Arrays, Strings, and Pointers

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 04

B3B36PRG – Programming in C

Overview of the Lecture

- Part 1 – Arrays
  - Arrays
  - Variable-Length Array
  - Multidimensional Arrays
  - Arrays and Pointers

K. N. King: chapters 8 and 12

- Part 2 – Strings
  - String Literals and Variables
  - Reading Strings
  - C String Library

K. N. King: chapters 13

- Part 3 – Pointers
  - Pointers
  - `const` Specifier
  - Pointers to Functions
  - Dynamic Allocation

K. N. King: chapters 11, 12, 17

- Part 4 – Assignment HW 03
- Part 5 – Coding examples (optional)

Part I

Arrays

Outline

Arrays

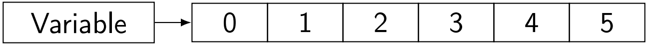
Variable-Length Array

Multidimensional Arrays

Arrays and Pointers

Array

- Data structure to store **a sequence of values of the same type**.



Array represents a continuous block of memory.

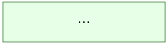
- The variable name (indentifier) represents the address of the memory where the first element of the array is stored.
- The array is defined as `type array_name[No. of elements]`.
  - No. of elements is an **constant expression**.
- In C99, the size of the array can be computed during the run time, that is why the array is called **Variable-Length Array** (**VLA**). *A non constant expression.*
- Array definition as a local variable allocates the memory on the stack.

If not defined as `static`*.*

- Array variable is passed to a function as a pointer (the address of the allocated memory).

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.



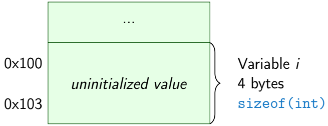
```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```

In the example, the variable allocation starts from the address 0 × 100 *for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.*

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.

```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```


...

0x100

0x103

uninitialized value

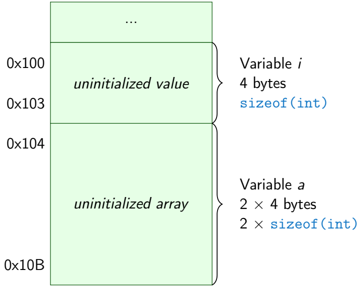
Variable *i*
4 bytes
`sizeof(int)`

In the example, the variable allocation starts from the address 0 × 100 for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.

```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```

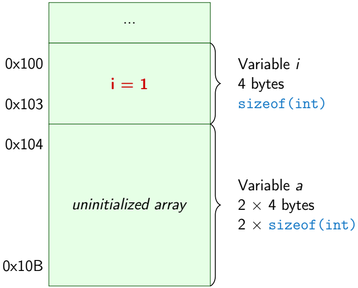


In the example, the variable allocation starts from the address 0 × 100 for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.

```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```

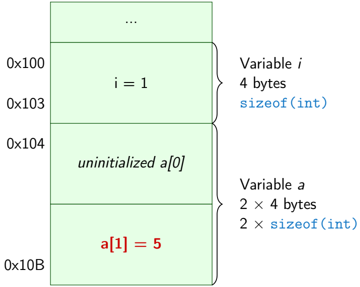


In the example, the variable allocation starts from the address 0 × 100 for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.

```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```

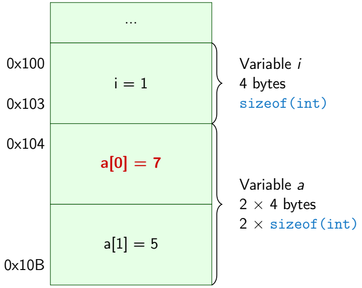


In the example, the variable allocation starts from the address 0 × 100 for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.

Array – Visualization of the Allocation and Assignment of Values

- An array type variable refers to the beginning of memory where individual array elements are allocated.
- Access to the array elements is realized by the index operator `[]` that computes the address of the particular element depending on the memory represent of the element type as `index * sizeof(type)`.

```c
int i;
int a[2];

i = 1;

a[1] = 5;
a[0] = 7;
```



Variable *i*
4 bytes
`sizeof(int)`

Variable *a*
2 × 4 bytes
2 × `sizeof(int)`

In the example, the variable allocation starts from the address 0 × 100 for visualization and understandability. Automatic variables on the stack are usually allocated from the upper address to the lower ones.

Arrays – Example 1/2 – Array Variable Definition

```c
#include <stdio.h>

int main(void)
{
   int array[10];

   for (int i = 0; i < 10; i++) {
      array[i] = i;
   }

   int n = 5;
   int array2[n * 2];

   for (int i = 0; i < 10; i++) {
      array2[i] = 3 * i - 2 * i * i;
   }

   printf("Size of array: %lu\n", sizeof(array));
   for (int i = 0; i < 10; ++i) {
      printf("array[%i]=%+2i \t array2[%i]=%6i\n", i, array[i], i, array2[i]);
   }
   return 0;
}
```

```
Size of array: 40
array[0]=+0     array2[0]=     0
array[1]=+1     array2[1]=     1
array[2]=+2     array2[2]=    -2
array[3]=+3     array2[3]=    -9
array[4]=+4     array2[4]=   -20
array[5]=+5     array2[5]=   -35
array[6]=+6     array2[6]=   -54
array[7]=+7     array2[7]=   -77
array[8]=+8     array2[8]=  -104
array[9]=+9     array2[9]=  -135
```

lec04/demo-array.c

Arrays – Example 2/2 – Array Variable Definition with Initialization

```c
#include <stdio.h>

int main(void)
{
   int array[5] = {0, 1, 2, 3, 4};

   printf("Size of array: %lu\n", sizeof(array));
   for (int i = 0; i < 5; ++i) {
      printf("Item[%i] = %i\n", i, array[i]);
   }
   return 0;
}
```

```
Size of array: 20
Item[0] = 0
Item[1] = 1
Item[2] = 2
Item[3] = 3
Item[4] = 4
```

lec04/array-init.c

- Array initialization

```c
double d[] = {0.1, 0.4, 0.5}; // initialization of the array

char str[] = "hallo"; // initialization with the text literal

char s[] = {'h', 'a', 'l', 'l', 'o', '\0'}; //elements

int m[3][3] = { { 1, 2, 3 }, { 4, 5 ,6 }, { 7, 8, 9 } }; // 2D array

char cmd[][10] = { "start", "stop", "pause" }; // we need to define no.  of columns
```

Arrays – Example 2/2 – Array Variable Definition with Initialization

```c
#include <stdio.h>

int main(void)
{
   int array[5] = {0, 1, 2, 3, 4};

   printf("Size of array: %lu\n", sizeof(array));
   for (int i = 0; i < 5; ++i) {
      printf("Item[%i] = %i\n", i, array[i]);
   }
   return 0;
}
```

```
Size of array: 20
Item[0] = 0
Item[1] = 1
Item[2] = 2
Item[3] = 3
Item[4] = 4
```

lec04/array-init.c

- Array initialization

```c
double d[] = {0.1, 0.4, 0.5}; // initialization of the array

char str[] = "hallo"; // initialization with the text literal

char s[] = {'h', 'a', 'l', 'l', 'o', '\0'}; //elements

int m[3][3] = { { 1, 2, 3 }, { 4, 5 ,6 }, { 7, 8, 9 } }; // 2D array

char cmd[][10] = { "start", "stop", "pause" }; // we need to define no.  of columns
```

Array Initialization

- An array (as any other variable) is not initialized by default.
- The array can be explicitly initialized by listing the particular values in `{` and `}`.

```
int a[5]; // elements of the array a are not initialized

/* elements of the array b are initialized
   to the particular values in the given order */
int b[5] = { 1, 2, 3, 4, 5 };
```

- In C99, **designated initializers** can be used to explicitly initialize specific elements only.
- Using designated initializers, the initialization can be in an arbitrary order.

```
int a[5] = { [3] = 1, [4] = 2 };

int b[5] = { [4] = 6, [1] = 0 };
```

Outline

Variable-Length Array (VLA)

- **C99** allows determining the array size during the program run time, not as compile-time constant expression, but the VLA cannot be initialized in the definition.
- Array size can be a function argument.

```
void fce(int n)
{
   // int local_array[n] = { 1, 2 }; initialization is not allowed
   int local_array[n]; // variable length array

   printf("sizeof(local_array) = %lu\n", sizeof(local_array));
   printf("length of array = %lu\n", sizeof(local_array) / sizeof(int));
   for (int i = 0; i < n; ++i) {
      local_array[i] = i * i;
   }
}
int main(int argc, char *argv[])
{
   fce(argc);
   return 0;
}
```

lec04/fce_var_array.c

Variable-Length Array (C99) – Example

```c
#include <stdio.h>
enum { ERROR_OK = 0, ERROR_NUMBER_VALUES = 100, ERROR_NUMBER = 101 };
int main(void)
{
   int i, n;
   printf("Enter the number of integers to be read: ");
   if (scanf("%d", &n) != 1 && n > 0) {
      return ERROR_NUMBER_VALUES;
   }

   int a[n]; /* variable length array */
   for (i = 0; i < n; ++i) {
      if (scanf("%d", &a[i]) != 1) {
         return ERROR_NUMBER;
      } // we always read n values or return ERROR_NUMBER
   }
   printf("Entered numbers in reverse order: ");
   for (i = n - 1; i >= 0; --i) {
      printf(" %d", a[i]);
   }
   printf("\n");
   return ERROR_OK;
}
```

lec04/vla.c

Outline

Arrays

Variable-Length Array

Multidimensional Arrays

Arrays and Pointers

Multidimensional Arrays

- Array can be defined as multidimensional, such as two-dimensional array for a matrix.

```
int m[3][3] = {
   { 1, 2, 3 },
   { 4, 5, 6 },
   { 7, 8, 9 }
};
```

```
Size of m: 36 == 36
1  2  3
4  5  6
7  8  9
```

```
printf("Size of m: %lu == %lu\n", sizeof(m), 3*3*sizeof(int));
for (int r = 0; r < 3; ++r) {
   for (int c = 0; c < 3; ++c) {
      printf("%3i", m[r][c]); // space only for 1-2 digit(s) numbers
   }
   printf("\n");
}
```

lec04/matrix.c

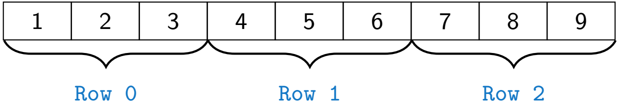
Multidimensional Array and Memory Representation

- Multidimensional array is **always** a continuous block of memory.

  For example, `int a[3][3];` *represents allocated memory of the size* `9*sizeof(int)`*, i.e., usually 36 bytes.*

```
int m[3][3] = { { 1, 2, 3 }, { 4, 5, 6}, { 7, 8, 9 } };

int *pm = (int *)m; // pointer to an allocated continuous memory block
printf("m[0][0]=%i m[1][0]=%i\n", m[0][0], m[1][0]); // 1 4
printf("pm[0]=%i pm[3]=%i\n", m[0][0], m[1][0]); // 1 4
```

`lec04/matrix.c`



- Two-dimensional array can be defined as pointer to a pointer, e.g., `int **a;`.
  - In general, a pointer (`int **a`) does not necessarily refer to a continuous memory.
  - Therefore, when accessing to `a` as to one-dimensional array

    `int *b = (int *)a;`

    the access to the second (and further) row is not guaranteed.
  - **It depends how the memory is allocated!**

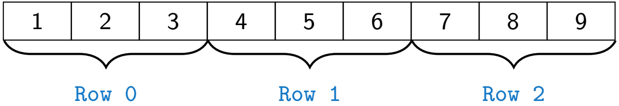
Multidimensional Array and Memory Representation

- Multidimensional array is **always** a continuous block of memory.

  For example, `int a[3][3];` *represents allocated memory of the size* `9*sizeof(int)`*, i.e., usually 36 bytes.*

```
int m[3][3] = { { 1, 2, 3 }, { 4, 5, 6}, { 7, 8, 9 } };

int *pm = (int *)m; // pointer to an allocated continuous memory block
printf("m[0][0]=%i m[1][0]=%i\n", m[0][0], m[1][0]); // 1 4
printf("pm[0]=%i pm[3]=%i\n", m[0][0], m[1][0]); // 1 4
```

lec04/matrix.c



- Two-dimensional array can be defined as pointer to a pointer, e.g., `int **a;`.
  - In general, a pointer (`int **a`) does not necessarily refer to a continuous memory.
  - Therefore, when accessing to `a` as to one-dimensional array

    `int *b = (int *)a;`

    the access to the second (and further) row is not guaranteed.
  - **It depends how the memory is allocated!**

Initialization of Multidimensional Array

- Multidimensional array can also be initialized during the definition.

Two-dimensional array is initialized row by row.

- Using designated initializers, the other elements are set to 0.

```c
void print(int m[3][3])
{
   for (int r = 0; r < 3; ++r) {
      for (int c = 0; c < 3; ++c) {
         printf("%4i", m[r][c]);
      }
      printf("\n");
   }
}

int m0[3][3];
int m1[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int m2[3][3] = { 1, 2, 3 };
int m3[3][3] = { [0][0] = 1, [1][1] = 2, [2][2] = 3 };

print(m0);
print(m1);
print(m2);
print(m3);
```

```
m0 - not initialized
-584032767743694227
   0   1   0
740314624   0   0

m1 - init by rows
   1   2   3
   4   5   6
   7   8   9

m2 - partial init
   1   2   3
   0   0   0
   0   0   0

m3 - indexed init
   1   0   0
   0   2   0
   0   0   3
```

lec04/array_inits.c

Outline

Arrays

Variable-Length Array

Multidimensional Arrays

Arrays and Pointers

Array vs Pointer 1/2

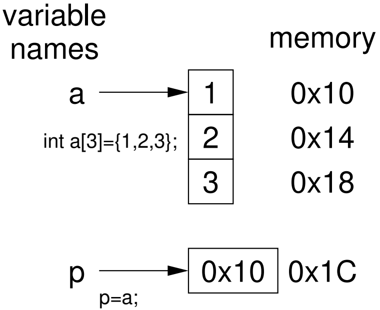
- Variable of the type array of `int` values `int a[3] = {1,2,3};`

  `a` refers to the address of the 1st element of `a`.

- Pointer variable `int *p = a;`

  Pointer `p` contains the address of the 1st element.

- Value `a[0]` directly represents the value at the address `0x10`.



- Value of `p` is the address `0x10`, where the value of the 1st element of the array is stored.
- Assignment `p = a` is legal. *The pointer value is set to the address of the first element.*
- Access to the 2nd element can be made by `a[1]` or `p[1]`.
- Both ways provide the requested elements; however, pointer access is based on the **Pointer Arithmetic**.

Array vs Pointer 2/2

- Pointer (variable) refers to the memory, typically allocated for some data/values.

  We consider a proper usage of the pointers (without dynamic allocation for now).
- Array (variable) refers to a continuous block of memory, where we store sequence of values of the same type.

```
int *p; //pointer (address) where a value of int type is stored
int a[10]; //a continuous block of memory for 10 int values

sizeof(p); //no.of bytes for storing the address (8 for 64-bit)
sizeof(a); //size of the allocated array is 10*sizeof(int)
```

- Both variables refer to a memory, but the compiler works differently with them.
  - Array variable is identified of the memory, where values of the array's elements are stored.

    Compiler (linker) substitute the name with a particular direct memory address.
  - Pointer contains an address, at which the particular value is stored (**indirect addressing**).

`http://eli.thegreenplace.net/2009/10/21/are-pointers-and-arrays-equivalent-in-c`

- However, an array is passed to a function as a pointer!

Example – Passing Array to Function 1/2

```
void fce(int array[])
{
   int local_array[] = {2, 4, 6};
   printf("sizeof(array) = %lu -- sizeof(local_array) = %lu\n",
   sizeof(array), sizeof(local_array));
   for (int i = 0; i < 3; ++i) {
      printf("array[%i]=%i local_array[%i]=%i\n", i, array[i], i,
    local_array[i]);
   }
}
...
int array[] = {1, 2, 3};
fce(array);
```

lec04/fce_array.c

- Compiled program (by `gcc -std=c99` at `amd64`) provides the following outputs.
  - `sizeof(array)` returns the size of 8 bytes (64-bit address).
  - `sizeof(local_array)` returns 12 bytes (3×4 bytes corresponding to three `int` values).
- **Array is passed to a function as a pointer to the first element!**

Example – Passing Array to Function 2/2

```
void fce(int array[]);
...
int array[] = {1, 2, 3};
fce(array);
```

lec04/fce_array.c

- `clang` (with default settings) warns the user about using `int*` instead of `int[]`.

```
fce_array.c:7:16: warning: sizeof on array function parameter will return
    size of 'int *' instead of 'int []' [-Wsizeof-array-argument]
        sizeof(array), sizeof(local_array));
              ^
fce_array.c:3:14: note: declared here
void fce(int array[])
             ^
1 warning generated.
```

- The program can be compiled anyway; however, we cannot rely on the value of `sizeof`.
- Pointer does not carry information about the size of the allocated memory!

Example – Passing Pointer to Array

- We need to pass the number of elements (size) of the array.

It works also for dynamically allocated arrays.

```c
#include <stdio.h>

void fce(int n, int *array); //array is local variable (pointer)
int main(void)
{
   int array[] = {1, 2, 3};
   fce(sizeof(array)/sizeof(int), array); // number of elements
   return 0;
}

void fce(int n, int *array) //array is local variable (pointer)
{ // we can modify the memory defined (allocated) in main()
   int local_array[] = {2, 4, 6};
   printf("sizeof(array) = %lu, n = %i -- sizeof(local_array) = %lu\n",
   sizeof(array), n, sizeof(local_array));
   for (int i = 0; i < 3 && i < n; ++i) { // !  Do the test for n
      printf("array[%i]=%i local_array[%i]=%i\n", i, array[i], i, local_array[i]);
   }
}
```

lec04/fce_pointer.c

- Using `array` in `fce()`, we can access to the array defined in `main()`.

2D Array as a Function Argument

- Function argument cannot be declared as the type `[][]`, e.g.,

  `int fce(int a[][])` × not allowed

  a compiler cannot determine the index for accessing the array elements, for `a[i][j]` the address arithmetic is used differently.

  For `int m[row][col]` the element `m[i][j]` is at the address `*(m + (col * i + j)*sizeof(int))`
- It is possible to declare a function as follows.
  - `int fce(int a[][13]);` – *the number of columns is provided*
  - or `int fce(int a[3][3]);`
  - or in C99 as `int fce(int n, int m, int a[n][m]);` or
  - `int fce(int n, int m, int a[][m]);`
- **We need to define the no. of columns** for accessing a continuous block of memory as 2D array (matrix).

  The compiler needs to be instructed how to determine the address of the matrix cell.

Casting Pointer to Array

- A pointer can be explicitly cast to an array of the particular size.

  The pointer has to refer to a continuous block of memory of the corresponding size, regardless how the memory has been allocated.

```c
int (*p)[3] = (int(*)[3])m; // pointer to array of int

printf("Size of p: %lu\n", sizeof(p));
printf("Size of *p: %lu\n", sizeof(*p)); // 3 * sizeof(int) = 12
```

```
Size of p: 8
Size of *p: 12
```

- It helps to use functions for 2D arrays with one dimensional array or a pointer, because

```c
void print(int rows, int cols, int array[rows][cols]);
...
int array[9];
int *p = array;

print(3, 3, p); //is not allowed
```

- would end with a warning (error).

```
warning: incompatible pointer types passing 'int *' to parameter of type 'int (*)[*]' [-
    Wincompatible-pointer-types]    print(3, 3, p);
```

Part II

Strings

Outline

String Literals and Variables

Reading Strings

C String Library

String Literals

- It is a sequence of characters (and control characters – escape sequences) enclosed within double quotes.

  `"String literal with the end of line \n"`

  - String literals separated by white spaces are joined together, e.g.,

    `"String literal"  " with the end of line \n"`

    is concatenated to

    `"String literal with the end of line \n"`.

- String literal is stored in an array of `char` values terminated by the character `'\0'`, e.g., string literal `"word"` is stored as follows.

`'w'`	`'o'`	`'r'`	`'d'`	`'\0'`

The length of the array must be longer than the text itself!

Referencing String Literal

- String literal can be used wherever `char*` pointer can be used.
- The pointer `p` defined as

  `char* p = "abc";`

  points to the first character of the given literal `"abc"`.
- String literal can be referenced by pointer to char; the type `char*`.

```
char *sp = "ABC";
printf("Size of ps  %lu\n", sizeof(sp));
printf(" ps ’%s’\n", sp);
```

```
Size of ps  8
 ps ’ABC’
```

  - Size of the pointer is 8 bytes (64-bit architecture).
  - String is terminated by `’\0’`.

String Literals, Character Literals

- Pointers can be subscripted (indexed as arrays), and thus also string literals can be subscripted.

  `char c = "abc"[2];`

  - A function to convert integer digit to hexadecimal character can be defined as follows.

    ```
    char digit_to_hex_char(int digit)
    {
       return "0123456789ABCDEF"[digit];
    }
    ```

    We need to assure (programatically) `digit` *would be within the range 0–15.*

- Having a pointer to a string literal, we can attempt to modify it.

  ```
  char *p = "123";

  *p = '0'; // This may cause undefined behaviour!
  ```

Notice, the program may crash or behave erratically!

Be aware of difference between text literals and string variables.

String Variables

- Any one-dimensional array of characters can be used to store a `string`.
- Initialization of a string variable.

```
char str[9] = "B3B36PRG"; // declaration with the size
```

  - Compiler automatically adds the `'\0'`.

There must be space for it!

  - Initialization can be also by particular elements.

```
char str[9] = { 'B', '3', 'B', '3', '6', 'P', 'R', 'G', '\0' };
```

Do not forget null character!

- If the size of the array is defined larger than the actual initializing string, the rest of elements is set to `'\0'`. *Consistent behavior of the array initialization.*
- Specification of the length of the array can be omitted – it is computed by the compiler.

```
char str[] = "B3B36PRG";
```

- **Strings are arrays terminated by `'\0'`.**

Example – Initialization of String Variables

- String variables can be initialized as an array of characters.

```
char str[] = "123";
char s[] = {’5’, ’6’, ’7’ };

printf("Size of str %lu\n", sizeof(str));
printf("Size of  s  %lu\n", sizeof(s));
printf("str ’%s’\n", str);
printf(" s  ’%s’\n", s);
```

```
Size of str 4
Size of  s  3
str ’123’
 s  ’567123’
```

lec04/array_str.c

- **If the string is not terminated by `’\0’`, as for the `char s[]` variable, the listing continues to the first occurrence of `’\0’`.**

Character Arrays vs. Character Pointers

- The string variable is a character array, while pointer can refer to string literal.

```
char str1[] = "B3B36PRG"; // initialized string variable
char *str2 = "B3B36PRG"; // pointer to string literal

printf("str1 \"%s\"\n", str1);
printf("str2 \"%s\"\n", str2);

printf("size of str1 %u\n", sizeof(str1));
printf("size of str2 %u\n", sizeof(str2));
```

lec04/string_var_vs_ptr.c

- Pointer refering to string literal cannot be modified.

It does not represents a writable memory!

- Pointer to the first element of the array (string variable) can be used.

```
#define STR_LEN 10    // best practice for string lengths
char str[STR_LEN + 1] // to avoid forgetting \0
char *p = str;
```

Notice the practice for defining size of string.

Outline

String Literals and Variables

Reading Strings

C String Library

Reading Strings 1/2

- Program arguments are passed to the program as arguments of the `main()` function.

`int main(int argc, char *argv[])`

Appropriate memory allocation is handled by the compiler and program loader.

- Reading strings in run time can be performed by `scanf()`.
- Notice, using a simple control character `%s` may case erratic behaviour, characters may be stored out of the dedicated size.

```c
char str0[4] = "PRG"; // +1 \0
char str1[5]; // +1 for \0
printf("String str0 = '%s'\n", str0);
printf("Enter 4 chars: ");
if (scanf("%s", str1) == 1) {
   printf("You entered string '%s'\n", str1);
}
printf("String str0 = '%s'\n", str0);
```

Example of the program output:

```
String str0 = 'PRG'

Enter 4 chars: 1234567
You entered string '1234567'

String str0 = '67'
```

lec04/str_scanf-bad.c

- Reading more characters than the size of the array `str1` causes overwriting the elements of `str0`.

Reading Strings 2/2

- The maximal number of characters read by the `scanf()` can be set to 4 by the control string `"%4s"`.

```c
char str0[4] = "PRG";
char str1[5];
...
if (scanf("%4s", str1) == 1) {
   printf("You entered string '%s'\n", str1);
}
printf("String str0 = '%s'\n", str0);
```

Example of the program output:

```
String str0 = 'PRG'
Enter 4 chars: 1234567
You entered string '1234'
String str0 = 'PRG'
```

`lec04/str_scanf-limit.c`

- `scanf()` skips white space before starting to read the next string.
- Alternative function to read strings from the `stdin` can be `gets()` or char-by-char using `getchar()`.
  - `gets()` reads all characters until it finds a new-line character. E.g., `'\n'`.
  - `getchar()` – read characters in a loop.
- `scanf()` and `gets()` automatically add `'\0'` at the end of the string.

For your custom `read_line`*, you need to handle it by yourself.*

Outline

String Literals and Variables

Reading Strings

C String Library

Getting the Length of the String

- In C, string is an array (`char[]`) or pointer (`char*`) refering to a part of the memory where the sequence of characters is stored.
- String is terminated by the `’\0’` character.
- Length of the string can be determined by sequential counting of the characters until the `’\0’` character.

```
int getLength(char *str)
{
   int ret = 0;
   while (str && (*str++) != ’\0’) {
      ret++;
   }
   return ret;
}

for (int i = 0; i < argc; ++i) {
   printf("argv[%i]: getLength = %i -- strlen = %lu\n", i,
         getLength(argv[i]), strlen(argv[i]));
}
```

- String functions are in standard string library **<string.h>**.
- String length – **strlen()**.
- **The string length query has linear complexity with its length – $O(n)$.**

lec04/string_length.c

Selected Function of the Standard C Library

- The `<string.h>` library contains function for copying and comparing strings.
  - `char* strcpy(char *dst, char *src);`
  - `int strcmp(const char *s1, const char *s2);`
  - Functions assume sufficient size of the allocated memory for the strings.
  - There are functions with explicit maximal length of the strings.

    `char* strncpy(char *dst, char *src, size_t len);`

    `int strncmp(const char *s1, const char *s2, size_t len);`
- Parsing a string to a number – `<stdlib.h>`.
  - `atoi()`, `atof()` – parsing integers and floats.
  - `long strtol(const char *nptr, char **endptr, int base);`
  - `double strtod(const char *nptr, char **restrict endptr);`

    Functions `atoi()` and `atof()` are „*obsolete*“, but can be faster.
  - Alternatively also `sscanf()` can be used.

See `man strcpy`, `strncmp`, `strtol`, `strtod`, `sscanf`.

Part III

Pointers

Outline

Pointers

`const` Specifier

Pointers to Functions

Dynamic Allocation

Pointers – Overview

- Pointer is a variable to store a memory address.
- Pointer is defined as an ordinary variable, where the name must be preceded by an asterisk, e.g., `int *p;`.
- Two operators are directly related to pointers.
  - **&** – **Address operator**.

    &variable

    - Returns address of the variable.
  - ***** – **Indirection operator**.

    ***pointer_variable**

    - Returns `l-value` corresponding to the value at the address stored in the pointer variable.
- The address can be printed using `"%p"` in `printf()`.
- Guaranteed invalid memory is defined as `NULL` or just as `0` (in C99).
- Pointer to a value of the empty type is `void *ptr;`.

Variables are not automatically initialized in C.

Pointers can refer to an arbitrary address.

Definition of Pointer Variables

- Definition of ordinary variables provide the way to "mark" a memory with the value to use the mark in the program.
- Pointers work similarly, but the value can be any memory address, e.g., where the value of some other variable is stored.

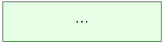
```
int *p;    // points only to integers
double *q; // points only to doubles
char *r;   // points only to characters

int i;      // int variable i
int *pi = &i; // pointer to the int value
              // the value of pi is the address where the value of i is stored
*pi = 10;     // will set the value of i to 10
```

- Without the allocated memory, we cannot set the value using pointer and indirection operator.

```
int *p;
*p = 10; //Wrong, p points to somewhere in the memory
         //The program can behave erratically
```

Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

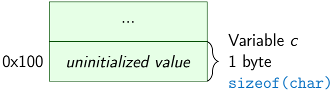


```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```

Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

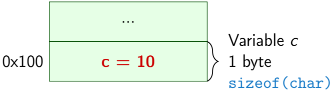
```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

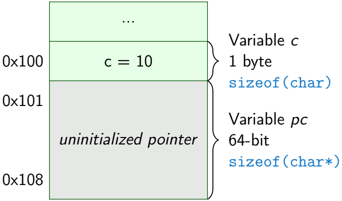
```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

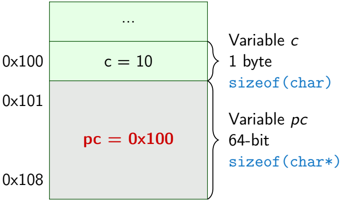
```
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

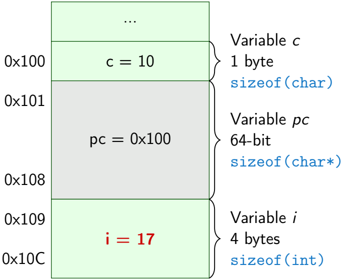
```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

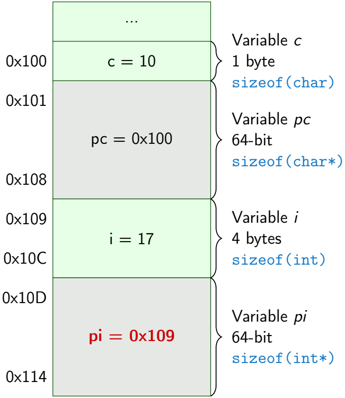
```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



...

0x100 c = 10

Variable *c*
1 byte
`sizeof(char)`

0x101 pc = 0x100 0x108

Variable *pc*
64-bit
`sizeof(char*)`

0x109 i = 17 0x10C

Variable *i*
4 bytes
`sizeof(int)`

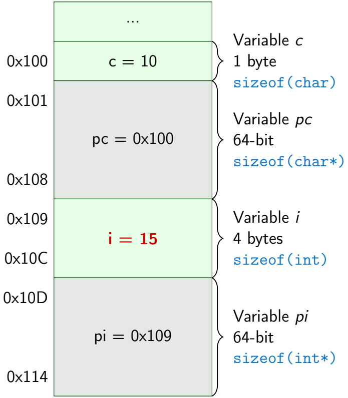
0x10D **pi = 0x109** 0x114

Variable *pi*
64-bit
`sizeof(int*)`

Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

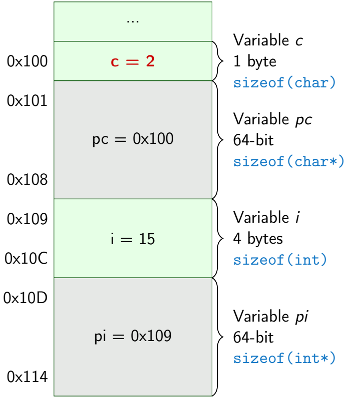
```
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

```c
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



...

0x100 c = 2

Variable *c*
1 byte
`sizeof(char)`

0x101
pc = 0x100
0x108

Variable *pc*
64-bit
`sizeof(char*)`

0x109
i = 15
0x10C

Variable *i*
4 bytes
`sizeof(int)`

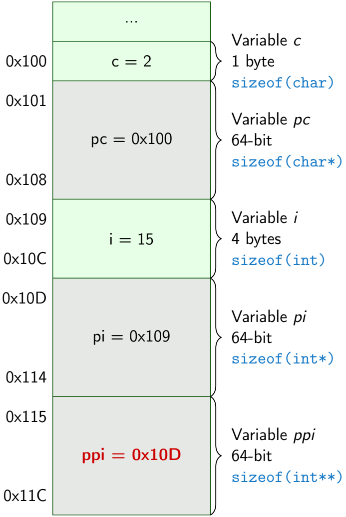
0x10D
pi = 0x109
0x114

Variable *pi*
64-bit
`sizeof(int*)`

Pointers – Visualization of the Allocation and Value Assignment

- Pointers are variables that stores addresses of other variables.

```
char c;

c = 10;

char *pc;

pc = &c;

int i = 17;
int *pi = &i;

*pi = 15;
*pc = 2;

int **ppi = &pi;
```



...

0x100 c = 2 — Variable *c*, 1 byte, `sizeof(char)`

0x101 – 0x108 pc = 0x100 — Variable *pc*, 64-bit, `sizeof(char*)`

0x109 – 0x10C i = 15 — Variable *i*, 4 bytes, `sizeof(int)`

0x10D – 0x114 pi = 0x109 — Variable *pi*, 64-bit, `sizeof(int*)`

0x115 – 0x11C ppi = 0x10D — Variable *ppi*, 64-bit, `sizeof(int**)`

Pointer Arithmetic

- Arithmetic operations $+$ and $-$ are defined for pointers and integers.
  - pointer = pointer of the same type +/- and integer number (int).
  - Shorter syntax can be used – `pointer += 1` and unary operators `pointer++`.
- Arithmetic operations are useful for pointers that refer to memory block where several values of the same type are stored.
  - Array, specifically when it is passed to a function.
  - Dynamically allocated memory, which behaves as array, but allocated in **heap** and not **stack**.
- Adding an int value and the pointer, the results is the address to the next element.

```
int a[10];
int *p = a;

int i = *(p+2); // refers to address of the 3rd element
```

  - The advance the address in the pointer accordingly, we need the size of element type; hence, a pointer to the value of a particular type.
  - `(p+2)` is equivalent to the address computed as follows.

`address of p` + `2*sizeof(int)`

Pointer Arithmetic, Arrays, and Subscripting

- Arrays passed as arguments to functions are pointers to the first element of the array.
- Using pointer arithmetic, we can address particular elements.
- We can use subscripting operator `[]` to access particular element.

The compiler uses `p[i]` as `*(p+i)`.

```c
#define N 10

int a[N];
int *pa = a;
int sum = 0;

for (int i = 0; i < N; ++i) {
   *(pa+i) = i; // initialization of the array a
}
int *p = &a[0]; // address of the 1st element
for (int i = 0; i < N; ++i, ++p) {
   printf("array[%i] = %i\n", i, pa[i]);
   sum += *p; // add the value at the address of p
}
```

- Even though the internal representation is different – we can use pointers as one-dimensional arrays almost transparently.

Special attention must be taken for memory allocation and multidimensional arrays!

Example – Pointer Arithmetic

```c
int a[] = {1, 2, 3, 4};
int b[] = {[3] = 10, [1] = 1, [2] = 5, [0] = 0}; //initialization

// b = a; It is not possible to assign arrays
for (int i = 0; i < 4; ++i) {
   printf("a[%i] =%3i   b[%i] =%3i\n", i, a[i], i, b[i]);
}

int *p = a; //you can use *p = &a[0], but not *p = &a
a[2] = 99;

printf("\nPrint content of the array 'a' with pointer arithmetic\n");
for (int i = 0; i < 4; ++i) {
   printf("a[%i] =%3i   p+%i =%3i\n", i, a[i], i, *(p+i));
}
```

```
a[0] =  1   b[0] =  0
a[1] =  2   b[1] =  1
a[2] =  3   b[2] =  5
a[3] =  4   b[3] = 10

Print content of the array 'a' using pointer arithmetic
a[0] =  1   p+0 =  1
a[1] =  2   p+1 =  2
a[2] = 99   p+2 = 99
a[3] =  4   p+3 =  4
```

lec04/array_pointer.c

Pointer Arithmetic – Subtracting

- Subtracting an integer from a pointer.

```
int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int *p = &a[8]; // p points to the 8th element (starting from 0)

int *q = p - 3; // q points to the 5th element (starting from 0)

p -= 6; // p points to the 2nd element (starting from 0)
```

- Subtracting two pointers results to distance between the pointers (no. of elements).

```
int i
int *q = &a[5];
int *p = &a[1];

i = p - q; // i is 4
i = q - p; // i is -4
```

- It is defined only for pointers refering to the same continuous block of memory (array).
- Performing arithmetic on a pointer that does not point to an array element causes undefined behaviour.

Pointers as Function Arguments

- Pointers can be used to pass the memory address of a variable to a function.
- Using the pointer, the memory can be filled with a new value, like in `scanf()`.
- Consider an example of swapping values of two variables.

```c
void swap(int x, int y)
{
    int z;
    z = x;
    x = y;
    y = z;
}
int a, b;
swap(a, b);
```

```c
void swap(int *x, int *y)
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
}
int a, b;
swap(&a, &b);
```

- The left variant does not propagate the local changes to the calling function.

Pointers as Return Values

- A function may also return a pointer value.
- Such a return value can be a pointer to an external variable.
- It can also be a local variable defined `static`.
- But never return a pointer to an automatic local variable.

```
int* fnc(void)
{
   int i;     // i is a local (automatic) variable
              // allocated on the stack
   ...        // it is valid only within the function
   return &i; // passsing pointer to the i is legal,
              // but the address will not be valid
              // address of the automatically
              // destroyed local variable a
              // after ending the function
}
```

- However, returning pointer to dynamically allocated memory is common.

Outline

Pointers

`const` Specifier

Pointers to Functions

Dynamic Allocation

Pointers to Constant Variables and Constant Pointers

- The keyword **const** can be writable before the type name or before the variable name.
- There are 3 options how to define a pointer with `const`.
  (a) `const int *ptr;` – pointer to a const variable.
    - Pointer cannot be used to change value of the variable.

  (b) `int *const ptr;` – constant pointer.
    - The pointer can be set during initialization, but it cannot be set to another address after that.

  (c) `const int *const ptr;` – constant pointer to a constant variable.
    - Combines two cases above.

`lec04/const_pointers.c`

Further variants of (a) and (c) are as follows.

  - `const int *` can be written as `int const *`.
  - `const int * const` can also be written as `int const * const`.

`const` can on the left or on the right side from the type name.

- Further complex definitions can be, e.g., `int ** const ptr;`

A constant pointer to refer to the int value.

Pointers to Constant Variables and Constant Pointers

- The keyword **const** can be writable before the type name or before the variable name.
- There are 3 options how to define a pointer with `const`.
  (a) `const int *ptr;` – pointer to a const variable.
    - Pointer cannot be used to change value of the variable.

  (b) `int *const ptr;` – constant pointer.
    - The pointer can be set during initialization, but it cannot be set to another address after that.

  (c) `const int *const ptr;` – constant pointer to a constant variable.
    - Combines two cases above.

  `lec04/const_pointers.c`

  Further variants of (a) and (c) are as follows.
  - `const int *` can be written as `int const *`.
  - `const int * const` can also be written as `int const * const`.

  `const` can on the left or on the right side from the type name.
- Further complex definitions can be, e.g., `int ** const ptr;`

  A constant pointer to refer to the int value.

Example – Pointer to Constant Variable

- It is not allowed to change variable using pointer to constant variable.

```c
int v = 10;
int v2 = 20;

const int *ptr = &v;
printf("*ptr: %d\n", *ptr);

*ptr = 11; /* THIS IS NOT ALLOWED! */

v = 11;  /* We can modify the original variable */
printf("*ptr: %d\n", *ptr);

ptr = &v2; /* We can assign new address to ptr */
printf("*ptr: %d\n", *ptr);
```

`lec04/const_pointers.c`

Example – Const Pointer

- Constant pointer cannot be changed once it is initialized.
- Definition `int *const ptr;` can be read from the right to the left.
  - `ptr` – variable (name) that is
  - `*const` – constant pointer
  - `int` – to a variable/value of the `int` type.

```c
int v = 10;
int v2 = 20;
int *const ptr = &v;
printf("v: %d *ptr: %d\n", v, *ptr);

*ptr = 11; /* We can modify addressed value */
printf("v: %d\n", v);

ptr = &v2; /* THIS IS NOT ALLOWED! */
```

lec04/const_pointers.c

Example – Constant Pointer to Constant Variable

- Value of the constant pointer to a constant variable cannot be changed, and the pointer cannot be used to change value of the addressed variable.
- Definition `const int *const ptr;` can be read from the right to the left.
  - `ptr` – variable (name) that is
  - `*const` – const pointer
  - `const int` – to a variable of the `const int` type.

```c
int v = 10;
int v2 = 20;
const int *const ptr = &v;

printf("v: %d *ptr: %d\n", v, *ptr);

ptr = &v2; /* THIS IS NOT ALLOWED! */
*ptr = 11; /* THIS IS NOT ALLOWED! */
```

lec04/const_pointers.c

Outline

Pointers

`const` Specifier

Pointers to Functions

Dynamic Allocation

Pointers to Functions

- Implementation of a function is stored in a memory, and similarly, as for a variable, we can refer a memory location with the function implementation.
- Pointer to function allows to dynamically call a particular function according to the value of the pointer.
- Function is identified (except the name) by its arguments and return value. Therefore, these are also a part of the definition of the pointer to the function.
- Function (a function call) is the function name and **()**, i.e.,

```
return_type function_name(function arguments);
```

- Pointer to a function is defined as

```
return_type (*pointer)(function arguments);
```

- It can be used to specify a particular implementation, e.g., for sorting custom data using the `qsort()` algorithm provided by the standard library `<stdlib.h>`.

Example – Pointer to Function 1/2

- Indirection operator `*` is used similarly as for variables.

```
double do_nothing(int v); /* function prototype */

double (*function_p)(int v); /* pointer to function */

function_p = do_nothing; /* assign the pointer */

(*function_p)(10); /* call the function */
```

- Brackets `(*function_p)` "*help us*" to read the pointer definition.

  We can imagine that the name of the function is enclosed by the brackets. Definition of the pointer to the function is similar to the function prototype.

- Calling a function using pointer to the function is similar to an ordinary function call. Instead of the function name, we use the variable of the pointer to the function type.

Example – Pointer to Function 2/2

- In the case of a function that returns a pointer, we use it similarly.

```
double* compute(int v);

double* (*function_p)(int v);
        ^^^^^^^^^^^^^---- substitute a function name


function_p = compute;
```

- Example of the pointer to function usage – `lec04/pointer_fnc.c`.
- Pointers to functions allows to implement a dynamic link of the function call determined during the program run time.

In object oriented programming, the dynamic link is a crucial feature to implement polymorphism.

Outline

Dynamic Storage Allocation

- A dynamic allocation of the memory block with the `size` can be performed by `malloc()`.

  `void* malloc(size);` from the `<stdlib.h>`

  - The **memory manager** handle the allocated memory (from the **heap** memory class).
  - **The size is not a part of the pointer**.
  - Return value is of the `void*` type – cast is required.
  - The programmer is fully responsible for the allocated memory.
- Example of the memory allocation for 10 values of the `int` type.

```
int *int_array;
int_array = (int*)malloc(10 * sizeof(int));
```

- The usage is similar to array (pointer arithmetic and subscripting).
- The allocated memory must be explicitly **released**.

  `void free(pointer);`

- By calling `free()`, the memory manager release the memory at the address stored in the pointer value.

  The pointer value is not changed! It has the previous address that is no longer valid!

Example – Dynamic Allocation 1/3

- If allocation may fail, `malloc()` returns `NULL` and we should test the return value.

  Unless, we intentionally take the risk of erratic behaviour of the program.

- The most straightforward handle of the allocation failure is to report the error and terminate the program execution. *We can implement our custom function for dynamic allocation.*

```
void* mem_alloc(size_t size)
{
   void *ptr = malloc(size);  //call malloc to allocate memory

   if (ptr == NULL) {
      fprintf(stderr, "Error: allocation fail"); // report error
      exit(-1); // and exit program on allocation failure
   }
   return ptr;
}
```

`lec04/malloc_demo.c`

Example – Dynamic Allocation 2/3

- Filling the dynamically allocated array, just the memory address is sufficient.

```c
void fill_array(int* array, int size)
{
   for (int i = 0; i < size; ++i) {
      *(array++) = random() % 10; // pointer arithmetic
      //array[i] = random() % 10; // array notation using subscript operator
   }
}
```

- After memory is released by `free()`, the pointer variable still contains the same address.
- Use a custom function to set the pointer to the guaranteed invalid address (`NULL` or `0`).

Passing pointer to a pointer is required to set the value of the variable, which is the pointer.

```c
void mem_release(void **ptr)
{
   // 1st test ptr is valid pointer, and also *ptr is a valid
   if (ptr != NULL && *ptr != NULL) {
      free(*ptr);
      *ptr = NULL;
   }
}
```

`lec04/malloc_demo.c`

Example – Dynamic Allocation 3/3

```c
int main(int argc, char *argv[])
{
   int *int_array;
   const int size = 4;

   int_array = mem_alloc(sizeof(int) * size);
   fill_array(int_array, size);
   int *cur = int_array;
   for (int i = 0; i < size; ++i, cur++) {
      printf("Array[%d] = %d\n", i, *cur);
   }
   mem_release((void**)&int_array); // we do not need type cast to void**, it is just to highlight we are passing pointer-to-pointer
   return 0;
}
```

lec04/malloc_demo.c

Standard Function for Dynamic Allocation

- `void* malloc(size_t size);` – allocates (no initialization) a block of the memory `size` bytes in length.
- `void* calloc(size_t number, size_t size);` – allocates memory for the **number** objects, each `size` bytes in length, and clears them.
- `void* realloc(void *ptr, size_t size)` – resizes a previously allocated block of memory `size` bytes in length.
  - It tries to enlarge the previous block; if there is a continuous block of the available memory of the `size` in length, starting from `ptr`.
  - If it it not possible, a new (larger) block is allocated.
    - The previous block is copied into the new one.
    - The previous block is released (calling `free()`. *The value `ptr` is not changed.*
    - The return values points to the enlarged block.
  - It returns `NULL` if allocation fails.
  - *It might release the allocated memory if a smaller size is given.* *It can act as `free()`.*

See `man malloc`, `man calloc`, `man realloc`.

Using `realloc()`

- The behaviour of the `realloc()` function is further specified.
    - It does not initialize the bytes added to the block.
    - If it cannot enlarge the memory, it returns a null pointer, and the old memory block is untouched.
    - If it is called with null pointer as the argument, it behaves as `malloc()`.
    - If it is called with 0 as the second argument (`size`), it frees the memory block as `free()`.

```c
int size = 10;
int *array = mem_alloc(size * sizeof(int)); // allocate 10 integers
... // do some code such as reading integers from a file

int *t = realloc(array, (size + 10)* sizeof(int)); // try to enlarge
if (t) {
   array = t;  // realloc handle possible allocation of new memory block, and thus
               // it is safe to overwrite array by t
   size += 10; // now, we are sure array can hold 10 more int values
} else {       // realloc fail, report and exit
   fprintf(stderr, "ERROR: realloc fail\n");
}
```

Restricted Pointers

- In C99, the keyword `restrict` can be used in the pointer definition.

```
int * restrict p;
```

- The pointer defined using `restrict` is called **restricted pointer**.
- The main intent of the restricted pointers is following.
  - If `p` points to an object that is later modified, the object is not accessed in any way other than through `p`.
- It is used in several standard functions, such as `memcpy()` from `<string.h>`.

```
void *memcpy(void * restrict dst, const void * restrict src, size_t len);
```

  - In `memcpy()`, it indicates `src` and `dst` should not overlap, but it is not guaranteed.
  - It provides useful documentation, but its main intention is to provide information to the compiler to produce more efficient code (similarly to `register` keyword).

Part IV

Part 4 – Assignment HW 03

HW 03 – Assignment

Topic: **Caesar Cipher**

Mandatory: **2 points**; Optional: none; Bonus : **2 points**

- **Motivation**: Experience a solution of the optimization task.
- **Goal**: Familiarize with the dynamic allocation.
- **Assignment**: `https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw03`
  - Read two text messages and print decode message to the output.
  - Both messages (the encoded and the poorly received) have the same length.
  - Determine the best match of the decoded and received messages based on the shift value of the Caesar cipher. `https://en.wikipedia.org/wiki/Caesar_cipher`
  - Optimization of the Hamming distance. `https://en.wikipedia.org/wiki/Hamming_distance`
  - Bonus assignment – an extension for missing characters in the received message. `https://en.wikipedia.org/wiki/Levenshtein_distance`
- Deadline: 06.04.2024, 23:59 AoE (bonus 24.05.2024, 23:59 CEST).

Summary of the Lecture

Topics Discussed

- Arrays
    - Variable-Length Arrays
    - Arrays and Pointers
- Strings
- Pointers
    - Pointer Arithmetic
    - Dynamic Storage Allocation

- Next: Data types: struct, union, enum, and bit fields

Topics Discussed

- Arrays
  - Variable-Length Arrays
  - Arrays and Pointers
- Strings
- Pointers
  - Pointer Arithmetic
  - Dynamic Storage Allocation

- Next: Data types: struct, union, enum, and bit fields