Maciej Sobieraj

Lecture 6

Outline

1. Aggregating data into arrays
   1. **Assigning values to strings**
   2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
   1. The real meaning of array indexing
   2. Using pointers: perils and disadvantages
   3. Arrays of arrays: multidimensional arrays
   4. Memory allocation and deallocation: malloc() and free()
3. Quiz

Initializing strings

- String initializations have another interesting extension.

```
char protagonist[] = "Snape";
```

- This means that we want the compiler itself to **count the characters**.
- The compiler will add an empty character to the string so the declaration works the same way as this one:
  - **char** protagonist[6] = "Snape";

Assigning strings

- Can we use the same clear methods to assign a string to the character arrays?

```
char protagonist[20];

protagonist = "Gandalf";
```

- **Unfortunately, not.**
- The compiler sees the following: there’s a **character array** on the left side of the = operator; on the opposite side there’s a **string**
- *protagonist* is a pointer of type char *

Assigning strings

- There’s a function that makes this task so much easier.
- This function is called *strcpy* (it's a conflation of two words: STRing CoPY).

```
char protagonist[20];

strcpy(protagonist, "Gandalf");
```

A very important distinction

- You already know that it results in the following:
  - an array of 10 characters will be created;
  - the following characters: 'F', 'r', 'o', 'd', 'o' and '\ 0' will be stored in the variable;
  - whenever you use the name *protagonist* it'll be interpreted as a **pointer** to the first element i.e. the one that contains the letter 'F'.

```
char protagonist[10] = "Frodo";
```

A very important distinction

```
char *hero = "Dumbledore";
```

- This is what happens:
  - the compiler **reserves the memory** of 11 bytes (10 for the hero's name itself + 1 for an empty char) and fills it with the characters 'D', 'u', 'm', 'b', 'l', 'e' , 'd', 'o', 'r' , 'e' and '\0';
  - the compiler **creates a variable** named *hero* of type *char **;
  - the compiler **assigns the pointer** to a newly reserved string to the *hero* variable.

A very important distinction

hero = "Sirius";

- The compiler will perform the following steps
  - reserve 7 bytes for the new string and fill it with “Sirius”, ending with the empty character;
  - store the pointer of the newly created string in the *hero* variable.
- The regular pointer variable (in contrast to the array name) is a valid **l-value**.

A very important distinction

- The *strcpy* has no intention of changing the pointer's value. It only copies the string "Pippin" along with its empty character into the location pointed to by the *hero* variable.

```
strcpy(hero, "Pippin");
```

- There are two important things that you have to consider before you use the *strcpy*:
  - Do you know for sure **where** the left argument points to?
  - Is there **sufficient room** to accommodate the string?

Outline

1. Aggregating data into arrays
   1. Assigning values to strings
   2. **Processing strings**
2. Arrays vs. structures: different aggregates for different purposes
   1. The real meaning of array indexing
   2. Using pointers: perils and disadvantages
   3. Arrays of arrays: multidimensional arrays
   4. Memory allocation and deallocation: malloc() and free()
3. Quiz

How do we print a string?

- The “C” language functions are described by specifying the *prototype*. The prototype consists of:
  - **a return type** (the type of the result)
  - **the name** of the function
  - **a list of its parameters** as well as their types
- The prototype of *puts* is as follows

```
int puts(char *s);
```

How do we print a string?

- Here are three valid forms of *puts*'s invocation:
  - with the name of the character array
  - with the name of the pointer of type char *
  - with the string literal

```
puts(protagonist);
puts(hero);
puts("Boromir");
```

How do we print a string?

- The *puts* prototype specifies that the function **returns a result** of type *int*.
  - it's a non-negative number if everything goes well and -1 if *puts* cannot meet our demands due to any reason.

```
int res;

res = puts("McGonagall");
puts("Saruman");
```

How do we print a string?

- The second function is *printf*

```
int printf(char *format, ...);
```

- It's *%s* where the letter "*s*" stands for *string*.

```
printf("%s and %s are the best\n", protagonist, hero);
```

Some useful functions

- More essential functions that let us work with strings efficiently and smoothly are contained in the header file named *string.h*.

```
#include <string.h>
```

```
char string[50];
char *ptr = "Computer";
```

strlen: STRing LENgth

- **strlen: STRing LENgth – the length of the string**
  - The *strlen* function is used to **count the characters** in a string, excluding the empty character at the end.

```
int strlen(char *s);
```

- An invocation like this:
  - strlen (ptr)
- returns 8 as a result

strcpy: STRing CoPY

- **strcpy: STRing CoPY – make a copy of a string**
  - The *strcpy* function **makes a copy** of a string pointed to by *source* and stores it at the location pointed to by *destination*.
  - The result of the function is the same pointer as the one specified as *destination*.

```
char *strcpy(char *destination, char *source);
```

strcpy: STRing CoPY

- An invocation like this:
  - strcpy(string, ptr);
- places a copy of the string “computer” at the location pointed to by the variable *string*.
- The invocation:
  - strcpy(string, "Alice has a cat");
- causes the *string* array to contain the phrase “Alice has a cat” along with the closing null character.

strncpy: STRing N CoPY

- **strncpy: STRing N CoPY – make an n-long copy of a string**
  - The *strncpy* function makes a **copy of a maximum *n* characters taken** from the string pointed to by *source* and stores them in the location pointed to by *destination*.
  - The finishing null character is only added to the copied string if this character is in n range.

```
char *strncpy(char *destination, char *source, int n);
```

strncpy: STRing N CoPY

- This invocation:
  - strncpy (string, ptr, 3);
- fills the array *string* with the letters 'C', 'o' and 'm'.
- This invocation:
  - strncpy (string, "Alice has a cat", 5);
- fills the array *string* with the string *“Alice”*.

strcat: STRing conCATenation

- **strcat: STRing conCATenation – append a string to another string**
  - The *strcat* function **appends a copy the string** pointed to by *source* to the end of the string pointed to by *destination*.
  - The null character that originally closes *destination* is removed.
  - Then a copy of *source* is appended to *destination* along with its closing null character.

```
char *strcat(char *destination, char *source);
```

strcat: STRing conCATenation

- This sequence of instructions:
  - strcpy(string, ptr);
  - strcat(string, ptr);
- causes the array *string* to contain “ComputerComputer” followed by the null character.
- This sequence of instructions:
  - strcpy (string, "Alice ");
  - strcat (string, "has no ");
  - strcat (string, ptr);
- fills the array *string* with “Alice has no Computer”.

Outline

1. Aggregating data into arrays
   1. Assigning values to strings
   2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
   1. **The real meaning of array indexing**
   2. Using pointers: perils and disadvantages
   3. Arrays of arrays: multidimensional arrays
   4. Memory allocation and deallocation: malloc() and free()
3. Quiz

Indexing vs pointers

- We’ve created a 10 element array of char and put a string “dump” there. Eventually, we consume 5 chars of the array.

```
char word[10] = "dump";
```

Indexing vs pointers

- Don't forget:
  - apostrophes: char
  - quotes: char *

```
word[1] = 'a';
puts(word);
```

Indexing vs pointers

- It won’t happen for sure:
  - the compiler won’t signal **either an error or a warning;**
  - the string contained in the array **will not be changed.**

```
word[-1] = 'x';
word[1000000] = 'y';
```

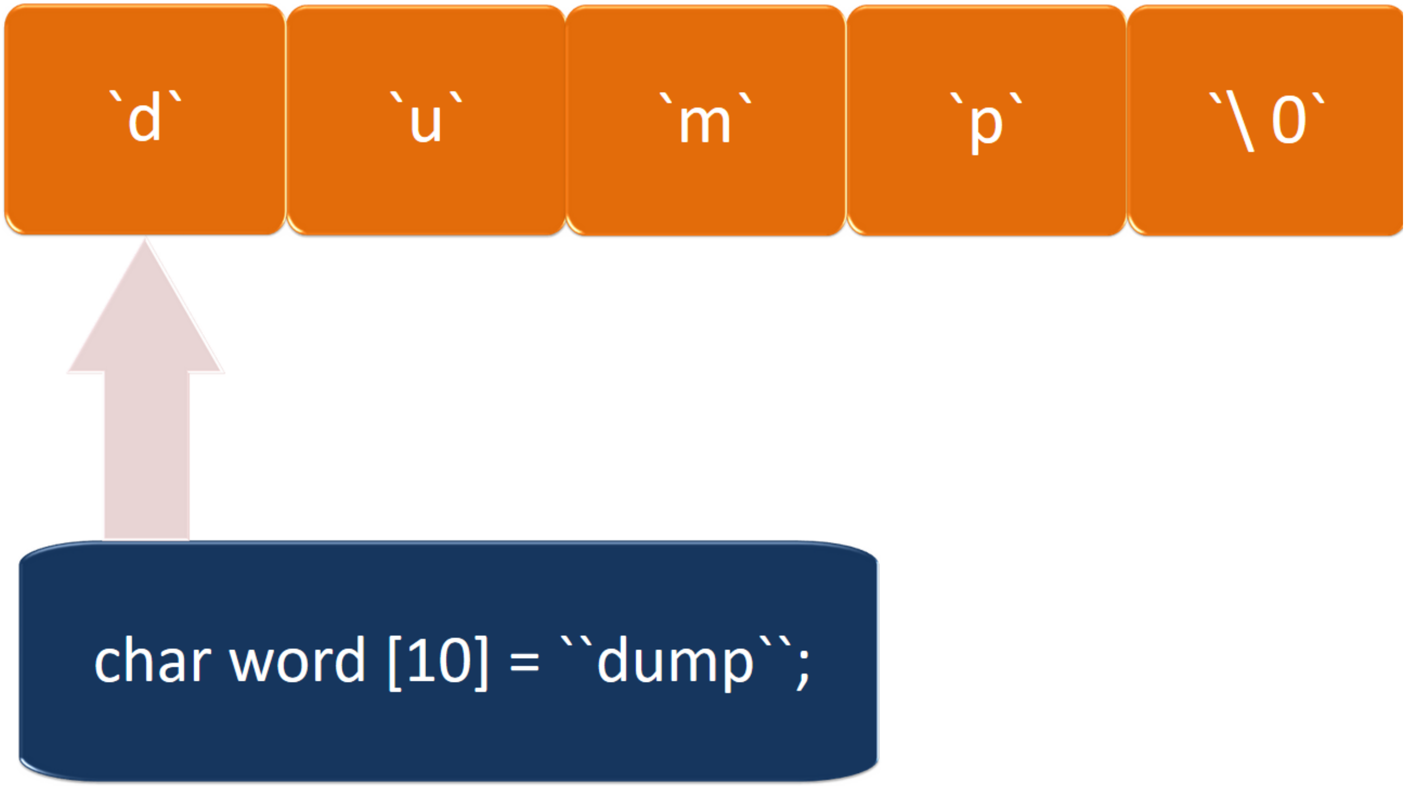
Indexing vs pointers

- The “C” language standard says: if any pointer is followed by an indexing operator, like this:
  - **t[i]**
- it’s always taken as:
  - ***(t + i).**

$$t[i] \equiv *(t + i)$$

Step 1

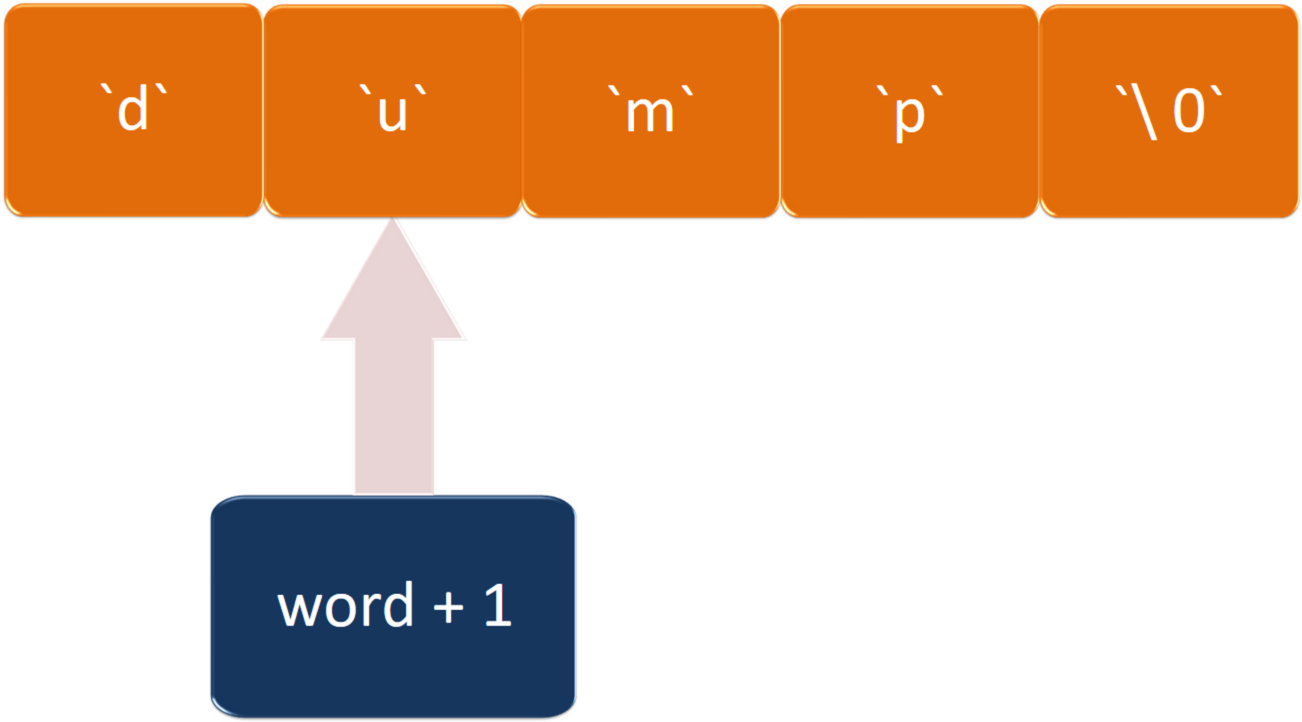
- The name *word* is interpreted as a pointer to the first element of the array.



Step 2

- The pointer is increased by one (*word + 1*).

char word [10] = ``dump``;



Step 2

- The increased pointer is an argument for the dereference operator, which means that it's of type char, at least from a syntactic and semantic point of view.
- This means that this **assignment is fully permissible**

```
*(word + 1) = 'a';
```

Step 2

```
*(word + 1) = 'a';
```

- The meaning of this assignment is exactly the same as the one here

```
word[1] = 'a';
```

Step 3

- Can you explain why we used the parentheses?

*(word + 1)

Indexing vs pointers

- A value of 1 is added to the dereferenced character

```
*word + 1
```

Indexing vs pointers

- if *t* is a pointer and *i* is an expression of type *int*, t[i] is equivalent to *(t + i);
- the addition is commutative, so we can write the previous expression in the following way: *(i + t);
- this also means that we're allowed to write the same indexing operation as i[t].

$$t[i] \equiv i[t]$$

Indexing vs pointers

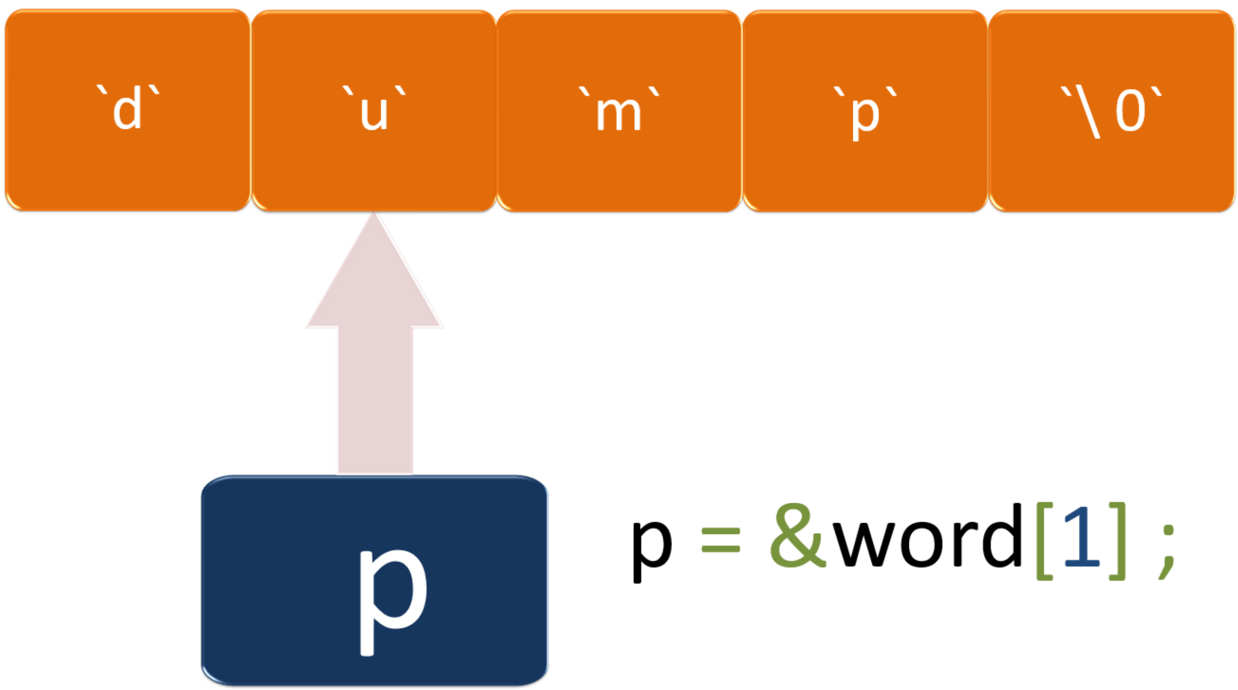
```
char string[] = "ABC";
char *p;
char c;
```

- Now we set *p* to point to the second element of the array string. The recommended form of this assignment is as follows:
  - **p = string + 1;**
- Acceptable, though less elegant (however, some would argue, clearer), is the following form:
  - **p = &string[1];**

Indexing vs pointers

- The *p* pointer will point to the second element of the array

```
char word [10] = ``dump``;
```




```
p = &word[1] ;
```

Indexing vs pointers

```
char string[] = "ABC";
char *p;
char c;
```

- Can you answer the question of what distinguishes these two instructions?
  - c = *p++;
- and
  - c = (*p)++;

Indexing vs pointers

- We can explain: the first assignment is as if the following two disjoint instructions have been performed;
  - c = *p;
  - p++;
- The second assignment is performed as follows:
  - c = *p;
  - string[1]++;

```
char string[] = "ABC";
char *p;
char c;
```

Indexing vs pointers

- Imagine the following assignment:
  - p = string + 2;
- *p* points to the **third** element of the *string* array. What happens now?
  - p[-1] = 'e';

Indexing vs pointers

- The compiler treats this as normal and thinks that we’re trying to do something like this:
  - *(p - 1) = 'e';

Outline

1. Aggregating data into arrays
    1. Assigning values to strings
    2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
    1. The real meaning of array indexing
    2. **Using pointers: perils and disadvantages**
    3. Arrays of arrays: multidimensional arrays
    4. Memory allocation and deallocation: malloc() and free()
3. Quiz

Pointers could be dangerous

- **Mistake no. 1: use of an uninitialized pointer**
  - Some compilers gives an error. „Uninitialized local variable”

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

int main(void) {
    char *ptr;
    strcpy(ptr, "you may get into trouble soon");
    puts(ptr);
    return 0;
}
```

Pointers could be dangerous

- **Mistake no. 1: use of an uninitialized pointer**
  - The other side of the same mistake

```
#include <stdio.h>
int main(void) {
    char *ptr;
    *ptr = 'C';
    printf("%c",*ptr);
    return 0;
}
```

Pointers could be dangerous

- **Mistake no. 2: exceeding the size of the array**
  - Your program may finish its work with a message about a memory violation error

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

int main(void) {
   char str[10];

   strcpy(str,"Welcome to troubles!");
   printf("%s",str);
   return 0;
}
```

Pointers could be dangerous

- **Mistake no. 3: non-terminated strings**

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

int main(void) {
    char str[10];
    int i;

    strcat(str,"Bump!");
    printf("%s",str);
    return 0;
}
```

Outline

1. Aggregating data into arrays
   1. Assigning values to strings
   2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
   1. The real meaning of array indexing
   2. Using pointers: perils and disadvantages
   3. **Arrays of arrays: multidimensional arrays**
   4. Memory allocation and deallocation: malloc() and free()
3. Quiz

Not only vectors

- Let’s consider the case when an **array's elements are just arrays**.
- A chessboard is composed of **rows** and **columns**. There are 8 rows and 8 columns.

```
int row[8];
```

Not only vectors

- Unfortunately, we have 8 of these rows. Does this mean that we have to declare 8 arrays like this?
  - **int** row1[8], row2[8], row3[8], row4[8], row5[8], row6[8], row7[8], row8[8];
- A *chessboard* is in fact an 8-element array of elements as single rows. Let's summarize our observations:
  - **elements of rows** are fields, 8 of them per row;
  - **elements of the chessboard are rows**, 8 of them per chessboard
  -

Not only vectors

- The chessboard variable is a **two dimensional array**. It’s also called, by analogy to algebraic terms, a **matrix**.

```
int chessboard[8][8];
```

- The appearance of two pairs of brackets tells the compiler that the declared array is not a vector – it's an **array whose elements are vectors**.

Not only vectors

	A	B	C	D	E	F	G	H	
1	[0] [0]	[0] [1]	[0] [2]	[0] [3]	[0] [4]	[0] [5]	[0] [6]	[0] [7]	1
2	[1] [0]	[1] [1]	[1] [2]	[1] [3]	[1] [4]	[1] [5]	[1] [6]	[1] [7]	2
3	[2] [0]	[2] [1]	[2] [2]	[2] [3]	[2] [4]	[2] [5]	[2] [6]	[2] [7]	3
4	[3] [0]	[3] [1]	[3] [2]	[3] [3]	[3] [4]	[3] [5]	[3] [6]	[3] [7]	4
5	[4] [0]	[4] [1]	[4] [2]	[4] [3]	[4] [4]	[4] [5]	[4] [6]	[4] [7]	5
6	[5] [0]	[5] [1]	[5] [2]	[5] [3]	[5] [4]	[5] [5]	[5] [6]	[5] [7]	6
7	[6] [0]	[6] [1]	[6] [2]	[6] [3]	[6] [4]	[6] [5]	[6] [6]	[6] [7]	7
8	[7] [0]	[7] [1]	[7] [2]	[7] [3]	[7] [4]	[7] [5]	[7] [6]	[7] [7]	8
	A	B	C	D	E	F	G	H	

Not only vectors

- we can set some chess pieces on our board. First, let's put all the rooks on the board:
  - chessboard[0][0] = ROOK;
  - chessboard[0][7] = ROOK;
  - chessboard[7][0] = ROOK;
  - chessboard[7][7] = ROOK;
- If we wanted to place a knight on C4, we would do this as follows:
  - chessboard[3][2] = KNIGHT;
- And now a pawn to E5:
  - chessboard[4][4] = PAWN;

Not only vectors

- To find any element of a two-dimensional array, we have to use two “***coordinates***”: a **vertical** (row number) one and a **horizontal** (column number) one.

```
float temp[31][24];
```

- This gives us a total of 24 * 31 = 744 values.

Not only vectors

```
float temp[31][24];
int day;
float sum = 0.0, average;
for(day = 0; day < 31; day++)
    sum += temp[day][11];
average = sum / 31;
printf("Average temperature at noon: %f", average);
```

Not only vectors

```
float temp[31][24];
int day,hour;
float max = -100.0;

for(day = 0; day < 31; day++)
   for(hour = 0; hour < 24; hour++)
      if(temp[day][hour] > max)
         max = temp[day][hour];
printf("The highest temperature was %f", max);
```

Not only vectors

```
float temp[31][24];
int day,hour;
int hotdays = 0;

for(day = 0; day < 31; day++)
   if(temp[day][11] >= 20.0)
      hotdays++;
printf("%d days were hot.", hotdays);
```

Not only vectors

```
float temp[31][24];
int d,h;

for(d = 0; d < 31; d++)
   for(h = 0; h < 24; h++)
      temp[d][h] = 0.0;
```

Not only vectors

- The “C” language **doesn’t limit the size of the array's dimensions**. Here we show an example of a 3-dimensional array.

```
int guests[3][15][20];
```

Not only vectors

```
int guests[3][15][20];
```

- Now imagine a hotel. It's a huge hotel consisting of three buildings, 15 floors each. There are 20 rooms on each floor. We need an array that can collect and process information on the number of guests registered in each room.

Not only vectors

- Let's check if there are any vacancies on the fifteenth floor of the third building:

```
int room;
int vacancy = 0;
for (room = 0; room <20; room++)
    if (guests[2][14][room] == 0)
        vacancy++;
```

- The *vacancy* variable contains 0 if all the rooms are occupied; otherwise it displays the number of available rooms.

Outline

1. Aggregating data into arrays
    1. Assigning values to strings
    2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
    1. The real meaning of array indexing
    2. Using pointers: perils and disadvantages
    3. Arrays of arrays: multidimensional arrays
    4. **Memory allocation and deallocation: malloc() and free()**
3. Quiz

void – the very exceptional type

```
void nothingatall(void);
```

- The function should be invoked **without parameters** and return no result.
- This is how we should invoke it:
  - nothingatall();

void – the very exceptional type

- Despite the fact that the *void* type doesn't represent any useful value, you can still declare pointers to this type

```
void *ptr;
```

- the type *void **, is called an **amorphous pointer** to emphasize the fact that it can point to any value of any type.
- a pointer of type *void ** **cannot be subject to the dereference operator**

Memory on demand

- To manage the allocating and freeing of memory, the “C” language provides a set of specialized functions.
- Using both functions, however, **requires** the inclusion of the header file ***stdlib**.h*.
- The first function is used to request access to the memory block of the specified size.
- When the allocated memory is no longer needed and/or utilized, it would be a good idea to return it to the operating system. We do this by using the second of these two functions.

Memory on demand

- The function that performs the first task has the following prototype

```
void *malloc(int size);
```

  - the name of the function is a conflation of *Memory ALLOCation*;
  - its only parameter provides information about the **size of the requested memory** and is expressed in **bytes**;
  - the **function returns a pointer** of type *void ** which points to the newly allocated memory block, or is equal to NULL to indicate that the allocation requested could not be granted;

Memory on demand

- The function that performs the first task has the following prototype

```
void *malloc(int size);
```

  - the function doesn't have a clue as to what we want to use the memory for and therefore the result is of type *void **; we'll have to convert it to another usable pointer type;
  - the allocated memory area **is not filled (initiated)** in any way, so you should expect it to contain **garbage.**

Memory on demand

- The function invoked when the memory is no longer necessary has the following prototype

```
void free(void *pointer);
```

  - the function name doesn't require any comments;
  - the function **does not return any results** so its type is defined as void;
  - the **function expects one parameter** – the pointer to the memory block that is to be released; usually it's a pointer previously received from the *malloc* or its kindred; using another pointer value may cause some kind of disaster;

Memory on demand

- The function invoked when the memory is no longer necessary has the following prototype

```
void free(void *pointer);
```

  - the function doesn't need to know the size of the freed block; you can only release the entire allocated block, not a part of it;
  - after performing the *free* function, all the pointers that point to the data inside the freed area become illegal; attempting to use them may result in abnormal program termination.

Memory on demand

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

int main(void) {
    int *ptr;

    ptr = (int *) malloc(sizeof(int));
    if(ptr != NULL) {
        *ptr = 200;
        printf("ptr points to value of %d", *ptr);
        free(ptr);
    } else
        printf("allocation failed");
    return 0;
}
```

Memory on demand

```
int *tabptr, i, sum = 0;

tabptr = (int *) malloc(5 * sizeof(int));
for(i = 0; i < 5; i++)
    tabptr[i] = i;
sum = 0;
for(i = 0; i < 5; i++)
    sum += tabptr[i];
free(tabptr);
```

Memory on demand

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

int main(void) {
   int *numbers, how_many_numbers;
   int i, aux;
   int swapped;

   printf("How many numbers are you going to sort?");
   scanf("%d", &how_many_numbers);
   if( how_many_numbers <= 0 || how_many_numbers > 1000000) {
      printf("Are you kidding?\n");
      return 1;
   }
   numbers = (int *) malloc(how_many_numbers * sizeof(int));
   if(numbers == NULL) {
      printf("Allocation failed – sorry.\n");
      return 1;
   }
   for(i = 0; i < how_many_numbers; i++) {
      printf("\nEnter the number #%i:\n",i + 1);
      scanf("%d",numbers + i);
   }
```

Memory on demand

```
do {
   swapped = 0;
   for(i = 0; i < how_many_numbers - 1; i++)
      if(numbers[i] > numbers[i + 1]) {
         swapped = 1;
         aux = numbers[i];
         numbers[i] = numbers[i + 1];
         numbers[i + 1] = aux;
      }
} while(swapped);
printf("\nThe sorted array:\n");
for(i = 0; i < how_many_numbers; i++)
   printf("%d ",numbers[i]);
printf("\n");
free(numbers);
return 0;
}
```

Outline

1. Aggregating data into arrays
    1. Assigning values to strings
    2. Processing strings
2. Arrays vs. structures: different aggregates for different purposes
    1. The real meaning of array indexing
    2. Using pointers: perils and disadvantages
    3. Arrays of arrays: multidimensional arrays
    4. Memory allocation and deallocation: malloc() and free()
3. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <string.h>
int main(void) {
   char t[20] = "ABCDEFGHIJK";
   int  s = strlen(t);
   t[3] = '\0';
   s = strlen(t);
   printf("%d",s);
   return 0;
}
```

- the program outputs 3
- the program outputs 1
- the program outputs 5

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <string.h>
int main(void) {
   char t[20] = "ABCDEFGHIJK";
   int  s = strlen(t);
   t[3] = '\0';
   s += strlen(t);
   printf("%d",s);
   return 0;
}
```

- the program outputs 14
- the program outputs 7
- the program outputs 21

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <string.h>
int main(void) {
   char t[10] = "";
   int  s;
   s = strlen(t);
   strcat(t,"ABCDEF");
   s += strlen(t);
   printf("%d",s);
   return 0;
}
```

- the program outputs 3
- the program outputs 0
- the program outputs 6

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int t[2][2];
   int i,j,s = 0;
   for(i = 0; i < 2; i++)
      for(j = 0; j < 2; j++)
         t[i][j] = 2 *i + j;
   printf("%d",t[1][0]);
}
```

- the program outputs 1
- the program outputs 2
- the program outputs 4

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
    int t[2][2];
    int i,j,s = 0;
    for(i = 0; i < 2; i++)
        for(j = 1; j >= 0; j--)
            t[i][j] = 2 * j + 1;
    printf("%d",t[1][0]);
    return 0;
}
```

- the program outputs 2
- the program outputs 3
- the program outputs 1