JTSK-320112

Programming in C II

C-Lab II

Lecture 1 & 2

Dr. Kinga Lipskoch

Spring 2017

Who are We?

- ▶ Kinga Lipskoch - university lecturer of Computer Science
- ▶ Jürgen Schönwälder - professor of Computer Science
- ▶ Contact details Lipskoch:
    - ▶ Office: Research I, Room 94
    - ▶ Telephone: +49 421 200-3148
    - ▶ E-Mail: k.lipskoch@jacobs-university.de
    - ▶ Office hours: Mondays 10:00 - 12:00
- ▶ Contact details Schönwälder:
    - ▶ Office: Research I, Room 87
    - ▶ Telephone: +49 421 200-3587
    - ▶ E-Mail: j.schoenwaelder@jacobs-university.de
    - ▶ Office hours: Thursdays 10:00 - 12:00

Course Resources

- ▶ Grader:
  https://grader.eecs.jacobs-university.de/
  Slides and programming assignment sheets will be posted
  there after the lab

- ▶ Programming assignments will be received during the lab

- ▶ Offline questions: Office hours or you can make other
  appointments if necessary

Literature

Textbooks:

- ▶ B. W. Kernighan & D. M. Ritchie: The C Programming Language, Second edition, Prentice Hall, 1988
- ▶ Stephen Prata: C Primer Plus, Fifth edition, Sams Publishing, 2004
- ▶ Steve Oualline: Practical C Programming, Third edition, O'Reilly, 1997
- ▶ Other C books as well, but stick to a few resources

Course Goals

- ▶ Continuation of the "Programming in C I" course
- ▶ Deepens the basic programming skills
- ▶ Advanced topics of C programming such as
    - ▶ Data structures
    - ▶ Simple algorithms
    - ▶ File handling, libraries, and debugging techniques
- ▶ Develop (i.e., design, code, test, and debug) more complex programs

Grading

- ▶ Same as the grading of the course "Programming in C I"
- ▶ 35% - programming assignments
- ▶ 65% - final exam

Programming Assignments

- ▶ To be solved individually
    - ▶ Discussion between students is encouraged, cheating is not. Each student needs to submit her/his own solution to the assignments
- ▶ Assignments will be graded by the TAs
    - ▶ Percentages
    - ▶ Formulated by the TAs: http://grader.eecs.jacobs-university.de/courses/ 320112/2017_1m1/Grading-Criteria-C2.pdf
- ▶ Every assignment has a deadline, no exceptions allowed after it expires
- ▶ Every student can use 6 extensions (i.e., 6 problems)

Structure of the Lectures

- ▶ Thursdays 14:15 - 16:15   Lecture
- ▶ Thursdays 16:15 - 18:30   Programming assignment
- ▶ Fridays   14:15 - 16:15   Lecture
- ▶ Fridays   16:15 - 18:30   Programming assignment
- ▶ Programming assignments are due on the next Tuesday and Wednesday at 10:00 in the morning, can be discussed at weekly tutorial offered by the TAs

Planned Syllabus

- **The C Preprocessor**
- **Bit Operations**
- **Pointers and Arrays (Dynamically Allocated Multi-Dimensional Arrays)**
- Pointers and Structures (Linked Lists)
- Compiling, Linking and the make Utility
- Pointers and Functions (Function Pointers)
- Stacks and Queues
- Modifiers and Other Keywords
- Binary I/O (File Handling)

Programming Environment

- ▶ We will use the Unix operating system and related GNU tools
  (gcc, gdb, codeblocks, geany, etc.)
    - ▶ You can use any Unix distribution or
    - ▶ If you use Windows you can try to install DevC++ or
      codeblocks
    - ▶ Your programs must compile without any warnings with gcc
    - ▶ Use gcc -Wall -o program program.c
- ▶ Once again: take the chance to learn some Unix
- ▶ Every problem has multiple associated testcases which are
  used to check your solution
- ▶ If testcases are not passed then the TAs will subtract points

The C Preprocessor (1)

- ▶ Before compilation, C source files are being preprocessed
- ▶ The preprocessor replaces tokens by an arbitrary number of characters
- ▶ Offers possibility of:
  - ▶ Use of named constants
  - ▶ Include files
  - ▶ Conditional compilation
  - ▶ Use of macros with arguments

The C Preprocessor (2)

- ▶ The preprocessor has a different syntax from C
- ▶ All preprocessor commands start with #
- ▶ A preprocessor directive terminates at the end-of-line
    - ▶ Do not put ; at the end of a directive
- ▶ It is a common programming practice to use all uppercase letters for macro names

The C Preprocessor: File Inclusion

- ▶ #include <filename>
  - ▶ includes file, follows implementation defined rule where to look for file, for Unix is typically /usr/include
  - ▶ Ex: #include <stdio.h>
- ▶ #include "filename"
  - ▶ looks in the directory of the source file
  - ▶ Ex: #include "myheader.h"
- ▶ Included files may include further files
- ▶ Typically used to include prototype declarations

The C Preprocessor: Motivation for Macros (1)

- ▶ Motivation for using named constants/macros
- ▶ What if the size of arrays has to be changed?

```c
int data[10];
int twice[10];
int main()
{
  int index;
  for(index = 0; index < 10; ++index) {
    data[index] = index;
    twice[index] = index * 2;
  }
  return 0;
}
```

The C Preprocessor: Motivation for Macros (2)

More generic program if using named constants/macros

```
1 #define SIZE 20
2 int data[SIZE];
3 int twice[SIZE];
4 int main()
5 {
6   int index;
7   for(index = 0; index < SIZE; ++index) {
8     data[index] = index;
9     twice[index] = index * 2;
10  }
11  return 0;
12 }
```

The C Preprocessor: Macro Substitution (1)

- ▶ Definition of macro
    - ▶ #define NAME replacement_text
- ▶ Any name may be replaced with any replacement text
    - ▶ Ex: #define FOREVER for (;;) defines new word
      FOREVER to be an infinite loop

The C Preprocessor: Macro Substitution (2)

- ▶ Possible to define macros with arguments
    - ▶ `#define MAX(A, B) ((A) > (B) ? (A) : (B))`
- ▶ Each formal parameter (A or B) will be replaced by corresponding argument
    - ▶ `x = MAX(p+q, r+s);` will be replaced by
    - ▶ `x = ((p+q) > (r+s) ? (p+q) : (r+s));`
- ▶ It is type independent

The C Preprocessor: Macro Substitution (3)

- ▶ Why are the ( ) around the variables important in the macro definition?
  - ▶ #define SQR(A) (A)*(A)
- ▶ Write a small program using this and see the effect without ( ) in (A)*(A) by calling SQR(5+1)
- ▶ Try also gcc -E program.c sends the output of the preprocessor to the standard output
- ▶ What happens if you call SQR(++i)?

The C Preprocessor: Macro Substitution (4)

▶ Spacing in macro definition is very important
▶ See the preprocessor output of the following source code

```
1  #include<stdio.h>
2  #define MAX =10
3  int main()
4  {
5    int counter;
6    for(counter =MAX; counter > 0; --counter)
7      printf("Hi there!\n");
8    return 0;
9  }
```

The C Preprocessor: Macro Substitution (5)

- ▶ Defined names can be undefined using
  - ▶ #undef NAME
- ▶ Formal parameters are not replaced within quoted strings
- ▶ If parameter name is preceded by # in replacement text, the actual argument will be put inside quotes
  - ▶ #define DPRINT(expr) printf(#expr " = %g\n", expr)
  - ▶ DPRINT(x/y) will be expanded to
  - ▶ printf("x/y" " = %g\n", x/y);

The C Preprocessor: Conditional Inclusion (1)

- ▶ Preprocessing can be controlled by using conditional statements which will be evaluated while preprocessor runs
- ▶ Enables programmer to selectively include code, depending on conditions
- ▶ #if, #endif, #elif (i.e., else if), #else

```
1 #ifdef DEBUG
2   printf("x: %d\n", x);
3 #endif
```

The C Preprocessor: Conditional Inclusion (2)

- ▶ `#ifdef`, `#ifndef` are special constructs that test whether name is (not) defined
- ▶ gcc allows to define names using the `-D` switch
- ▶ Ex: gcc -DDEBUG -c program.c
- ▶ Previous line is equivalent to `#define` DEBUG

The C Preprocessor: Conditional Inclusion (3)

- ▶ Write a small program in which you illustrate the use of conditional inclusion for debugging purposes
- ▶ Ex: If the name DEBUG is defined then print on the screen the message "This is a test version of the program"
- ▶ If DEBUG is not defined then print on the screen the message "This is the production version of the program"
- ▶ Also experiment with gcc -D

Bit Operations

- ▶ The bit is the smallest unit of information
  - ▶ Represented by 0 or 1
- ▶ Eight bits form one byte
  - ▶ Which data type could be used for representation?
- ▶ Low-level coding like writing device drivers or graphic programming require bit operations
- ▶ Data representation
  - ▶ Octal (format %o), hexadecimal (format %x, representation prefix 0x)
- ▶ In C you can manipulate individual bits within a variable

Bitwise Operators (1)

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal	128	64	32	16	8	4	2	1
Binary number	0	1	0	1	1	1	0	1

▶ Allow you to store and manipulate multiple states in one variable

▶ Allows to set and test individual bits in a variable

Bitwise Operators (2)

Operator	Function	Use
~	bitwise NOT	~expr
<<	left shift	expr1 << expr2
>>	right shift	expr1 >> expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
|	bitwise OR	expr1 | expr2
&=	bitwise AND assign	expr1 &= expr2
^=	bitwise XOR assign	expr1 ^= expr2
|=	bitwise OR assign	expr1 |= expr2

Bitwise and Logical AND

```
1 #include <stdio.h>
2 int main()
3 {
4   int i1, i2;
5   i1 = 6; // set to 4 and suddenly check 3 fails
6   i2 = 2;
7   if ((i1 != 0) && (i2 != 0))
8     printf("1: Both are not zero!\n");
9   if (i1 && i2)
10     printf("2: Both are not zero!\n");
11   // wrong check
12   if (i1 & i2)
13     printf("3: Both are not zero!\n");
14   return 0;
15 }
```

The Left-Shift Operator

- ▶ Moves the data to the left a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are 0's
- ▶ Ex: 10101101 << 3 results in 01101000

The Right-Shift Operator

- ▶ Moves the data to the right a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are:
  - ▶ 0's if variable is unsigned
  - ▶ Value of the sign bit if variable is signed
- ▶ Ex:
  - ▶ 7 = 00000111 >> 2 results in 00000001
  - ▶ −7 = 11111001 >> 2 results in 11111110

Using Masks to Identify Bits

MASK	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**

&

flag	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**1**

=

						1		
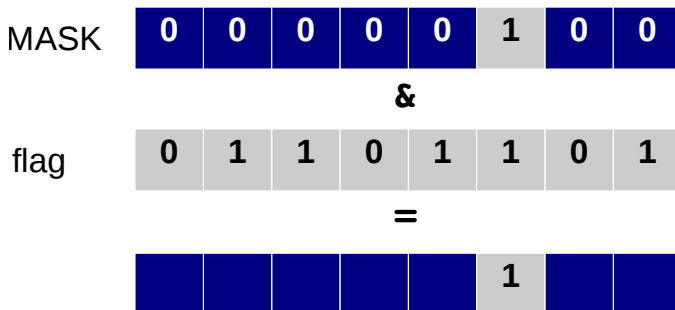
Using Masks

- ▶ Bitwise AND often used with a mask
- ▶ A mask is a bit pattern with one (or possibly more) bit(s) set
- ▶ Think of 0's as opaque and the 1's being transparent, only the mask 1's are visible
- ▶ If result > 0 then at least one bit of mask is set
- ▶ If result == MASK then the bits of the mask are set

binary.c

```
 1  #include <stdio.h>
 2  char str[sizeof(int) * 8 + 1];
 3  const int maxbit = sizeof(int) * 8 - 1;
 4  char* itobin(int n, char* binstr) {
 5    int i;
 6    for (i = 0; i <= maxbit; i++) {
 7      if (n & 1 << i) {
 8        binstr[maxbit - i] = '1';
 9      }
10      else {
11        binstr[maxbit - i] = '0';
12      }
13    }
14    binstr[maxbit + 1] = '\0';
15    return binstr;
16  }
17  int main()
18  {
19    int n;
20    while (1) {
21      scanf("%i", &n);
22      if (n < 0) break;
23      printf("%6d: %s\n", n, itobin(n, str));
24    }
25    return 0;
26  }
```

How to Turn on a Particular Bit

- ▶ To turn on bit 1 (second bit from the right), why does
  flags += 2 not work?
    - ▶ If flags = 2 = $00000010_{(2)}$
    - ▶ Then flags +=2 will result in
    - ▶ flags = 4 = $00000100_{(2)}$ which "unsets" bit 1
- ▶ Correct usage:
    - ▶ flags = flags | 2 is equivalent to
    - ▶ flags |= 2 and turns on bit 1

How to Toggle a Particular Bit

- ▶ To toggle bit 1
    - ▶ flags = flags ^ 2;
    - ▶ flags ^= 2; toggles on bit 1
- ▶ General form
    - ▶ flags ^= MASK;

How to Test a Particular Bit

▶ To test bit 1, why does `flags == 2` not work?
▶ Testing whether any bit of `MASK` are set:
  ▶ `if (flags & MASK)` ...
▶ Testing whether all bits of `MASK` are set:
  ▶ `if ((flags & MASK) == MASK)` ...

Using Bits Operations: A Problem

- ▶ Think of a low-level communication program
- ▶ Characters are stored in some buffer
- ▶ Each character has a set of status flags
    - ▶ ERROR   true if any error is set
    - ▶ FRAMING_ERROR   framing error occurred
    - ▶ PARITY_ERROR   wrong parity
    - ▶ CARRIER_LOST   carrier signal went down
    - ▶ CHANNEL_DOWN   power was lost on device

Size Considerations

- ▶ Suppose each status is stored in additional byte
  - ▶ 8k buffer   (real data)
  - ▶ But 40k status flags   (admin data)
- ▶ Need to pack data

A Communication System

- ▶ 0 - ERROR
- ▶ 1 - FRAMING_ERROR
- ▶ 2 - PARITY_ERROR
- ▶ 3 - CARRIER_LOST
- ▶ 4 - CHANNEL_DOWN

How to Initialize Bits

- `const int ERROR = 0x01;`
- `const int FRAMING_ERROR = 0x02;`
- `const int PARITY_ERROR = 0x04;`
- `const int CARRIER_LOST = 0x08;`

- If more states needed: `0x10, 0x20, 0x40, 0x80`
- It is not intuitive to know which hexadecimal-value has which bit set

How to "Nicely" Set Bits

- ▶ const int ERROR = (1 << 0);
- ▶ const int FRAMING_ERROR = (1 << 1);
- ▶ const int PARITY_ERROR = (1 << 2);
- ▶ const int CARRIER_LOST = (1 << 3);
- ▶ const int CHANNEL_DOWN = (1 << 4);

Everyone will directly understand encoding of the bits, additional documentation can be greatly reduced

Review: Pointers, Arrays, Values

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int length[2] = {7, 9};
 5 int *ptr1, *ptr2;
 6 int n1, n2;
 7
 8 int main() {
 9   ptr1 = &length[0];  // &length[0] is pointer to first elem
10   ptr2 = length;      // length is pointer to first elem
11                       // therefore same as above
12
13   n1 = length[0];     // length[0] is value
14   n2 = *ptr2;         // *ptr2 is value
15                       // therefore same as above
16
17   printf("ptr1: %p ptr2: %p\n", ptr1, ptr2);
18   printf("n1: %d, n2: %d\n", n1, n2);
19 }
```

Arrays in C

- ▶ See "Programming in C I" for introduction
- ▶ In C you declare an array by specifying the size between square brackets
- ▶ `int my_array[50];`
- ▶ The former is an array of 50 elements
- ▶ The first element is at position 0, the last one is at position 49

Accessing an Array in C

▶ To write an element, you specify its position

```
my_array[2] = 34;
my_array[0] = my_array[2];
```

▶ Pay attention: if you specify a position outside the limit, you will have unpredictable results
  ▶ Segmentation fault, bus error, etc
  ▶ And obviously wrong

▶ Note the different meaning of brackets

▶ Brackets in declaration describe dimension, while in program they are index operator

Initialization of Arrays

- ▶ C allows also the following declarations
  ```
  int first_array[] = {12, 45, 7, 34};
  int second_array[4] = {1, 4, 16, 64};
  int third_array[4] = {0, 0};
  ```
- ▶ It is not possible to specify more values than the declared size of the array
  - ▶ The following is not possible:
  - ▶ `int wrong[3] = {1, 2, 3, 4};`

Finding the Maximum Value in an Array

```
 1 /* Returns the biggest element in v
 2    v[]      array of ints
 3    dim      number of elements in v
 4 */
 5 int findmax(int v[], int dim) {
 6   int i, max;
 7   max = v[0];
 8   for (i = 1; i < dim; i++) {
 9     if (v[i] > max)
10       max = v[i];
11   }
12   return max;
13 }
```

Looking for an Element

```
1 /* v[]     array of integers
2    dim     number of elements in v
3    t       element to find
4    Returns   1 if t is not present in v or
5    its position in v
6 */
7 int find_element(int v[], int dim, int t) {
8    int i;
9    for (i = 0; i < dim; i++) {
10     if (v[i] == t)
11       return i;
12   }
13   return -1;
14 }
```

Pointers and Arrays

- Ex: `char array[5];`
    `char *array_ptr1 = &array[0];`
    `char *array_ptr2 = array;`
    `// the same as above`
- C allows pointer arithmetic:
    - Addition
    - Subtraction
- `*array_ptr` equivalent to `array[0]`
- `*(array_ptr+1)` equivalent to `array[1]`
- `*(array_ptr+2)` equivalent to `array[2]`
- What is `(*array_ptr)+1`?

Multidimensional Arrays in C

- ▶ It is necessary to specify the size of each dimension
    - ▶ Dimensions must be constants
    - ▶ In each dimension the first element is at position 0

    ```c
    int matrix[10][20]; /* 10 rows, 20 columns */
    float cube[5][5][5]; /* 125 elements */
    ```

- ▶ Every index is specified between brackets
    ```c
    matrix[0][0] = 5; /* which element is 5? */
    ```

Variably Sized Multidimensional Arrays

- ▶ Unidimensional arrays can be allocated "on the fly" using the `malloc()` function
- ▶ Possible also for multidimensional arrays, but more tricky
- ▶ Underlying idea: a pointer can point to the first element of a sequence
- ▶ A pointer to a pointer can then point to the first element of a sequence of pointers
  - ▶ And each of those pointers can point to first element of a sequence

Pointers to Pointers for Multidimensional Arrays (1)

- Consider the following
  char **table;
- We can make table to point to an array of
  pointers to char
  table = (char **) malloc(sizeof(char *)
   * N);
- Every element in the array of N rows is a
  char*

table

char *
char *
char *
char *
char *
char *
char *
char *
char *
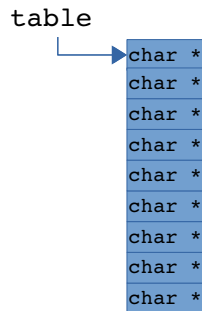
Pointers to Pointers for Multidimensional Arrays (2)

- ▶ Every pointer in the array can in turn point to an array
- ▶ In this way a two-dimensional array with N rows and M columns has been allocated

```
1 for (i = 0; i < N; i++)
2   table[i] = (char *) malloc(sizeof(char) * M);
```

Pointers to Pointers for Multidimensional Arrays (3)
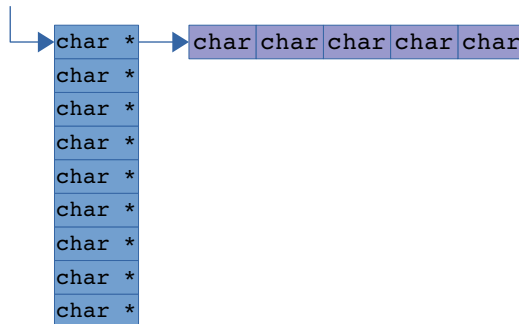
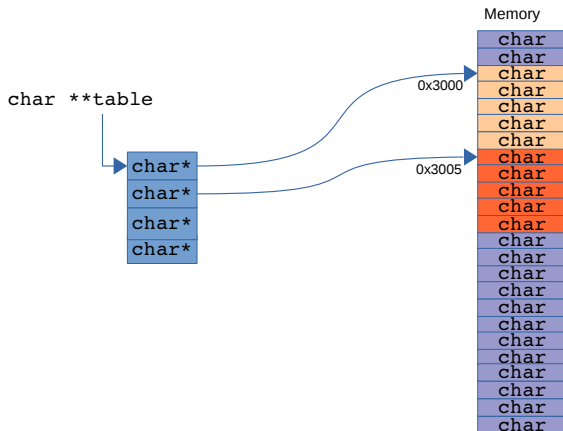To access a generic element in the dynamically allocated matrix a matrix-like syntax can be used. Let us see why ...

Allocating Space for a Multidimensional Array (1)
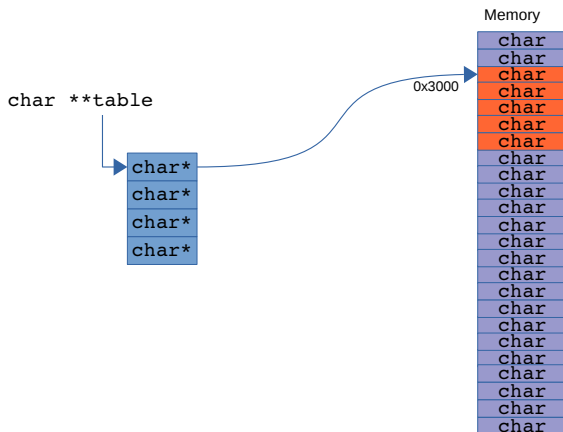
Case i=0

```
1  for (i = 0; i < 4; i++)
2    table[i] = (char *) malloc(sizeof(char) * 5);
```
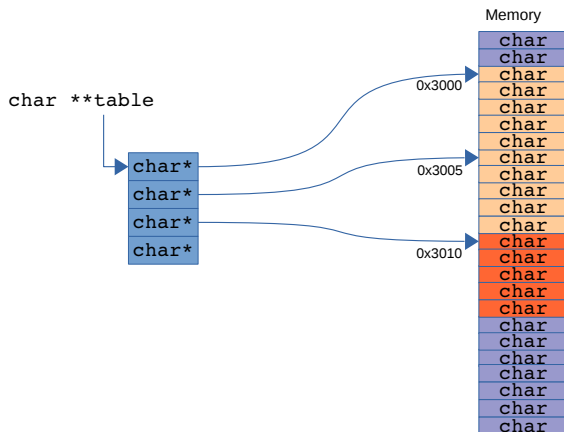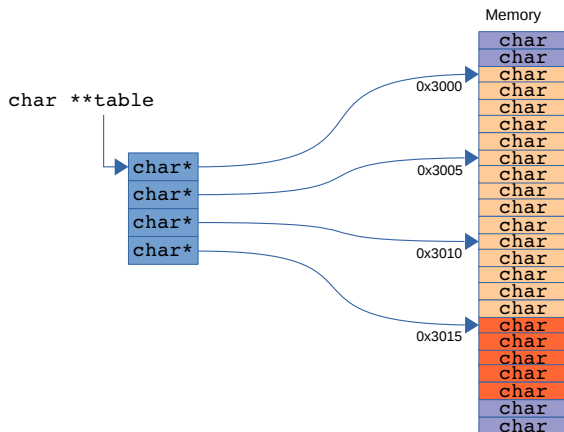
Allocating Space for a Multidimensional Array (2)

Memory

char **table

0x3000

0x3005

Case i=1

```
1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

Allocating Space for a Multidimensional Array (3)

Case i=2

```
1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

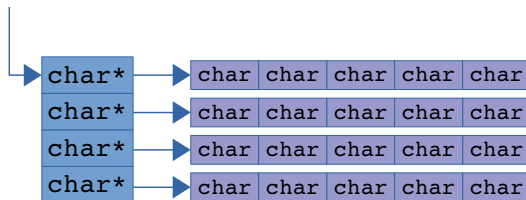## Allocating Space for a Multidimensional Array (4)

Case i=3

```
1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

Drawing Memory in a Different Way: The Result is a Table


```
1 for (i = 0; i < 4; i++)
2   table[i] = (char *) malloc(sizeof(char) * 5);
```

Pointers to Pointers vs. Bidimensional Arrays

- Bidimensional arrays occupy less memory
- Pointers to pointers allow tables where every row can have its own dimension
- Pointers to pointers need a more involved initialization
- One can have pointers to pointers to pointers (e.g., `int ***`) to have 3D data structures

De-allocating a Pointer to Pointer Structure

- ▶ Everything you have allocated via malloc() must be de-allocated via free()
- ▶ Ex: De-allocation of a 2D array with N elements

```
1 int i;
2 for (i = 0; i < N; i++)
3   free(table[i]);
4 free(table);
```

Working with 2D Dynamic Arrays

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 void set_all_elements(int **arr, int numrow, int numcol) {
 4   int r, c;
 5   for (r = 0; r < numrow; r++)
 6     for (c = 0; c < numcol; c++)
 7       arr[r][c] = r * c;  // some value ...
 8 }
 9 int main() {
10   int **table, row;
11   table = (int **) malloc(sizeof(int *) * 3);
12   if (table == NULL)
13     exit(1);
14   for (row = 0; row < 3; row++) {
15     table[row] = (int *) malloc(sizeof(int) * 4);
16     if (table[row] == NULL)
17       exit(1);
18     }
19   set_all_elements(table, 3, 4);
20 }
```

Locating a Matrix Element in the Memory

- ▶ Consider the following
  int table[ROW][COL];
  where ROW and COL are constants
- ▶ table holds the address of the pointer to the first element
- ▶ *table holds the address of the first element
- ▶ What is the address of table[i][j]?
  *(table + (i * COL + j))
- ▶ Find out the formula for an arbitrary multidimensional array

Pointer Arithmetic with Arrays

```
 1 #include <stdio.h>
 2 #define ROW 2
 3 #define COL 3
 4 int main () {
 5   int arr[ROW][COL] = { {1, 2, 3}, {11, 12, 13} };
 6   int i = 1;
 7   int j = 2;
 8   int* p = (int*) arr;            // needs explicit cast
 9   printf("Address of [1][2]: %p\n", &arr[1][2]);
10   printf("Address of [1][2]: %p\n", p +  (i * COL + j));
11   printf("Value of [1][2]: %d\n", arr[1][2]);
12   printf("Value of [1][2]: %d\n", *(p +  (i * COL + j)));
13   printf("\n");
14   printf("Address of [0][0]: %p\n", p +  (0 * COL + 0));
15   printf("Address of [0][1]: %p\n", p +  (0 * COL + 1));
16   printf("Address of [0][2]: %p\n", p +  (0 * COL + 2));
17   printf("Address of [1][0]: %p\n", p +  (1 * COL + 0));
18   printf("Address of [1][1]: %p\n", p +  (1 * COL + 1));
19   printf("Address of [1][2]: %p\n", p +  (1 * COL + 2));
20   return 0;
21 }
```