Basics of C

A Simple Program.

simple.c:

```
/* CPE 101                                               Alex Dekhtyar  */
/* A simple program                                                     */
/* This is a comment!
                                                        It ends here -> */

#include <stdio.h>                           /*       preprocessor directive */

#define KMPERMILE 1.6                        /* preprocessor directive       */

float convertMiles(int miles);               /* function declaration         */

int main() {                                 /*  main()  function: beginning */

   int miles;                                /* integer variable declaration */
   float km;                                 /* float  variable declaration  */

   printf("\n\nMiles2Kilometers'R'US!\n");        /* output                  */
   printf("\n");
   printf("How many miles? ");

   scanf("%d", &miles);                      /* input                        */

   km  = convertMiles(miles);                /* assignment statement         */

   printf("%d miles is %f kilometers\n", miles, km);    /* formatted output */

   return 0;                                 /* return statement             */

   }                                         /* end of main() function       */

float convertMiles(int miles) {              /* function definition          */
    return miles*KMPERMILE;                  /* expression, return statement */
}
```

Anatomy

C Program. A C program consists of a sequence of comments, preprocessor directives, declarations and definitions.

simple.c program contains comments, preprocessor directives and a function definition (but no declarations. More on declarations - later in the course).

Comments. Comments are ignored by the C compiler.

Comments must be enclosed in `/*` and `*/`.

A single comment can span multiple lines.

```
/* This is a comment in C */
/* This is
                        a comment too */
```

Preprocessor directives. C comes with a preprocessor, a program that runs prior to submitting the C program to the compiler. C preprocessor plays a number of functions, but the most important ones are:

- Inclusion of multiple (library) files into the C program being compiled.
- Declaration of constants.

Preprocessor directives are instructions in the C source file that are read (parsed) and executed by the C preprocessor. We concentrate on two preprocessor directives now, will learn more about others later in the course:

- Inclusion of external C library files is achieved using the `#include` directive.

  Example. The `#include` preprocessor directive from `simple.c` brings in the external library of input-output functions. Both printf and scanf (see below) are functions from the `stdio.h` library.
- Constants are declared using `#define` preprocessor directive.

```
#include <stdio.h>
#define KMPERMILES 1.6
```

main() function definition. In order to *compile into an executable file*, a C program **must** contain the definition of the main() function. In any C program, the first statement to be executed is the first statement of its main() function. (i.e, execution starts at the beginning of the main() function).

In ANSI C (and we are studying the ANSI C standard) the definition of main() is as follows:

```
int main() {
 <declarations>

 <statements>

 return 0;
}
```

Here, `<declarations>` is a sequence of variable declarations and `<statements>` is the sequence of steps (instructions) to be executed.

Variables. Variable is a named collection (sequence) of memory cells, which, at *different times* can contain different content (values). Variables have three key properties:

- Name. Variable name allows us to refer to the variable in the program (rather than referring to a specific memory address or addresses).

  It provides a *level of abstraction* between the code in a high-level langauge (like C) and the code in machine language.

- Type. The type of a variable is information about the class of values that can be stored in the variable. Type of a variable affects two things:

  1. the number of memory cells reserved for the variable, and
  2. the interpretation of the content of the variable.

- Content. The actual value(s) stored in the memory.

Variable Types. C has a wide range of types: from very simple to very complex. A table of some simple variable types is shown below.

C type	Explanation	Range	number of bytes
int	Integer numbers	-32767 ... 32767	2
float	Floating point decimals	$-3.4^{38} \dots 3.4^{38}$	6 (?)
double	Double precision floating point decimals	$-1.7^{308} \dots 1.7^{308}$	10 (?)
char	a single character	-128 ... 127	1
void	no type	none	0

Variable declarations. A variable declaration has the syntax:

```
<type> <VariableName1>, <VariableName2>,...<VariableNameN>;
```

Here, <type> is one of C types (or one of user-created types — more about it at the end of the course), and <VariableName1>, ...,<VariableNameN> are the names of the variable. A variable declaration can include one or more variable names. It must end with a semicolon (";").

Variable Names Reserved Words and Identifiers. Variable names, dubbed user-defined identifiers are formed as follows:

- An identifier must consist of only letters ('a'...'z', 'A',...'Z'), digits ('0',..., '9') and underscores ('_').
- An identifier **cannot** begin with a digit.
- An identifier cannot be a **C reserved word**, nor can it be a **C standard identifier**.

Reserved Words. Some words (identifiers) are used in C language and have special meaning. They cannot be used as variable names, and their occurrence in C programs is interpreted according to special rules of C syntax.

Some of the reserved words in C are:

type	reserved words
Type names	int, float, double, char, void
C control structures	if, while, for, return, case, switch, break, continue, goto, do
misc.	sizeof, typedef, union, struct

Standard Identifiers. Standard identifiers are the names of C functions from the standard C library. The complete list of these identifiers is found in **Appendix B** of your textbook.

Some standard identifiers you have already seen are `printf` and `scanf`.

Function declarations. A *function* is a **named** block of code that performs a certain computation and can be *reused* in a program.

Generally speaking, a `C` *function* is similar to a mathematical function. Functions in mathematics:

- have names (f, sin, log, etc. . .);
- take as input one or more parameters;
- perform a mathematical computation (e.g. $f(x, y) = x + y - 1$);
- return a value (e.g., $f(3, 4) = 3 + 4 - 1 = 6$).

In `C`, functions:

- have *names* that are unique within a program (e.g., `main`, `convertMiles`);
- take as input one or more parameters;
- perform computations as specified by the statements they consist of;
- return computed values using `return` statement.

Function declarations. A **function declaration** is a description of a function that contains the following information:

- function name;
- return type of the function;
- names and types of the function arguments/parameters.

The syntax is:

```
<Type> <FunctionName>( <parameterList>);
```

Here `<Type>` is the return type of the function, `<FunctionName>` is the name of the function and `<parameterList>` is a comma-separated list of function parameters. Each parameter is declared as a pair

```
<Type> <Name>
```

Function definitions. A **function definition** is a description of a function that contains the following information:

- function name;
- return type of the function;
- names and types of the function arguments/parameters;
- **function code**.

The syntax is:

```
<Type> <FunctionName>( <parameterList>) {
   <code>
}
```

Function declarations allow `C compiler` to recognize function calls in `C` code.

Function definitions actually provide the code for the function.

I/O functions. C does not have statements for input and output. But it does have two standard library functions which are used for input and output.

printf(). `printf()` is the output function.

`printf("string");` outputs the string. Strings must be enclosed in double quotes.

`printf("formatted string", <variable1>,...,<variableN>);` — formatted output. Format placeholders ("%d" for integer, "%f" for floating point, "%c" for character) are inserted into the the `"formatted string"`. When output is generated, they are replaced with the values of variables in the variable list, in the order of appearance.

Example.

```
printf{"%d miles is %f kilometers\n", miles, km);
```

Let's assume that `miles` has value 1 and `km` has value 1.6. What the line above will produce as output is:

```
1 miles is 1.600000 kilometers
```

scanf(). `scanf()` is the input function.

`scanf("format", &<variable1>,..., & <variableN>)` reads from the standard input stream the values of variables whose names are `<variable1>`,... `<variableN>`.

The format string consists of the format placeholders: "%d", "%lf" (for floating point numbers), "%c".

Example

```
scanf("%d", &miles);
```

reads an integer value from the standard input (keyboard, by default) into the variable `miles`.

Note the use of the ampersand (&) in front of the variable name. While the *real* meaning of this symbol will be discussed at the end of the course, for now, it suffices to say that the *presence of the & in front of the variable name, gives the `scanf` function the **permission/ability** to **change the value of the variable**.*

Note: Both `printf` and `scanf` functions are part of the standard C library of functions. Standard C library consists of a number of header files, each containing the code for a set of "like-minded" functions.

All standard input/output functions reside in the `stdio.h` header file. This is why, **any C program that contains input or output** must contain the

```
#include <stdio.h>
```

preprocessor directive.

Constants

Constants are values used throughout the program. In C, each constant must come with a type, and each type has its own syntax for constant values.

Via the preprocessor `#define` directive, we can *name* some of the constants and use their names, rather than values in computations.

Integer constants. Integer constants can be either signed or unsigned. Examples of signed integer constants are:

```
-1  +123  -643054 +23456
```

Examples of unsigned integer constants are:

```
0 1 999  34235  785
```

Note: Integer constants in C **may not** contain commas separating the orders. That is, `12,390` is **NOT** a correct integer constant.

Floating point constants. Floating point constants come in two different forms: the decimal point notation and the exponent notation and can be signed or unsigned. The decimal point constants have the form `IntegerPart.DecimalPart`. Examples of such constants are:

```
1.0  0.001  -12.12  3.1415926  +100000.00001  .434
```

The constants in the exponent notation have the form

```
<Integer>.<Decimal>e<Exponent>
```

or

```
<Integer>.<Decimal>e-<Exponent>
```

The value of such a constant is $\mathsf{Integer.Decimal} \times 10^{\mathsf{Exponent}}$ (or $\mathsf{Integer.Decimal} \times 10^{-\mathsf{Exponent}}$). The following are valid floating point constants:

```
2e-2  1.2e3  4.5e-5 6e+2
```

Their values are respectively:

$$2 \times 10^{-2} = 0.02; 1.2 \times 10^{3} = 1200; 4.5 \times 10^{-5} = 0.000045; 6 \times 10^{2} = 600.$$

The following are **invalid** floating point constants

```
20        /* no decimal point - it's an integer constant */
45e0.45   /* the exponent must be an integer             */
3.342e    /* no exponent                                 */
344,453e200 /* comma is not allowed                      */
```

Character constants. Character constants are individual (ASCII) characters in single quotes (’). Examples of character constants are

```
'a'  'b' 'C' 'D' 'T' '3' '&' '"' ' ' '_' '~'
```

The following are **NOT** valid character constants:

```
"a"        /* this is a one-character string, but not a character */
'ab'       /* only one character allowed inside single quotes     */
```

String constants. A string constant is any text inside double quotation marks (”). Examples are:

```
"a"  "Hello, world!" "Train station" "LOL" "1234"  "20,234" "/.?>$342?5^%%" "\n" "\t"
```

String constants can include special characters (escape sequences).

Symbol	meaning
`\n`	new line
`\r`	carriage return
`\t`	tab
`\b`	backspace
`\\`	\ (backslash)
`%%`	% (percent)
`\'`	’ (single quote)
`\"`	” (double quote)

(Arithmetic) Expressions. The three general forms of an expression are

1. `<Operand> <Operation>   (e.g., x++)`

2. `<Operation> <Operand>   (e.g., - 4)`

3. `<Operand> <Operation> <Operand>   (e.g., 5 + x)`

C has the following operations:

Operation	Arity	Meaning	Example
`+`	binary	addition	`5 + 6` evaluates to 11
`-`	binary	subtraction	`6 - 5` evaluates to 1
`*`	binary	multiplication	`4 * 3` evaluates to 12
`/`	binary	integer division	`4/5` evaluates to 0
`/`	binary	division	`4.0/5.0` evaluates to 0.8
`%`	binary	remainder	`10 % 7` evaluates to 3
`-`	unary	unary minus	`- 4` evaluates to -4
`+`	unary	unary plus	`+4` evaluates to 4

(additional operations will be introduced later).

`<Operand>` can be any of the following:

- constant value (`2, 5.6,` etc.)
- defined constant (`HUNDRED`)
- variable name (`currentStateVote`)

- function call (`log10(x)`)
- (arithmetic) expression in parentheses (`(log10(x) + 5)`)