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

Jan Faigl

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

Lecture 03

B3B36PRG – Programming in C

Overview of the Lecture

Part I

Data Types

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.

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.

```
1 int i;
2 printf("%lu\n", sizeof(int));
3 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

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-bit 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
`short`	-32,768	32,767
`int`	-2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295

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;
4 printf("uc: %i\tsu: %i\n", uc, su);
5 uc = uc + 2;
6 su = su + 2;
7 printf("uc: %i\tsu: %i\n", uc, su);
```

```
$ clang -c signed_unsigned_char.c -o
    signed_unsigned_char
$ ./signed_unsigned_char
uc: 127 su: 127
uc: 129 su: -127
```

lec03/signed_unsigned_char.c

Coding of Negative Values

- Signed magnitude representation – the sign is encoded by the first bit (from the left), which supports an easy determination of the absolute value. The representation has two zeros. *Direct encoding*
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $-121_{(10)}$ $1111\ 1001_{(2)}$
  - $0_{(10)}$ $0000\ 0000_{(2)}$
  - $-0_{(10)}$ $1000\ 0000_{(2)}$
- Ones' complement – a negative value corresponds to the bit negation of the positive value. The representation has two zeros. *Inverse encoding*
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $-121_{(10)}$ $1000\ 0110_{(2)}$
  - $0_{(10)}$ $0000\ 0000_{(2)}$
  - $-0_{(10)}$ $1111\ 1111_{(2)}$
- **Two's-complement** – the negative value is stored as a positive value after bit negation increased by one.
  - **A single representation of zero.**
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $-121_{(10)}$ $1000\ 0111_{(2)}$
  - $127_{(10)}$ $0111\ 1111_{(2)}$
  - $-128_{(10)}$ $1000\ 0000_{(2)}$
  - $-1_{(10)}$ $1111\ 1111_{(2)}$

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

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*
  - **float** – 32 bits (4 bytes) with sign (1 bit), exponent (8 bits), and mantissa (23 bits).

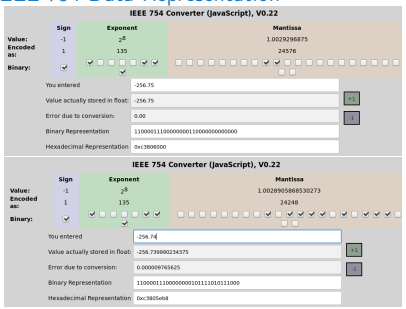
  - **double** – 64 bits (8 bytes) with sign, exponent, and mantissa.
    - s – 1 bit sign (+ or −).
    - **Exponent** – 11 bits, i.e., 2048 numbers.
    - **Mantissa** – 52 bits ≈ 4.5 quadrillions numbers.

4 503 599 627 370 496

  - A rational number x is stored according to $x = (-1)^s \textbf{Mantissa} \cdot 2^{\textbf{Exponent}-\textbf{Bias}}$.
  - **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.

Example of the IEEE 754 Data Representation



- Finite precision of the number representation -256.75 vs -256.74.
- `Infinity` (0x7f800000), `-Infinity` (0xff800000), a `NaN` (0x7fffffff).

https://www.h-schmidt.net/FloatConverter/IEEE754.html

Example of `float` Values Representation

Representation of the value 85.125 (`float`)

- 85 corresponds $1010101_{(2)}$.
- 0.125 corresponds 001
  - $0.125/2^{-1} = 0.25 \mid 0$
  - $0.125/2^{-2} = 0.50 \mid 0$
  - $0.125/2^{-3} = 1.00 \mid 1$
- 85.125 corresponds $1010101.001_{(2)} =$
  $= 1.010101001_{(2)} \times 2^6$.
- Bias for `float` is 127.
- Exponet is $127 + 6 = 133$
- 133 corresponds $10000101_{(2)}$.
- Normalized mantissa $010101001_{(2)}$ that is filled by zeros to 23 bits (from the right, it is decimal number).
- **0 - 1000 0101 - 0101 0100 1000 0000 0000 000**.
- **01000010 10101010 01000000 00000000**.
- In hexadecimal system **0x42 0xaa 0x40 0x00**, thus **0x42aa4000**.

Representation of the value 0.1 (`float`)

- 0.1 is periodic in binary system.
  1. $0.1 * 2 = 0.2 \mid 0$
  2. $0.2 * 2 = 0.4 \mid 0$
  3. $0.4 * 2 = 0.8 \mid 0$
  4. $0.8 * 2 = 1.6 \mid 1$
  5. $0.6 * 2 = 1.2 \mid 1$
  6. $0.2 * 2 = 0.4 \mid 0$
- Repeated pattern 0011, 23 bits represents lower value.
- $0.1_{(10)} \sim 0.0001\,1001\,1001\,1001\,1001\,1001\,100_{(2)} =$
  $= 1,1001\,1001\,1001\,1001\,1001\,100_{(2)} \times 2^{-4}$.
- Exponet is $127 - 4 = 123$ that corresponds $0111\,1011_{(2)}$.
- Normalized mantissa 1100 1100 1100 1100 1100 1100.
- **0 - 0111 1011 - 100 1100 1100 1100 1100 1100**.
- **00111101 11001100 11001100 11001100**.
- In hexadecimal system **0x3d 0xcc 0xcc 0xcc**, thus **0x3dcccccc**.
- In practice, 0.1 is encoded as a litte bit higher value **0x3dcccccd**, because its absolute error is lower.

lec03/floats.c

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

lec03/char.c

```
clang char.c -o char && ./char
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

Boolean type – `_Bool`

- In **C99**, the logical data type `_Bool` has been introduced.

```
1 _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.

```
1 #define false 0
2 #define true  1
4 #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.

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

- Implicit type conversion is safe.

Explicit Type Conversion

- Tranformation of values of the **double** type to the **int** type has to be explicitely prescribed by the **cast operator**.
- The franctional part is truncated.

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

- Explicit type conversion can be potentially dangerous.

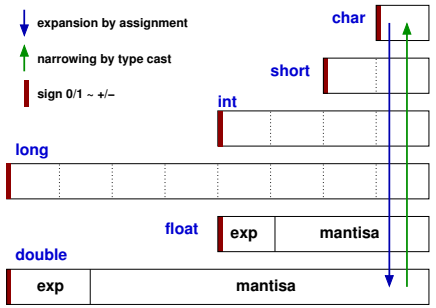
```
1 double d = 1e30;
2 int i = (int)d;
4 // i is -2147483648
5 // which is ~ -2e9 vs 1e30
```

```
1 long l = 5000000000L;
2 int i = (int)l;
4 // i is 705032704
5 // (truncated to 4 bytes)
```

lec03/demo-type_conversion.c

Type Cast of Numeric Types

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



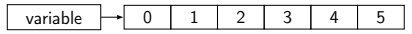
Array

- A data structure to store **several data values of the same type**, e.g., `int array[10];`.

*Values are stored in a **continuous 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, 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 array_variable [];

- [] is also the array subscripting operator.

array_variable [index]

- An example of array of `int` elements. *I.e., 10 × `sizeof(int)`*

```
1 int array[10];
3 printf("Size of array %lu\n",  sizeof(array));
4 printf("Item %i of the array is %i\n", 4, array[4]);
6 Size of array 40
7 Item 4 of the array is -5728
```

Values of individual elements are not initialized!

C does not check the validity of the array index during the program runtime!

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>
3  int main(void)
4  {
5     int array[5];
7     printf("Size of array: %lu\n", sizeof(array));
8     for (int i = 0; i < 5; ++i) {
9        printf("Item[%i] = %i\n", i, array[i]);
10    }
11    return 0;
12 }
```

```
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.

```
1 void fce(int n)                    The scope of the local variable is only within the block (function).
2 {
3    int array[n];
4    // we can use array here
5    {
6       int array2[n*2];
7    } // end of the block destroy local variables
8    // here, array2 no longer exists
9 } // 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, large arrays might be rather allocated 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 storing an **address** where the value of the particular type is stored.

For 64-bit systems, it is like a `long` variable, but its value is iterpreted as a memory address.

- Pointer *refers* to the memory location where a value is stored.
- **Pointer is of type** of the data it can refer.

Type is important for the pointer arithmetic and accessing the value referred to by the pointer.

  - Pointer to a value (variable) of primitive types: `char`, `int`, ...
  - "Pointer to an array"; pointer to function; pointer to a pointer.
- Pointer can also be of general (without) type, `void` pointer. *A general memory address.*
  - Size of the variable (data) cannot be determined from the void pointer.
- In general, pointer can point to any address.

Programmer is responsible to point to memory, where data are stored.

- Empty address is defined by the symbolic constant **NULL**. *Provably invalid address.*

C99 – `int` value 0 can be used as well.

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.

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 – ***
  - Returns the **l-value** corresponding to the value at the address stored in the pointer variable.

***variable** *Variable is of the pointer type.*

  - Allows reading and writing values from/to the memory addressed by the pointer's value.

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

- The address can be printed using `"%p"` in the `printf()` function.

```
1 int a = 10;
2 int *p = &a;
4 printf("Value of a %i, address of a %p\n", a, &a);
5 printf("Value of p %p, address of p %p\n", p, &p);
7 Value of a 10, address of a 0x7fffffffe95c
8 Value of p 0x7fffffffe95c, address of p 0x7fffffffe950
```


Pointer – Examples 1/2

```
1  int i = 10; // variable of the int type
2              // &i - address of the variable i
4  int *pi;    // declaration of the pointer to int
5              // pi pointer to the value of the int type
6              // *pi value of the int type
8  pi = &i;    // set address of i to pi
10 int b;      // int variable
12 b = *pi;    // set content of the addressed reference
13             // by the pi pointer to the to the variable b
```


Pointer – Examples 2/2

```
1  printf("i: %d -- pi: %p\n", i, pi); // 10 0x7fffffffe8fc
2  printf("&i: %p -- *pi: %d\n", &i, *pi); // 0x7fffffffe8fc 10
3  printf("*(&)i: %d -- &(*pi): %p\n", *(&i), &(*pi));
5  printf("i: %d -- *pj: %d\n", i, *pj);   // 10 10
6  i = 20;
7  printf("i: %d -- *pj: %d\n", i, *pj);   // 20 20
9  printf("sizeof(i): %lu\n", sizeof(i));  // 4
10 printf("sizeof(pi): %lu\n", sizeof(pi));// 8
12 long l = (long)pi;
13 printf("0x%lx %p\n", l, pi); /* print l as hex  -- %lx */
14 // 0x7fffffffe8fc 0x7fffffffe8fc
16 l = 10;
17 pi = (int*)l; /* possible but it is nonsense */
18 printf("l: 0x%lx %p\n", l, pi); // 0xa 0xa
```

lec03/pointers.c

Pointers and Coding Style

- The **pointer type** is denoted by the * character.
- * 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;
```

Only `a` is the pointer.

```
char *a, *b, *c;
```

All variables are pointers.

- Pointer to a pointer to a value of `char` type is `char **a;`.
- Writting 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 after definition.
- Thus, it may be suitable to **explicitly** initialize pointers to `0` or `NULL`.

E.g., `int *i = NULL;`

Part II

Functions and Memory Classes

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.

It is a new local variable allocated on the stack and initialized by the passed value.

- However, by passing a pointer, we have access to the address of the original variable; so, we can change the value at the passed address.

We can achieve a similar behaviour as passing by reference.

Passing Arguments – Example

- The variable `a` is passed by it 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

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;
}
```

```
$ clang demo-arg.c -o arg
$ ./arg one two three
Number of arguments 4
argv[0] = ./arg
argv[1] = one
argv[2] = two
argv[3] = three
```

lec03/demo-arg.c

- 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`.

Program Interaction using `stdin`, `stdout`, and `stderr`

- The main function `int main(int argc, char *argv[])`.
  - The program arguments are passed 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.
    - "File names" `stdin`, `stdout` and `stderr` are defined in `<stdio.h>`.

Program Output Redirection – Example

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int ret = 0;
   fprintf(stdout, "Program has been called as %s\n", argv[0]);
   if (argc > 1) {
      fprintf(stdout, "1st argument is %s\n", argv[1]);
   } else {
      fprintf(stdout, "1st argument is not given\n");
      fprintf(stderr, "At least one argument must be given!\n");
      ret = -1;
   }
   return ret;
}
```

lec03/demo-stdout.c

- Example of the output: `clang demo-stdout.c -o demo-stdout`.

```
$ ./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

- Program instructions are read from the computer memory.
- It provides great flexibility in creating the list of instructions.

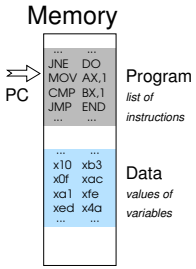
The program can be arbitrarily changed.

- Von Neumann architecture is the computer architecture with program and data in the same memory type.

John von Neumann (1903–1957)

  - Address of the currently executed instruction is stored in the Program Counter (**PC**).
  - Calling a function is setting the PC to the address where the function implementation (as a sequence of instructions) starts.
  - Branching is a "jump" to a block of instructions, e.g., `if-else`.
  - Calling function or jumps need to manage passing function arguments, return values, or location variables.

Auto variables using stack.



- 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

Program Memory Organization

- The memory of the program can be categorized into five parts.
- **Stack** – local variables, function arguments, return value.

Automatically managed

- **Heap** – dynamic memory (`malloc()`, `free()`).

Managed by the programmer.

- **Static** – global or "local" `static` variables.

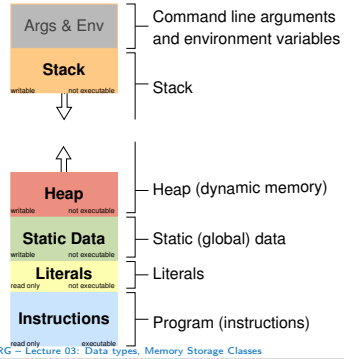
Initialized at the program start.

- **Literals** – values written in the source code, e.g., strings.

Initialized at the program start.

- **Program** – machine instructions.

Initialized at the program start.



Scope of Local Variables

- Local variables are declared (and valid) inside a block or function.

```
int a = 1;  // global variable

void function(void)
{  // here, a represents the global variable
   int a = 10; // local variable a shadowing the global a
   if (a == 10) {
      int a = 1;  // new local variable a; access to the
                  // former local a is shadowed
      int b = 20; // local variable valid inside the block
      a = b + 10; // the value of the variable a is 11
   } // end of the block
   // here, the value of a is 10, it is the local
   // variable from the line 5
   b = 10; // b is not valid (declared) variable
}
```

- Global variables are accessible "everywhere" in the program, but they can be shadowed by a local variable of the same name.

Shadowed variables can be accessed using the specifier `extern` in a block.

http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm

Variables and Memory Allocation

- Memory allocation is determination of the memory space for storing variable value.
- For local variables (the 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**.

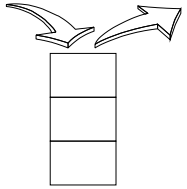
The memory is released for the further usage.

  - 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 in 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 calls, the memory reserved for the stack can be depleted, and the program is terminated with an error.

Recursive Function Call – Example

- Try yourself to execute the program with a limited stack size, set by `ulimit`.

```
1  #include <stdio.h>
3  void printValue(int v);
5  int main(void)
6  {
7     printValue(1);
8  }
10 void printValue(int v)
11 {
12    printf("value: %i\n", v);
13    printValue(v + 1);
14 }
```

```
$ 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
```

lec03/demo-stack_overflow.c

Comment – Coding Style and **return** 1/2

- `return` terminates the function call and pass the value (if any) to the calling function.

```
1  int doSomeThingUseful() {
2     int ret = -1;
3     ...
4     return ret;
5  }
```

- How many times **return** should be placed in a function? *Inversion*

See "Why You Shouldn't Nest Your Code" – https://youtu.be/CFRhGnuXG-4.

```
1  int doSomething() {
2     if (
3           !cond1
4        && cond2
5        && cond3
6        ) {
7        ... do some long code ...
8     }
9     return 0;
10 }
```

```
1  int doSomething() {
2     if (cond1) {
3        return 0;
4     }
5     if (!cond2) {
6        return 0;
7     }
8     if (!cond3) {
9        return 0;
10    }
11    ... some long code ....
12    return 0;
13 }
```

http://llvm.org/docs/CodingStandards.html

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.

- However, 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., see the example.

```
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.
  - **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>`.

  - **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).

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

Variable Definition/Declaration

- The variable definition/declaration has the general form
  declaration-specifiers declarators;
- Declaration specifiers are:
  - **Storage classes**: at most one of the `auto`, `static`, `extern`, `register`.

Using `extern` it becomes the variable declaration.

  - **Type quantifiers**: `const`, `volatile`, `restrict`.

None 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 first lectures.

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** – Suggest (to the compiler) to store the variable in the CPU register (hint).

*Note that a register variable does not have memory address; thus, it is not **l-value**!*

It might not be necessary for modern compilers with advanced optimizations.

- **static**
  1. 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**).
  2. 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 global module variables from the module to the other parts of the program.
  - Global variables with the `extern` specifier are stored in the **static data**.

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 **l-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");
}
```

lec03/assign.c

Part III

Part 3 – Assignment HW 02

Part IV

Part 4 – Coding Examples (optional)

Coding Example – Pointers in Swap Function 1/2

- Implement a function that swap values of two variables `swap`.
- The swap of the variables' values can be implemented using temporary variable.

```
int a = 10;
int b = 20;

int t = a;
a = b;
b = t;
```

- However, passing the integer values of the variables into a function `void swap(int a, int b);` does not yield the expected result.
- It is because new local variables are defined.

```
...
int main(void)
{
   int a = 10;
   int b = 20;
   printf("a: %d b: %d\n", a, b);
   swap(a, b);
   printf("a: %d b: %d\n", a, b);
...
void swap(int a, int b)
{
   int t = a;
   a = b;
   b = t;
}
```

```
$ clang swap.c -o swap && ./swap
a: 10 b: 20
a: 10 b: 20
```


Coding Example – Pointers in Swap Function 2/2

- We need to pass addresses of the local variables `a` and `b` defined in the calling (`main`) function.
- Then, we can access the values at the passed addresses using indirect addressing operator `*`, e.g., `int t = *a;`.
- The variables `a` and `b` in the `main` function are integer values.

  Most likely `sizeof(a)` would be 4 bytes.
- The variables `a` and `b` in the `swap` function are pointers to integer values.

  Most likely `sizeof(a)` would be 8 bytes.
  Most likely `sizeof(*a)` would be 4 bytes (`int`).
- Try and experiment with the code yourself!

  Add prints, e.g., using `printf("%p" ...);`.

```
void swap(int *a, int *b);
int main(void)
{
   int a = 10;
   int b = 20;
   printf("a: %d b: %d\n", a, b);
   swap(&a, &b);
   printf("a: %d b: %d\n", a, b);
...
void swap(int *a, int *b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
```

```
$ clang swap.c -o swap && ./swap
a: 10 b: 20
a: 20 b: 10
```


Summary of the Lecture

Topics Discussed

- Data types
- Arrays
- Pointers
- Memory Classes

- Next: Arrays, strings, and pointers.