

Data types, arrays, pointer, memory storage classes, function call

Jan Faigl

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

Lecture 03

B3B36PRG – C Programming Language

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 1 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

Basic Data Types

- Basic (built-in) types are numeric integer and floating types
 - Logical data type has been introduced in C99*
- C data type keywords are
 - Integer types: `int`, `long`, `short`, and `char`
Range "modifiers": `signed`, `unsigned`
 - Floating types: `float`, `double`
May also be used as `long double`
 - Character type: `char`
 - Can be also used as the integer type*
 - Data type with empty set of possible values: `void`
 - Logical data type: `_Bool`
- Size of the memory representation depends on the system, compiler, etc.
 - The actual size of the data type can be determined by the `sizeof` operator
- New data type can be introduced by the `typedef` keyword

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 5 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

Signed and Unsigned Integer Types

- In addition to the number of bytes representing integer types, we can further distinguish
 - `signed` (default) and
 - `unsigned` data types
 - A variable of unsigned type cannot represent negative number*
 - Example (1 byte):
 - `unsigned char`: values from 0 to 255
 - `signed char`: values from -128 to 127
- ```
1 unsigned char uc = 127;
2 char su = 127;
3
4 printf("The value of uc=%i and su=%i\n", uc, su);
5 uc = uc + 2;
6 su = su + 2;
7 printf("The value of uc=%i and su=%i\n", uc, su);
```
- [lec03/signed\\_unsigned\\_char.c](#)

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 8 / 57

## Overview of the Lecture

- Part 1 – Data Types
  - Numeric Types
  - Character Type
  - Logical Type
  - Type Cast
  - Arrays
  - Pointers

*K. N. King: chapters 7, 8, and 11*
- Part 2 – Functions and Memory Classes
  - Functions and Passing Arguments
  - Program I/O
  - Hardware Resources
  - Scope of Variables
  - Memory Classes

*K. N. King: chapters 9, 10, and 18*
- Part 3 – Assignment HW 03

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 2 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

## Basic Numeric Types

- Integer Types – `int`, `long`, `short`, `char`
  - `char` – integer number in the range of single byte or character
  - Size of the allocated memory by numeric variable depends on the computer architecture and/or compiler
    - Type `int` usually has 4 bytes even on 64-bits systems*
  - The size of the memory representation can be find out by the operator `sizeof()` with one argument name of the type or variable.

```
int i;
printf("%lu\n", sizeof(int));
printf("ui size: %lu\n", sizeof(i));
```

[lec03/types.c](#)
- Floating types – `float`, `double`
  - Depends on the implementation, usually according to the IEEE Standard 754 (1985) (or as IEC 60559)*
  - `float` – 32-bit IEEE 754
  - `double` – 64-bit IEEE 754
  - [http://www.tutorialspoint.com/cprogramming/c\\_data\\_types.htm](http://www.tutorialspoint.com/cprogramming/c_data_types.htm)

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 6 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

## Integer Data Types with Defined Size

- A particular size of the integer data types can be specified, e.g., by the data types defined in the header file `<stdint.h>`
  - IEEE Std 1003.1-2001*

```
int8_t uint8_t
int16_t uint16_t
int32_t uint32_t
```

[lec03/inttypes.c](#)

<http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html>

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 9 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

# Part I Data Types

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 3 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

## Integer Data Types

- Size of the integer data types are not defined by the C norm but by the implementation
  - They can differ by the implementation, especially for 16-bits vs 64-bits computational environments.*
- The C norm defines that for the range of the types, it holds that
  - `short` ≤ `int` ≤ `long`
  - `unsigned short` ≤ `unsigned` ≤ `unsigned long`
- The fundamental data type `int` has usually 4 bytes representation on 32-bit and 64-bit architectures
  - Notice, on 64-bit architecture, a pointer is 8 bytes long vs int*
- Data type size the minimal and maximal value

| Type                      | Min value      | Max value     |
|---------------------------|----------------|---------------|
| <code>short</code>        | -32,768        | 32,767        |
| <code>int</code>          | -2,147,483,648 | 2,147,483,647 |
| <code>unsigned int</code> | 0              | 4,294,967,295 |

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 7 / 57

Numeric Types Character Type Logical Type Type Cast Arrays Pointers

## Floating Types

- C provides three floating types
  - `float` – Single-precision floating-point
    - Suitable for local computations with one decimal point*
  - `double` – Double-precision floating-point
    - Usually fine for most of the programs*
  - `long double` – Extended-precision floating-point
    - Rarely used*
- C does not define the precision, but it is mostly IEEE 754
  - ISO/IEC/IEEE 60559:2011*
- `double` – 64 bits (8 bytes) with sign, exponent, and mantissa
  - `s` – 1 bit sign (+ or -)
  - `Exponent` – 11 bits, i.e., 2048 numbers
  - `Mantissa` – 52 bits ≈ 4.5 quadrillions numbers
- A rational number  $x$  is stored according to  $x = (-1)^s \text{Mantissa} \cdot 2^{\text{Exponent}-\text{Bias}}$ 
  - 4 503 599 627 370 496*
- `Bias` allows to store exponent always as positive number
  - It can be further tuned, e.g., Bias =  $2^{eb-1} - 1$ , where  $eb$  is the number bits of the exponent.*

Jan Faigl, 2018 B3B36PRG – Lecture 03: Data types, Memory Storage Classes 10 / 57

## Character – char

- A single character (letter) is of the `char` type
- It represents an integer number (byte)
  - Character encoding (graphics symbols), e.g., ASCII – American Standard Code for Information Interchange.
- The value of `char` can be written as *constant*, e.g., `'a'`.

```
1 char c = 'a';
2
3 printf("The value is %i or as char '%c'\n", c, c);
```

```
clang char.c && ./a.out
The value is 97 or as char 'a'
```

- There are defined several control characters for output devices
  - The so-called *escape sequences*
  - `\t` – tabular, `\n` – newline,
  - `\a` – beep, `\b` – backspace, `\r` – carriage return,
  - `\f` – form feed, `\v` – vertical space

lec03/char.c

## Boolean type – `_Bool`

- In **C99**, the logical data type `_Bool` has been introduced
  - `_Bool` `logic_variable`;
- The value `true` is any value of the type `int` different from 0
- In the header file `stdbool.h`, values of `true` and `false` are defined together with the type `bool`

Using preprocessor

```
#define false 0
#define true 1
#define bool _Bool
```

- In the former (ANSI) C, an explicit data type for logical values is not defined
  - A similar definition as in `<stdbool.h>` can be used

```
#define FALSE 0
#define TRUE 1
```

## Type Conversions – Cast

- Type conversion transforms value of some type to the value of different type
- Type conversion can be
  - **Implicit** – automatically, e.g., by the compiler for assignment
  - **Explicit** – must be prescribed using the **cast operator**
- Type conversion of the `int` type to the `double` type is implicit

Value of the `int` type can be used in the expression, where a value of the `double` type is expected. The `int` value is automatically converted to the `double` value.

### Exempl

```
double x;
int i = 1;
x = i; // the int value 1 is automatically converted
// to the value 1.0 of the double type
```

- **Implicit type conversion is safe**

## Explicit Type Conversion

- Transformation of values of the `double` type to the `int` type has to be **explicitly** prescribed by the **cast operator**
- The fractional part is truncated

### Příklad

```
double x = 1.2; // declaration of the double variable
int i; // declaration of the int variable
int i = (int)x; // value 1.2 of the double type is
// truncated to 1 of the int type
```

- **Explicit type conversion can be potentially dangerous**

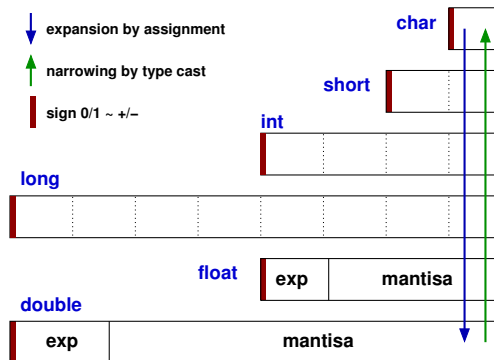
### Examples

```
double d = 1e30; // i is -2147483648
int i = (int)d; // which is ~ -2e9 vs 1e30
long l = 5000000000L; // i is 705032704
int i = (int)l; // (truncated to 4 bytes)
```

```
lec03/demo-type_conversion.c
```

## Type Cast of Numeric Types

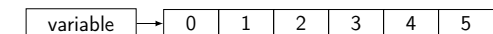
- The basic data types are mutually incompatible, but their values can be transformed by type cast



## Array

- A data structure to store **several data values of the same type**
  - Values are stored in a *continues block of memory*
- Each element has identical size, and thus its relative address from the beginning of the array is uniquely defined
  - Elements can be addressed by order of the element in the array

“address”=size\_of\_element \* index\_of\_element\_in\_the\_array



- The variable of the **array type** represents the address of the memory, where the particular values are stored
  - Address = `1st_element_address + size_of_the_type * index_of_the_element`
- The memory is allocated by the definition of the array variable
  - The array always has a particular size, i.e., defined by the number of the elements or automatically allocated by the compiler
- **Once the array is defined, its size cannot be changed!**

## Array Definition

- Definition consists of the type (of the array elements), name of the variable, and size (the number of elements) in the `[]` brackets
  - type variable [];**
- `[]` is also the array subscripting operator
  - array\_variable [index]**

### Example of array of `int` elements

```
int array[10]; // i.e., 10 x sizeof(int)
```

```
printf("Size of array %lu\n", sizeof(array));
printf("Item %i of the array is %i\n", 4, array[4]);
```

```
Size of array 40
Item 4 of the array is -5728
```

Values of individual elements are not initialized!

**C does not check validity of the array index during the program run time!**

## Arrays – Example

- Definition of 1D and **two-dimensional** arrays
  - `/* 1D array with elements of the char type */`
  - `char simple_array[10];`
  - `/* 2D array with elements of the int type */`
  - `int two_dimensional_array[2][2];`
- Accessing elements of the array
  - `m[1][2] = 2*1;`
- Example of the array definition and accessing its elements

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 int array[5];
6 printf("Size of array: %lu\n", sizeof(array));
7 for (int i = 0; i < 5; ++i) {
8 printf("Item[%i] = %i\n", i, array[i]);
9 }
10 return 0;
11 }
```

```
Size of array: 20
Item[0] = 1
Item[1] = 0
Item[2] = 740314624
Item[3] = 0
Item[4] = 0
```

lec03/array.c

## Array in a Function and as a Function Argument

- Array defined in a function is a local variable
  - The of the local variable is only within the block (function).

```
void fce(int n)
{
 int array[n];
 // we can use array here
 {
 int array2[n*2];
 } // end of the block destroy local variables
 // here, array2 no longer exists
} // after end of the function, a variable is automatically destroyed
```

- Array (as any other local variable) is automatically created at the definition, and it is automatically destroyed at the end of the block (function);
  - The memory is automatically allocated and released.
- Local variables are stored at the **stack**, which is usually relatively small
- Therefore, it may be suitable to allocate a large array dynamically (in the so called **heap** memory) using **pointers**

- Array can be argument of a function
  - `void fce(int array[]);`

**However, the value is passed as pointer!**

### Pointer

- Pointer is a variable which **value is an address** where the value of some type is stored
- Pointer *refers* to the memory location where a value (e.g., of another variable) is stored
- **Pointer is of type** of the data it can refer

*Type is important for the pointer arithmetic*

- Pointer to a value (variable) of primitive types: `char`, `int`, ...
- "Pointer to an array"; pointer to function; **pointer to a pointer**
- Pointer can be also without type, i.e., **void pointer**
  - Size of the variable (data) cannot be determined from the void pointer
  - The pointer can point to any address
- Empty address is defined by the symbolic constant **NULL**

*C99 - int value 0 can be used as well*

#### Validity of the pointer address is not guaranteed!

*Pointers allow to write efficient codes, but they can also be sources of many bugs. Therefore, acquired knowledge of the indirect addressing and memory organization is crucial.*

### Pointer - Examples 2/2

```
printf("i: %d -- pi: %p\n", i, pi); // 10 0x7fffffff8fc
printf("&i: %p -- *pi: %d\n", &i, *pi); // 0x7fffffff8fc
printf("*(&i)i: %d -- &(*pi): %p\n", *(&i), &(*pi));
printf("i: %d -- *pj: %d\n", i, *pj); // 10 10
i = 20;
printf("i: %d -- *pj: %d\n", i, *pj); // 20 20
printf("sizeof(i): %lu\n", sizeof(i)); // 4
printf("sizeof(pi): %lu\n", sizeof(pi)); // 8

long l = (long)pi;
printf("0x%lx %p\n", l, pi); /* print l as hex -- %lx */
// 0x7fffffff8fc 0x7fffffff8fc

l = 10;
pi = (int*)l; /* possible but it is nonsense */
printf("l: 0x%lx %p\n", l, pi); // 0xa 0xa
```

lec03/pointers.c

### Passing Arguments to Function

- In C, **function argument is passed by its value**
- Arguments are local variables (allocated on the stack), and they are initialized by the values passed to the function
 

```
void fce(int a, char *b)
{ /*
a - local variable of the int type (stored on the stack)
b - local variable of the pointer to char type (the value is address) the variable b is stored on the stack */
}
```
- Change of the local variable does not change the value of the variable (passed to the function) outside the function
- However, by passing a pointer, we have access to the address of the original variable

*We can achieve a similar behaviour as passing by reference.*

### Address and Indirect Operators

- **Address operator - &**
  - It returns the address of the memory location, where the value of the variable is stored **&variable**
- **Indirect operator - \***
  - It returns the **l-value** corresponding to the value at the address stored in the pointer variable **\*variable\_of\_the\_pointer\_type**
  - It allows to read and write values of the memory location addressed by the value of the pointer, e.g., pointer to the `int` type as `int *p`

```
*p = 10; // write value 10 to the address stored in the p variable
int a = *p; // read value from the address stored in p

The address can be printed using "%p" in the printf() function
int a = 10;
int *p = &a;

printf("Value of a %i, address of a %p\n", a, &a);
printf("Value of p %p, address of p %p\n", p, &p);

Value of a 10, address of a 0x7fffffff95c
Value of p 0x7fffffff95c, address of p 0x7fffffff950
```

### Pointers and Coding Style

- The **pointer type** is denoted by the `*` symbol
- `*` can be attached to the type name or the variable name
- `*` attached to the variable name is preferred to avoid oversight errors
 

```
char* a, b, c; char *a, *b, *c;
Only a is the pointer All variables are pointers
```
- Pointer to a pointer to a value of `char` type is `char ***a`;
- Writing pointer type (without variable): `char*` or `char**`
- Pointer to a value of empty type `void *ptr`
- Guaranteed not valid address has the symbolic name **NULL**
  - Defined as a preprocessor macro (0 can be used in C99)*
- Variables in C are not automatically initialized, and therefore, pointers can reference any address in the memory
- Thus, it may be suitable to **explicitly** initialize pointers to `0` or **NULL**
  - E.g., `int *i = NULL;`*

### Passing Arguments - Example

- The variable `a` is passed by its value
  - The variable `b` "implements calling by reference"
- ```
void fce(int a, char* b)
{
a += 1;
(*b)++;
}

int a = 10;
char b = 'A';
printf("Before call a: %d b: %c\n", a, b);
fce(a, &b);
printf("After call a: %d b: %c\n", a, b);
```
- Program output
Before call a: 10 b: A
After call a: 10 b: B

lec03/function_call.c

Pointer - Examples 1/2

```
int i = 10; // variable of the int type
           // &i - adresa of the variable i

int *pi;   // declaration of the pointer to int
           // pi pointer to the value of the int type
           // *pi value of the int type

pi = &i;   // set address of i to pi

int b;     // int variable

b = *pi;   // set content of the addressed reference
           // by the pi pointer to the to the variable b
```

Part II Functions and Memory Classes

Passing Arguments to the Program

- We can pass arguments to the `main()` function during program execution


```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("Number of arguments %i\n", argc);
for (int i = 0; i < argc; ++i) {
printf("argv[%i] = %s\n", i, argv[i]);
}
return argc > 1 ? 0 : 1;
```
- The program return value is passed by **return** in `main()`

```
./arg >/dev/null; echo $?
1
./arg first >/dev/null; echo $?
0
```
- In shell, the program return value is stored in `$?`, which can be print by `echo`
- `>/dev/null` redirect the standard output to `/dev/null`

Reminder

Program Interaction using stdin, stdout, and stderr

- The main function `int main(int argc, char *argv[])`
 - We can pass arguments to the program as text strings
 - We can receive return value of the program
 - By convention, 0 without error, other values indicate some problem
 - At runtime, we can read from `stdin` and print to `stdout`
 - E.g., using `scanf()` or `printf()`
 - We can redirect `stdin` and `stdout` from/to a file
 - In such a case, the program does not wait for the user input (pressing "Enter")
 - In addition to `stdin` and `stdout`, each (terminal) program has standard error output (`stderr`), which can be also redirected
 - `./program <stdin.txt >stdout.txt 2>stderr.txt`
- Instead of `scanf()` and `printf()` we can use `fscanf()` and `fprintf()`
 - The first argument of the functions is a file, but they behave identically
 - Files `stdin`, `stdout` and `stderr` are defined in `<stdio.h>`

Program Output Redirection – Example

```

1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int ret = 0;
6
7     fprintf(stdout, "Program has been called as %s\n", argv[0]);
8     if (argc > 1) {
9         fprintf(stdout, "1st argument is %s\n", argv[1]);
10    } else {
11        fprintf(stdout, "1st argument is not given\n");
12        fprintf(stderr, "At least one argument must be given!\n");
13        ret = -1;
14    }
15    return ret;
16 }

```

lec03/demo-stdout.c

```

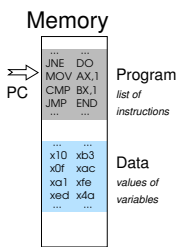
./demo-stdout; echo $?
Program has been called as ./demo-stdout
1st argument is not given
At least one argument must be given!
255

./demo-stdout 2>stderr
Program has been called as ./demo-stdout
1st argument is not given
./demo-stdout ARGUMENT 1>
stdout; echo $?
0

```

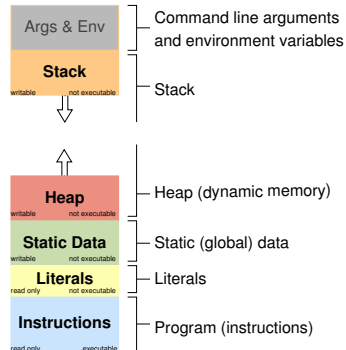
Computers with Program Stored in the Operating Memory

- A sequence of instructions is read from the computer operating memory
 - It provides great flexibility in creating the list of instructions
 - The program can be arbitrarily changed
- The computer architectures with the shared memory for data and program
 - Von Neumann architecture
 - John von Neumann (1903–1957)
 - Program and data are in the same memory type
 - Address of the currently executed instruction is stored in the Program Counter (PC)
- The architecture also allows that a pointer can address not only to data but also to the part of the memory where the program is stored
 - Pointer to a function



Basic Memory Organization

- The memory of the program can be categorized into five parts
 - Stack** – local variables, function arguments, return value
 - Automatically managed
 - Writable, not executable
 - Heap** – dynamic memory (`malloc()`, `free()`)
 - Managed by the programmer
 - Writable, not executable
 - Static** – global or "local" static variables
 - Initialized at the program start
 - Writable, not executable
 - Literals** – values written in the source code, e.g., strings
 - Initialized at the program start
 - Read only, not executable
 - Program** – machine instructions
 - Initialized at the program start
 - Read only, executable



Scope of Local Variables

- Local variables are declared (and valid) inside a block or function
 - Global variables are accessible "everywhere" in the program
 - A global variable can be shadowed by a local variable of the same name, which can be solved by the specifier `extern` in a block
 - http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm

```

1 int a = 1; // global variable
2
3 void function(void)
4 { // here, a represents the global variable
5     int a = 10; // local variable a shadowing the global a
6     if (a == 10) {
7         int a = 1; // new local variable a; access to the
8                 // former local a is shadowed
9         int b = 20; // local variable valid inside the block
10        a = b + 10; // the value of the variable a is 11
11    } // end of the block
12    // here, the value of a is 10, it is the local
13    // variable from the line 5
14
15    b = 10; // b is not valid (declared) variable
16 }

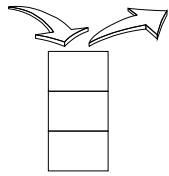
```

Variables and Memory Allocation

- Memory allocation is determination of the memory space for storing variable value
 - For local variables a function arguments the memory is allocated during the function call
 - The memory is allocated until the function return
 - It is automatically allocated from reserved space called **Stack**
 - The memory is released for the further usage.
 - The exceptions are local variables with the specifier `static`
 - Regarding the scope, they are local variables
 - But the value is preserved after the function/block end
 - They are stored in the static part of the memory
 - Dynamic allocation of the memory – library, e.g., `<stdlib.h>`
 - The memory allocation is by the `malloc()` function
 - Alternative memory management libraries exist, e.g., with garbage collector – `boehm-gc`
 - The memory is allocated from the reserved part of the memory called **Heap**

Stack

- Memory blocks allocated to local variables and function arguments are organized into **stack**
 - The memory blocks are "pushed" and "popped"
 - The last added block is always popped first
 - LIFO – last in, first out
 - The function call is also stored in the stack
 - The return value and also the value of the "program counter" denoted the location of the program at which the function has been called.
 - The variables for the function arguments are allocated on the stack
 - By repeated recursive function call, the memory reserved for the stack can be depleted, and the program is terminated with an error



Recursive Function Call – Example

```

#include <stdio.h>
void printValue(int v)
{
    printf("value: %i\n", v);
    printValue(v + 1);
}

int main(void)
{
    printValue(1);
}

```

lec03/demo-stack_overflow.c

```

clang demo-stack_overflow.c
ulimit -s 1000; ./a.out | tail -n 3
value: 31730
value: 31731
Segmentation fault

ulimit -s 10000; ./a.out | tail -n 3
value: 319816
value: 319817
Segmentation fault

```

Comment – Coding Style and return 1/2

- The `return` statement terminates the function call and pass the value (if any) to the calling function
 - How many times `return` should be placed in a function?
 - Comparison of two coding styles:
 - Style 1 (left): `return` is placed at the end of the function block.
 - Style 2 (right): `return` is placed at the end of each conditional branch.

Comment – Coding Style and return 2/2

- Calling **return** at the beginning can be helpful

E.g., we can terminate the function based on the value of the passed arguments.

- Coding style can prescribe to use only a single return in a function
Provides a great advantage to identify the return, e.g., for further processing of the function return value.
- It is not recommended to use **else** immediately after **return** (or other interruption of the program flow), e.g.,

```

case 10:
if (...) {
...
return 1;
} else {
if (cond) {
...
return -1;
} else {
break;
}
}

case 10:
if (...) {
...
return 1;
} else {
if (cond) {
...
return -1;
}
}
break;
    
```

Variables

- Variables denote a particular part of the memory and can be divided according to the type of allocation
 - **Static** allocation is performed for the definition of **static** and global variables. The memory space is allocated during the program start. The memory is never released (only at the program exit).
 - **Automatic** allocation is performed for the definition of local variables. The memory space is allocated on the **stack**, and the memory of the variable is automatically released at the end of the variable scope.
 - **Dynamic** allocation is not directly supported by the C programming language, but it is provided by library functions

E.g., malloc() and free() from the standard C library <stdlib.h> or <malloc.h>

http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html

Variable Declaration

- The variable declaration has general form
declaration-specifiers declarators;
- Declaration specifiers are:
 - **Storage classes:** at most one of the **auto, static, extern, register**
 - **Type quantifiers:** **const, volatile, restrict**
Zero or more type quantifiers are allowed
 - **Type specifiers:** **void, char, short, int, long, float, signed, unsigned.** In addition, **struct** and **union** type specifiers can be used. Finally, own types defined by **typedef** can be used as well.

Reminder from the 1st lecture.

Variables – Storage Classes Specifiers (SCS)

- **auto** (local) – Temporary (automatic) variable is used for local variables declared inside a function or block. Implicit specifier, the variables is on the **stack**.
- **register** – Recommendation (to the compiler) to store the variable in the CPU register (to speedup).
- **static**
 - Inside a block **{...}** – the variable is defined as static, and its value is preserved even after leaving the block It exists for the whole program run. It is stored in the **static (global) part of the data memory (static data)**.
 - Outside a block – the variable is stored in the **static data**, but its visibility is restricted to a module
- **extern** – extends the visibility of the (static) variables from a module to the other parts of the program Global variables with the **extern** specifier are in the **static data**.

Definitions – Example

```

Header file vardec.h
1 extern int global_variable;                                lec03/vardec.h

Source file vardec.c
1 #include <stdio.h>
2 #include "vardec.h"
3
4 static int module_variable;
5 int global_variable;
6
7 void function(int p)
8 {
9     int lv = 0; /* local variable */
10    static int lsv = 0; /* local static variable */
11    lv += 1;
12    lsv += 1;
13    printf("func: %d, lv %d, lsv %d\n", p, lv, lsv);
14 }
15 int main(void)
16 {
17     int local;
18     function(1);
19     function(1);
20     function(1);
21     return 0;
22 }
    
```

■ Output
 1 func: p 1, lv 1, slv 1
 2 func: p 1, lv 1, slv 2
 3 func: p 1, lv 1, slv 3

Comment – Variables and Assignment

- Variables are defined by the type name and name of the variable
 - Lower case names of variables are preferred
 - Use underscore **_** or **camelCase** for multi-word names
<https://en.wikipedia.org/wiki/CamelCase>
 - Define each variable on a new line
`int n;`
`int number_of_items;`
- The assignment statement is the assignment operating **=** and **;**
 - The left side of the assignment must be the **I-value – location-value, left-value** – it has to represent a memory location where the value can be stored
 - Assignment is an expression, and it can be used whenever an expression of the particular type is allowed

Storing the value to left side is a side effect.

```

/* int c, i, j; */
i = j = 10;
if ((c = 5) == 5) {
    fprintf(stdout, "c is 5 \n");
} else {
    fprintf(stdout, "c is not 5\n");
}
    
```

HW 03 – Assignment

Topic: Caesar Cipher

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

- **Motivation:** Experience a solution of the optimization task
- **Goal:** Familiar yourself 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 message and the poorly received message) 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
 - **Optional assignment** – an extension for considering missing characters in the received message and usage of the Levenshtein distance
https://en.wikipedia.org/wiki/Levenshtein_distance
- **Deadline: 17.03.2018, 23:59:59 PST** PST – Pacific Standard Time

Part III

Part 3 – Assignment HW 03

Summary of the Lecture

Topics Discussed

- Data types
- Arrays
- Pointers
- Memory Classes

■ Next: Arrays, strings, and pointers.