Dynamic memory allocation (1)

A variable may have local lifetime (they get memory when the program enters into the block in which they are declared and lose the memory when the program leaves this block) or global lifetime (they get memory when the program starts to run and lose it when the program exits). Consequently, the allocation of memory for a variable is strictly predefined by the rules of C/C++.

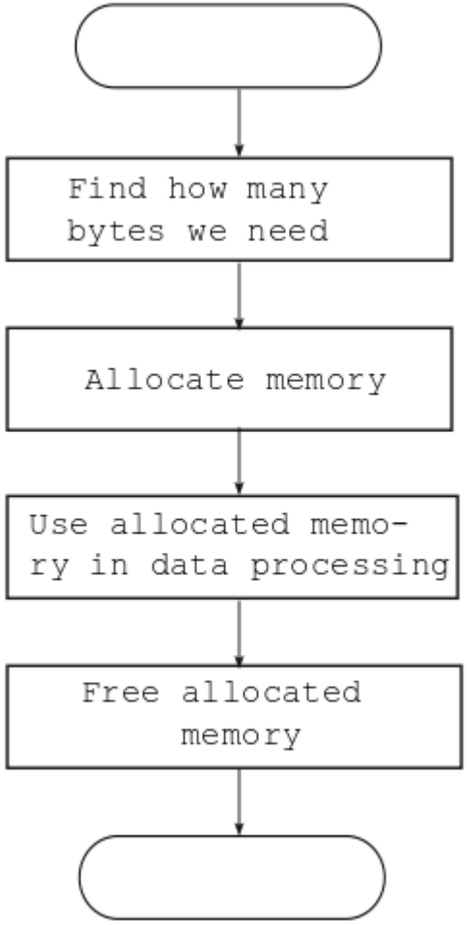
Unfortunately, it means that for example:

```
int n;
double md[n]; // error, allocation of memory for md is impossible because the compiler
              // does not know how many bytes are needed
```

Very often when we are writing the code we cannot know what should be the dimensions of arrays. For example, if our program must read the contents of a disk file into an array, the length of array depends on the file size but the file may have any size. Estimating the maximum length is not the solution: in most cases it is simply impossible.

So we need a mechanism to allocate memory dynamically, i.e. not beforehand when writing code but when the program is already running and the amount of needed memory has become clear.

Dynamic memory allocation (2)



For allocating memory we use standard function
#include "stdlib.h
<pointer_to_allocated_memory> = malloc(<number_of_bytes_to_allocate>);

To use the allocated memory we need to know the pointer arithmetic.

To free the allocated memory we use standard function

free(<pointer_to_allocated_memory>);

Function *malloc* sends to the operating system a request asking to allocate the needed amount of memory.

Function *free* informs the operating system that the allocated memory field is not needed any more and the operating system may use it for other purposes.

The region from which the memory is allocated is the heap.

Pointers (1)

int i = 1;

Variable *i* is located on a 4-byte memory field. Each byte of this memory field has its own address. Here we need not to know those addresses, because the identifier *i* is enough for accessing the variable.

However, when we have allocated a memory field dynamically, this memory field cannot have predefined identifier. Function *malloc* returns the address of the first byte of the allocated memory field. To store that address we need a new data type: the pointer.

int i, mi[5], *pi;

Here we declare 3 variables: *i*, *mi* and *pi*. Brackets after *mi* tell that *mi* is an array of 5 integers. Asterisk before pi tells that pi IS NOT AN INTEGER BUT A POINTER that will point to memory field containing integers. In other words it means that *pi* will be used to store the address of the first (also told as the highest) byte of a memory field and this memory field is used or will be used for storing values of type *int*.

How many bytes occupies *pi* itself? It depends on the platform. In Windows it is 4 bytes. What will be exactly written onto those bytes, i.e. what is an address physically? It also depends on the platform.

Pointers (2)

```
double d, md[5], *pd;
char c, mc[5], *pc;
```

Now variable *pd* is a pointer that will point to memory field containing double numbers and variable *pc* is a pointer that will point to memory field containing characters. Although *d* occupies 8 bytes and *c* one byte, both *pd* and *pc* occupy 4 bytes (valid for Windows).

We may also need pointers that point to a memory field that will also contain pointers. Those are declared like:

```
double **ppd; // pointer to pointers that point to doubles
```

We may continue in this way (pointer to pointers to pointers…, the number of asterisks before the pointer name has no limits) but such a complicated data structures should be avoided.

We may also have arrays with fixed size containing pointers, for example:

```
double *pmd[5]; // array of 5 pointers pointing to double numbers
```

Letter *'p'* as the first character of the pointer variable identifier is not a rule from C syntacs but a good tradition.

Pointers (3)

```
int n, *pi;
…… // find the actual value of n
pi = (int *)malloc(n * sizeof (int));
```

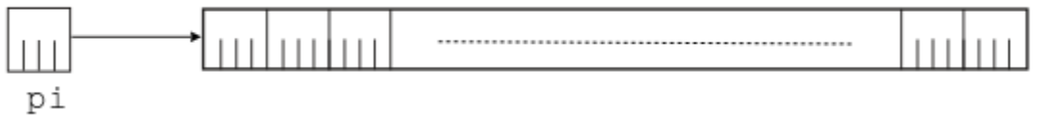
The argument of *malloc* is the number of requested bytes. Here we need memory for *n* numbers of type *int*. However, the C/C++ standard does not set how many bytes an *int* number occupes – it depends on the platform and nowadays may be 2, 4 or 8. In Windows it is currently 4. If we want to write the portable code (i.e. code that may be compiled on any platform), we must not write the number of bytes directly into the call to *malloc*. Unary operator *sizeof* followed by type enclosed in *( )* returns the amount of memory needed for one unit from this type on the current platform. So in Windows *sizeof (int)* returns 4, *sizeof (double)* returns 8, *sizeof (unsigned short int)* returns 2, etc. Operator *sizeof* followed by an expression or variable (which is also considered as a simple expression) returns the number of bytes needed to store the result of expression:

```
int i, mi[5];
printf("%d %d\n", sizeof i, sizeof mi); // prints 4 and 20
```

The return value of *malloc* is of type *void* * or a pointer to data of any type. The *pi* is of type *int* *. Assignment of *void* * pointers to pointers of strictly predetermined type without casting is not allowed. Therefore before assignment we have to convert the type.

Pointers (4)

```
int n, *pi;
…… // find the actual value of n
pi = (int *)malloc(n * sizeof (int));
                    // pi = malloc(n *sizeof(int)); error, no casting
                    // pi = (int *)malloc(n * 4); formally correct but not portable
…… // do something with the allocated memory
free (pi); // we do not need this memory field, release it
```



On the picture we suppose that *sizeof pi* is 4 and *sizeof (int)* is also 4.
Similarly:

```
int n;
double *pd;
char *pc;
…… // find the actual value of n
pd = (double *)malloc(n * sizeof (double));
pc = (char *)malloc(n); // as char is always 1-byte integer, sizeof is not needed
…… // do something with the allocated memory
free (pd);
free (pc);
```

Pointers (5)

```
int n, *pi;
…… // find the actual value of n
pi = (int *)malloc(n * sizeof (int));
*pi = 0;
```

* and * are both asterisks but they have different meaning:
* tells that *pi* is not an integer but a pointer to integer.
* marks the unary dereference operator (or indirection operator) applied on variable *pi*.

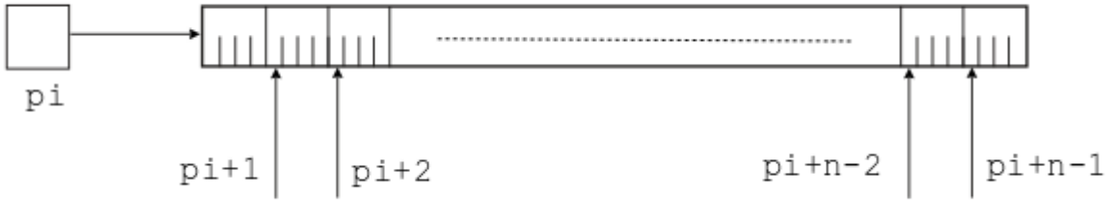
Expression **pi* may be read as the contents of memory field to which *pi* is pointing. So
*<pointer> = <expression>;
means: perform the expression and assign the result to the memory field to which the pointer is pointing. So **pi = 0* means that the first integer gets value 0.

```
int j = *pi;
```

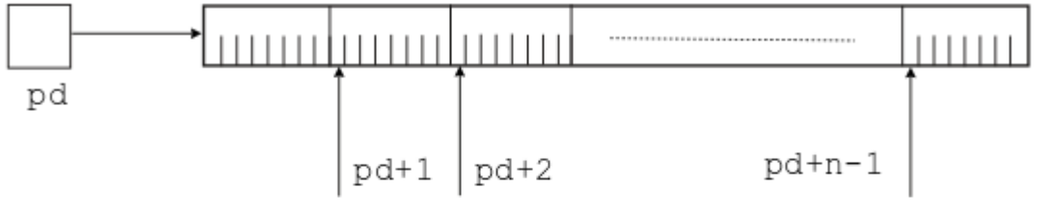
means: read the contents of memory field to which *pi* is pointing (i.e. an integer) and assign it to *j*.
Examples:

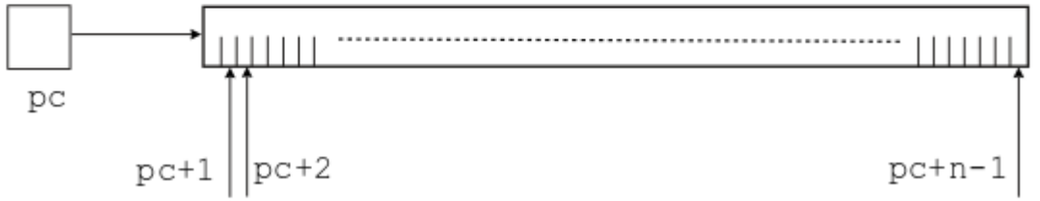
```
int j = *pi + 5; // read the integer to which pi is pointing, add to it 5 and assign the
                 // sum to j
int j = *pi * 5; // read the integer to which pi is pointing, multiply it with 5 and assign
                 // the result to j (the precedence of dereference is higher than the
                 // precedence of multiplication, therefore we do not need parentheses)
```

Pointers (6)



An integer may be added to a pointer. But although *pi* is the address of the first byte, *pi + 1* is not the address of the second byte – it is the address of the first byte of second integer. Similarly, *pi + 2* is the address of the first byte of third integer and *pi + n – 1* is the address of the first byte of last integer. Adding of an integer *k* to a pointer means that the pointer is shifted to the k-th unit (and not to the k-th byte) on the memory field.





In case of *char *pc*, of course, *pc + 1* means shifting to the second byte.

Subtracting of integers from pointers (like pi – 1) is performed similarly.
Arithmetics with *void* * pointers is not allowed: as a void * pointer may point to any data, we cannot now how to shift.

Pointers (7)

Now, knowing the deference operator and the pointer arithmetic, we are able to work with the allocated memory.

Example: write a function that creates an array of pseudo-random numbers. The dimension of array is the input parameter. The function must return the pointer to the created array.

```
int *RandomArray(int n) // int * is the type of return value
{
  int *pRand = (int *)malloc(n * sizeof(int));
  for (int i = 0;  i < n;  i++)
  {
        *(pRand+i) = rand(); // assign to integer to which pRand + i is pointing,
                             // we need parentheses because the precedence of deference
                             // is higher than the precedence of addition but we need to
                             // increment the pointer first
   }
   return pRand; // deallocation of array is the task for function that calls RandomArray
}
```

Usage:

```
int *pr = RandomArray(100);
……..
free(pr);
```

Pointers (8)

Exercise:

Write a function that creates the reverse of given string, for example if the input string is "abcd" then the result is "dcba". Write also *main* to test the result.

Tips:

- The prototype should be
  ```
  char *Reverse(char *);
  ```
- In *main* use standard function *printf* for showing the result. The format specifier for strings is *%s*, for example:
  ```
  char *pc; // pointer to a string
  printf("%s\n", pc); // prints the string to which pc points
  ```
- In *main* instead of input from keyboard with *gets_s* generate input data with function *CreateRandomString* (see the next slide). It simplifies the debugging.
- Your function must at first find the number of characters in input string (remember about the terminating zero), then allocate the memory for result and at last fill it in reverse order.
- Checking of the input data is obligatory. If the input pointer is zero or if it points to byte that contains zero, this input pointer is also the output value.

Pointers (9)

```
char *CreateRandomString(int n)
{ // n is the length of requested string. It does not include the terminating zero byte.
  const char alphabet[] = "abcdefghijklmnopqrstuvwxyz";
    // This initialization will be discussed later. Here it is enough to know that
    // alphabet[0] gets value 'a', alphabet[1] gets value 'b', etc.
  char *pResult = (char *)malloc(n + 1);
       // Remember: the function that calls CreateRandomString must take care of
       // deleting the memory field allocated here.
  for (int i = 0; i < n; i++)
  {
        *(pResult + i) = alphabet[rand() % 26];
        // The English alphabet contains 26 characters
        // When dividing with 26, the remainder is in range 0…25
  }
  *(pResult + n) = 0; // set the terminating 0
  return pResult;
}
```

Pointers (10)

Exercise:
Write a function that reformats a name from pattern <*given_name*><*space*><*family_name*> to pattern <*family_name*><*comma*><*space*><*given_name*>, for example from "John Smith" to "Smith, John". Write also *main* to test the result.

Tips:

- The protoype should be:
  char *ReformatName(char *);
- The input string is considered to follow the input pattern, i.e. there are no several spaces, missing components, etc. Therefore the specific cheking of input data correctness is not needed: if the input pointer is not zero and the byte to which it points does not contain zero, the input is considered to be correct.
- In *main* use standard function *printf* for showing the result.
- In *main* instead of input from keyboard with *gets_s* generate input data with function *CreateRandomName* (see the next slide). It simplifies the debugging.
- Your function must at first find the number of characters in input name, then allocate the memory for result (do not forget about the additional byte for comma) and at last copy the components from the original string into the result.

Pointers (11)

```
char *CreateRandomName(int n1, int n2)
{ // n1 – length of given name, n2 – length of family name
  if (n1 <= 0 || n2 <= 0)
  {
      return 0;
  }
  const char alphabet[] = "abcdefghijklmnopqrstuvwxyz";
  char *pResult = (char *)malloc(n1 + n2 + 2); // names + space + terminating 0
  for (int i = 0; i < n1; i++)
  {
      *(pResult + i) = alphabet[rand() % 26]; // given name
  }
  *pResult = toupper(*pResult); // must start with uppercase letter
  *(pResult + n1) = ' ';
  for (int i = 0; i < n2; i++)
  {
      *(pResult + n1 + 1 + i) = alphabet[rand() % 26]; // family name
  }
  *(pResult + n1 + 1) = toupper(*(pResult + n1 + 1)); // must start with uppercase letter
  *(pResult + n1 + n2 +1) = 0; // terminating zero
  return pResult;
}
```

Pointers (12)

Pointers may be incremented and decremented.

```
int n, *pi;
…… // find the actual value of n
pi = (int *)malloc(n * sizeof (int)); // pi points to the first integer
pi++; // now pi points to the second integer, the alternatives are pi = pi +1
      // and pi += 1
pi--;  // now pi is shifted back to the first integer, the alternatives are pi = pi -1
      // and pi -= 1
*pi++ = 0; // the first integer is set to 0 and the pointer is shifted to the second integer
           // actually we have made two assignments here:
           // *pi = 0;
           // pi += 1;
*++pi = 0; // the pointer is shifted to the third integer which is also set to zero,
           // the second integer is not accessed:
           // pi += 1;
           // *pi = 0;
```

Important:

```
free(pi); // error, pi is not pointing to the first byte of allocated memory field
```

Pointers (13)

Example: write a function that creates an array of pseudo-random numbers. The dimension of array is the input parameter. The function must return the pointer to the created array.

```
int *RandomArray(int n) // int * is the type of return value
{ // compare with slide Pointers (7)
  int *pRand = (int *)malloc(n * sizeof(int)), i = 0;
  for (int *q = pRand;  i < n;  i++)
  { // introduce auxiliary variable q
        *q++ = rand();
   }
   return pRand; // returns the value to the first byte
}
```

Usage:

```
int *pr = RandomArray(100);
……..……
free(pr);
```

The argument of function *free* must be equal with the value returned by *malloc*. It is not possible to free only a part of allocated memory field.

When freeing a memory field you also destroy the data stored on it.

Pointers (14)

Let

```
char c, *pc = (char *)malloc(2);
*pc = 'M';
*(pc + 1) = 'A';
```

Expression	Resulting c	Resulting pc	Resulting array
c = *pc++;	M	pc + 1	MA
c = ++*pc;	N	pc	NA
c = *++pc	A	pc + 1	MA
c = (*pc)++;	M	pc	NA
c = ++*pc++;	N	pc + 1	NA
c = ++*++pc;	B	pc + 1	MB

Do not try to be too clever. Use increment and decrement with pointers only if you are sure that you understand what will happen when the program is running.

Pointers (15)

Example: suppose that the first n characters of two strings match (for example, for strings *"programmer"* and *"programming"* n is 8). Write a function to find n.

```
int Match(char *p1, char *p2)
{
  char *q1 = p1;  // remember the initial value of p1
  while (*q1++ == *p2++);
  return q1 – p1 – 1; // subtraction of two pointers
}
```

Subtraction of two pointers has sense if and only if they both point to bytes from the same memory field. The result is the number of objects (i.e. characters, integers, etc.) between them.

Turn attention that in this example we increment the value of pointer *p2*. It is not an error here because *p2* is a local variable:

```
char *pWord1, *pWord2;
……….
int n = Match(pWord1, pWord2); // value of pWord2 is assigned to formal parameter p2,
                               //  function Match changes the value of p2 but pWord2
                               // keeps its value
```

Pointers (16)

Pointer with value 0 is actually a pointer without value. It is allowed to write:

```
int *p = 0;
int *p = NULL; // #define NULL 0 is in stdio.h
int *p = nullptr; // in C++
```

If *malloc* returns null pointer, it has failed because the memory is exhausted.

If your program crashes with message "access violation", then in most cases you have tried to apply the dereference operator to a pointer with wrong value or without value.

Relational operations (<, <=, >, >=, ==, !=) between pointers are allowed but they have sense only when the operands points to the bytes from the same memory field. It is also possible to compare the pointer with zero:

```
if (p) { …. } // do if p is not zero, actually if (p != NULL) or if (p != 0) or if (p != nullptr)
if (!p) { ….} // do if p is zero
```

Pointers may be the operands of logical operations (&&, ||, !) in usual way: pointer without value (i.e. null pointer) is FALSE and non-null pointer is TRUE.

Pointers (17)

It may happen that the allocated memory field turns out too short or senselessly long. In that case helps standard function *realloc*:

<pointer_to_new_field> = realloc(<pointer_to_old_field>, <new_number_of_bytes>);

Example:

```
int *pi = (int *)malloc(n * sizeof(int));
……..
pi = (int *)realloc(pi, m * sizeof(int)); // m may be greater or less than n
```

Pointers (18)

Instead of *malloc* you may use standard function *calloc:*

<pointer_to_allocated_memory> = (<item_type>)calloc(<number_of_items_to_allocate>,
<sizeof(<item_type>));

Examples:

int *pi = (int *)calloc(100, sizeof(int)); // memory for100 integers

char *pc = (char *)calloc(100, 1);

Function *calloc* initializes all the allocated bits to zero. *malloc* does not perform any initializations.

C++ has its own tools for memory allocation:

<pointer_to_allocated_memory> = new <item_type>[<number_of_items_to_allocate>];

Examples:

int *pi = new int[100]; // memory for 100 integers

int **ppi = new int *[100]; // memory for 100 pointers that will point to integers

To release memory allocated with the *new* operator use *delete* operator:

delete <pointer_to_allocated_memory>;

Example:

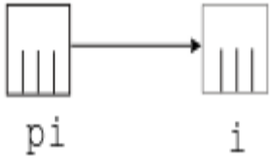
delete pi;

Pointers (19)

Let

```
int i, *pi;
pi = &i;
```

The unary & operator (address-of operator) gives the address on variable or in other word: the pointer to the variable:



Now we may write

```
i = 10;
```

or

```
*pi = 10;
```

the result is the same.

The address-of operator is very useful when we need to write functions with more than one output values.

Pointers (20)

The following function is written for interchanging of two values, but it is useless:

```
void swap(int x, int y)
{
  int z = y; // suppose x is 5 and y is 6, then now z is 6
  y = x; // now y is 5
  x = z; // now x is 6, yes we have interchanged the values
} // leave the function, local x and y are deleted, wasted efforts

int a = 5, b= 6;
swap(a, b);
printf("%d %d\n", a, b); // prints 5 6
```

Solution:

```
void swap(int *px, int *py)
{
  int z = *py; // py points to b, we read the value of b and assign it to z
  *py = *px; // px points to a, we read this value and store to the field to where py points
  *px = z; // store value of z to the field to where px points
}
swap(&a, &b);
printf("%d %d\n", a, b); // prints 6 5
```

Pointers (21)

Exercise:
Write a function that checks the input integer array and computes the sum of its members. Write also *main* to test the code. The prototype is

```
int CheckSum(int *, int, int *);
```

where the first parameter is the pointer to array, the second parameter is the number of elements in array and the third (actually output) parameter is the pointer to integer for storing the error code.

The array is incorrect, if some of its members are negative (error code 1), zero (error code 2), greater that 1000 (error code 3). The array is also incorrect if the pointer to it is zero (error code 4) or the array is empty (i.e. zero elements, error code 5). In that case instead of sum the function must return 0. If the input array was correct, error code is 0.

Mark the errors with *#define* preprocessor constants (see slide *Informing about errors (1)* from chapter *Deeper into C*).

The *main* function must include the *switch* statement (see slides *Multiple choice* from chapter *Deeper into C*) for analyzing the errors and print detailed messages.

To test use function *CreateRandomArray* (see the next slide). With it you can generate arrays containing negative elements or too large elements (but not zero elements).

Pointers (22)

```
int *CreateRandomArray(int n, int lower, int upper)
{ // n is the length of array to be generated
   // The random numbers will be in range lower….upper
   // lower and upper may be negative
  if (n <= 0 || lower >= upper)
  {
        return 0;
  }
  int *pResult = (int *)malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
 {
        *(pResult + i) = lower + rand() % (upper - lower + 1);
  }
  return pResult;
}
```

Pointers (23)

As now we know pointers and address-of operator, we may use a more convenient standard function for input:

scanf(<format_string>, <list_of_pointers_to_locations_of_data>);

Examples:

```
int i, i1, i2;
double x;
printf("Type input values\n");
scanf("%d", &i); // if you type 100 and ENTER, to i is assigned value 100
scanf("%d %d", &i1, &i2); // if you type 50 60 (space between them and ENTER
                          // at the end, then i1 gets value 50 and i2 60
scanf("%lg", &x); // to input value 1000000 you may type it in different ways:
                  // 1000000.0 or 1000000 or 1e6 or 1E6
```

Generally, the format specifiers are as for *printf*.

You may use *scanf* for text data input (specifier *%s*) but there are some specific problems: it reads input characters only until the first space. It is better use to use *gets_s*.

The full specification of *scanf* is on http://www.cplusplus.com/reference/cstdio/scanf/.

In Visual Studio the *scanf* is considered to be unsafe and the compiler refuses to accept it. To suppress this error start your source code file with line

```
#pragma warning ( disable : 4996 )
```

Pointers (24)

A table (matrix) is a two-dimensional array, for example:

```
int Table[10][5]; // 10 row, 5 columns
for (int i = 0;  i < 10;  i++)
{ // fill with pseudo-random integers
      for (int j = 0;  j < 5;  j++)
      {
            Table[i][j] = rand();
       }
}
```

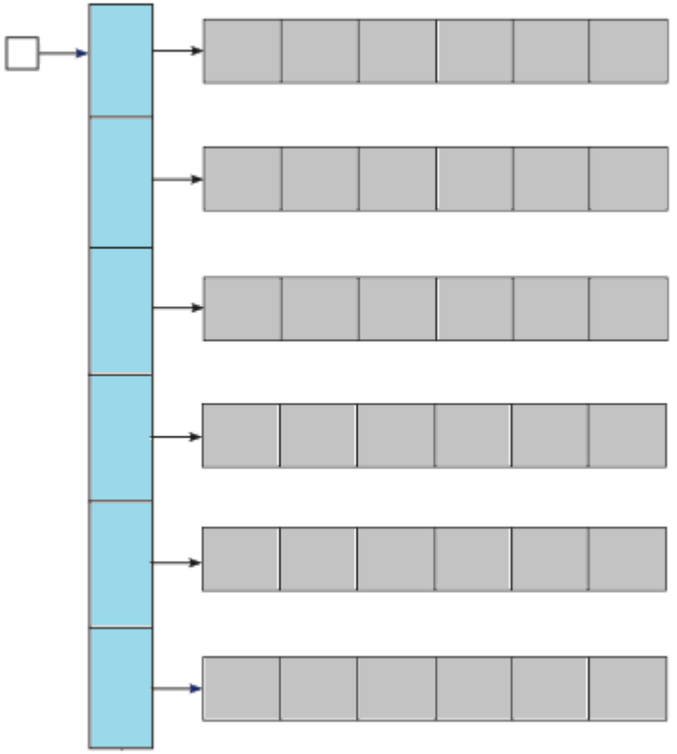
If the dimensions are not known beforehand:

```
int *pTable = (int *)malloc(nRows * nColumns * sizeof(int));
for (int i = 0;  i < 10;  i++)
{ // fill with pseudo-random integers
      for (int j = 0;  j < 5;  j++)
      {
            *(pTable + i + j) = rand();
       }
}
```

If *nRows* and *nColumns* are large, pTable points to a colossal chunk of memory. This approach is not advised.

Pointers (25)

The much better way is to present the table with the following data structure:



Each row (gray) is a separate memory field. To tie them together, we need a memory field containing pointers (blue).

To access the table we need only one variable: the pointer to array of pointers to rows (white). Its type is *int* **.

To create the table we have at first allocate memory for the array (or vector) of pointers and then fill itt with pointers to rows.

```
int **CreateTable(int nRows, int nColumns)
{ // allocate memory for table and build the data structure
  int **ppTable = (int **)malloc(nRows * sizeof(int *));
  for (int i = 0;  i < nRows;  i++)
        *(ppTable + i) = (int *)malloc(nColumns *sizeof(int));
   return ppTable;
}
```

Pointers (26)

ppTable is the pointer to pointers that point to integers. Consequently it is of type *int* **:

```
int **ppTable;
```

ppTable at this moment points to nothing. The memory field we need will contain nRow pointers that will point to arrays of integers, i.e. the type of elements of this array is *int* *. After

```
ppTable = (int **)malloc(nRows * sizeof(int *));
```

we have 

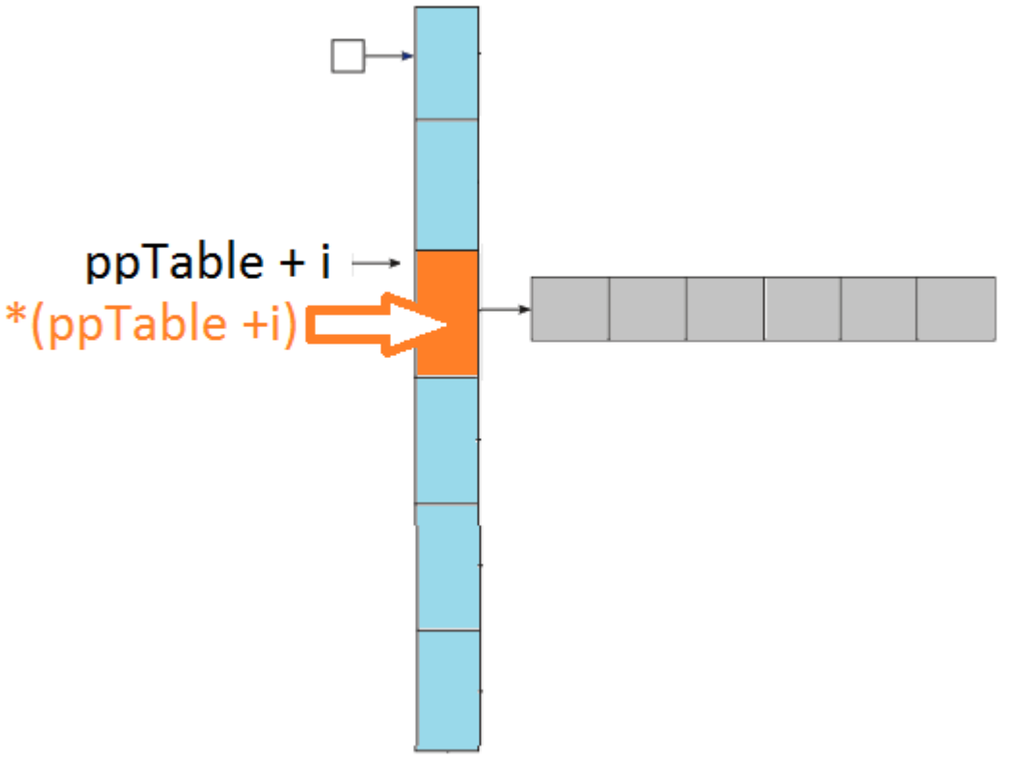
The array we got is not initialized and we have no memory for integer members of our table. Therefore we have to write the loop

```
for (int i = 0;  i < nRows;  i++)
        *(ppTable + i) = (int *)malloc(nColumns *
                                        sizeof(int));
```

*(int *)malloc(nColumns * sizeof(int))* gives us the pointer to array of nColumn integer.

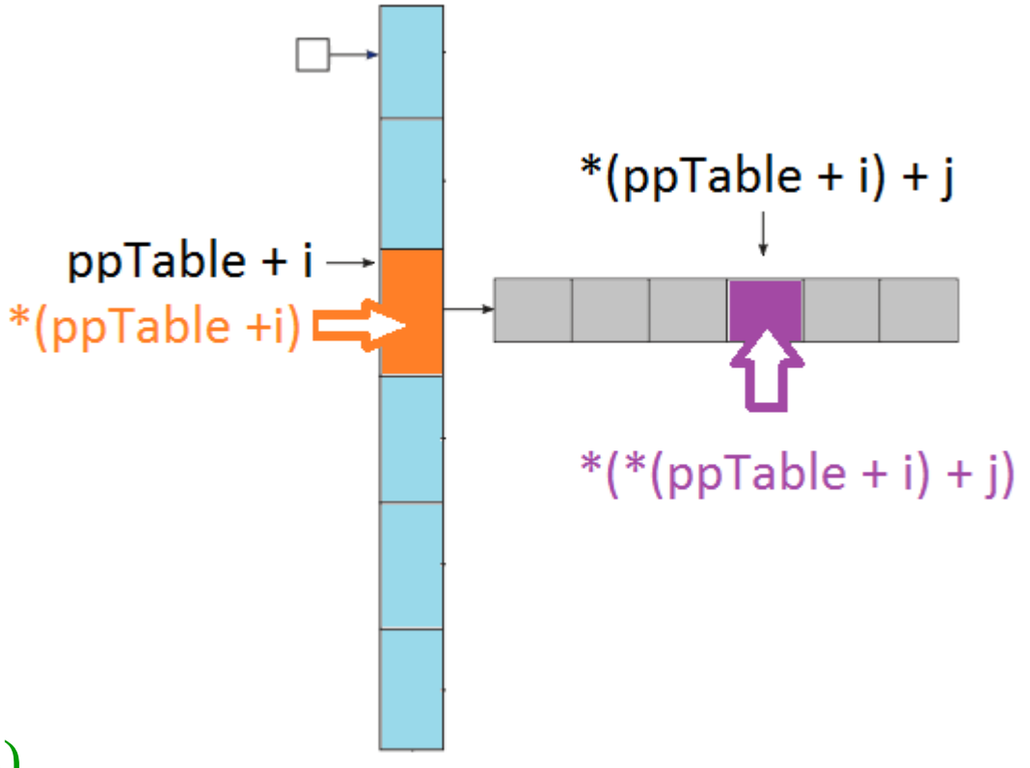
ppTable + i points to the i-th member in the array of pointers.

**(ppTable +i) = <expression>* means that the result of expression must be stored as the i-th element of the array of pointers.



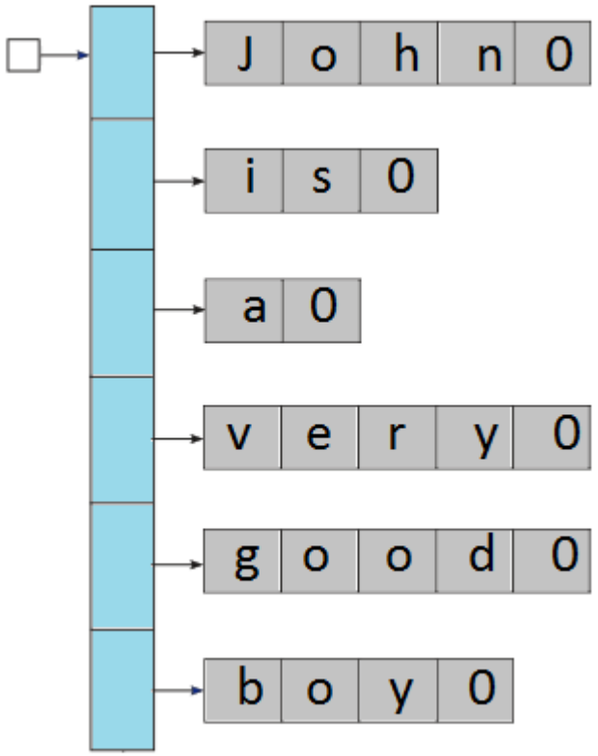
Pointers (27)

```
void FillTable(int **ppTable, int nRows, int nColumns)
{
  for (int i = 0;  i < nRows;  i++)
  {
        for (j = 0;  j < nColumns;  j++)
        {
            *( *(ppTable + i) + j) = rand();
         }
   }
}
void DeleteTable(int **ppTable, int nRows)
{
  for (int i = 0;  i < nRows;  i++)
      free(*(ppTable +i));
   free (ppTable); // Do not delete in reverse order!
}
```



Pointers (28)

Exercise: write a function that splits a sentence into a table of words. For example, from input sentence "John is a very good boy" we should get a table like:



The prototype of the function must be:

char **SentenceSplit(char *, int *);

where the first parameter is the pointer to input sentence (string) and the second parameter is the pointer to second output value presenting the number of words. The function must return the pointer to the table.
Write also the *main* function for testing and printing the results. Use function *CreateRandomSentence* (see the next slide) to generate test data. Usage example:
*char *p = CreateRandomSentence(5, 3, 4, 6, 1, 2);*
we get a sentence of 5 words; 3, 4, 6, 1, 2 are the lengths of words.

Tips:

- The sentence may include any number of words. The words are separated by one (and only one) space. The sentence does not contain punctuation marks like comma or point. The only allowed characters are space and the uppercase and lowercase English letters.
- If the first character of input sentence is the terminating zero, the sentence is empty and the return values are zeroes.
- The function must check the correctness of input parameters (no null pointers)!

Pointers (29)

```
char *CreateRandomSentence(int nWords, ...)
{ // this code is out of scope of this course.
   // #include "stdarg.h" is needed
   // nWords is the number of words, it is followed by sequence specifying the word lengths
   // nWords may be any positive number
  va_list indic;
  va_start(indic, nWords);
  int nChars = 0, j = 0;
  for (int i = 0; i < nWords; i++, nChars += va_arg(indic, int) + 1);
  const char alphabet[] = "abcdefghijklmnopqrstuvwxyz";
  char *pResult = (char *)malloc(nChars);
  va_start(indic, nWords);
  for (int i = 0; i < nWords; i++)
  {
        int nWord = va_arg(indic, int);
        for (int k = 0; k < nWord; k++, *(pResult + j++) = alphabet[rand() % 26]);
        *(pResult + j++) = ' ';
  }
  *(pResult + nChars - 1) = 0;
  return pResult;
}
```

Pointers (30)

Suppose we have a function with prototype

```
void fun(int *);
```

Then, as we know, we can write:

```
int *pi = (int *)malloc(1024 * sizeof(int));
fun(pi);
```

We have also:

```
int mi[1024]; // not dynamical, with size set beforehand
```

How to use *fun* for processing of *mi*?

```
fun(&mi[0]); // pointer to the first byte of mi
```

or simply

```
fun(mi);
```

The name of array is handled as pointer. Actually, array[0] is the syntactic shorthand of *(array + 0).

We may write in easy to understand mode

```
scanf("%d", &mi[1]);
mi[1] = 0;
```

or the same in a bit unaccustomed mode

```
scanf("%d", mi + 1);
*(mi + 1) = 0;
```

Pointers (31)

As *array[0]* is the syntactic shorthand of **(array + 0)*, we may also write instead of:

```
int *pi = (int *)malloc(1024 * sizeof(int));
for (int i = 0;  i < 1024;  i++)
    *(pi + i) = 0;
```

with using shorthand:

```
for (int i = 0;  i < 1024;  i++)
    pi[i]= 0;
```

Function FillTable from slide *Pointers (27)* may be rewritten as:

```
void FillTable(int **ppTable, int nRows, int nColumns)
{
  for (int i = 0;  i < nRows;  i++)
  {
        for (j = 0;  j < nColumns;  j++)
        {
            ppTable[i][j]= rand();
         }
   }
}
```

Operator precedence (1)

Precedence	Operator	Description	Associativity
1	++ and -- ( ) [ ]	Increment and decrement, postfix Function call Reading element from array	Left -> Right
2	++ and -- - ! (type) * & sizeof	Increment and decrement, prefix Sign conversion Logical NOT Type cast Dereference Address-of Size-of	Right->Left
3	* / %	Multiplication Division Modulus	Left -> Right
4	+ -	Addition Subtraction	Left -> Right

Operator precedence (2)

Precedence	Operator	Description	Associativity
5	<= < >= >	Less or equal Less Greater or equal Greater	Left -> Right
6	== !=	Equal Not equal	Left -> Right
7	&&	Logical AND	Left -> Right
8	||	Logical OR	Left -> Right
9	?:	Conditional	Right->Left
10	= += -= *= /* %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right->Left
11	,	Comma	Left -> Right

Standard functions for memory management (1)

```
#include "memory.h"
```

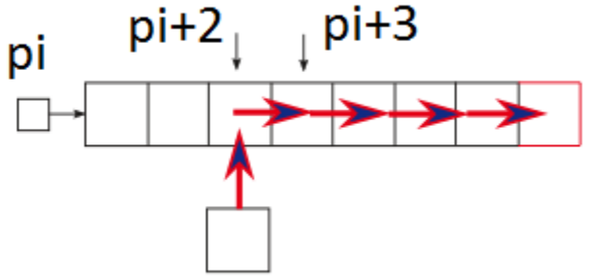
memcpy(<pointer_to_destination>, <pointer_to_source>, <number_of_bytes_to_copy>);

Copies from one memory field to another. Example:

```
int mi[] = { 0, 1, 2, 3, 4, 5, 6 };
int *pi = (int *)malloc(7 * sizeof(int));
memcpy(pi, mi, sizeof(int) * 7); // now *pi is 0, *(pi + 1) is 1, etc.
```

The memory fields must not overlap. If they do, the results of *memcpy* are unpredictable.

Function *memmove* is slower but in case of overlapping memory fields copies correctly:



Here we need to insert a new value to position pi + 2. As it is already occupied, we need to shift all the values starting from position pi +2 one position to right. Of course, we must be sure that there is at least one free position at the end of array. If not, our program will fail.

memmove has the same arguments as *memcpy*:

```
memmove(pi + 3, pi + 2, 5 * sizeof(int));
*(pi + 2) = 100;
```

Standard functions for memory management (2)

memset(<pointer_to_field_to_fill>, <value_to_set>, <number_of_bytes_to_set>);

Sets each byte of the memory field to the specified value. Example:

```
int mi[8];
memset(mi, 0, sizeof(int) * 8); // now mi is filled with zeroes.
```

The value to set is automatically converted to type *unsigned char*.

int <result> = memcmp(<pointer_to_field_1>, < pointer_to_field_2>,
<number_of_bytes_to_compare>);

Compares bytewise the specified memory fields. Returns:

- 0 if the memory fields match
- a negative integer if the first byte that does not match has a smaller value in *field_1* than in *field_2*
- a positive integer if the first byte that does not match has a greater value in *field_1* than in *field_2*

Examples:

```
int mi1[] = { 0, 1, 2, 3, 4, 5, 6 };
int mi2[] = { 0, 1, 2, 3, 4, 5, 6 };
int mi3[] = { 0, 1, 2, 3, 6, 5, 4};
int r1 = memcmp(mi1, mi2, 7 * sizeof(int)); // result is 0
int r2 = memcmp(mi2, mi3, 7 * sizeof(int)); // result is a negative integer
int r3 = memcmp(mi3, mi2, 7 * sizeof(int)); // result is a positive integer
```

Standard functions for memory management (3)

Exercise: let us have a text that contains a list of names separated by comma, for example *"John,Mary,James,Elizabeth"*. Write a function that inserts a new name to the beginning of the list. The prototype of this function must be:

```
int InsertName(char *pNames, int n, char *pNameToInsert);
```

Here *n* is the number of bytes in memory field to which *pName* points.

The return value is 0 (i.e. *FALSE*), if there was no enough memory for the expanded list. If the name was successfully inserted, the return value is 1 (i.e. *TRUE*).

Write also the *main* function for testing and printing the results.

Tips:

- Use standard function *scanf* to read the two texts from keyboard (here it is possible because our texts do not contain spaces).
- Use standard function *memmove* to expand array *pNames*.
- Do not forget that all the texts have terminating zero.

Example: if *pNames* points to string *"John,Mary,James,Elizabeth"* and *pNameToInsert* points to string *"David"*, at the end of function *pNames* should point to string *"David,John,Mary,James,Elizabeth"*. In this example the function failes (returns *FALSE*) if *n <32*.

Standard functions for memory management (4)

Exercise: let us have a text that contains a list of names separated by comma, for example *"John,Mary,James,Elizabeth"*. Write a function that removes one of the names. The prototype of this function must be:

```
int RemoveName(char *pNames, char *pNameToRemove);
```

The return value is 0 (i.e. *FALSE*), if the name to remove was not found. If the name was successfully removed, the return value is 1 (i.e. *TRUE*). Write also the *main* function for testing and printing the results.

Tips:

- Use standard function *scanf* to read the two texts from keyboard (here it is possible because our texts do not contain spaces).
- Use standard function *memmove* to compress array *pNames*.
- Use standard fuction *memcmp* to compare names.
- Do not forget that all the texts have terminating zero.
- When testing, try to remove:
  - name from the middle of input text.
  - the first name.
  - the last name.

Example: if *pNames* points to string *"John,Mary,James,Elizabeth"* and *pNameToRemove* points to string *"James"*, at the end of function *pNames* should point to string *"John,Mary, Elizabeth"*.

String constants and constant pointers (1)

Texts enclosed in quotation marks like *"abc"* are string constants (or string literals). The compiler builds the string (i.e. allocates the memory, fills it with specified characters and appends the terminating zero). *""* is the empty string containing only the zero. If the string constant is very long, type backslash right after the last character of row and continue from the very beginning of the next row.

To initialize an array with needed text use the following rule:
char <array_name>[] = <string_constant>;

Example:
```
char text[] = "Hello!\n";
```

The same result we may get writing:
```
char text[] = {  'H', 'e', 'l', 'l', 'o', '!', '\n', '\0' };
```

Later we may replace characters, for example:
```
text[5] = '.';
```
because array *text* contains just a copy of the original constant. But we may not change the number of characters. If we declare the array with modifier *const*, the later modification is impossible:
```
const char text[] = "Hello,\
everybody!";
text[5] = ' '; // error
```

String constants and constant pointers (2)

It is possible to get the pointer to string constant. However:

```
char *p = "Hello!\n"; // error
```

Correct is:

```
const char *<pointer_name> = <string_constant>;
```

Objects to which a constant pointer points are unchangeable. Therefore for example

```
*(p + 5) = '.'; // error
```

but it is allowed to set a constant pointer to point to another object:

```
p = "Good bye!\n"; // correct
```

It is possible to ban everything:

```
const char * const <pointer_name> = <string_constant>;
```

means that neither changing the object nor resetting the pointer is not allowed. Example:

```
const char * const p = "Hello\n";
*(p + 5) = '.'; // error
p = "Good bye!\n"; // error
```

String constants and constant pointers (3)

Let us have:

```
char text[] = "Hello!\n";
const char ctext[] = "Goodbye!\n";
```

and a function:

```
void fun(char *p) {
  printf("%c %c\n", *p, *(p + 1));
  ……..……
}
```

Then, as variable *text* is actually a pointer:

```
fun(text); // OK, prints "H e"
```

but as variable *ctext* is actually a constant pointer:

```
fun(ctext); // error
```

We may use casting:

```
fun((char *)ctext) // OK, prints "G o"
```

but it is risky – if function *fun* changes its input text, our constant gets corrupted.

Also

```
fun("Hello!\n"); // error
```

because string constant is actually a constant pointer.

```
fun((char *)"Hello!\n");
```

is formally correct, but if *fun* tries to change its input, the program will crash.

String constants and constant pointers (4)

Let us have:

```
char text[] = "Hello!\n";
const char ctext[] = "Goodbye!\n";
```

and a function:

```
void fun(const char *p) {
  printf("%c %c\n", *p, *(p + 1));
  ……..……
}
```

Then the following calls are correct:

```
fun(text); // although here variable text is not constant
fun(ctect);
fun("Hello!\n");
const char *pHello = "Hello!\n";
fun(pHello);
```

because now *fun* cannot change its input text.

String constants and constant pointers (5)

To initialize a multidimensional array use the following statement:

```
<data_type> <array_name>[<number_of_rows>] [< number_of_columns>] =
{
  {< sequence_of_initial_values_of_row_1> },
  {< sequence_of_initial_values _of_row_2> },
   ……..
   {< sequence_of_initial_values _of_row_n> }
};
```

For example:

```
int mmi[3][2] = { {1, 1}, { 2, 2}, {3, 3} };
```

Number of columns as well as number of rows are optional:

```
int mmi[][] = { {1, 1}, { 2, 2}, {3, 3} };
```

It is possible to set only a part of initial values. In that case the omitted values are initialized to zero but the dimensions must be present. Examples:

```
int mi[5] = { 1, 2, 3 }; // we get array 1, 2, 3, 0, 0
int mmi1[2][2] = { { 1, 1} }; // the second row is filled with zeroes
int mmi2[2][2] = { {1}, {1} }; // the second column is filled with zeroes
int mmi3[][2] = { {1}, {1} }; // as mm2, the compiler is able to guess the number of rows
int mmi4[2][] = { { 1, 1}, { 2 }}; // error, number of columns is missing
```

String constants and constant pointers (6)

To initialize a twodimensional array with needed texts use the following rule:

```
char <array_name>[][max_length_of_strings] = { <string_constant>, <string_constant>,
                                                ... , <string_constant>};
```

Example:

```
char Students[][8] = { "Mary", "John", "Richard" };
```

The same result we may get writing:

```
char Students[][8] = {
                      { 'M', 'a', 'r', y', '\0' },
                      { 'J', 'o', 'h', 'n', '\0' },
                       {'R', 'i', 'c', 'h', 'a', 'r', 'd', '\0' }
                     };
```

Later we may replace characters, for example:

```
Students[0][3] = 'a'; // the first name is now Mara
```

but we may not change the number of characters.

If we declare the array with modifier *const*, the later modification is impossible:

```
const char Students[][8] = { "Mary", "John", "Richard" };
Students[0][3] = 'a'; // error
```

String constants and constant pointers (7)

We have seen that in

```
char text[] = "Hello!\n";
```

variable *text* is handled as pointer to *char*. But in

```
char Students[][8] = { "Mary", "John", "Richard" };
```

variable *Students* is neither a pointer to *char* nor a ponter to pointers pointing to *char*. Physically, *Students* is the address of the first byte of array, i.e. it contains *'M'*. The question is, if we want to handle it as pointer, what is type of pointer?

A twodimensional array is an array of arrays. So, *Students[i]* is handled as pointer to the i-th row. For example, if we have

```
void fun(char *p) {
  printf("%s\n", p);
}
```

then

```
fun(Students[1]);  // prints "John"
```

Also, according to the general concepts of C:

```
fun(*(Students + 1)); // prints "John"
```

But this pointer arithmetic is possible only if the number of columns is known – *Students + i* means step of 8 bytes. Consequently the type of pointer corresponding to *Students* must somehow contain the number of columns.

String constants and constant pointers (8)

The type is:

char (* pointer_name) [number_of_colums]

here the parentheses are needed because the precedence of brackets is higher.

Example:

```
void fun(char (*p)[8], int nRow)
{
      for (int i = 0;  i < nRow;  i++)
            printf("%s\n", p + i);
 }
```

and we may call this functions as

```
fun(Students, 3);
```

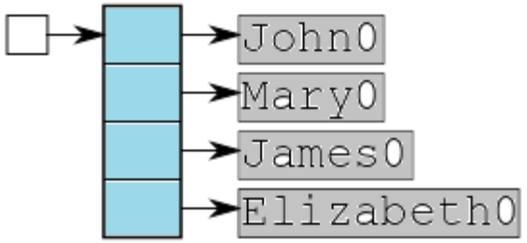
String constants and constant pointers (9)

const char *<array_name>[] = { <string_constant>, <string_constant>, ... ,
<string_constant>};

defines an array of constant pointers to string constants. Example:

const char *pStudents[] = { "John", "Mary", "James", "Elizabeth" };

Actually, we have got an array of pointers to strings created by the compiler:



Modifying of student names is not possible. For example if we try to replace name *Mary* with *Mara*, then:

*(pSudents[1] + 3) = 'a'; // error

However, we can replace a string constant with the other one:

pStudents[1] = "Mara"; // instead of "Mary"

*(pStudents +3) = "Richard"; // instead of "Elizabeth"

Important: initializing an array with string constant we make a copy of this constant. Initializing a pointer we do not make a copy, we just set the pointer to point to a locked memory field.

String constants and constant pointers (10)

Let us have a function with prototype

```
void fun(const char **, int);
```

and data

```
const char *pStudents[] = { "John", "Mary", "James", "Elizabeth" };
```

Then we may write

```
fun(pStudents, 4);
```

because actually *pStudents* points to compiler-created vector containing pointers pointing to string constants.

```
void fun(const char **ppŃames, int nNames)
{
  for (int i = 0;  i < nNames;  i++)
       printf("%s\n", *(ppNames + i);
   ……..……
 }
```

In our example **(ppNames +0)* points to *"John"*, **(ppNames + 1)* points to *"Mary"*, etc.

Standard functions for string handling (1)

```
#pragma warning ( disable : 4996 )
#include "string.h"
// see details from https://cplusplus.com/reference/cstring/
```

<integer_variable> = strlen(<const_pointer_to_string>);
returns the length (number of characters) in this string.

strcpy(<pointer_to_copy>, <const_pointer_to_source>);
makes the copy of source string.

Example:

```
char buffer[81];
gets_s(buffer);
int n = strlen(buffer);
char *pCopy = (char *)malloc(n + 1); // not n but n + 1
strcpy(pCopy, buffer);
```

Here at first we read a text into *buffer*. In most cases, the most of bytes in *buffer* stay empty. Using *strlen* we request the actual number of characters in input string. But the output value of *strlen* does not include the terminating zero. Therefore, for copy we need one byte more. At last we copy the input text (together with the terminating zero) to the allocated memory.

If the memory for copy is not allocated or is too short, the program will fail.

If the source and copy fields overlap, the result is unpredictable.

Standard functions for string handling (2)

strncpy(<pointer_to_copy>, <const_pointer_to_source>, <number_of_bytes_to_copy>);
copies only the specified number of bytes starting from the beginning of source string.
Terminating zero is not added.

Example:

```
char buffer[81], name[31];
printf("Type the name, max 30 characters\n");
gets_s(buffer);
int n = strlen(buffer);
if (n <= 30) // the user has followed the instructions
    strcpy(name, buffer);
else
{   // the user has ignored the instructions
      strncpy(name, buffer, 30); // copies bytes 0 … 29
      name[30] = 0; // add terminating zero
}
```

However, if the number of bytes to copy exceeds the length of source string, *strncpy* acts as *strcpy*.

Standard functions for string handling (3)

<pointer_to_found_character> = strchr(<pointer_to_string>, <character_to_find>);
<const_pointer_to_found_character> = strchr(<const_pointer_to_string>,
<character_to_find>);

searches the first occurence of specified character from the string and returns the pointer to it. If the character was not found, the result is null pointer.

Example:

```
char buffer[81];
printf("Type the given name and family name separated by space\n");
gets_s(buffer);
char *pSpace = strchr(buffer, ' ');
if (pSpace) {
  *pSpace = 0; // now we have 2 strings: buffer is the pointer to the given name and
                  // pSpace + 1 points to the family name.
  char *pGivenName = (char *)malloc(strlen(buffer) + 1);
  strcpy(pGivenName, buffer);
  char *pFamilyName = (char *)malloc(strlen(pSpace +1) +1);
  strcpy(pFamilyName, pSpace +1);
 *pSpace = ' '; // restore the original input
}
else
   printf("Input data not correct\n");
```

Standard functions for string handling (4)

<pointer_to_found_character> = strrchr(<pointer_to_string>, <character_to_find>);

<const_pointer_to_found_character> = strrchr(<const_pointer_to_string>,
<character_to_find>);

searches the last occurence of specified character from the string (i.e. starts searching from the end) and returns the pointer to it. If the character was not found, the result is null pointer.

Example:

```
int fun(const char *pBuf)
{  // two last characters in the buffer must be digits
    const char *p = strrchr(pBuf, ' ');
    if (p != 0 && isdigit(*(p + 1)) && isdigit(*(p + 2)) && *(p + 3) == 0)
       return 1;
     else
       return 0;
 }
```

Standard functions for string handling (5)

<pointer_to_found_character> = strpbrk(<pointer_to_string>,
<const_pointer_to_string_of_characters_to_find>);
<const_pointer_to_found_character> = strpbrk(<const_pointer_to_string>,
<const_pointer_to_string_of_characters_to_find>);

searches the first occurence of any of the characters to find and returns the pointer to it. If nothing was found, the result is null pointer. Example:

```
char buffer[81];
printf("Type the text\n");
gets_s(buffer); // we have to extract the number
char *pNumber = strpbrk(buffer, "0123456789"); // get pointer to the first digit
int n;
if (pNumber) {
    char *pSpace = strchr(pNumber, ' '); // if the number was the last, pSpace is 0
    if (pSpace) {
      *pSpace = 0;
      n = atoi(pNumber); // the extracted value as integer
      *pSpace = ' ';  // restore the buffer
     }
     else
        n = atoi(pNumber);
}
```

Standard functions for string handling (6)

<pointer_to_found_substring> = strstr(<pointer_to_string>,
<const_pointer_to_substring_pattern>);
<const_pointer_to_found_substring> = strstr(<const_pointer_to_string>,
<const_pointer_to_substring_pattern>);

searches the first occurence of specified pattern string and returns the pointer to it. If the substring was not found, the result is null pointer. Example:

```
char buffer[81];
printf("Type the names following the format <given name_1> <family name_1>, <given \
name_2> <family name_2>,…. \n"); // like Mary Clerk,John Smith,James Sailor
gets_s(buffer);
char *pJohn = strstr(buffer, "John "); // we want to extract the full name of John
if (pJohn) {
  char *p = strchr(pJohn + 5, ','); // 0 if John is the last in sequence
if (p)
    *p = 0; // replace comma with zero
char *pFullName = (char *)malloc(strlen("John ") + strlen(pJohn + 5) + 1);
strcpy(pFullName, "John ");
strcpy(pFullName + 5, pJohn + 5);
if (p)
    *p = ',';  // restore the buffer
}
```

Standard functions for string handling (7)

int <result>= strcmp(<const_pointer_to_string_1>, <const_pointer_to_string_2>);
compares the strings. Returns:

- 0 if the strings match
- a negative integer if the first character (i.e. the corresponding to it ASCII code) that does not match has a smaller value in *string_1* than in *string_2*.
- a positive integer if the first byte that does not match has a greater value in *string_1* than in *string_2*

It is told that if the result is negative, the first string is less than the second. In other words, it means that in a list sorted lexically the first string is located higher. And vice versa, if the result is positive, the first string is greater and located lower. So "Joan" is less than "John".

int <result>= strncmp(<const_pointer_to_string_1>, <const_pointer_to_string_2>,
<number_of_bytes>);
is similar, but compares no more than the specified number of bytes.

int <result>= stricmp(<const_pointer_to_string_1>, <const_pointer_to_string_2>);
compares without case sensitivity. For example:

stricmp("IDE", "ide");

returns zero.

Standard functions for string handling (8)

Example:

```
char *pAddresses[10];
printf("Type 10 addresses\n");
for (int i = 0;  i < 10;  i++)
{
    char buffer[81];
    gets_s(buffer);
    pAddresses[i]= (char *)malloc(strlen(buffer) + 1);
    strcpy(pAddresses[i], buffer);
}
int nTallinn = 0;
for (int i = 0;  i < 10;  i++)
{
  if (!strncmp(pAddresses[i], "Tallinn", 7)) // checks the first 7 bytes
     nTallinn++;
 }
 printf("%d persons live in Tallinn", nTallinn);
```

Standard functions for string handling (9)

strcat(<pointer_to_string_1>, <const_pointer_to_string_2>);
concatenates the strings, i.e. appends the copy of *string_2* to *string_1*. The terminating zero of *string_1* is overwritten by the first character of *string_2*.

Important: the memory field to which *pointer_to_string_1* points must be large enough to contain the result. If not, the program will fail.

Example:

```
char buffer1[81], buffer2[81];
printf("Please type the family name\n");
gets_s(buffer1);
printf("Please type the given name\n");
gets_s(buffer2);
char *pFullName = (char *)malloc(strlen(buffer1) + strlen(buffer2) + 2);
strcpy(pFullName, buffer2);
strcat(pFullName, " ");
strcat(pFullName, buffer1);
```

strncat(<pointer_to_string_1>, <const_pointer_to_string_2>,
<number_of_bytes_to_append>);
appends only the specified number of bytes starting from the beginning of *string_2*. If the length of *string_2* is less than the number of bytes to append, the complete *string_2* is appended.

Standard functions for string handling (10)

char *strtok(<pointer_to_string>, <const_pointer_to_string_of_delimiters>);

Here we suppose that our string consists of tokens (i.e. sequencies of characters) separated by delimiters. Examples:
"Here I am" tokens are *Here*, *I*, *am*; delimiter is space.
"*John:51*, *James:61*" tokens are *John*, *51*, *James*, *61*; delimiters are comma, colon and space.

Function *strtok* is for splitting the string into tokens.

```
char text[] = "Here I am";
char *pToken1 = strtok(text, " "); // pToken1 points to the first token
printf("%s\n", pToken1); // prints "Here"
for (int i = 0;  i < 10;  printf("%d ", text[i++]); // prints 71 101 114 101 0 73 32 97 109 0
                                                    // the first delimiter is replaced by zero
char *pToken2 = strtok(0, " "); // pToken2 points to the second token
printf("\n%s\n", pToken2); // prints "I"
for (int i = 0;  i < 10;  printf("%d ", text[i++]); // prints 71 101 114 101 0 73 0 97 109 0
                                                    // the second delimiter is replaced by zero
char *pToken3 = strtok(0, " "); // pToken3 points to the third token
printf("\n%s\n", pToken3); // prints "am"
for (int i = 0;  i < 10;  printf("%d ", text[i++]); // prints 71 101 114 101 0 73 0 97 109 0
char *pToken4 = strtok(0, " "); // no more tokens, pToken4 gets value 0
```

Standard functions for string handling (11)

"*John: 51*, *James: 61*" tokens are *John*, *51*, *James*, *61*; delimiters are comma, colon and space.

```
char text[] = "John: 51, James: 61";
char *pToken1 = strtok(text, ",: "); // pToken1 points to the first token
printf("%s\n", pToken1); // prints "John"
for (int i = 0;  i < 10;  printf("%d ", text[i++]);
    // prints 74 111 104 110 0 32 53 49 44 32 74 97 109 101 115 58 32 54 50 0
   // the first delimiter (colon) is replaced by zero
char *pToken2 = strtok(0, ",: "); // pToken2 points to the second token
printf("\n%s\n", pToken2); // prints "51"
for (int i = 0;  i < 10;  printf("%d ", text[i++]);
 // prints 74 111 104 110 0 32 53 49 0 32 74 97 109 101 115 58 32 54 50 0
 // the second delimiter (comma) is replaced by zero
```

In sequence of delimiters the first of them is replaced by zero, the following ones are ignored.

Standard functions for string handling (12)

Exercise:

Write a function that counts occurences of a character in a given string. For example, if the input string is *Jesse James* and the searched character is *e*, the function must return 3. Write also *main* to test the function.

Requirements:

- The prototype must be
  ```
  int CharFreq (char *, char);
  ```
- Use standard function *strchr.*
- The function must not crash if the input pointer is null or points to an empty string. In those cases the output value must be 0.

Tip:

You may use string constants to create actual parameters of your function. Thus input of test data from keyboard is not needed. The testing, for example may look like:

```
char input[] = "Jesse James";
printf("%d\n", CharFreq (input, 'e'));
```

Standard functions for string handling (13)

Exercise:

Write a function that counts occurences of all the English letters in a given string. Write also *main* to test the function.

Requirements:

- The prototype must be
  int *CharFreqs(char *);
- The output is the pointer to the array of occurences. The first member of array is the occurence of *a*, the second is the occurence of *b*, etc. For example, if the input string is *Butch Cassidy*, the array starts with 1, 1, 2, 1, 0, 0, …. The counting is case insensitive, i.e. the lowercase and uppercase letters are equivalent.
- Characters that are not English letters should be ignored.
- Use function *CharFreq* from the previous exercise.
- The function must not crash if the input pointer is null or points to an empty string. In those cases the output value must be 0.
- The *main* must print the results as a table, for example:
  a – 1
  b - 1
  c - 2
  …….

Tip:

Before counting convert the text into lowercase using the standard function *tolower*.

Standard functions for string handling (14)

Exercise:

Write a function that counts the number of words in a given string. Write also *main* to test the function.

Requirements:

- The prototype must be
  int WordFreq (char *);
  The argument specifies the input string.
- Use standard function *strchr.*
- Words in the input text are separated by one space.
- The function must not crash if one or the both of the input pointers are null or point to an empty string. In those cases the output value must be 0.

Standard functions for string handling (15)

Exercise:

Write a function that counts occurences of a given word in a given string. Write also *main* to test the function. Do not use the functions developed in the previous exercises.

Requirements:

- The prototype must be
  int WordSearch (char *, const char *);
- The first argument points to the text to analyze, the second to the word to find.
- Use standard function *strstr.*
- Words in the input text are separated by one space.
- The function must not crash if one or the both of the input pointers are null or point to an empty string. In those cases the output value must be 0.

Tips: do not forget to test the special cases when:

- The word is the first one.
- The word is the last one.
- There are words a part of which matches the given word. For example if we search *and* then in sentence *and the band started to play* the return value is 1.

Standard functions for string handling (16)

Exercise:

Write once more the function *SentenceSplit* specified on slide *Pointers (28)*. Write also *main* to test the function and to print the results.

Additional requirement:

- Use standard functions *strlen*, *strcpy*, *strchr*, *strtok*

Standard functions for string handling (17)

Exercise:

Using the results of function *SentenceSplit* from the previous exercise write a function that checks whether the given word is present in the table or not. Write also *main* to test the function and to print the results.

Requirements:

- The prototype must be
  int WordExist (char **, int, const char *);
  (the parameters are the pointer to table, the number of words in table and the pointer to word to search).
- Use standard function *strcmp*.
- The output value is 1 (TRUE, exists) or 0 (FALSE, not found).
- The function must not crash if one or the both of the input pointers are null or the second pointer points to an empty string. In those cases the output value must be -1.

Standard functions for string handling (18)

Exercise:

Write a function that replaces in a given string the first occurence of a given word with another word. For example, if the input string is *I asked Mary to call me* and the task is to replace *Mary* with *Elizabeth*, the function must create string *I asked Elizabeth to call me*. Write also *main* to test the function.

Requirements:

- The prototype must be
  char *WordReplace (char *, const char *, const char *);
  The return value is the pointer to the new string or 0 if the replacement failed.
- Use standard functions *strlen*, *strstr* and *strcat*.
- Words in the input text are separated by one space.
- The function must not crash if some of the input pointers is null or point to an empty string. In those cases the output value must be 0.

Tips: do not forget to test the special cases when:

- The word to replace is the first one.
- The word to replace is the last one.
- There are words a part of which matches the word to replace. For example if we must replace *and*, word *band* must be included into output without any changes.
- The replacing word is longer or shorter of the word to replace.