High Performance Programming, Lecture 2

Programming in C

What will we do today?

- Pointers, 1D arrays
- Dynamic memory allocation
- File I/O
- Extra staff (depending on time)

Before we start

Assignment 1 is available on the Studentportalen. Deadline next Friday (1/2)!

Lab 1 is available on the Studentportalen. Come to 1515 tomorrow, work in groups.

Assignment 2 will be available on Monday 28.01. Register to a group!

New staff on Student Portal

CONTENT $\longrightarrow$ Deadlines

CONTENT $\longrightarrow$ Course material overview
— added short info about each lecture

CONTENT $\longrightarrow$ Suggested reading
— I will try to add some reading suggestions before lectures

DOCUMENT $\longrightarrow$ Extra files
— C_build_process (will be updated soon)

History of C

C programming language

- Developed by Dennis Ritchie between 1969 and 1973 at Bell Labs
- The Unix system is written in C
- C is designed on a way to provide a full control of the computer.

C is the most widely used programming languages of all time.

C standarts

Software developers writing in C are encouraged to conform to the standards, as doing so aids portability between compilers.

- C89 (ANSI C)
- C90 (ISO standard)
- C99
- C11
- C18 (C17)

If you want to compile your programs according to the C standard C99, you should type
`gcc -std=c99 -pedantic-errors` or
`clang -std=c99 -pedantic-errors`

How to find out compiler version?

When reporting performance results/timings, important to include information about the compiler version used.

Important to not just say "gcc" or "clang" but **include the version number**, like "gcc 8.2.0".

```
  gcc --version
---------->
gcc (GCC) 8.2.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying
 conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS
 FOR A PARTICULAR PURPOSE.
```

The version number (8.2.0 above) is the most important info there.

Structure of a program

```
#include <stdio.h> // system header files
// #include "myheader.h" for your own header files

int main()
{
        printf("Hello world!\n");   // note newline
    character
        return 0;  // return EXIT_SUCCESS;
}
```

Function documentation and tutorials

`https://en.cppreference.com/w/c`
`https://www.learn-c.org/`

See suggestions in Student portal under CONTENT $\longrightarrow$ "C language and library documentation"

Functions

The syntax of a function definition:

```
return_type function_name( parameter list ) {
        /* body of the function */
}
```

Example of function declaration:

```
void f1();
int f2(int num1, int num2);  // call by value
void f3(char *s);   // call by reference
int* f4(int* arr);  // call by reference
```

Call by value - takes a copy of parameter ⇒ no changes made to original parameter.
Call by reference - takes an address of parameter ⇒ modification of original is possible.

main function arguments

```c
#include <stdio.h>      /* printf */
#include <string.h>     /* strcpy */
#include <stdlib.h>     /* atoi */

int main(int argc, char const *argv[])
{
 if(argc != 4) {printf("Usage: %s string int double\n", argv[0]);
     return -1;} // Usage: a.out string int double
 char str[10];
 strcpy(str, argv[1]);
 int i = atoi (argv[2]);
 double f = atof (argv[3]);
 printf("%s %d %.1f\n", str, i, f);  // hello 5 6.5
 return 0;
}
```

We can run the program like this:
./a.out hello 5 6.5

Example programs

See examples in Student Portal under DOCUMENT/Example programs:

`hello_world.c`

`input_args.c` (how to use main function input arguments)

`input_menu_scanf.c`

Input/output

```
#include <stdio.h>
int main (){
char str [80]; int i, ihex;

scanf ("%s",str); // enter string
printf ("Entered string %s\n", str);

scanf ("%d",&i); // enter integer
printf ("Entered integer %d\n", i);

scanf ("%x",&ihex); // enter hexadecimal number (
    Example: 0x4A or 0X4A)
printf ("Entered hexadecimal %x (%d in decimal)\n",
    ihex, ihex); // output:  Entered hexadecimal 0x4A
     (74 in decimal)
```

Input/output

```
#include <stdio.h>
int main (){
 double f; char ch;

scanf ("%lf",&f); // enter double
printf ("Entered double %.3lf\n", f); // note format %.3lf

printf ("Address of f %p\n", &f);

scanf (" %c",&ch); // NOTE whitespace before %c, scanf
// does not skip any leading whitespace when reading char
printf ("Entered char %c\n", ch);

return 0;}
```

Conditional selection/loops

The usage of the basic constructions such as **if...else**, **for**, **while**, **switch** see in the example file `basic_C_loops.c` in the Student Portal, under DOCUMENT/Example programs.

Pointers

C allows to a programmer directly access and manipulate computer memory. It applies also to C++.

Pointer is a variable that can hold the address of other variable

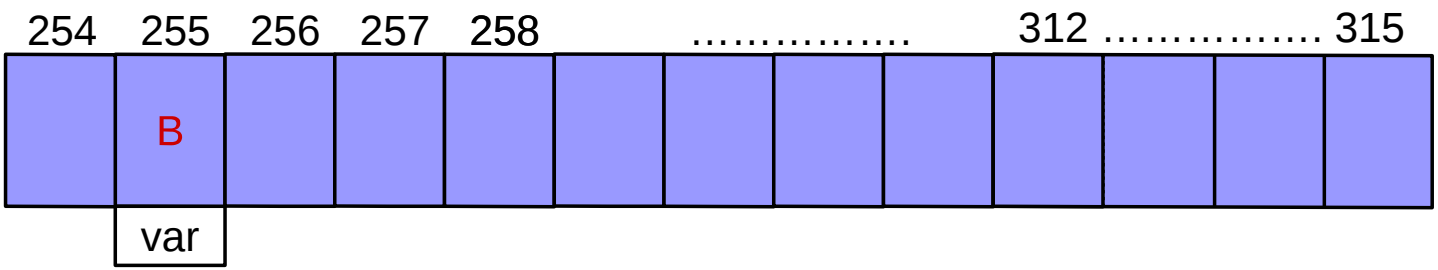
Why do we need pointers?

- to get address of the variable
- pointing to functions
- declare strings
- pass large structure to other functions to avoid complete copy of the structure (sending function arguments by reference)
- to create complex data structures like binary trees

Declare a variable

Declare a char variable:

```
char var = 'B';
```



sizeof

sizeof operator gives size in bytes of the object or type.

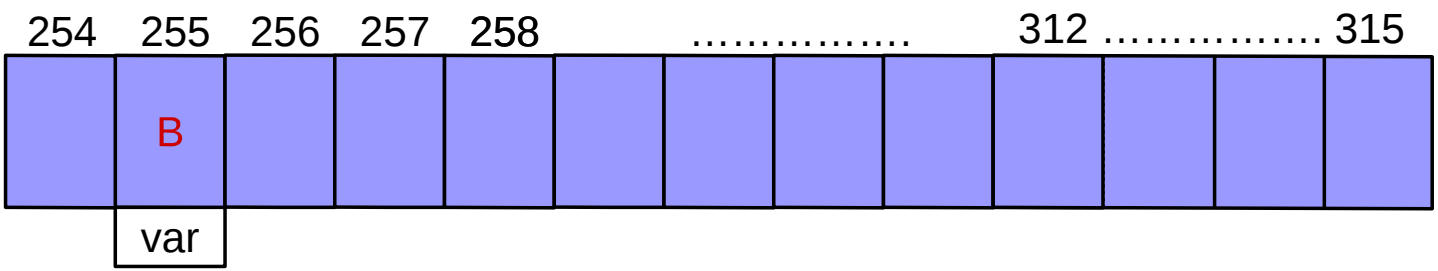
`sizeof(expression)` (or `sizeof expression`)

`sizeof(typename)`

Declare a variable

Declare an integer variable and get its size:

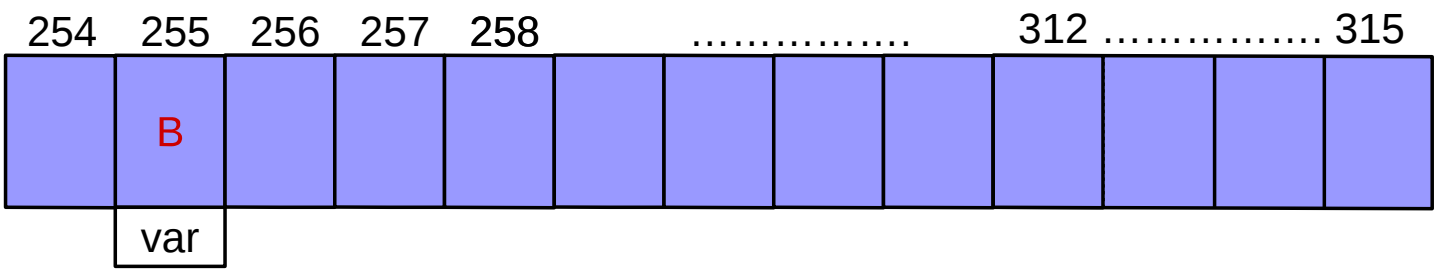
```
char var = 'B';
printf("Sizeof(var): %lu\n", sizeof(var));  // 1 byte
printf("Sizeof(char): %lu\n", sizeof(char)); // 1
    byte
```



The address of a variable

The operator & - returns an address of a variable.
Operator & just returns the address of this memory location (the address of the first byte in case the representation of the variable var needs more bytes).

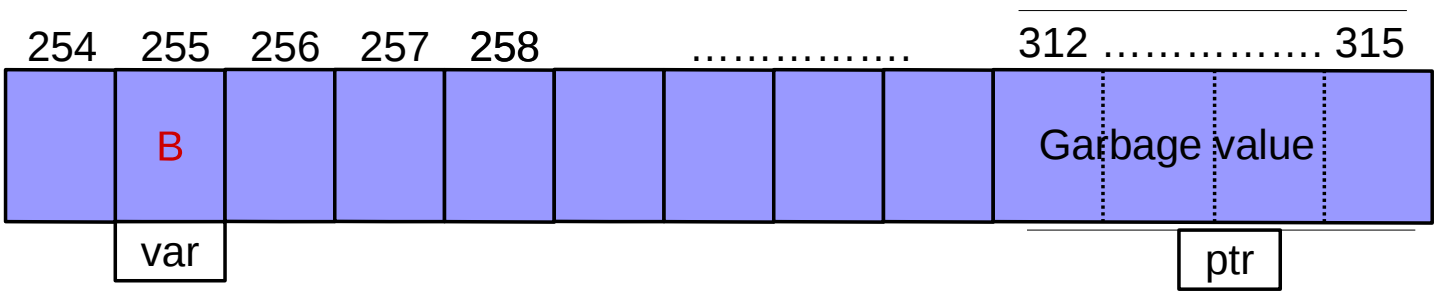
```
char var = 'B';
printf("The address is %p \n", &var);
// 0xff in hexadecimal, 255 in decimal
```



Pointers

Declare a pointer `ptr`:

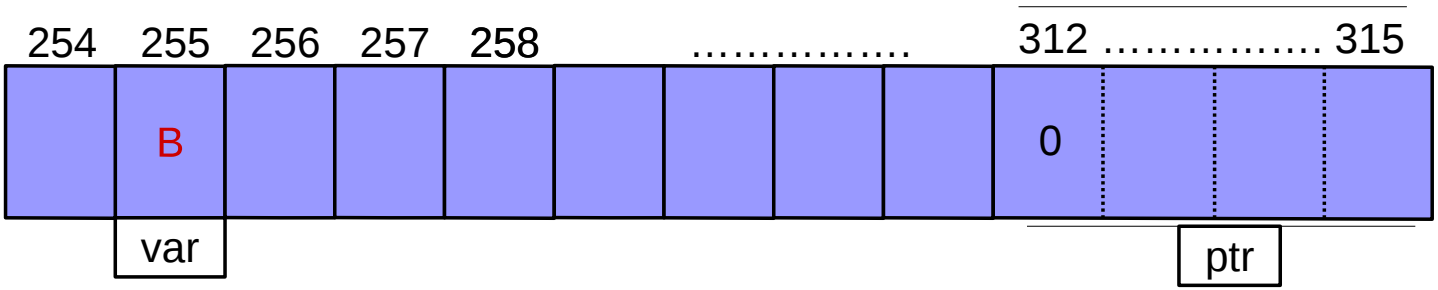
```
char var = 'B';  // allocates 1 byte of memory
char *ptr;   // or char* ptr;
/* ptr is a variable with type char*. In our example
    we use 32bit system and ptr allocates 4 bytes of
    memory. */
```



Pointers

Set `ptr` to NULL (usually macros for 0):

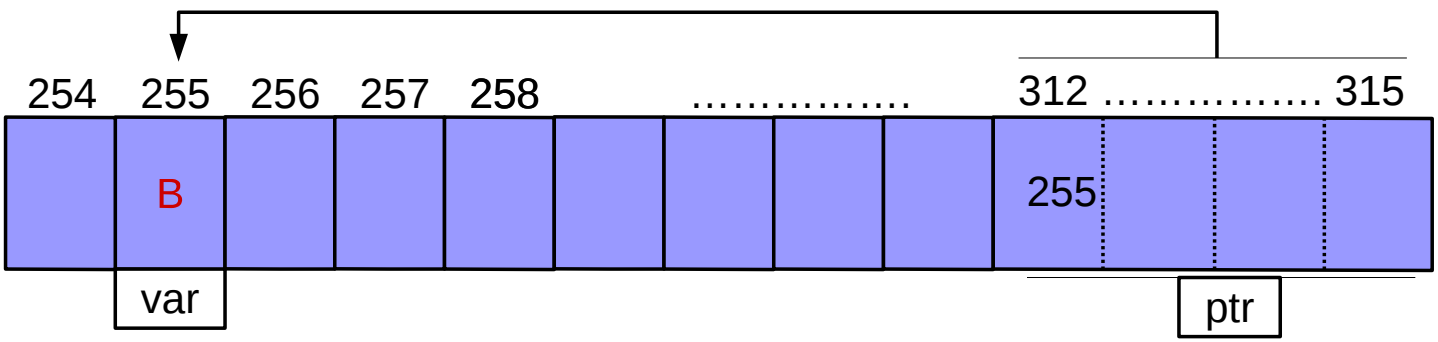
```
char var = 'B';
char *ptr = 0; // char *ptr = NULL // this telling to
     a developer that pointer is not used
```



Pointers

Assign an address of `var` to `ptr`:

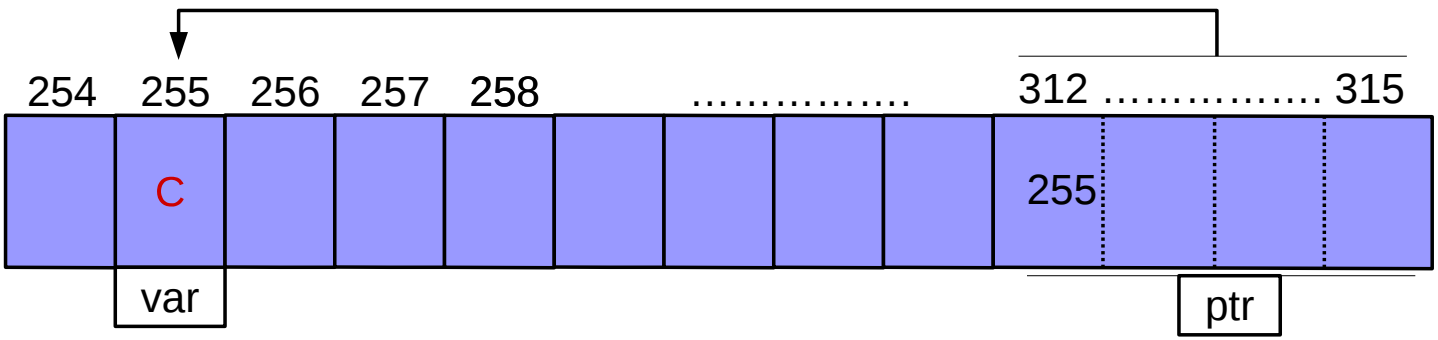
```
char var = 'B';
char *ptr = &var;  // 255
```



Dereferencing pointer

Operator * access the content of the memory pointed by a pointer.

```
char var = 'B';
char *ptr = &var; // 255
*ptr = 'C';  // indirect way to change a value of var
```



Examples:

```
char var = 'A';
char *ptr = &var; // var = 'A'
char var2 = *ptr; // var = 'A', var2 = 'A'
*ptr = 'B';       // var = 'B', var2 = 'A'
var = 'C';        // var = 'C', var2 = 'A'
ptr = &var2;
*ptr = 'D';       // var = 'C', var2 = 'D'
```

Example

What is the output of the following program?

```
#include<stdio.h>

void f(int *y)
{
    *y = 3;
}
int main()
{
 int x = 5;
 f(&x);
 printf("%d", x);
 return 0;
}
```

Example

What is the output of the following program?

```
#include<stdio.h>

void f(int *y)
{
    *y = 3;
}
int main()
{
 int x = 5;
 f(&x);
 printf("%d", x);
 return 0;
}
```

OUTPUT: 3

Example - try at home

What is the output of the following program?

```
#include<stdio.h>

void f(int **y, int a)
{
    **y = a;
}
int main()
{
 int x = 5;
 int *px = &x;
 int a = 3;
 f(&px, a);
 printf("%d", x);
 return 0;
}
```

Example - try at home

What is the output of the following program?

```
#include<stdio.h>

void f(int **y, int a)
{
    **y = a;
}
int main()
{
 int x = 5;
 int *px = &x;
 int a = 3;
 f(&px, a);
 printf("%d", x);
 return 0;
}
```

OUTPUT: 3

Pointer arithmetics

See here for explanation and examples: `http://www.c4learn.com/c-programming/c-pointer-arithmetic-operations/`

`p` and `r` are pointers
Operator precedence:
`++p` gives pointer to the next variable
`p - r` get distance between pointers
`p + 5` moves pointer 5 positions forward
`p + r` not supported

Operator precedence:
`*p++` increment pointer not value pointed by it
`(*p)++` increment value pointed by `p`

Debugging

To help both avoiding and fixing bugs, turn on all compiler warnings using `-Wall` and make sure to **fix the code to get rid of all warnings**.

General rule when debugging: when you run into some problem, before starting to track down the bug **start by finding the smallest possible test case that still leads to the problem**.

Use `gdb` debugger to find out why a program "hangs" (Lab 1 and Lab 2).

Structures

Declare a structure with 2 members:

```
struct person
{
char  name[50];
int   age;
};
```

If we want to define an object: `struct person A;`
To access a member: `A.age = 19;`
Define a shortcut "person_t" for "struct person" using typedef:

```
typedef   struct person    person_t;
```

Now we can define an object using `person_t A;`

Pointers and structures

```
struct person
{
  char  name[50];
  int   age;
};
typedef   struct person    person_t;

person_t A;  // or struct person A;

A.age = 18; // set values of the structure members
strcpy(A.name, "Maria");
```

NOTE: see more about strings and strcpy at the end of this slides.

Pointers and structures

```
struct person
{
 char  name[50];
 int   age;
};
typedef   struct person    person_t;

person_t *A = (person_t *)malloc(sizeof(person_t));  //
    pointer to a structure

(*A).age = 18; // set values of the structure members
strcpy((*A).name, "Maria");
//or
A->age = 18; // set values of the structure members
strcpy(A->name, "Maria");
```

Pointers

A lots of information and examples here:
`https://www.learn-c.org/en/Pointers//`
`http://www.c4learn.com/index/pointer-c-programming/`

Check this short summary about the pointers:
`http://nuclear.mutantstargoat.com/articles/pointers_explained.pdf`

Here you can read also about `void` pointers which can keep an address of the variable but cannot be dereferenced!

Top 20 C pointer mistakes and how to fix them
`https://www.acodersjourney.com/top-20-c-pointer-mistakes/`

Arrays

`T[n]` is an array of n elements of type `T`: $0, \ldots, n-1$
Arrays values are stored in contiguous memory locations.

Initialization:

```
type arrayName [ size ]; // size is constant !
```

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
//or
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Arrays

Arrays are closely related to pointers.

```
int a[] = {1, 2, 3, 4};
int *p = a;      // pointer to the first element
// equivalent:
int *p = &a[0]; // pointer to the first element
int *p = &a[3]; // pointer to the fourth element
```

Moreover:

```
int *p = a+2; // pointer to the third element
int v = *(a+2); // the third element
```

Arrays

Program writing in the standard output elements of the array `arr`:

```
#include <stdio.h>
int main()
{
        int arr[5] = {11,22,33,44,55};
        int i;
        for(i=0;i<5;i++) {
                printf("%d ",arr[i]);
        // or
        //        printf("%d ",*(arr + i));
        }
        return 0;
}
```

Undefined behavior

The behavior of some constructs may not be defined: you cannot predict what will happen at runtime.

Example: dereferencing a null pointer

```
int *p = NULL;
printf("%d\n", *p); // Segmentation fault (core
    dumped) on most systems
```

Example: out of bound

```
int arr[3] = {1, 2, 3};
for (int i=0; i<=3; i++)
printf("%d ", arr[i]);  // access to arr[3] in last
    iteration.
```

Static vs dynamic memory

Static memory allocation: memory is allocated by the compiler on *stack*.
Dynamic memory allocation: memory is allocated at runtime on *heap*.
You would use DMA if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data, if you want to use dynamic data structure.

Static allocation:

```
int arr[10];
```

Dynamic allocation

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

malloc, calloc, realloc from <stdlib.h>

`malloc` is used to allocate an amount of memory of size bytes during the execution of a program.

```
void* malloc (size_t size_of_block);
```

The argument to malloc() is the size of the memory **in bytes**!

`calloc` allocates and zero-initialize array

```
void* calloc (size_t number_of_elements, size_t
    size_of_each_element);
```

`realloc` changes the size of the memory block pointed to by ptr

```
void* realloc (void* ptr_to_block, size_t
    size_of_block);
```

Allows increase the size of dynamically allocated array.

Example of using malloc or realloc

```
/* Allocate memory block for 3 double numbers. */
double* p = (double*)malloc(3*sizeof(double));
p[0] = 0.1;
p[1] = 0.2;
p[2] = 0.3;

/* Change size of memory block to fit 5 double
numbers. */
p = realloc(p, 5*sizeof(double));
p[3] = 0.4;
p[4] = 0.5;
```

Note: you do not need to free memory between malloc (calloc) and realloc! It is done automatically if needed.

free(p)

No garbage collector in C! You must free memory allocated with `malloc`, `calloc` or `realloc` using

```
void free (void* ptr);
```

A **memory leak** occurs when dynamically allocated memory has become unreachable. If the leak is severe enough, your program will eventually run out of address space and future allocation attempts will fail.

Typical program:

```
int *ptr = (int *)malloc(sizeof(int));
if (ptr == 0)
{   printf("ERROR: Out of memory\n");
        return 1;}
/* code used ptr*/
free(ptr);
```

File I/O

A file represents a sequence of bytes, regardless of it being a text file or a binary file.
Text files contain only textual data, binary files contain custom binary data.

```
FILE *fopen( const char *filename, const char *mode )
```

`filename` is a string literal, the name of a file.
Most common file modes:

- 'r' : Opens an existing text file for reading purpose.
- 'w' : Opens a text file for writing. If it does not exist, then a new file is created.

Function returns NULL pointer if file cannot be opened.

Close file: `int fclose( FILE *fp );`

Text files

```
#include <stdio.h>
int main() {
FILE *fp1;   // POINTER to an object FILE
FILE *fp2;
char buff[255];
fp1 = fopen("test.txt", "r");  // open file test.txt for reading
fp2 = fopen("out.txt", "w");   // open file out.txt for writing
fscanf(fp1, "%s", buff);   // read a string to buff (reads untill
     the space is encountered)
fprintf(fp2, "%s", buff);  // write data from buff to the file
    fp2
fclose(fp1);
fclose(fp2);
return 0; }
```

Binary files

```
#include <stdio.h>
int main() {
FILE *fp1;   // POINTER to an object FILE
FILE *fp2;
char buff[255];
fp1 = fopen("test.bin", "r");  // open file test.bin for reading
fp2 = fopen("out.bin", "w");   // open file out.bin for writing
size_t mem_block_size=10;
fread(buff, sizeof(char), mem_block_size, fp1);  // read a memory
      block to buff
fwrite(buff, sizeof(char), mem_block_size, fp2);  // write data
     from buff to the file fp2
fclose(fp1);
fclose(fp2);
return 0; }
```

String literal

String literal: `"hello"`, `"this is a string"`
Note: character: `'a'`, string `"a"`

Staring literal has a type `const char[size]`
The null character ('\0') is always appended to the string literal:

`"hello"` is a `const char[6]` holding the characters
'H', 'e', 'l', 'l', 'o', and '\0'.

String literal and array of char

String literals can be assigned to `char` arrays:

```
char array1[] = "Hello";
// same as
char array2[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Common functions:

- strlen – gives number of characters in the string, computed at the run time
- sizeof – gives size of the array, computed at the compile time

Note the difference!

Compare

```
char *s1 = "Hello";
char s2[] = "Hello";
```

The first sentence will place "Hello world" in the read-only parts of the memory, and making s a pointer to that makes any writing operation on this memory illegal.
The second will put the literal string in read-only memory and copies the string to newly allocated memory on the stack. Thus making `s[0] = 'J';` legal.

Strings

```
char arr[] = "Hello world";
int len = strlen(arr);
```

where

```
size_t strlen ( const char * str );
```

Implicit cast from array type `char[]` to the pointer to char `char*`.

Function gets the pointer to the first element of the array.
How does `strlen` knows the size of `str`?

Strings

```
char arr[] = "Hello world";
int len = strlen(arr);
```

where

```
size_t strlen ( const char * str );
```

Implicit cast from array type `char[]` to the pointer to char `char*`.

Function gets the pointer to the first element of the array.
How does `strlen` knows the size of `str`?
(String ends with a terminating null character '\0')

```
int len = strlen(arr); // answer is 11
```

Example

What is the output of the following program?

```
#include<stdio.h>

int main()
{
  char s[20] = "Hello\0Hi";
  printf("%d %d", strlen(s), sizeof(s));
}
```

Example

What is the output of the following program?

```
#include<stdio.h>

int main()
{
  char s[20] = "Hello\0Hi";
  printf("%d %d", strlen(s), sizeof(s));
}
```

Answer: 5 20
strlen given the length of a string, so it is counting characters up to '0'.
sizeof reports the size of the array.

Strings

strcpy copies the string pointed by `source` into the array pointed by `destination`:

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

Example:

```
char a[6];
//a = "hello";          This is an error!
strcpy(a, "hello");   // This is correct!
```

Strings, example program

Demonstration how the char array can be used as a text string `char_arr_as_string.c` in the Student Portal, under DOCUMENT/Example programs.