Topic 1 (Revision): Overview of C Language for Microprogrammed Embedded Systems and Memory Hierarchy Use in NDS

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

The C language

- Nowadays, the most commonly-used programming language for embedded systems

- Powerful and easy-to-use to express algorithmic steps
  - Only 32 reserved words

- Considered “high-level” assembly …
  - Low-level control of what the processor does
  - Efficient compilers available almost for every existing architecture
  - High efficiency of the produced code

- … but highly portable
  - From mainframes to microprocessors and micro-controllers

A C Language Tutorial

The examples have been extracted and adapted from:

Programming in C – A Tutorial

Brian Kernighan, Bell Labs

http://www.lysator.liu.se/c/bwk-tutor.html

"Introduction au langage C", Bernard Cassagne (CNRS),
(free on-line access and PDF in Moodle website of our course)

The program structure in C

```
main() {
        printf("hello, world\n");
}
```

printf is a standard library (stdio.h) function that format and prints texts to the standard output

Execution starts with the first statement of the main function

- One or more functions
  - At least, there is one main function
  - Different functions are called in order to do the work
- Functions are limited by: { }
- Statements end up in: ;

A functional program in C:
An example of variables, types and declarations

- This program sums three elements and returns the sum
  - *a, b, c*, and *sum* are *declared* as integer type variables
  - Variables names should not contain strange characters (a-z, A-Z, 0-9, _)

```
main() {
        int a, b, c, sum;
        a = 1;  b = 2;  c = 3;
        sum = a + b + c;
        printf("sum is %d", sum);
}
```

Basic assignments
a = 1; b = 2; c = 3;

Basic Types in C

- Variables should be declared before used (specifying their type)
- Basic Types
  - int — Integer of a word size
  - char — Character (or integer) of 1 byte size
  - float — Real number in simple precision floating point
  - double — Real number in double precision floating point

Constants in C

- '\n' new line
- '\t' tab
- '\b' backspace
- '\0' end
- '\\' character \

- 77 decimal
- 077 octal
- 0x77 hexadecimal

```
int a; char quest, newline, flags;
a = 1;
quest = '?';
newline = '\n';
flags = 077;
```

printf (stdio.h)

- %d formatted as decimal number
- %o formatted as octal number
- %x formatted as hexadecimal number
- %s it places a text string

```
main( ) {
        int n;
        n = 511;
        printf ("What is the value of %d in octal notation?", n);
        printf ("%s! %d decimal is %o octal\n", "Correct", n, n);
}
```

The format

The text to be formatted

Conditional statement
if, relation and logical operations

- Basic conditional statement
  - if (*expression*) *statement*;

- Two types of operations
  - Relational Operation: == != < > <= >=
  - Logical Operation: && || !

- Example: if (a < b && (a < 0 || a > 100))

```
Int c = 16;
if(c == 0x0010)
        printf(”c is equal to 16");
```

It evaluates if the expression is ≠0

false: = 0
true: ≠ 0

Basic Logic Expressions

- a == b — 1 if a equals to b
- a != b — 1 if a differs from b
- a > b — 1 if a is greater than b
- a < b — 1 if a is less than b
- a >= b — 1 if a is greater than/equal to b
- a < b — 1 if a is less than/equal to b
- a && b — 1 if a $\neq 0$ and b $\neq 0$
- a || b — 1 if a $\neq 0$ or b $\neq 0$

Conditional expression:
The *else* clause

- Each *if* statement might have an associate *else* clause
  - if (*expression*) *statement1*;
    else *statement2*;

Statement1 is executed if the *expression* is ≠ 0

- A different way
  - *expression1 ? statement1 : statement2*
    - It evaluates *expresion1*
    - If true (≠0), then evaluates *statement1*
    - If it is false (=0), then evaluates *statement2*

Conditional ternary operator

- As it is an statement, different if-then-else clauses can be nested
  - if (*expression1*) *if(expression2) statement2_1*;
    else *statement2_2*;

else clauses associate to the closest if statement

Blocks and the *else* clause

- Blocks, limited by ***{}***, are used delimit groups of statements
  - They are considered as a statement

```
if (a < b) {
        t = a;
        a = b;
        b = t;
}
```

A block instead of a single statement is executed when the condition is true

Arithmetic

- Arithmetic Operators: + - * / %

Modulus operator

- Characters can be treated as 8-bit integers

```
main( ) {
        char c = 'R';
        if(c % 2 == 0)
                printf("%c is an even char (%d) \n", c, c);
        else
                printf("%d is an odd char (%d) \n", c, c);
}
```

If c is an even character (e.g. 0, 2, 4…)

C characters:
They follow the ASCII code order

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Operator precedence

- Strict preference between operators and symbols
  - 1: Highest preference
  - 15: Least preference

- For equal preference, then from left to right on the same statement

	Category	Operator	Associativity
1	Postfix	() [] -> . ++ - -	Left to right
2	Unary	+ - ! ~ ++ - - (type) * & `sizeof`	Right to left
3	Multiplicative	* / %	Left to right
4	Additive	+ -	Left to right
5	Shift	<< >>	Left to right
6	Relational	< <= > >=	Left to right
7	Equality	== !=	Left to right
8	Bitwise AND	&	Left to right
9	Bitwise XOR	^	Left to right
10	Bitwise OR	|	Left to right
11	Logical AND	&&	Left to right
12	Logical OR	||	Left to right
13	Conditional	?:	Right to left
14	Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
15	Comma	,	Left to right

Operators precedence:
with and without parenthesis

- *(c = a + b) < 33*
  - It assigns to c the result of a+b
  - and compares it to 33

The **<** operator has higher priority than **=**

- *c = a + b < 33*
  - It performs a+b
  - and compares it to 33
  - and assigns the result of the comparison operation to c

Basic loop
while, assignment expression

- While loop
  - while (*expression*) *statement*;
- Behavior:
  1. It evaluates the expression
  2. If it is true (≠0) then it executes the statement and returns to the step 1.

```
main() {
        int c = 0;
        while(c < 16) {
                printf(“%d ”, c);
                c = c+1;
        }
}
```

Expressiveness of the C language

- More compact does not necessarily means superior
  - It is easier to understand the first expression
  - But advanced users rather use the second one because it is more compact

- It is said that C is an expressive programming language because
  - It is mainly composed of expressions
  - It is capable of expressing complex behaviors in few lines

```
Int c = 0;
while(c <16) printf(“%d “, c++);
```

Increments and Decrements

- *++n* is equivalent to n = n + 1
- *--n* is equivalent to n = n – 1
  - It is more clear, specially if n is complicated

- The post-fix expression returns the value before the increment or decrement expression

```
int a, b, x, y;
a = b = 5;
x = a++;
y = ++b;
```

```
a == 6      b == 6
x == 5      y == 6
```

Arrays (vectors)

- Arrays can be declared and used in the following way:

```
int x[10]; int y[10][10];
x[0] = 1; x[8] = x[9] = 5;
y[0][0] = y[1][0] = 8;
```

- At declaration time, the number of elements must be stated
  - It has to be constant
  - Index goes from 0..N-1
  - They can be N-dimensional

Arrays example

- Counting upper-case characters in a sentence

```
main( ) {
        int n = 0, count = 0;
        char line[] = "This Is My Line \n";
        while(line[n] != '\n') {
                if((line[n] >= 'A') && (line[n] <= 'Z'))
                        count++;
                n++;
        }
}
```

The array size has to be known at compilation time (Length of the sentence!!)

The *for* loop

- This is a special case of the loop while
  - for(begin; expression; increment)
        statement;

- It is equivalent to
  - initialization;
    while(expression) {
        statement;
        increment;
    }

for loop examples

- Character arrays copy

```
for(i=0; (t[i]=s[i]) != '\0'; i++);
```

- Array elements sum

```
sum = 0;
for( i=0; i<n; i++) sum = sum + array[i];
```

- Bi-dimensional array initialization

```
for( i=0; i<n; i++ )
        for( j=0; j<m; j++ )
                array[i][j] = 0;
```

It does not need composed statements (blocks with '{}')

Functions
Definitions and Declarations

- Function definition

```
int min(int a, int b) {
        return a < b ? a : b;
}
```

A ***void*** value can be returned = nothing is returned

- Function declaration

```
int min(int a, int b);
...
i = min(10, 6);
...
```

It declares argument types and return type

The compiler checks that the function is correctly used

Implicit Declaration

- Functions have to be always declared

- If they are not explicitly declared, an implicit declaration is generated:
  - *int function();*

Unknown arguments

- The implicit declaration prevents the compiler from checking the function arguments

The *switch* statement

- Some nested *if/else* can be more efficiently implemented using the *switch* statement:

```
switch(c) {
case 'a':
            aflag++;
            break;
case 'b':
            bflag++;
            break;
case 'c':
            cflag++;
            break;
default:
            printf("%c?\n", c);
}
```

```
switch(c) {
case 'a': case 'A':
            aflag++;
            break;
case 'b': case 'B':
            bflag++;
            break;
            ...
}
```

break and *continue* statements

- The switch statement uses *break;* to exit
  - Similarly, using *break* in a while or for loop results in the loop being finished

```
for(;;) {
        ...
        if (ExitCondition) break;
        ...
}
```

- The *continue* statement is used to step into the next iteration without executing the current one

Variables initialization

- The initial value is provided during the variable definition
  - Initializations can be more efficient than assignments

```
int x = 20; int a = 'k'; char c = 0177;
int y[] = { 1, 2, 3, 4 };
int* p = &y[1];
char* msg = "Format error";
char buf[] = "Insert your data";
```

msg and *buf* are not equivalent. The *msg* string cannot be modified. Why not?

The pre-processor

- Pre-processor: compilation previous step
  - Directives are interpreted
  - Directives starts with #

```
#include <stdio.h>

FILE* f;
```

Includes the *stdio.h* file (i.e., declarations from the standard C library)

```
#define MAXBUFSIZE 25

int buf[MAXBUFSIZE];
```

Macro-constant

- Comments are placed in between /* */
  - In C99 it is possible to also use: // To the end of the line

```
void strcopy(char s1[], char s2[]) { /* copy s1 to s2 */
        int i;
        for(i = 0; (s2[i] = s1[i]) != '\0'; i++);
}
```

Comments are eliminated by the pre-processor so the compiler ignores them

Bitwise operations and Special Assignments

- Operations are evaluated bit by bit among the operands
  - a & b   AND
  - a | b   OR (or inclusive)
  - a ^ b   XOR (or exclusive)
  - ~a   1's Complement
  - !   Logic negation
  - a << b   Shift b bits to the left
  - a >> b   Shift b bits to the right

- Special assignments
  - a -= b   a = a - b
  - a &= b   a = a & b
  - a <<= b   a = a << b
  - …

Floating point

- Calculating the average of a real simple precision vector

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
        sum =+ y[i];
avg = sum/n;
```

Different numeric types can be mixed. There is a implicit type promotion

goto statement and labels

- Normally, the use of ***goto*** statement is not required (and **<u>VERY STRONGLY</u>** discouraged!)
  - Beginners might fall into abusing of goto statement, making the code hard to understand

```
loopA:
        ...
        if (a > b) goto loopA;
        ...
```

- There is always a structured way to program the same functionality, while being much clearer to follow for any additional user or maintainer of the code

Automatic and Extern variables

- Automatic variables are local to the block

```
if (a < b) {
        int i;
        ...
}
```

They disappear at the end of the block

- Extern variables are always available

```
int i;
void f() {
        ...
}
```

Extern variables are automatically initialized to 0

```
void g() {
        extern int i;
        ...
}
```

The i variable is available to different functions, even from different files

Advanced data management in C: Pointers and addresses

- Pointer: Object containing memory address of an object

Except void*

- Type of pointer: determined by the type of pointed object

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

& retrieves the memory address of the object

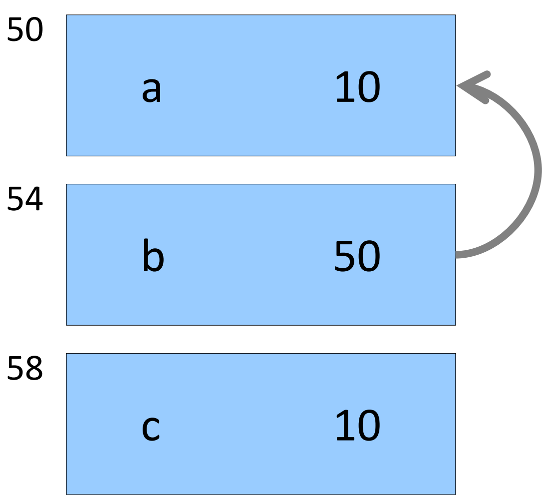
* retrieves the object pointed by the pointer

Pointers

- Pointer = the address of ...
  - A variable address is retrieved by using &
  - The value of an object pointed by a pointer is retrieved by *

```
int a, *b, c;
a = 10;
b = &a;
c = *b;
```

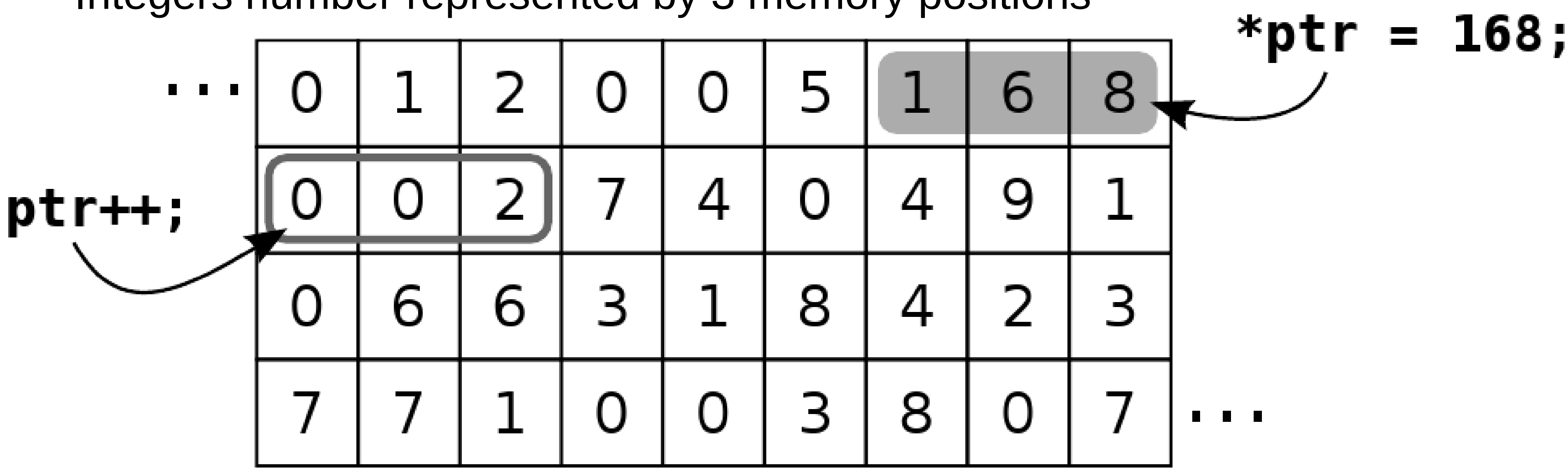
Pointer to an integer



Advanced data Management in C: Pointer arithmetic

- Arithmetic operations over pointers modify the object referred by the pointer
  - Incrementing/decrementing the pointer means referring to the respective next/previous element: *ptr++*, *ptr--*, *ptr=ptr+2*, *ptr=ptr-2* ...

Integers number represented by 3 memory positions

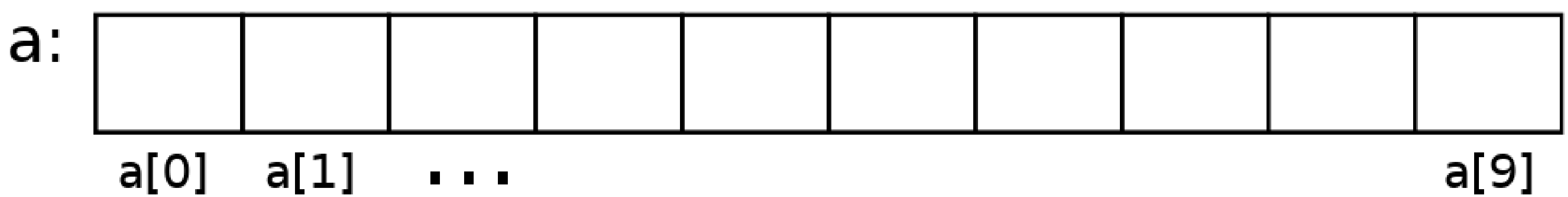


Advanced data management in C: Arrays (or vectors)

- Arrays are homogeneous sequences
  - Index from 0 to N-1

```
int a[10];
```

- They are stored in consecutive positions in memory



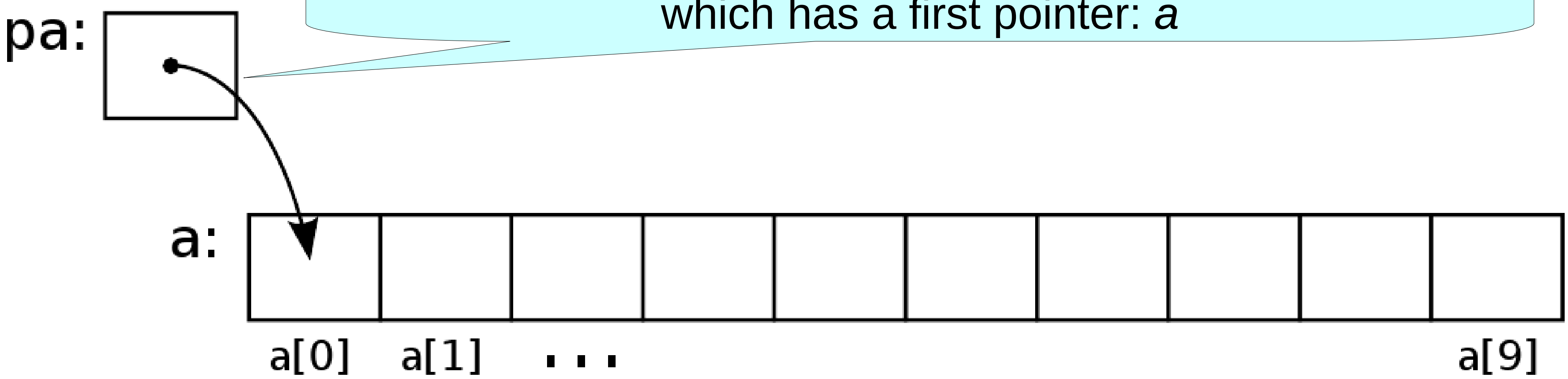
Advanced data management in C: Arrays and pointers

- Pointers and arrays representations are closely related

```
int a[10];
int* pa;
pa = &a[0];
```

Equivalent to pa = a

*int *pa* points to the first element of the array *int a[10],* which has a first pointer: *a*

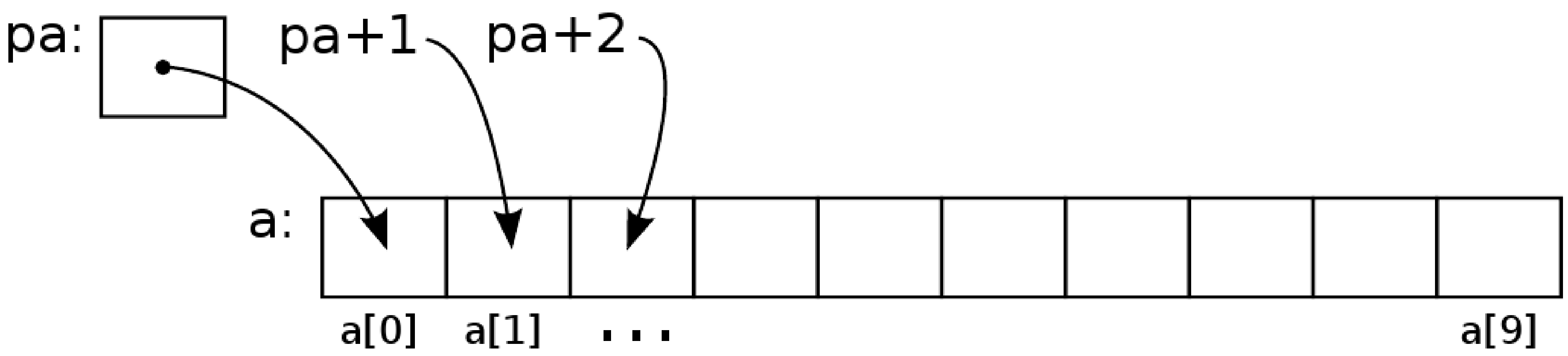


Advanced data management in C:
Indexing arrays and pointer arithmetic

- Indexing arrays/matrices is like moving positions of pointers

```
pa + 1 == &a[1]
pa + 2 == &a[2]
pa[i] == *(pa + i) == a[i]
```

I am just increasing positions in the array!



Pointers and arrays
Pointer Arithmetic

- Using the array name in an expression is equivalent to use the address of the first element of the array

```
char *y;
char x[100];

y = &x[0];
y = x;
```

Equivalent

```
char buf[100];
extern void f(char* p);

f(buf);
```

Arrays are passed to functions as a pointer to the first element

- Pointer arithmetic can be used to access the array elements

```
*(y+1) == x[1];
*(y+i) == x[i];
```

```
y = &x[0]; y++;
*y == x[1];
```

Pointers and arrays
A Pointer Arithmetic Example

- The name of an array, used in an expression, holds the address of the array first element

```
int length(char s[]) {
        int n;
        for(n=0; s[n] != '\0';)
                n++;
        return n;
}
```

```
int length(char* s) {
        int n;
        for(n=0; *s != '\0'; s++)
                n++;
        return n;
}
```

Nowadays, the code generated by compilers are equally efficient in all these situations.

```
int length(char* s) {
        int n;
        for(n=0; *s++!='\0'; n++);
        return n;
}
```

Passing Arguments
Passing values and references

- In C, arguments are passed by value
  - Passing the pointer is how references are passed to functions

```
void flip(int *x, int* y) {
        int temp;
        temp = *x; *x = *y;
        *y = temp;
}
```

```
int a, b;
a = 1; b = 2;
flip(&a, &b);
// a == 2 , b == 1
```

- Which value is printed by this fragment?

```
void fill0(int a[]) {
        a[0] = 10;
}
```

```
int buf[10];
fill0(buf);
printf("%d", buf[0]);
```

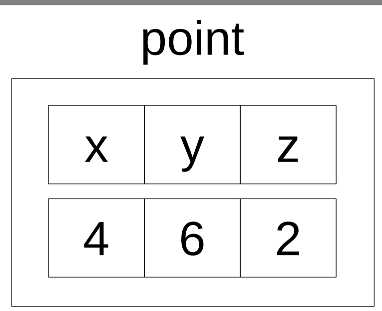
Data structures

- Structures are used to create new complex types

Example 1:

Coordinates in 3D space

```
struct coordinates {
            int    x;
            int    y;
            int    z;
};
struct coordinates point;
point.x = 4;
point.y = 6;
point.z = 2;
```

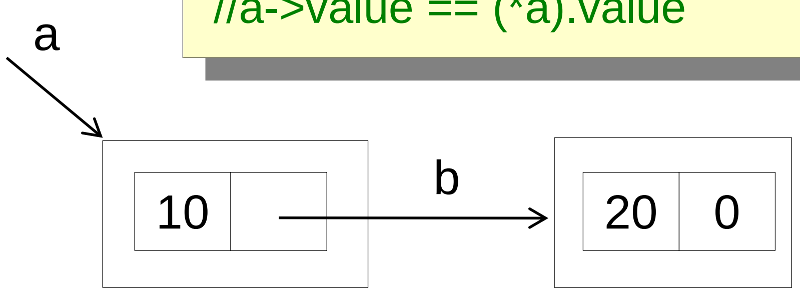


Example 2:

Linked list

```
Struct element {
            int value;
            int *Next;
};

struct element *a, *b;
a->value = 10;
a.Next = a;
b->value = 20;
b.Next = 0;
//a->value == (*a).value
```



Communicating data between parts of the programs: Passing arguments

- Two ways to pass arguments in *C* functions
  - Pass by value
  - Pass by reference

Pass by value

A copy of each argument is passed to the function

```
int sum(int x, int y, int z){
    z = x + y ;
    printf (“z == %d”, z);
}

//main function
int main(){
    int a=10; int b=5; int c=0;
    sum(a, b, c);
    //a, b and c are passed by value
    printf (“c == %d”, c);
}
```

<u>Screen output</u>

z == 15

c == 0

Pass by reference

Pointer argument are used to pass the adress of a variable

```
int sum(int x, int y, int *z){
    *z = x + y ;
    printf (“*z == %d”, *z);
}
//main function
int main(){
    int a=10; int b=5; int c=0;
    sum(a, b, &c);
    //a and b are passed by value
    //c is passed by reference
    printf (“c == %d”, c);
}
```

<u>Screen output</u>

z == 15

c == 15

Static vs. dynamic memory management

- Two ways to allocate an array in memory
  - Static allocation:
    - The size of the memory to be allocated is known before execution

      int array[10];
  - Dynamic allocation
    - Useful when the size is known only at run-time of the application

      *void *malloc(size_t size) ; /*similar to <u>new int [ ]</u> in c++*/*
    - Dynamic allocated memory must be freed once it not used anymore

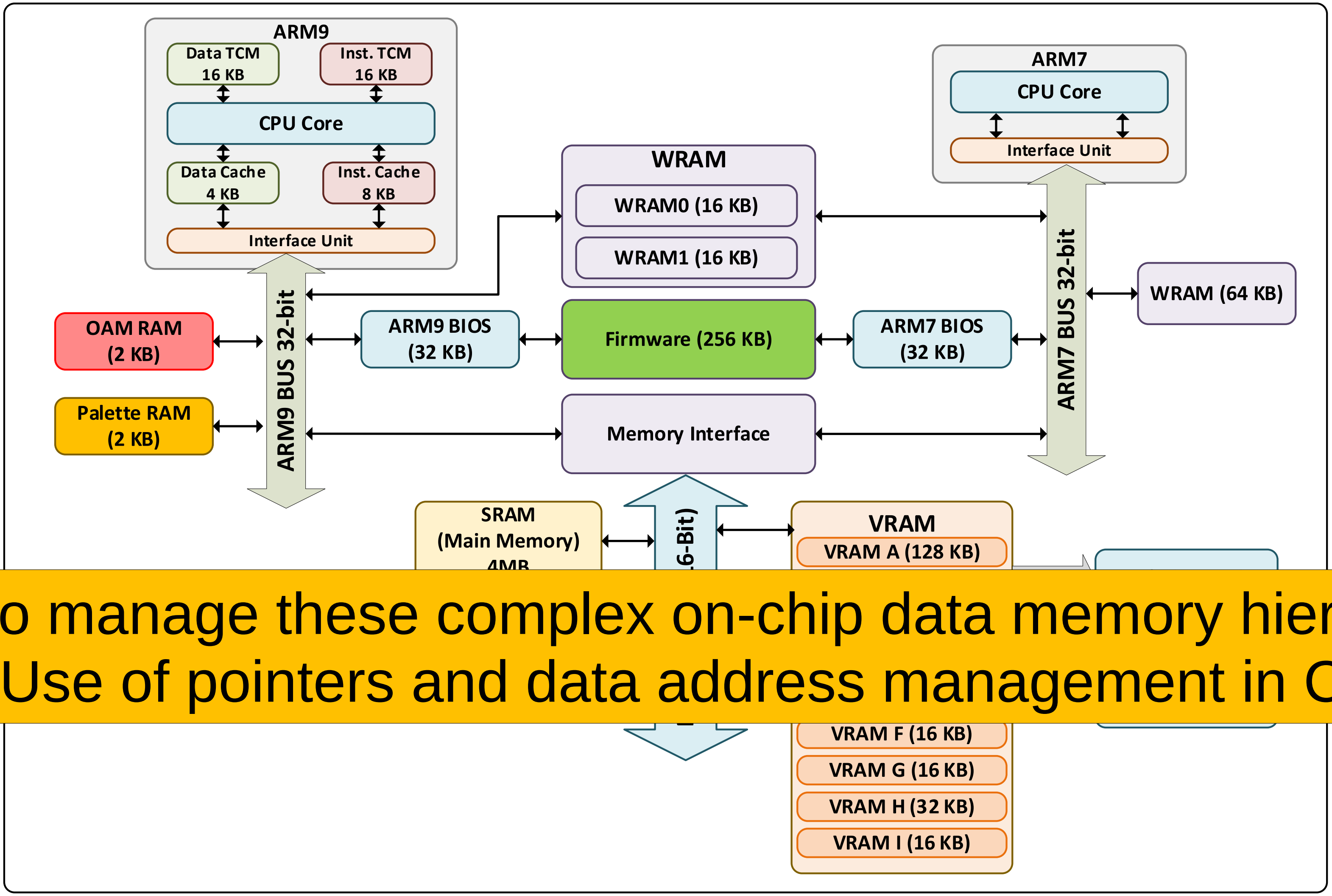
      *void free(void *pointer); /*similar to <u>delete [ ] pointer</u> in c++*/*

```
//Static memory
int StaticArray[10];

//Dynamic memory
int *DynamicArray; //Static pointer
…
DynamicArray=malloc( n*sizeof(int) ); //n=size of the array
ProcessArray(DynamicArray);
free(DynamicArray); //de-allocate memory
```

Practical Work 3: Advanced C Programming on the Nintendo DS

- Acessing multiple memory locations: complex memory hierarchy
  - ARM9 SW-controlled memories: D-/I- tightly coupled memories (TCMs)



How to manage these complex on-chip data memory hierarchy? Use of pointers and data address management in C!

Practical Work 3: Advanced C Programming on the Nintendo DS

- Matrix representations can be declared in 2 different ways

NDS Visualization

Algebraic

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Static Declaration

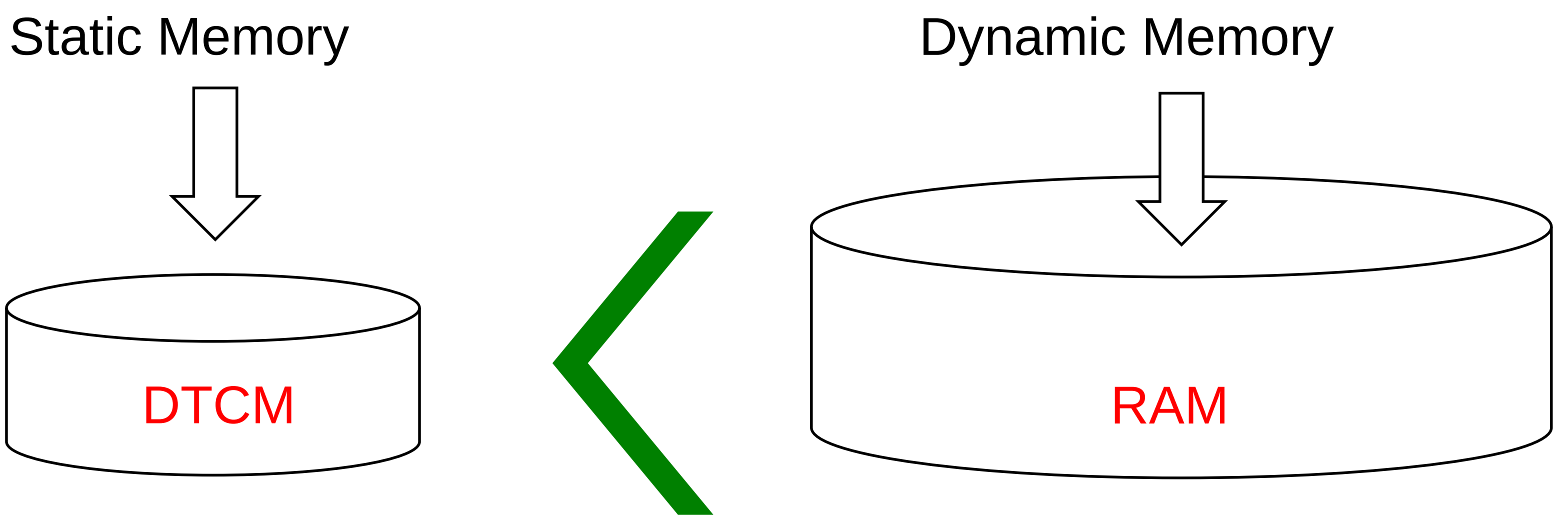
```c
//Matrix with 9 components preinitialized
int myMatrix[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Dynamic Declaration

```c
//Matrix with 9 components
int *myMatrix2 = malloc(sizeof(int)*9);
//Initialization
for(i=0; i<9;i++) myMatrix2[i] = i+1;
```

Practical Work 3: Advanced C Programming on the Nintendo DS

- The matrices are then stored in different parts of the NDS memory hierarchy



- Can we run out of memory in these two cases below? How?
  - Automatic static variables
  - Dynamic Memory

Practical Work 3: Advanced C Programming on the Nintendo DS

- Matrix representation
  - Combining structures and dynamic memory

Algebraic

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Dynamic Declaration

```
//3x3 Matrix
tMatrix *myDynamicMatrix = malloc(sizeof(tMat
myDynamicMatrix->cols = 3;
myDynamicMatrix->rows = 3;
myDynamicMatrix->mat = malloc(sizeof(int)*3*3
//Initialization
for(i=0;i<9;i++)
    myDynamicMatrix->mat[i] = i+1;
```

Struct

```
typedef struct{
    int *mat;
    int rows;
    int cols;
}tMatrix;
```

NDS Visualization

Practical Work 3: Advanced C Programming on the Nintendo DS

- Exercises (and homework)
  - Exercise 1 – Separating function prototypes from implementations
  - Exercise 2 – Displaying matrices on NDS console
  - Exercise 3 – Initialization of matrices
  - Exercise 4 – Summation of arrays and matrices
  - Exercise 5 – Vector sorting
  - Exercise 6 – Matrices multiplication
  - *Exercise 7 – Run-time resources errors on the NDS
  - *Exercise 8 – Managing the different types of memory resources in the NDS
  - *Exercise 9 – Allocating/deallocating memory
  - *Exercise 10 – Understanding how arguments are passed between NDS functions

* Data management for the NDS

Questions?

Let’s manage the NDS using the C language