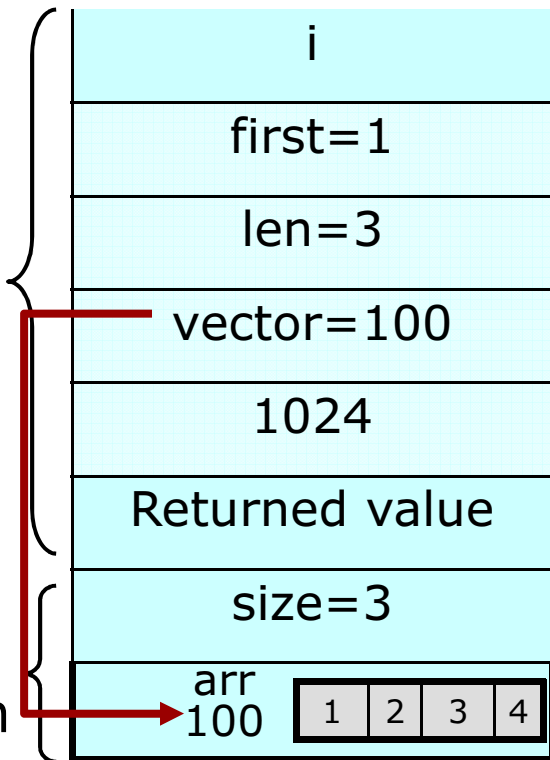
After passing of
arguments

```
void main()
{
    int arr[4]={1,2,3,4}, size=3;
    initialize(arr, size, arr[0]);
}
```

```
int initialize(int vector[], int len, int first)
{
    int i;
    for(i=1; i<len; i++)
        vector[i] = first;
    len = first = 8;
    return len;
}
```

initialize

main



Array as Parameter – Stack: Example

After initial execution

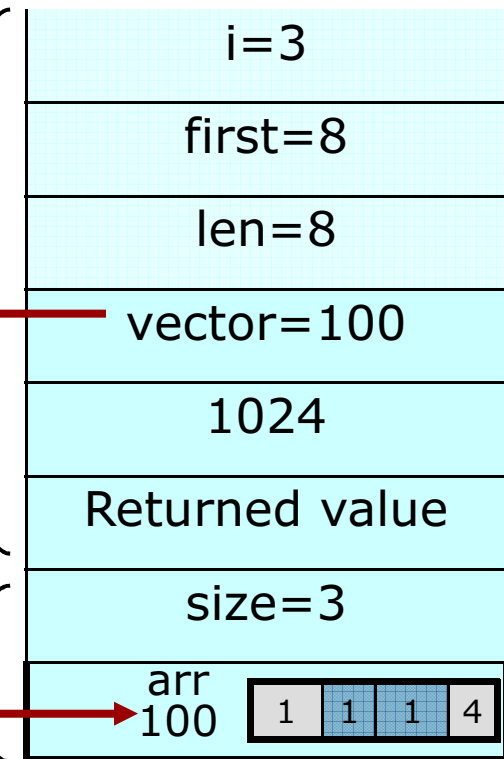
```
void main()
{
    int arr[4]={1,2,3,4}, size=3;
    initialize(arr, size, arr[0]);
}
```

```
int initialize(int vector[], int len, int first)
{
    int i;
    for(i=1; i<len; i++)
        vector[i] = first;
    len = first = 8;
    return len;
}
```

vector[i]:
100+i*sizeof(int) []

initialize

main



Array as Parameter – Stack: Example

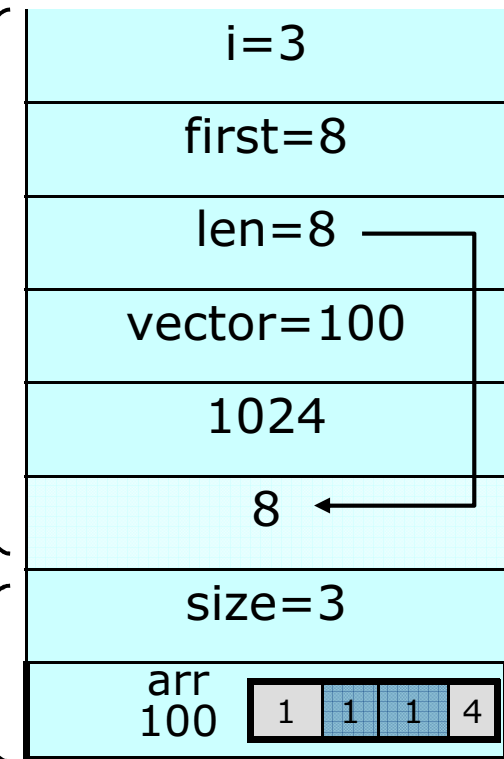
After returned value
updating

```
void main()
{
    int arr[4]={1,2,3,4}, size=3;
    initialize(arr, size, arr[0]);
}
```

```
int initialize(int vector[], int len, int first)
{
    int i;
    for(i=1; i<len; i++)
        vector[i] = first;
    len = first = 8;
    return len;
}
```

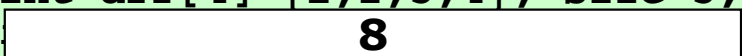
initialize

main



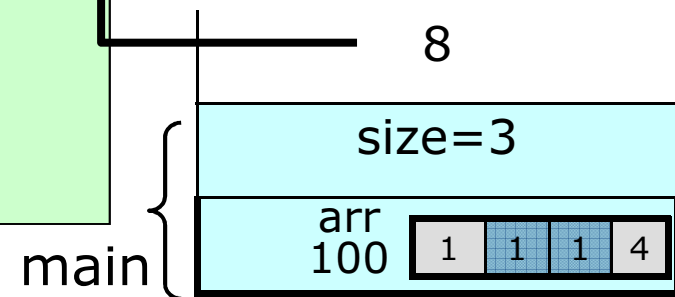
Array as Parameter – Stack: Example

After return of value

```
void main()  
{  
    int arr[4]={1,2,3,4}, size=3;  
      
}
```

```
int initialize(int vector[], int len, int first)  
{  
    int i;  
    for(i=1; i<len; i++)  
        vector[i] = first;  
    len = first = 8;  
    return len;  
}
```

Conclusion:
size wasn't changed
arr was changed



Call by Value

- Passing arguments by value: the arguments are copied to the local parameters of the function.
- Conclusion: if we want to pass an argument so as to change its value in the function, we can **not** pass it by value.

Call by Address

- Passing an array as argument: its address is passed by value (a local parameter stores the address). But, the array's elements are accessible by using the operator [].
- Conclusion: if we want to pass an argument in order to change it in the function, we should pass it by address.

2D-array as Parameter

- 2D array name is also an address. Its type is: `type[][]`.
- When defining a function that gets a 2D array, it's necessary to add the size of the second [] (columns).
- Explanation: let's describe what happens with access to an element by `[][]`:

```
int matrix[2][4];
```

```
matrix[1][2] = 4;
```

- `matrix` is a vector of two 1D vectors.
- `matrix[1]`: `matrix+1*sizeof(matrix[0])` – assume `matrix` is at address 100; then `matrix[1]` is at address 100+16.
- `(matrix[1])[2]`: `116+2*sizeof(int)`. The operator `[]` accesses the value of the address 124 (4 is assigned to this value).

2D-array as Parameter

- When passing an array to a function, only its address is passed.
- When accessing an array in the function with the following command:
matrix[1][2] = 4;

The compiler doesn't know to decode the term matrix[1], because its address is known but sizeof(matrix[0]) depends on 1D vector length.

```
void func(int mat[][]);  
void main()  
{  
    int matrix[2][4];  
    func(matrix);  
}  
void func(int mat[][])  
{  
    mat[1][2] = 4;  
}
```

matrix
address
100

?	?	?	?
?	?	?	?

mat 100

The compiler shouts that it doesn't know the size of each vector in mat!!!

2D-array as Parameter

- So note the correct declaration now.

```
void func(int mat[][4]);  
void main()  
{  
    int matrix[2][4];  
    func(matrix);  
}  
void func(int mat[][4])  
{  
    mat[1][2]=4;  
}
```

matrix
address
100

?	?	?	?
?	?	?	?

mat 100

**Now the compiler
knows that sizeof(mat[0])
is 4*4=16**