Applied Programming

Arrays and Pointers

in C

More details in: G. Semeraro, “Numerical Programming in C”, chapter 4, (in Mycourses)

Arrays and Pointers

- In C, ***arrays of primitive types*** are implemented as blocks of contiguous memory.

```
int a[10];       /* 1D array of 10 integers */
double A[3][5]; /* 2D array of floats ("3 rows, 5 columns") */
```

- Arrays are can be initialized as follows

```
int a[] ={1,2,6,7,8}; /* 1D array of 5 integers */
```

- Array elements can be accesses by ***indexing*** and **pointer arithmetic**

Ex: The following are all equivalent `A[i][j]`

```
*( A[i] + j )
*( (A+i) )[j]
*( (*(A+i)) + j )
*(&A[0][0]+ 5*i+j )
```

The compiler uses "a storage mapping function" to convert the indices to a memory address.

Arrays and Pointers

- In C, the **name of an array** is a pointer to the memory address where the first array value is stored

```
a same as &a[0]       /* 1D array */
A same as &A[0][0] /* 2D array */
```

- In our previous example, the *storage mapping function* for the 2D array `A[3][5]` is

```
s(i,j) = 5*i+j
```

- Notes:
  - The storage mapping function is used by the compiler to generate object code to access the correct array element in memory. The compiler *only needs* to know the *last dimension of the array* (e.g., 5) to generate the *correct memory offset*.
  - The compiler uses pointer arithmetic.

  If `ptr=&A[0][0]` then `A[2][3]` is `*( ptr + 5*(2)+(3) )`

Memory Storage of 2D Arrays

- C stores 2D-arrays in ***row-major form***, (as in Java and C++). Some languages, e.g., FORTRAN, use column-major form.

Example: The 2D array **`A[3][3]`** in memory.

2D array

1	2	3
4	5	6
7	8	9

Memory storage

1	2	3	4	5	6	7	8	9

Warning: When calling FORTRAN code (*e.g.*, optimized linear algebra routines) from C code this must be taken into account !

Using Arrays

- The most common use of arrays in numerical computations is to **represent Matrices**.
- A straight forward representation of a “matrix” in C is using a “fixed” 2D array

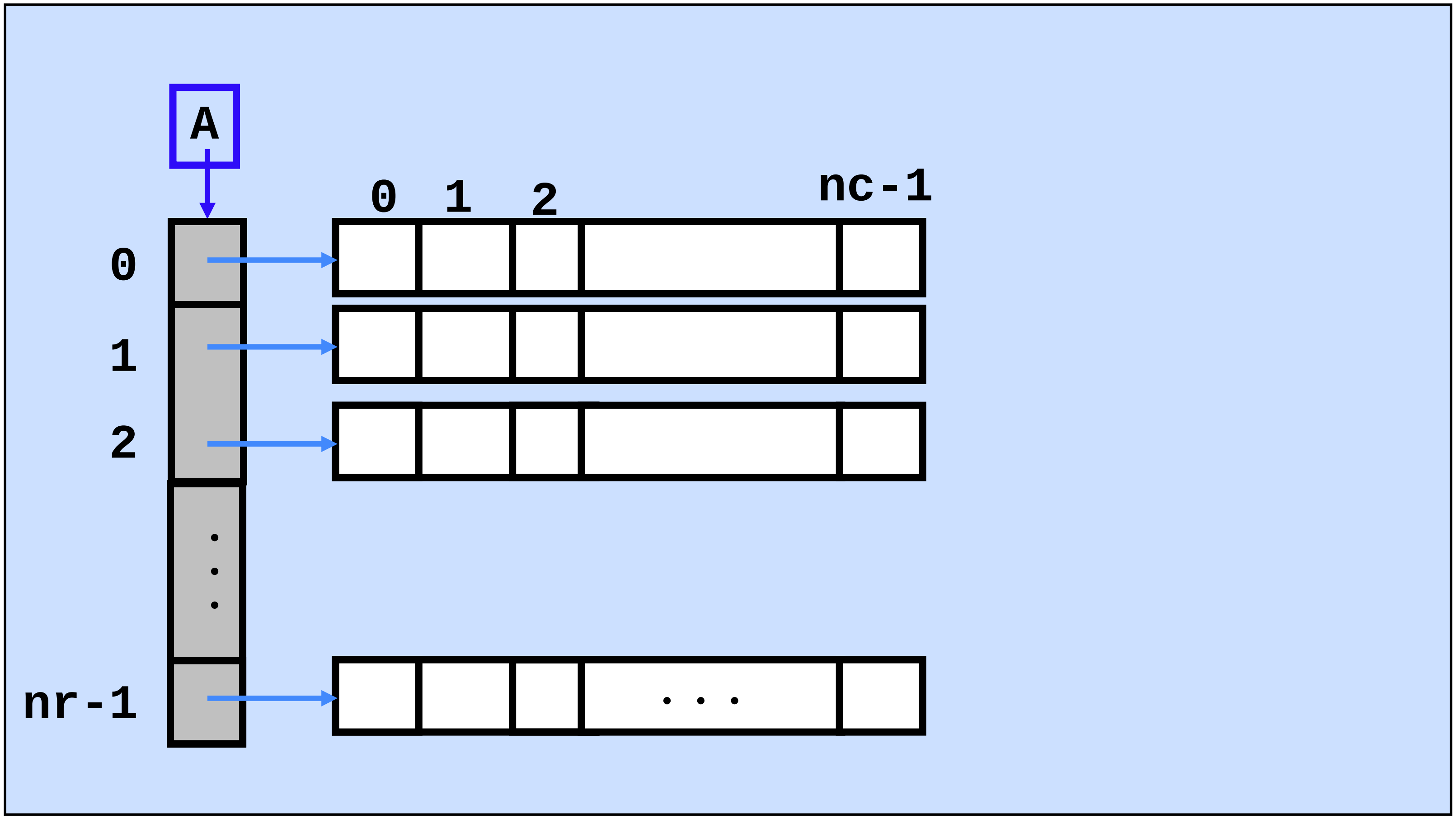
```
#define N_ROWS (20)
#define N_COLS (40)
double C[N_ROWS][N_COLS];
```

- Practical Limitation

  Cannot use 2D arrays in function calls unless we know dimensions *a priori* (and in practice we almost never do !)

  How to we address this limitation ?

Dynamically Allocated Arrays



Dynamically Allocated 2D Array in Memory (“row-major” form)

Dynamically Allocated Arrays

- A better way to represent a matrix in C is as pointers to (arrays of) pointers
- The size of the matrix neither needs to be known a priori nor be coded as a constant.

```
#include <stdlib.h>     /* for calloc and malloc */
typedef double MatElement;
.   .   .
int i, /* row index counter    */
nr,nc; /* # rows and # columns */

MatElement **A = NULL; /* declare matrix A */
. . .
/* Allocate space for array of pointers */
A = malloc( nr * sizeof(MatElement *));
/* Allocate space for matrix elements and set to zero */
for (i=0; i<nr; i++)
  A[i] = calloc( nc, sizeof(MatElement) );
```

Representing (Dense) Matrices

- The space ***must be*** allocated in **row-major form** to be able to access its elements using the ***standard indexing*** **syntax `A[i][j]`**

```
#include <stdlib.h>   /* for malloc      */
typedef double MatElement;
int        i,j,nr,rc;
MatElement **A = NULL;      /* declare matrix */
. . . .
A = malloc( nr*sizeof(MatElement *) );
for (i=0; i<nr; i++)
  A[i] = malloc( nc*sizeof(MatElement) );
  . . . .
for (i=0; i<nr; i++) /* Initialize to random */
  for (j=0; i<nc; j++)
    A[i][j] = rand();
```

Releasing Matrix Space

- To release the allocated memory space *"undo" the allocation operations in reverse order.*

```
void free_matrix_space(MatElement **A, int m) {
  int i;                          /* row counter */
  for (i=0; i<m; i++)
    free(A[i]);   /* free rows   */
    A[i] = NULL;  /* "grounding" pointers */
  free(A); /* Then column vector of row pointers */
  A = NULL;       /* grounding */
}
```

Warning: Doing `free` A will not work !

Efficient Matrix Allocation

- It is *more efficient* to *allocate all memory space at once* as shown below

```
#include <stdlib.h>     /* for calloc and malloc */
typedef double MatElement;
. . .
int i,              /* row index counter    */
nr,nc;              /* # rows and # columns */
MatElement *ptr = NULL;  /* pointer to rows       */
MatElement **A = NULL;   /* ptr to matrix        */
. . .
/* Allocate array of row pointers */
A = malloc(nr * sizeof(MatElement *) );
/* Allocate all matrix at once */
ptr = calloc( nr * nc , sizeof(MatElement));
/* Set array of row pointers properly */
for (i=0;i<nr;i++)
   A[i] = ptr + nc*i;
```

Efficient Matrix Allocation

- You can release the memory of an array allocated all at once as follows:

```
void free_all_matrix (MatElement **A) {
  free(A[0]);  /* free array of elements     */
  A[0] = NULL;
  free(A);     /* free array of ptrs to rows */
  A = NULL;
}
```

Note: We do not need to pass the size of the array for deallocation (this is not the case if we allocate space row by row).

Example: row_major.c

- Allocate using row by row: A [6][3]
- Allocate all at once: C [6][3]

Why does the row by row allocation take more memory?

Elements of A (row by row allocation)

1	(10000)	2	(10004)	3	(10008)
4	(10032)	5	(10036)	6	(10040)
7	(10064)	8	(10068)	9	(10072)
10	(10096)	11	(10100)	12	(10104)
13	(10128)	14	(10132)	15	(10136)
16	(10160)	17	(10164)	18	(10168)

Total bytes: 172

Elements of C (all at once allocation)

1	(10000)	2	(10004)	3	(10008)
4	(10012)	5	(10016)	6	(10020)
7	(10024)	8	(10028)	9	(10032)
10	(10036)	11	(10040)	12	(10044)
13	(10048)	14	(10052)	15	(10056)
16	(10060)	17	(10064)	18	(10068)

Total bytes: 72

(Memory address in parenthesis)

Example: Changing Indexing

- C is a very flexible programming language so it allows you to do many things (you are in charge)

- In most *matrix computations algorithms indexing starts at 1*, *e.g.*, A[1][1] is the first element of matrix A. That is also how it is in **FORTRAN** (**FOR**mula **TRAN**slator)

- Here we show an unorthodox way to “change indexing” of matrices so that it starts at 1 (instead of 0 as is the default in **C** 2d arrays)

Adjusting Subscript Range

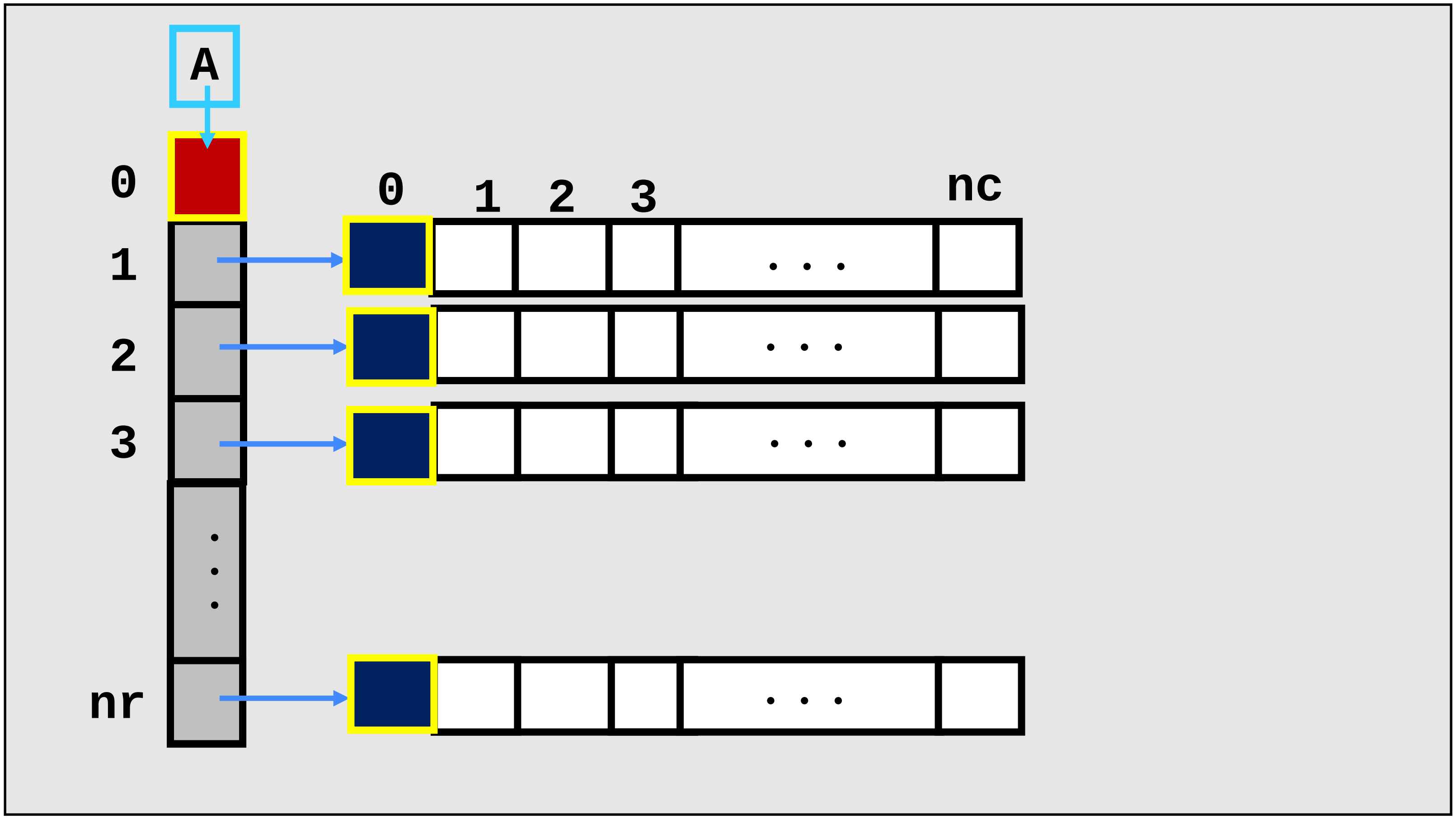
- Most mathematical algorithms use subscripts that start at 1 not at 0

  Q: How do we handle that in C ?

  A: Offset the pointers

```
typedef double MatElement;
MatElement **matrix1_simpleAlloc(int m, int n) {
   int i;
   MatElement **A = NULL;
   A = malloc(m * sizeof(MatElement*)); /*m x n matrix */
   --A;                /* offset row pointer back by one */
   for (i=1; i<=m; ++i) {
      A[i] = calloc( n, sizeof(MatElement));
      --A[i];          /* offset col pointer back by one */
      } /* End for i*/
   return(A); } /* End matrix1_simpleAlloc */
```

Dense Matrix: Dynamically Allocated



Dynamically Allocated Matrix with offset

Adjusting Subscript Range

- The following sample code illustrates the use of *subscript adjusted matrices*

```
typedef double MatElement;
MatElement **A;
   .. ..

   A = matrix1_simpleAlloc(m,n); /* offset mxn matrix */
   for (i=1; i<=m; i++)
     for (j=1; j<=n; j++)
         A[i][j] = ....
  .. ..
  matrix1_simpleFree(A, m, n) /* how ? */
```

- How do we *de-allocate* the memory space ?

Releasing Matrix Space

- To release the allocated memory space we need to point to the right location

```
void matrix1_simpleFree(MatElement **A, int m, int n)
   {
   int i;
   for (i=1; i<=m; ++i)
      {
      /* Return the col pointer BACK to the
         original then FREE */
      free(++A[i]);
      A[i] = NULL;
      }
   /* Return the row pointer BACK to the
      original, THEN free */
   free(++A);
   A = NULL;
} /* End matrix1_simmpleFree */
```

More efficient version

- The following version with allocation of matrix space all at once is more efficient

```
MatElement **matrix1_cleverAlloc(int m, int n) {
   int i;                     /* row index counter    */
   MatElement *ptr=NULL;      /* pointer to rows       */
   MatElement **A=NULL;       /* ptr to matrix    */

   /* Allocate array of row pointers and shift back */
   A = malloc(m * sizeof(MatElement *) );
   --A;
   /* Allocate all matrix data at once */
   ptr = calloc( m * n, sizeof(MatElement));
      /* Set array of row pointers properly */
   for (i = 1; i <= m; ++i)
      {
         A[i] = ptr + (n*(i-1)) -1;
      }
   return(A);
   }
```

More efficient Version

- To release the allocated memory space need to point to the right location

```
void matrix1_cleverFree (MatElement **A) {
  free(A[1]+1); /* base address of array elements */
  A[1] = Null;
  free(A+1);
  A = NULL;
}
```

- Note that, in this case, deallocation is simple, we don’t need the dimensions of A.

Summary: Arrays and Matrices in C

- ***Dense Matrices*** are usually declared as ***arrays of pointers to pointers*** (to the element type of interest) and allocated dynamically.
- **Vectors** (1D arrays) are naturally represented as a ***pointers*** (to the element type of interest) and allocated dynamically as well.
- Regular *matrix indexing* can be used to access the elements of a matrix.
- *Matrices* ***in C*** *are stored in* ***row-major*** **form**. Therefore to use indexing we need to dynamically allocate space also in “row-major form”

Strings

- Strings: (not a data type)
  - **Array of characters** terminated by the **sentinel** `\0`
  - Constant strings enclosed with doubt quotes:
    - The string : `“A”` `[65][\0]`
    - The character : `‘A’` `[65]`

Examples:

```
/* declare and initialize a string */
char[] course=“CMPE-380”;
course[2]==‘P’;  /* is True */
```

- The standard library `<string.h>` has functions to operate on strings, such as:

  `strcpy, strcmp, strcat, strstr, strchr`

Comparing Strings

- **You can't compare strings directly in C**
  - Only character by character

- Given:

```
char *str1 = "Some Text1";
char *str2 = "Some Text2";
```

- Consider: **if (str1 == str2)**
  - NEVER true, only comparing POINTERS str1 and str2

- Consider: **if (*str1 == *str2)**
  - ALWAYS true in this case
  - Only comparing the FIRST CHARACTERS in str1 and str2

- Use the function **strcmp()**

`strcmp()`

```
int strcmp(const char *s1, const char *s2);
```

- s1, s2 - ***strings to be compared,*** ***byte by byte***
- Returns –
  - 0 - if strings are identical
  - Negative - if, character by character, s1 is less than s2
  - Positive - if, character by character. s1 is greater than s2

How it works:

strcmp("abc", "ABC") is positive

because “a” (0x61) is greater than “A” (0x41)

Use: if **(!strcmp(s1, s2)**) { printf(“**Identical**”);}

strcpy()

```
char *strcpy(char *dest, const char *src);
```

- Copies **src** into **dest**
- byte by byte until 0x00 in src is reached
- ***Assumes*** dest is big enough (dangerous)
- Returns – Pointer to dest

How it works:

```
char dest [20];
strcpy(dest, "ABC");
```

dest [0] contains “A”

dest [1] contains “B”

dest [2] contains “C”

dest [3] contains 0x00

strncpy()

```
char *strncpy(char *dest, const char *src , int num);
```

- Copies **src** into **dest**
- byte by byte until 0x00 in src is reached
- Will not copy more than num bytes (safer)
  - ***Won't overflow the destination buffer!***
- Returns – Pointer to dest

How it works:

char dest [20];

strncpy(dest, "ABC“, 20);

dest [0] contains “A”

dest [1] contains “B”

dest [2] contains “C”

dest [3] contains 0x00

`strcat()`

```
char *strcat(char *dest, const char *src);
```

- Appends **src** to the **END of dest**
- byte by byte until 0x00 is src is reached
- Assumes dest is big enough
- Returns – Pointer to dest

How it works:

```
char dest [20];
strcpy(dest, "abc");
strcat(dest, "123");
```

results in dest containing "abc123"

strncat()

```
char *strcat(char *dest, const char *src, int num);
```

- Appends **src** to the **END of dest**
- byte by byte until 0x00 is src is reached
- Will not copy more than num bytes (safer)
- Returns – Pointer to dest

How it works:

char dest [20];

strcpy(dest, “abc“, 20);

strcat(dest, “123“, 20);

results in dest containing “abc123”

Problem 1

Crashes when I use "poly" in main.
What's **wrong?**

main.c:

```
polynomial poly;
createPoly(poly, 10);
poly.polyCoef [0] = 1;          /* crash */
.....
void createPoly(polynomial *p, int n){
    p = malloc(sizeof(polynomial));
    p->polyCoef = malloc(sizeof(complex) * n);
}
```

Error checking removed for clarity

Problem 1

- Addresses for example only

polynomial poly — Creates storage @100

createPoly(poly, 10); — passes the value p= 100, n= 10

createPoly(polynomial *p, int n)

p = malloc(); — A new p is created @400

p->polyCoef = malloc() — 400->polyCoef = @1000

Back in main

poly.polyCoef [0] = 1 — 100.polyCoef undefined!

Problem 2

Crashes when I use “poly” in main.

main.c:

```
polynomial *poly;
poly = malloc(sizeof(polynomial));
createPoly(poly, 10);
poly.polyCoef [0] = 1;          /* crash */
.....
void createPoly(polynomial *p, int n){
    p = malloc(sizeof(polynomial));
    p->polyCoef = malloc(sizeof(complex) * n);
}
```

Just like before, changing the pointer in the subroutine does nothing!

Question

- **What would strcat("123", "456") do?**

- Logically bad things.
  - The compiler puts "123" someplace in memory
  - strcat will try to append "456" after the end of "123" BUT
  - the compiler has put other variables in ram after the "123" SO
  - this operation will result in those other variables being destroyed