JTSK-320111

Programming in C I

C-Lab I

Lecture 3 & 4

Dr. Kinga Lipskoch

Fall 2016

This Week's Agenda

- Type conversions and some more operators
- Booleans
- Decision and Control Statements
- Looping Statements
- Everything about functions:
  - Prototypes
  - Header files
  - Variable scope
  - Recursion
- Strings

Type Conversions

- When data of different types are combined (via operators) some rules are applied
- Types are converted to a common type
  - Usually, to the larger one (called promotion)
  - **Example:** while summing an `int` and a `float`, the `int` is converted into a `float` and then the sum is performed
- A demotion is performed when a type is converted to a smaller one
  - **Example:** a function takes an `int` parameter and you provide a `float`
- A demotion implies possible loss of information
- Therefore, be careful with what to expect
  - In the above example, the fractional part will be lost

Casting

- It is possible to overcome standard conversions (casting)
- To force to a different data type, put the desired data type before the expression to be converted
  (type name) expression
- Casting is a unary operator with high precedence

Casting: An Example

```c
int a;
float f1 = 3.456;
float f2 = 1.22;
/* these operations imply demotions */
a = (int) f1 * f2;     /* a is now 3 */
a = (int) (f1 * f2);  /* a is now 4 */
```

Incrementing and Decrementing

- The unary operators ++ and -- can be applied to increase or decrease a variable by 1

```
int a, b;
a = b = 0;
a++; b-- ; ++a ; --b;
```

- Note that they can be both prefix and postfix operators
  - The two versions are different

Prefix and Postfix Modes

- Prefix means that first you modify and then you use the value
- Postfix means that first you use and then you modify the value
- `int a = 10, b;`

Expression	**New value of `a`**	**New value of `b`**
`b = ++a;`	11	11
`b = a++;`	11	10
`b = --a;`	9	9
`b = a--;`	9	10

The sizeof() Operator

- sizeof() returns the number of bytes needed to store a specific object
- Useful for determining the sizes of the different data types on your system

```
int a;
printf("size int %lu\n", sizeof(a));
printf("size float %lu\n", sizeof(float));
printf("size double %lu\n", sizeof(double));
```

- For strings do not confuse sizeof() with strlen()
- Compile-time operator, will not work for dynamically allocated memory

Boolean Variables

- A boolean variable can assume only two logic values: **true** or **false**
- Boolean variables and expressions are widely used in computer languages to control branching and looping
- Some operators return boolean values
- A boolean expression is an expression whose value is **true** or **false**

Boolean Operators

- Boolean operators can be applied to boolean variables
  - AND, OR, NOT

A	**NOT A**	**A**	**B**	**A AND B**	**A**	**B**	**A OR B**
false	true	false	false	false	false	false	false
true	false	false	true	false	false	true	true
		true	false	false	true	false	true
		true	true	true	true	true	true

Booleans in C

- C does not provide an ad-hoc boolean type but uses rather the `int` type
- 0 is false, everything different from 0 is true
- C also provides the three Boolean operators
  - `&&` for AND,
  - `||` for OR,
  - `!` for NOT
- Applied to booleans they return booleans

Boolean Operators: Example

```
int main() {
  int a, b, c;
  a = 0;                /* a is false */
  b = 57;               /* b is true */
  c = a || b;           /* c is true */
  c = a && b;           /* c is false */
  a = !a;               /* a is now true */
  c = a && b;           /* c is now true */
  c = (a && !b) && (a || b);
  return 0;
}
```

Relational Operators

- Relational operators are applied to other data types (numeric, character, etc.) and produce boolean values

```
(b > 5) --> true
```

- Relational operators with boolean operators produce boolean expressions

```
(b > 5) && (a < 1) --> true && false --> false
```

Relational operator	**Meaning**
==	Equality test
!=	Inequality test
>	Greater
<	Smaller
>=	Greater or equal
<=	Smaller or equal

Relational Operators: Example

```c
int main() {
  int a = 2, b, c;
  float f1 = 1.34;
  float f2 = 3.56;
  char ch = 'D';
  b = f1 >= f2;
  c = !b;
  b = c == b;
  b = b != c;
  c = f2 > a;
  c = ch > a;
  return 0;
}
```

Branching

- Up to now programs seem to execute all the instructions in sequence, from the first to the last (a linear program)
- Change the control flow of a program with branching statements
- Branching allows to execute (or not to execute) certain parts of a program depending on boolean expressions or conditions

Selection: `if ... else`

- In general selection constructs allow to choose a way in a binary bifurcation
- De facto you can use it in three ways
  - `if ()` single selection
  - `if ()`
    `else` double selection
  - `if ()`
    `else if ()`
    `else if ()`
    `...`
    `else` multiple selection

The if Syntax (1)

- General syntax:

```
if (condition)
  statement 1;
else
  statement 2;
other_statement; /* always executed */
```

- The else part can be omitted
- Statement: single statement or multiple statements
- Multiple statements need to be surrounded by braces { }

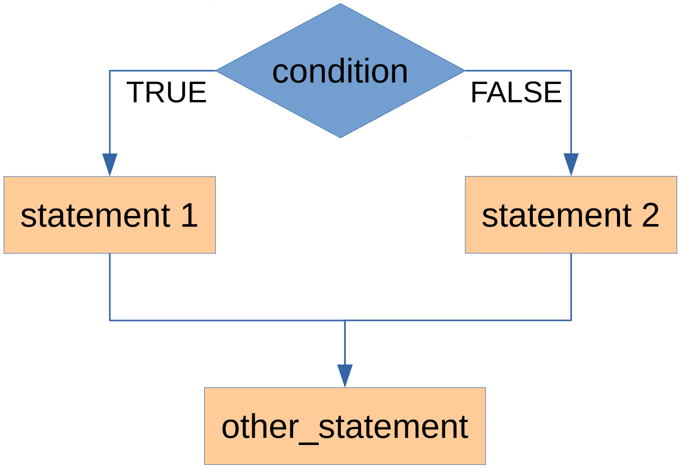
The if Syntax (2)

- Preferred syntax (always use braces)

```
if (condition) {
  statements;
}
else {
  statements;
}
```

- If you add statements, program flow is not changed (less errors)
- Using indentation, you can easily see where block starts and ends

`if`: Flow Chart



if: Example

```c
#include <stdio.h>
int main() {
  int first, second;
  printf("Type the first number:\n");
  scanf("%d", &first);
  printf("Type the second number:\n");
  scanf("%d", &second);
  if (first > second) {
    printf("Bigger one is %d\n", first);
  }
  else {
    printf("Bigger one is %d\n", second);
  }
  printf("Can you see the error?\n");
  return 0;
}
```

Statements and Compound Statements

- Statements can be grouped together to form compound statements
- A compound statement is a set of statements surrounded by braces

```
int a = 3;
if (a > 0) {
  printf("a is positive %d\n", a);
  a = a - 2 * a;
  printf("now a is negative %d\n", a)
}
```

Multiple Choices: `switch`

- `switch` can be used when an expression should be compared with many values
- The same goal can be obtained with multiple `if`'s
- The expression must return an integer value

switch: The Syntax

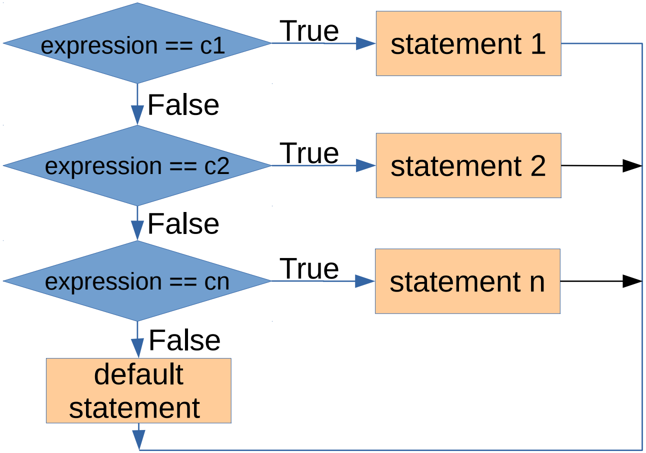
```
switch (expression)  {
  case c1:
    statement1;
    break;

  case c2:
    statement2;
    break;

  ...

  default:
    default statement;
}
```

`switch`: Flow Chart



switch: Example

```c
#include <stdio.h>
int c;
int main() {
  for (c = 0; c <= 3; c++) {
    printf("c: %d\n", c);

    switch (c) {
      case 1:
        printf("Here is 1\n");
        break;
      case 2:
        printf("Here is 2\n");
        /* Fall through */
      case 3:
      case 4:
        printf("Here is 3, 4\n");
        break;
      default:
        printf("Here is default\n");
    }
  }
  return 0;
}
```

Iterations

- In many cases it is necessary to repeat a set of operations many times
- Example: compute the average grade of the exam
  - Read all the grades, and sum them
  - Divide the sum by the number of grades
- C provides three constructs

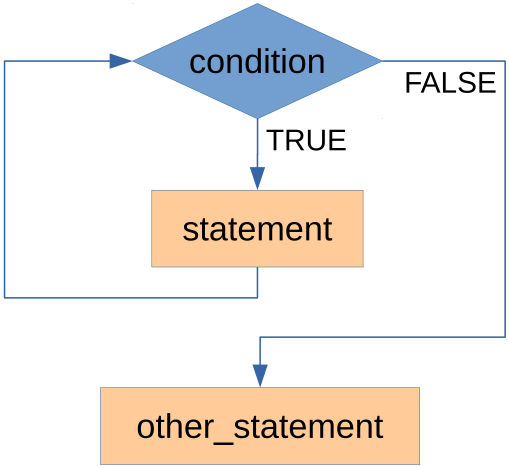
Iterations: while

- General syntax:

```
while (condition) {
  statement;
}
```

- Keep executing the statement as long as the condition is true

`while`: Flow Chart



Example:
Compute the Sum of the First n Natural Numbers

```c
#include <stdio.h>
int main() {
  int idx, n, sum = 0;
  printf("Enter a positive number ");
  scanf("%d", &n);
  idx = 1;
  while (idx <= n) {
    sum += idx;
    idx++;
  }
  printf("The sum is %d\n",sum);
  return 0;
}
```

Iterations: `for`

- General syntax:

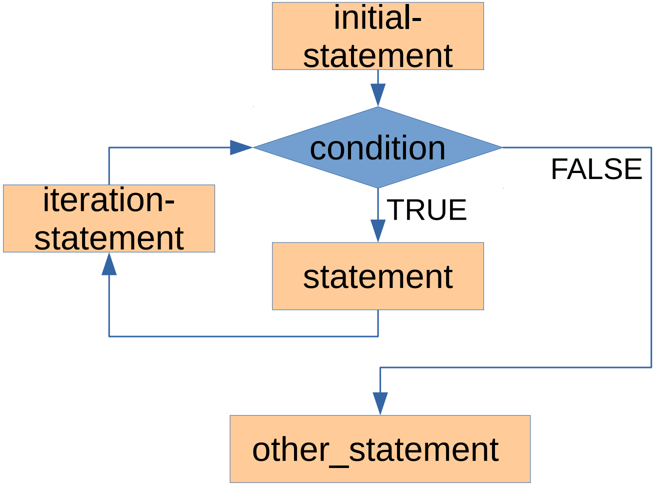
```
for (initial-statement; condition; iteration-
    statement)
  statement;
```

- Example:

```
for (n = 0; n <= 10; n++)
  printf("%d\n", n);
```

- The `for` and `while` loops can be made interchangeable

`for`: Flow Chart



for: Example Revised

```c
#include <stdio.h>
int main() {
  int idx, n, sum = 0;
  printf("Type a positive number ");
  scanf("%d", &n);
  for (idx = 1; idx <= n; idx++) {
    printf("Processing %d..\n", idx);
    sum += idx;
  }
  printf("The sum is %d\n", sum);
  return 0;
}
```

Boolean Operators and if

```c
for (n = 0; n < 3; n++) {
  for (i = 0; i < 10; i++) {
    if (n < 1 && i == 0) {
      printf("n is < 1, i is 0\n");
    }
    if (n == 2 || i == 5) {
      printf("HERE n: %d i:%d\n", n, i);
    }
    else {
      printf("n:%d, i:%d\n", n, i);
    }
  }
}
```

Easier or Harder to Read?

```
for (n = 0; n < 3; n++)
  for (i = 0; i < 10; i++) {
    if (n < 1 && i == 0) {
      printf("n is < 1, i is 0\n"); }
      if (n == 2 || i == 5) {
       printf("HERE n: %d i:%d\n", n, i); }
      else {
        printf("n:%d, i:%d\n", n, i); }}}
```

Iterations: `do ... while`

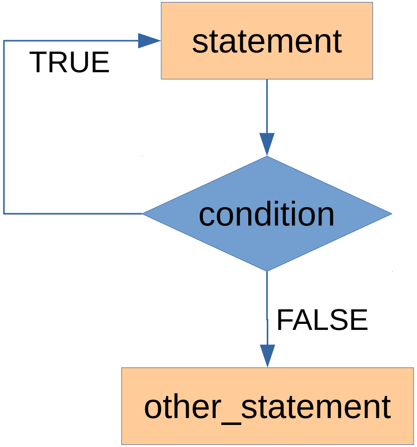
- General syntax:

```
do
  statement;
while (condition);
```

```
do {
  statement1;
  statement2;
} while (condition);
```

- In this case the end condition is evaluated at the end
- The body is always executed at least once

`do ... while`: Flow Chart



do ... while: Example

```c
#include <stdio.h>
int main() {
  int n, sum = 0;
  do {
    printf("Enter number (<0 ends)");
    scanf("%d", &n);
    sum += n;
  } while (n >= 0);
  sum -= n; /* Remove last negative value */
  printf("The sum is %d\n", sum);
  return 0;
}
```

Jumping Out of a Cycle: break

- The keyword break allows to jump out of a cycle when executed
- We have already seen this while discussing switch

```c
int num, i = 0;
scanf("%d", &num);
while (i < 50) {
  printf("%d\n", i);
  i++;
  if (i == num)
    break;
}
```

Jumping Out of a Cycle: `continue`

- `continue` jumps to the expression governing the cycle
- The expression is evaluated again and so on

```
char c;
while ((c = getchar()) != '\n') {
  // ignore the letter b
  if (c == 'b')
    continue;
  printf("%c", c);
}
```

Jumping Out of a Cycle

- Do not abuse `break` and `continue`
- You can always obtain the same result without using them
  - This at the price of longer coding
- By using them your code gets more difficult to read
- When you are experienced you will master their use
  - Meanwhile, learn the basics

Iterations: General Comments

- Inside the body of the loop you must insert an instruction that can cause the condition to become false
- If you do not do that, your program will fall into an infinite loop and will be unable to stop (Press `Ctrl-C` to stop such a program)
- `do ... while` is far less used than `while` and `for`
- The same constructs are provided in the majority of other programming languages

Arrays in C

- See first lecture for introduction
- In C you declare an array by specifying the size between square brackets
- Example: `int my_array[50];`
- The former is an array of 50 elements
- The first element is at position 0, the last one is at position 49

Accessing an Array in C

- To write an element, you specify its position

```
  my_array[2] = 34;
  my_array[0] = my_array[2];
```

- Pay attention: if you specify a position outside the limit, you will have unpredictable results segmentation fault, bus error, etc.
- And obviously wrong
- Note the different meaning of brackets
- Brackets in declaration describe the dimension, while in program they are the index operator

Arrays with Initialization

- C allows also the following declarations:

```
int first_array[]    = {12, 45, 7, 34};
int second_array[4] = {1, 4, 16, 64};
int third_array[4]  = {0, 0};
```

- It is not possible to specify more values than the declared size of the array
- The following is wrong:

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

Typical Structure of a C Program

```c
#include <stdio.h>
int rect_area(int length, int width) {
  int area;
  area = length * width;
  return area;
}

int main() {
  int a, b;
  a = rect_area(5, 7);
  printf("Area of first rectangle is %d\n", a);
  b = rect_area(3, 4);
  printf("Area of second rectangle is %d\n", b);
  return 0;
}
```

Predefined and User Defined Functions

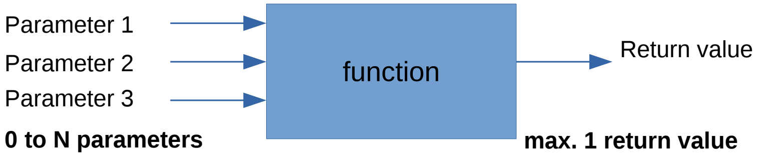
- Predefined functions are functions provided by the language or by the host
- Operating system
  - Library functions: they usually provide general purpose functionalities
- User defined functions are defined by the program
  - Usually targeted to the problem being solved

Functions: Motivation

- Writing a 50000 lines long main function can be really difficult
- Splitting the code into many small pieces has many advantages:
  - Easier to develop
  - Easier to maintain and debug
  - Increased opportunities to reuse the code
- An example: the `printf` function
  - Developed by specialists
  - Up to now we used it without knowing how it works internally
  - Should there be a bug in it, by just using an updated version you can fix your code at once

Some Analogies

- A function can be thought as a mathematical function
- A function can be thought as a black box performing some functionality



Functions in C

- **Function declaration** (prototyping)
- **Function call** (use)
- **Function definition**
- Call should be preceded by prototyping (ANSI C (American National Standards Institute) strongly advises this)
- There can be many declarations and many calls
- There must be exactly one definition

Prototyping

- The prototype is a statement declaring `return_type functionname(parameters);`
- Returned type is the type of the data
  - may be empty, default type is `int`
  - always declare the `return_type` explicitly
- Name follows the usual rules
- Parameters specify the number and types of the possible parameters
  - may be empty
  - always use explicit `void`, if function does not take arguments

The `void` Keyword

- `void` can be used to specify that
  - The function does not return any value
  - The function does not take any parameter
- `int unknown(void);`
  - function does not take any parameters
- `int unknown();`
  - function takes arbitrary number of parameters (to be compliant with the old Kernighan & Ritchie style)

Remember the Difference

- `void`
  - No return value
  - No parameter
- `void *`
  - Generic pointer (a pointer with no specific type which can be casted to any type)

Prototyping: Why?

- By having a prototype the compiler can check if the calls are performed correctly
  - Number of parameters, types, etc.
- It is now clear why prototypes should always appear before calls

Prototypes: Examples

- Prototypes of functions in `math.h`

```
double sqrt(double x);
double pow(double x, double y);
```

- User defined function prototypes

```
int find_max(int v[], int dim);
void print_menu(char *options[], int dim);
void do_something(void);
```

- `void` specifies no return value and empty parameters list

Typical Structure of a C Program

```
#include <stdio.h>
int rect_area(int length, int width);
float b_func(int a, int b);
int main() {
  ...
  c = rect_area(5, 7);
  b_func(11, 6);
  return 0;
}
int rect_area(int length, int width) {
  ...   /* do some operations */
  return area;
}
float b_func(int a, int b) {
  ...   /* do some operations */
  return c;
}
```

Calling a Function

- To call a function you insert its name
  - Function call is a statement
- You have to provide suitable parameters
  - Number and type of parameters must match function declaration
- The result of a function can be ignored

An Example

```
#include <math.h>
#include <stdio.h>
int main() {
  double number, root;
  scanf("%lf", &number);
  if (number >= 0) {
    root = sqrt(number);
    printf("Square root is %f\n", root);
    sqrt(number); /* useless but legal */
    /* What can I print now? */
  }
  else
    printf("Cannot calc square root\n");
}

      gcc -Wall -lm -o example example.c
```

Function Definition

- The function definition specifies what a functions does
- Function definitions can contain everything (variables definitions, cycles, branches, etc) but NOT other function definitions
- A function terminates when
  - it executes the last instruction
  - it encounters a return statement
- Definition starts with the function header
  `return type, name, parameters info`
- Braces to define where the function starts and ends
- Business statements (instructions for carrying out the function's task)

Finding the Maximum Value in an Array

```c
/*  v[]: array of ints
    dim: number of elements in v
    Returns the greatest element in v
*/
int findmax(int v[], int dim) {
  int i, max;

  max = v[0];
  for (i = 1; i < dim; i++) {
    if (v[i] > max)
      max = v[i];
    }
  return max;
}
```

Looking for an Element

```
/*  v[]: array of ints
    dim: number of elements in v
    t: element to find
    Returns   1 if t is not present in v or
    its position in v
*/
int find_element(int v[], int dim, int t)  {
  int i;
  for (i = 0; i < dim; i++) {
    if (v[i] == t)
      return i;
  }
  return -1;
}
```

What Happens when a Function is Called?

- The given parameters are copied into the corresponding entry in the parameters list
- The control is transferred to the function
- When the called function terminates, the control goes back to the caller function

Flow of Execution

```c
#include <stdio.h>

int main() {
  int array[] = {2, 4, 8, 16, 32};
  int result;

  result = find_element(array, 5, 37);
  if (result == -1)
    printf("37 is not present\n");

  return 0;
}
```

Comment your Functions

- Every function should be commented
  - Describe what the function does
  - Describe each parameter (type and meaning)
  - Describe what the function returns
- Look at the UNIX man pages to have an idea of how function documentation should look like

  ```
  man strcmp
  ```

Local Variables

- Variables can be declared inside any function
    - These are called local variables
    - Local variables are created when the function is called (e.g., the control is transferred to the function) and are destroyed when the function terminates
- Local variables do not retain their values between different calls

The Concept of Scope

- The scope of a name (function, variable, constant) is the part of the program where that name can be used
- The scope of a local variable is the function where it is defined
    - From the point of its definition
- Names having different scopes do not clash

Global Scope

- The scope of the names of functions goes from the prototype/definition to the end of file
- After their name is known they can be used, i.e., called
- It is possible to define global variables, i.e., variables outside function
  - Their scope is from the point of definition to the end of the file
  - After their definition is given they can be used, i.e., written and read

Local and Global Scope

```c
#include <stdio.h>

//global variable
int x = 7;

void xlocal(int y)  {
  int x;
  x = y * y;
  printf("xlocal: %d\n", x);
  return;
}

void xglobal(int y)  {
  x = y * x;
  printf("xglobal: %d\n", x);
  return;
}
```

```c
int main() {
  //int x;
  // try to explain if not
  // commented out
  x = 8;
  printf("main: %d\n", x);
  xlocal(x);
  printf("main: %d\n", x);
  xglobal(x);
  printf("main: %d\n", x);
  return 0;
}
```

Do not Misuse Global Variables

- Global variables can be used to communicate parameters between functions
- They can introduce subtle bugs in your code
- In general try to avoid them unless enormous advantages can be gained at a price of low risk
    - Document why you insert them
- Bigger projects will avoid using global variables

Parameters

- Function parameters are treated as local variables
- Local variables within functions and parameters must have different names
- Therefore the scope of a parameter is its function

Parameters: by Value and by Reference

- **By value**: variables are copied to parameters
  - Changes made to parameters are not seen outside the function
- **By reference**: variables and parameters coincide
  - Changes made to parameters are seen outside the function
  - In C this is obtained by mean of pointers

Example

```c
#include <stdio.h>
void increase(int par) {
  par++;
}
/*  In this case no prototype:
    can you tell why? */
int main() {
  int number = 5;
  increase(number);
  printf("Increased number is %d\n", number);
  /* not as expected? */
}
```

Parameters by Reference in C

- C passes only parameters by value
- For references it is necessary to provide a pointer to the variable
- In order to make a modification visible
- Outside it is necessary to use the dereference ($*$) operator

Parameters by Reference: Example

```c
#include <stdio.h>

void increase(int *par) {
  *par = *par + 1;
}

int main() {
  int number = 5;
  increase(&number); /* pass pointer */
  printf("Increased number is %d", number);
}
```

Indentation Styles (1)

- Use spaces between operators: `a = b + 5;`
- Exception: `b++;`
- Do not use spaces if parentheses act as delimiter (functions)
  `printf("Number %d", b);`
- But use spaces before after `if`, `for`, `while`:
  `while (i <= 10)`
- Always put a space after comma
- Do not put a space before semicolon:
  `printf("Number %d", b);`

Indentation Styles (2)

- Put the opening brace either behind last word (including space) or put it on the next line
- Indent the block inside by tab or 4 (8) spaces
- The closing brace should be on the same column as the opening statement

```c
for (i = 0; i < 10; i++) {    // K&R style
  printf("%d\n", i);
}
```

or

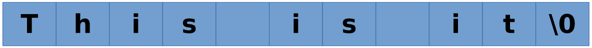
```c
for (i = 0; i < 10; i++)     // Allman style
{
  printf("%d\n", i);
}
```

Strings

- A string is a sequence of characters
- Strings are often the main way used to communicate information to the user
- Many languages provide a string data type, but C does not
- In C strings are treated as arrays of characters
- `char my_string[30];`

C strings

- A string is represented as a sequence of chars enclosed by double quotes
  - `"This is it"`
- String are stored in arrays of chars
  - An extra character is always added at the end to mark the end of the string
  - The extra character is the `'\0'` character i.e., the character whose ASCII code is 0



fgets versus gets (1)

- gets does not check if you type more characters than allowed:
  ```
  char inputString[50];
  gets(inputString);
  ```
- fgets allows additional parameters:
  ```
  char line[50];
  fgets(line, sizeof(line), stdin);
  ```
  - Reads up to 49 characters from the input stream
  - The 50th one is used to store the null character '\0'

`fgets` versus `gets` (2)

- `gets` replaces the trailing '\n' with a '\0'
- `fgets` does not replace '\n', but it leaves it in the string
- Read the man pages for learning more on these functions
  - `man gets`
  - `man fgets`
- To make your life easier use `fgets` and convert to integer via `sscanf`
- Avoid using `gets`, it is unsafe

fgets and scanf together

- scanf and fgets do not work well together
- Your code should look like this, if you use both

```
scanf("%d", &number);
getchar();
...
fgets(line, sizeof(line), stdin);
sscanf(line, "%d", &number);
```

String Functions

- Defined in `string.h`

`strlen`	Determines the length of a string
`strcat`	Concatenates two strings
`strcpy`	Copies one string into another
`strcmp`	Compares two strings
`strchr`	Searches a char in a string

- See man pages
  - Do not reinvent the wheel, there are many many functions that will help you

Pointers and Address Arithmetic

- The arithmetic operators for sum and difference (+, -, ++, --, etc) can be applied also to pointers
  - After all a pointer stores an address, which is an integer
- These operators are subject to the "address arithmetic".
- Increasing a pointer means that the pointer will point to the following element
  - You can also add a number other than 1
- From a logic point of view the pointer is increased by one. From a physical point of view, the increment depends on the size of the pointed type

Address Arithmetic: Example (1)

```c
int main() {
  char a_string[] = "This is a string";
  char *p;
  int count = 0;
  printf("The string: %s\n", a_string);
  for (p = &a_string[0]; *p != '\0'; p++)
    count++;
  printf("The string has %d chars.\n", count);
  p--;
  printf("Printing the reverse string: ");
  while (count > 0) {
    printf("%c", *p);
    p--;
    count--;
  }
  printf("\n");
}
```

Address Arithmetic: Example (2)

```
int main() {
  char a_string[] = "This is a string";
  char *p;
  int count = 0;
  printf("The string: %s\n", a_string);
  p = a_string;
  while (*p != '\0') {
    p++;
    count++;
  }
  printf("The string has %d characters.\n", count);
  printf("Printing the reverse string: ");
  p--;
  while (count > 0) {
    printf("%c", *p);
    p--;
    count--;
  }
  printf("\n");
}
```

Increasing a Pointer will Increase the Memory Address Depending on the Size of Type

```c
#include <stdio.h>
#include <stdlib.h>
char ch_arr[2] = {'A', 'B'};
char *ch_ptr;
float f_arr[2] = {1.1, 2.2};
float *f_ptr;
int main() {
  ch_ptr = &ch_arr[0];          /* same as ch_ptr = ch_arr */
  printf("%p\n", ch_ptr);   /* address of 1st elem */
  ch_ptr++;                 /* increase pointer   */
  printf("%p\n", ch_ptr);   /* address of 2nd elem */
  printf("%c\n", *ch_ptr);  /* content of 2nd elem */
  f_ptr = f_arr;            /* same as &f_arr[0]  */
  printf("%p\n", f_ptr);    /* address of 1st elem */
  f_ptr++;                  /* increase pointer   */
  printf("%p\n", f_ptr);    /* address of 2nd elem */
  printf("%f\n", *f_ptr);   /* content of 2nd elem */
}
```

Where to Study?

- Chapter 2
- Chapter 3: all, except 3.8
- Chapter 5 (some parts to be covered next week)