JTSK-320112

Programming in C II

C-Lab II

Lecture 1 & 2

Dr. Kinga Lipskoch

Spring 2019

Who am I?

- PhD in Computer Science at the Carl von Ossietzky University of Oldenburg
- University lecturer at the Computer Science Department
- Joined Jacobs University in January 2013
- Office: Research I, Room 94
- Telephone: +49 421 200-3148
- E-Mail: k.lipskoch@jacobs-university.de
- Office hours: Mondays 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

Grader not Publicly Visible

- You can access Grader from campus without any additional connection or software
- To access Grader from outside of campus you need to use a VPN (Virtual Private Network) connection
- Tutorials from the Jacobs IRC IT team on how to install a VPN client: `https://www.youtube.com/user/jacobsircit`

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 and Grading

- 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
  - `https://grader.eecs.jacobs-university.de/courses/320112/2019_1gA/Grading-Criteria-C2.pdf`
- Two types of assignments:
  - Automatically graded problems with testcases only - for example, 10 uploaded testcases and your solution passes only 7 then your grade will be 70%
  - Manually graded problems - with feedback from TAs and a grade between 0% and 100%
- Extension of deadline possible only with an official excuse

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

Missing Homework, Quizzes, Exams according to AP

- https://www.jacobs-university.de/sites/default/files/bachelor_policies_v1.1.pdf (page 9)
- Illness must be documented with a sick certificate
- Sick certificates and documentation for personal emergencies must be submitted to the Student Records Office by the third calendar day
- Predated or backdated sick certificates will be accepted only when the visit to the physician precedes or follows the period of illness by no more than one calendar day
- Students must inform the Instructor of Record before the beginning of the examination or class/lab session that they will not be able to attend
- The day after the excuse ends, students must contact the Instructor of Record in order to clarify the make-up procedure
- Make-up examinations have to be taken and incomplete coursework has to be submitted by no later than the deadline for submitting incomplete coursework as published in the Academic Calendar

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?

```
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

```
#define SIZE 20
int data[SIZE];
int twice[SIZE];
int main()
{
  int index;
  for(index = 0; index < SIZE; ++index) {
    data[index] = index;
    twice[index] = index * 2;
  }
  return 0;
}
```

Works but it no type information is associated with macros, so using `const` for this problem is a better solution.

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
  - Ex:
```
#define ODD(A, B) { unsigned char abit=A & 1;\
                    unsigned char bbit=B & 1; \
                    ... }
```

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 and syntax in macro definition is very important
- See the preprocessor output of the following source code

```
#include<stdio.h>
#define MAX =10
int main()
{
  int counter;
  for(counter =MAX; counter > 0; --counter)
    printf("Hi there!\n");
  return 0;
}
```

wrong_macro.c

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`

```
#if defined(DEBUG)  // short: #ifdef DEBUG
  printf("x: %d\n", x);
#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`

The Structure of a Header File with Conditional Inclusion

```c
/* Student.h */
#ifndef _LIST_H
#define _LIST_H
struct list {
  int info;
  struct list *next;
};
void printList(struct list *);
struct list * push_front(struct list *, int);
...
#endif // this matches the initial #ifndef
```

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)

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

Bitwise and Logical AND

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

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

```c
#include <stdio.h>
char str[sizeof(int) * 8 + 1];
const int maxbit = sizeof(int) * 8 - 1;
char* itobin(int n, char* binstr) {
  int i;
  for (i = 0; i <= maxbit; i++) {
    if (n & 1 << i) {
      binstr[maxbit - i] = '1';
    }
    else {
      binstr[maxbit - i] = '0';
    }
  }
  binstr[maxbit + 1] = '\0';
  return binstr;
}
int main()
{
  int n;
  while (1) {
    scanf("%i", &n);
    if (n < 0) break;
    printf("%6d: %s\n", n, itobin(n, str));
  }
  return 0;
}
```

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 = 000000010`$_{(2)}$
  - 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

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

int length[2] = {7, 9};
int *ptr1, *ptr2;
int n1, n2;

int main() {
  ptr1 = &length[0];  // &length[0] is pointer to first elem
  ptr2 = length;      // length is pointer to first elem
                      // therefore same as above

  n1 = length[0];   // length[0] is value
  n2 = *ptr2;       // *ptr2 is value
                    // therefore same as above

  printf("ptr1: %p ptr2: %p\n", ptr1, ptr2);
  printf("n1: %d, n2: %d\n", n1, n2);
}
```

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

```
/* Returns the biggest element in v
   v[100]    array of ints
   dim     number of elements in v
*/
int findmax(int v[100], int dim) {
  int i, max;
  max = v[0];
  for (i = 1; i < dim; i++) {
    if (v[i] > max)
      max = v[i];
  }
  return max;
}
```

Looking for an Element

```
/* v[100]    array of integers
   dim       number of elements in v
   t         element to find
   Returns    1  if t is not present in v or
   its position in v
*/
int find_element(int v[100], int dim, int t)  {
  int i;
  for (i = 0; i < dim; i++) {
    if (v[i] == t)
      return i;
  }
  return -1;
}
```

Pointers and Arrays

- Ex:
```c
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

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

- Every index is specified between brackets

```
matrix[0][0] = 5; /* which element is 5? */
```

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

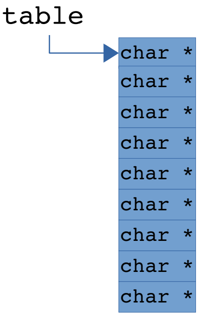
```
#include <stdio.h>
#define ROW 2
#define COL 3
int main() {
  int arr[ROW][COL] = { {1, 2, 3}, {11, 12, 13} };
  int i = 1;
  int j = 2;
  int* p = (int*) arr;          // needs explicit cast
  printf("Address of [1][2]: %p\n", &arr[1][2]);
  printf("Address of [1][2]: %p\n", p + (i * COL + j));
  printf("Value of [1][2]: %d\n", arr[1][2]);
  printf("Value of [1][2]: %d\n", *(p + (i * COL + j)));
  printf("\n");
  printf("Address of [0][0]: %p\n", p + (0 * COL + 0));
  printf("Address of [0][1]: %p\n", p + (0 * COL + 1));
  printf("Address of [0][2]: %p\n", p + (0 * COL + 2));
  printf("Address of [1][0]: %p\n", p + (1 * COL + 0));
  printf("Address of [1][1]: %p\n", p + (1 * COL + 1));
  printf("Address of [1][2]: %p\n", p + (1 * COL + 2));
  return 0;
}
```

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*`

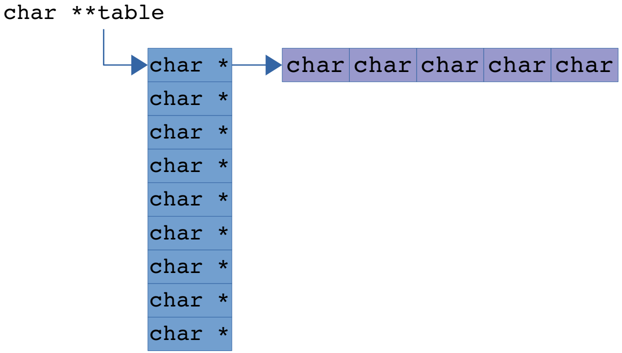


Pointers to Pointers for Multidimensional Arrays (2)

- 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

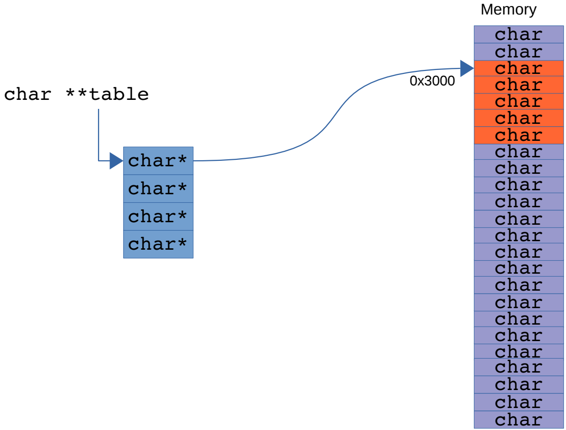
```
for (i = 0; i < N; i++)
  table[i] = (char *) malloc(sizeof(char) * M);
```

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 ...

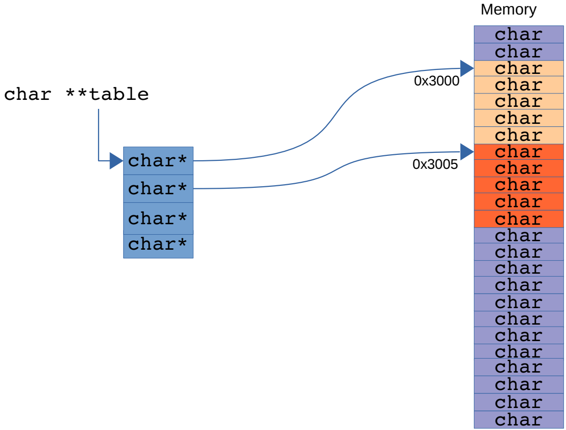
Allocating Space for a Multidimensional Array (1)



Case i=0

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

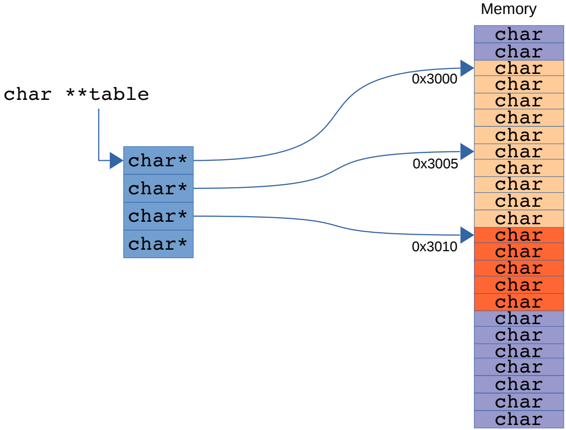
Allocating Space for a Multidimensional Array (2)



Case i=1

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

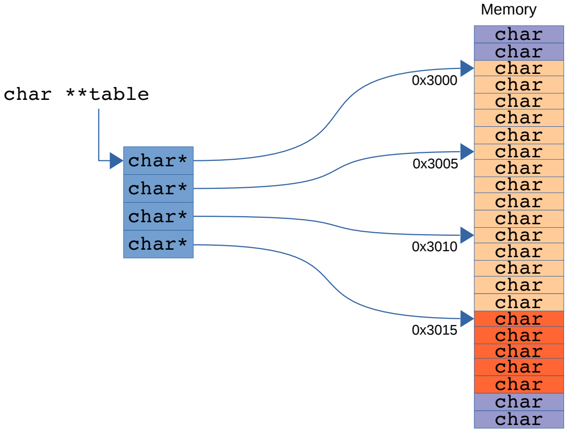
Allocating Space for a Multidimensional Array (3)



Case i=2

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

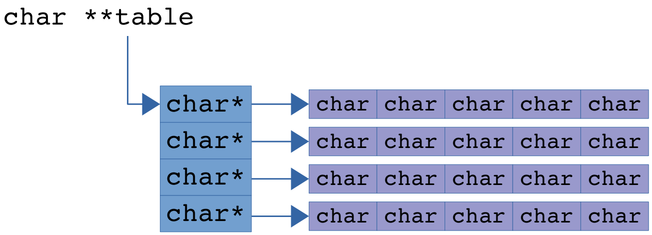
Allocating Space for a Multidimensional Array (4)



Case i=3

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

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




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

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

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

Working with 2D Dynamic Arrays

```c
#include <stdio.h>
#include <stdlib.h>
void set_all_elements(int **arr, int numrow, int numcol) {
  int r, c;
  for (r = 0; r < numrow; r++)
    for (c = 0; c < numcol; c++)
      arr[r][c] = r * c;  // some value ...
}
int main() {
  int **table, row;
  table = (int **) malloc(sizeof(int *) * 3);
  if (table == NULL)
    exit(1);
  for (row = 0; row < 3; row++) {
    table[row] = (int *) malloc(sizeof(int) * 4);
    if (table[row] == NULL)
      exit(1);
    }
    set_all_elements(table, 3, 4);
}
```

Static vs. Dynamic Array Allocation (1)

- `int a[n][m]` leads to an index offset calculation using the known array dimensions
- `int **a` treats `a` as an array `int *[]` and once indexed the result as an array of `int []`
- Statically allocated arrays occupy less memory
- Pointers to pointers allow tables where every row can have its own dimension
- One can have pointers to pointers to pointers (e.g., `int ***`) to have 3D data structures

Static vs. Dynamic Array Allocation (2)

- Static allocation
  - `int a[100][50]; int b[n][m];`
  - Syntax for allocation is easy
  - Release/reallocation not possible at runtime
  - Allocated memory is contiguous
- Dynamic allocation
  - `int **a; int *b[100], int ***c; ...`
  - Call(s) of `malloc` is needed
  - Syntax for allocation is more difficult
  - Release/reallocation possible at runtime using `free`, `realloc`
  - Allocated memory can be, but in general is not contiguous
- Passing arrays to functions: `static_dyn_allocation.c`
- Further reading/study:
  `https://www.cse.msu.edu/~cse251/lecture11.pdf`