Joanna Klukowska
joannakl@cs.nyu.edu

# Lecture 2: Introduction to C Programming Language

Notes include some materials provided by Andrew Case, Jinyang Li, Mohamed Zahran, and the textbooks.

**Reading materials** Chapters 1-6 in The C Programming Language, by B.W. Kernighan and Dennis M. Ritchie
Section 1.2 and Aside on page 4 in Computer Systems, A Programmer's Perspective by R.E. Bryant and D.R. O'Hallaron

## Contents

Joanna Klukowska
joannakl@cs.nyu.edu

# 1 Intro to C and Unix/Linux

Brian W. Kernighan

Dennis M. Ritchie

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

## 1.1 Why C?

Mainly because it produces code that runs nearly as fast as code written in assembly language, but is a high level programming language. Some examples of the use of C:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Language Interpreters
- Utilities

## 1.2 C vs. Java

**C**

- procedure oriented
- compiled to machine code: runs directly on a machine hardware

**Java**

- object oriented
- compiled to byte code: runs within another piece of software (JVM = Java Virtual Machine)

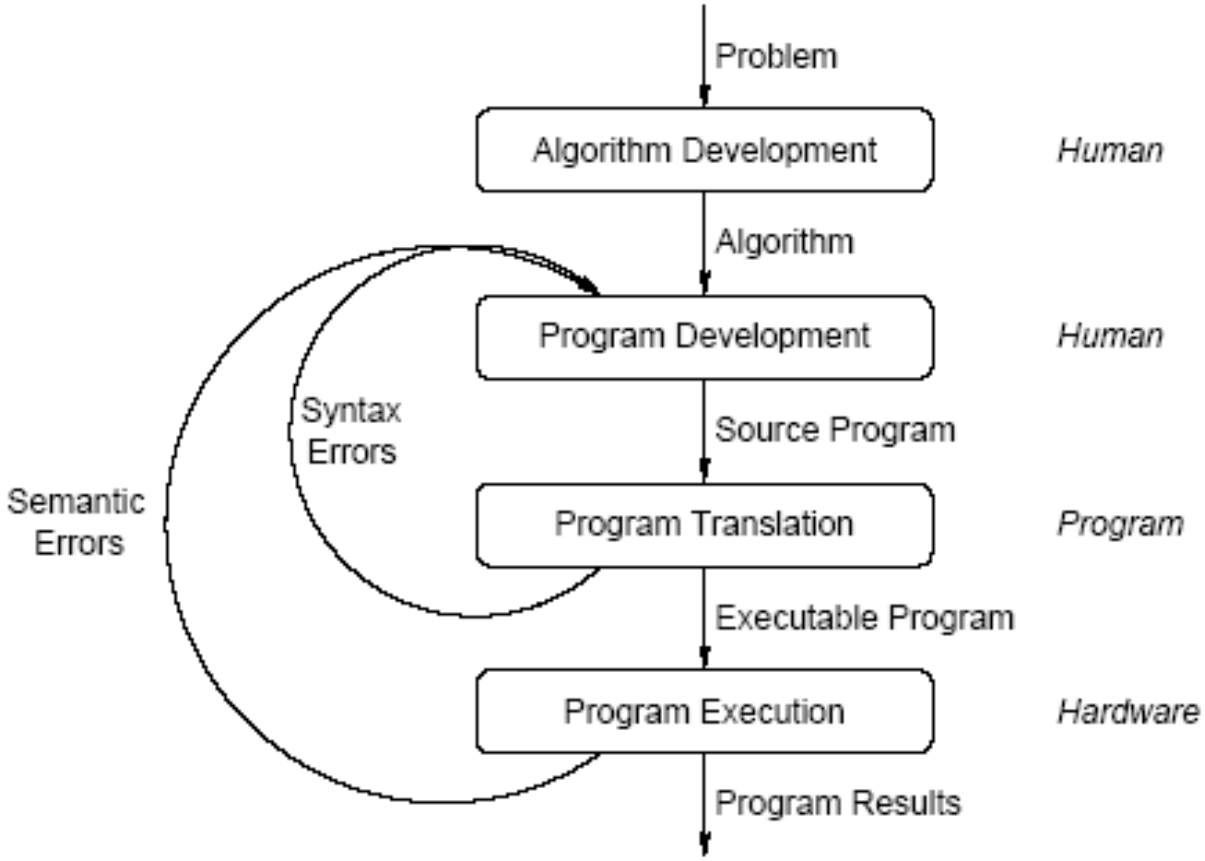
**About C**

- Hardware independent

Joanna Klukowska
joannakl@cs.nyu.edu

- Programs portable to most computers (in source code format, not executable format)
- Case-sensitive
- Four stages of program/code development:
  - Editing: Writing the source code by using some IDE or editor
  - Preprocessing or libraries: Already available routines
  - Compiling: translates or converts source to object code for a specific platform source code -¿ object code
  - Linking: resolves external references and produces the executable module

## 1.3 Code Development Process (Not Only in C)




## 1.4 Basic Unix Commands

In this class you will be working a lot in a command line or terminal environment.

Joanna Klukowska
joannakl@cs.nyu.edu

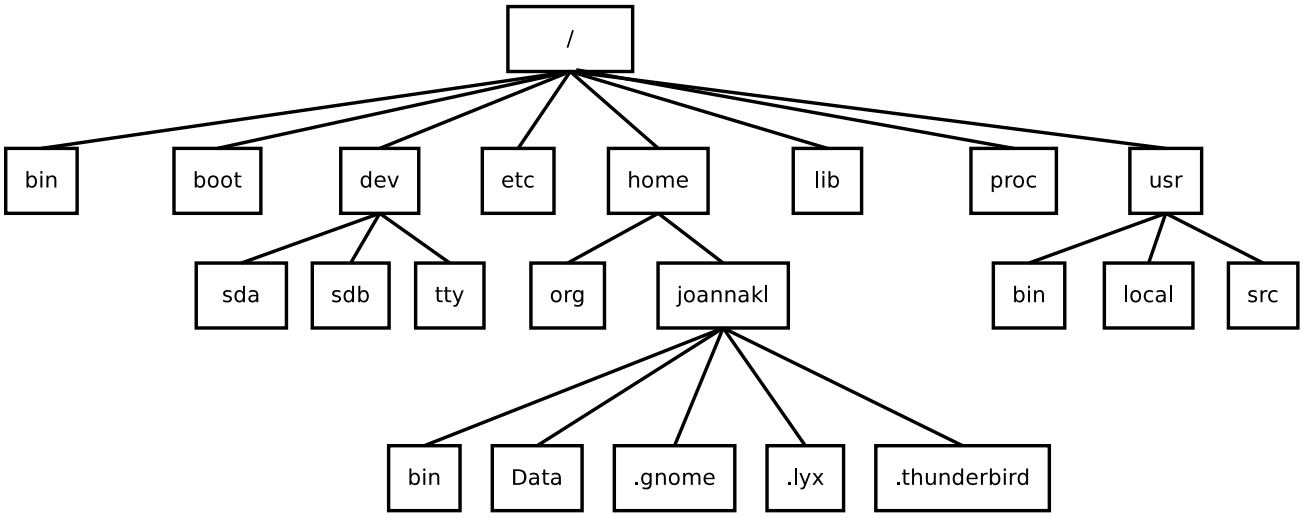
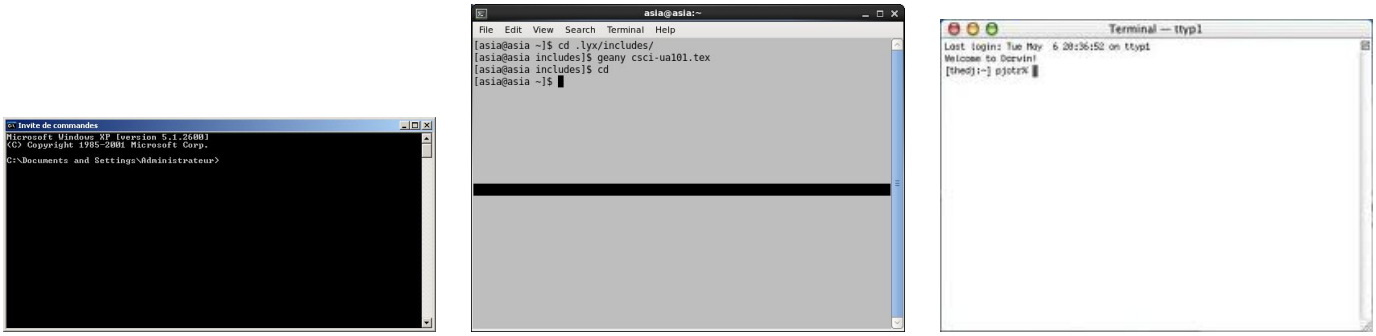



Figure 1: Partial directory structure on a typical Linux/Unix system.




When you log in to a computer system remotely, very often this is the only way of interacting with the computer.

**Finding your way around Unix/Linux**

Whenever you are logged into a Unix/Linux system, you have a unique, special directory called your **current** or **present working directory** (PWD for short). The present working directory is where you "are" in the file system at the moment, i.e., the directory that you feel like you are currently "in". This is more intuitive when you are working with the command line, but it carries over to the graphical user interface (GUI) as well.

Many commands operate on the PWD if you do not give them an argument. We say that the PWD is their default argument. (Defaults are fall-backs – what happens when you don't do something.) For example, when you enter the "`ls`" command (list files) without an argument, it displays the contents of your PWD. The dot "`.`" is the name of the PWD:

```
ls .
```

and

```
ls
```

both display the files in the PWD, first one because the name of the directory is provided, second one because it is the default behavior for `ls`.

When you first login, your present working directory is set to your **home directory**. Your home directory is a directory created for you by the system administrator. It is the top level of the part of the file system that belongs to you. You can create files and directories within your home directory, but usually nowhere else. Usually your home directory's name is the same as your username.

In Unix/Linux, files are referred to by **pathnames**. A pathname is like a generalization of the file's name. A pathname specifies the location of the file in the file system by its position in the hierarchy. In Unix/Linux, a forward slash "/" separates successive directory

Joanna Klukowska
joannakl@cs.nyu.edu

and file names in the pathname, as in:
`joannakl/Data/NYU_Teaching/website/csci201/daily.php`

There are two kinds of pathnames: absolute pathnames, and relative pathnames. The absolute pathname is the unique name for the file that starts at the root of the file system. It always starts with a forward slash:
`/home/joannakl/Data/NYU_Teaching/website/csci201/daily.php`

A relative pathname is a pathname to a file relative to the PWD. It never starts with a slash: if the PWD is
`/home/joannakl/Data/NYU_Teaching/`
then the relative path to the file `daily.php` is `website/csci201/daily.php`.

**Dot-dot** or **..** is a shorthand name for the parent directory. The parent directory is the one in which the directory is contained. It is the one that is one level up in the file hierarchy. If my PWD is currently `/home/joannakl/Data/NYU_Teaching`, then `..` is the directory `/home/joannakl/Data` and `../../` is the directory `/home/joannakl`.

**Unix/Linux commands to get you started**

**`man command`** display a manual page (or simply help) for the `command` (this is the easiest way to learn about options to the commands that you know and about new commands)

**`pwd`** print the name of the present working directory

**`ls`** list content of the current working directory

**`ls dir_name`** list content of the directory named `dir_name`

**`cd dir_name`** `cd` stands for change directory, changes the current working directory to `dir_name`

**`cd ..`** move one directory up in the directory tree

**`cd`** change the current working directory to your home directory

**`cp file1 file2`** copy `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names

**`mv file1 file2`** move `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names

**`rm file`** remove a file (there is no undoing it, so be very careful!)

**`mkdir path`** make a directory at the specified `path`

**`rmdir path`** remove the directory specified by the `path` (there is no undoing it, so be very careful!)

**`file file_name`** determine the type of a file

**`less file_name`** view the file in the terminal

**`more file_name`** view the file in the terminal

**`cat file_name(s)`** concatenate files and print them to standard output

# 2 First C Program and Stages of Compilation

## 2.1 Writing and running hello world in C

The usual first program in any language is the "Hello World" program. Here it is in C:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("hello world\n");
6  }
```

Joanna Klukowska
joannakl@cs.nyu.edu

How do we actually write, compile and run the program?

1. Write the code in your favorite text editor (make sure it is a **text** editor, not rich text or any other document format that allows text formatting). For example: Emacs, gedit, geany, vi, sublime, .... Save and name your file. The C source code files usually have .c extension. Their name can be anything (should be something that is related to what the program does).
2. Run the "compiler" to convert the program to executable code (or binary):

   ```
   gcc -Wall -g -o programName programSource.c
   ```

   **gcc** is the name of the compiler.

   **-Wall** is an option of `gcc` that tells it to generate and print all warnings (it's always good to know what they are).

   **-g** option tells `gcc` to generate debugging information (we will need that fairly soon).

   **-o programName** option tells `gcc` to save the executable code in the file named `programName`. If you do not provide this option, `gcc` will save your executable code in the file named `a.out` - this file gets overwritten without warning when the same program or a different program is compiled in the same directory.

   **programSource.c** is the name of the text file that contains the source code.
3. If any errors occur during step 2, go back to step 1, fix the errors and recompile. If any warnings occur during step 2, read through them and decide if they need fixing.
4. Run the program: this is done using the following syntax

   ```
   ./programName
   ```

   (Given your environment setup, you may be able to run the program by typing only the name of the program without `./` )
5. If errors occur during running of your program, go back to step 1 and fix them.

## 2.2 Hello world line by line

Let's look at the hello world program line by line. Here it is again (in a slightly different version):

```
1  #include <stdio.h>
2  /* A simple C program */
3  int main( int argc, char ** argv )
4  {
5      printf("hello world\n");
6      return 0;
7  }
```

**Line 1** `#include` inserts another file.
The `.h` files are called **header** files. They contain stuff needed to interface to libraries and code in other `.c` files. They contain lists of functions declarations (not definitions) - like a table of contents for an actual source code file that does not give away the details of the implementation.

**Line 2** this is a comment

**Line 3** The **`main()`** function is always where your program starts running. The parameters are optional. They are a way for the program to read the options/arguments provided on the command line.

**Lines 4, 7** The body of any function (**`main()`** is a function) is enclosed in curly braces.

**Line 5** **`printf()`** function prints the formatted text to the screen.

**Line 6** This returns 0 from this function. This line could be skipped, but proper C requires that **`main()`** returns an integer indicating termination status (0 for success, any other value for failure).

Joanna Klukowska
joannakl@cs.nyu.edu

## 2.3 What really happens when we compile a program?

The compilation occurs in three steps: preprocessing, actual compilation (convert to assembly and then binary) and then linking.

During **preprocessing**, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the preprocessor. Such lines are called **preprocessor directives**.

- Include files are "pasted in" (`#include`).
- Macros are "expanded" (`#define`)
- Comments are stripped out ( `/* */` , `//` )
- Continued lines are joined ( `\` )

We can run only this stage by providing ing the `-E` option to `gcc`. Assuming that our program above is saved as `hello.c`, running

```
gcc -E hello.c > hello.i
```

produces another text file called `hello.i`. This file contains over 800 lines. That's because the entire file `stdio.h` is "pasted" at the beginning. If we scroll all the way down, we find our own program at the bottom. It does not include the comments though.

During the **compilation**, the compiler reads the code and makes sure that it is syntactically correct. The compilation itself can be viewed as two steps: 1) create the assembly code and then 2) create the executable code. We can tell `gcc` to stop right after the first stage and look at the assembly code by using the following options:

```
gcc -S hello.c
```

By default, the output is written to a file with `.s` extension. `hello.s` file is fairly short. It contains the assembly instructions. It may look like this:

```
  .file  "hello.c"
  .section  .rodata
.LC0:
  .string  "hello world"
  .text
.globl main
  .type  main, @function
main:
.LFB0:
  .cfi_startproc
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movl  $.LC0, %edi
  call  puts
  movl  $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size  main, .-main
  .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-3)"
  .section  .note.GNU-stack,"",@progbits
```

(NOTE: In just a few weeks you will be able to read and understand the above code.)

Joanna Klukowska
joannakl@cs.nyu.edu

The second part of compilation converts the above assembly code into the binary code. To tell `gcc` to stop after that stage, we can use `-c` option. Running

```
gcc -c hello.c
```

produces `hello.o` file (this is called an object file). It is no longer a text file, so trying to look at it will produce gibberish (and many editors will refuse to display it). It contains the binary version of our `hello.c` program.

The final step is **linking**. At this point the binary code for our `hello.c` program exists, but it needs to be linked together with the code from `C` libraries (this is where `printf` lives, for example). Our original `gcc` command runs all of the steps above (including the linking) and produces a runnable/executable binary file.

# 3 Basics of C

If you know any other programming language you should be able to read simple C programs and understand what they do. In this section we will quickly review the basics and look at a few code examples.

## 3.1 Data types (primitive types)

The following table lists the data types in C together with their minimum size (in bytes) on a 32-bit and 64-bi systems.

| type | size (32bit) | size (64bit) | example |
|---|---|---|---|
| `char` | 1 | 1 | `char c = 'a';` |
| `short` | 2 | 2 | `short s = 175;` |
| `int` | 4 | 4 | `int i = 2147483647;` |
| long | 4 | 8 | long int l = 2147483647; |
| long long | 8 | 8 | long long int ll = 9223372036854775807; |
| float | 4 | 4 | float f = 1.0; |
| double | 8 | 8 | double d = 1.0; |
| pointer | 4 | 8 | int *x = NULL; |

Notice that there is no type to represent **Boolean data**. Instead, C uses int or char to represent true/false. The value of zero is always interpreted as false, any other value is interpreted as true. For example:

```
1  int i = 0;
2  if (i)   /* same as i!=0 */
3    printf("true");
4  else
5    printf("false");
```

The values of the Boolean expressions are always evaluated to either `0` or `1`.

## 3.2 Using `printf` to print different data types

The `printf` function provides way of printing values of variables of different data types. We need to know the format specifier that `printf` expects for each given type. The very basic `printf` calls using all of the types mentioned in the previous section are listed below:

Joanna Klukowska
joannakl@cs.nyu.edu

```
    char c = 'a';
    short s = 32767;
    int i = 2147483647;
    long int l = 2147483647;
    long long int ll = 9223372036854775806;
    float f = 1.0;
    double d = 1.0;
    int *iptr = NULL;
    char* string = "test string";

    printf("c: %c\n", c);
    printf("s: %i\n", s);
    printf("i: %i\n", i);
    printf("l: %li\n", l);
    printf("ll: %lli\n", ll);
    printf("f: %f\n", f);
    printf("d: %f\n", d);
    printf("iptr: %p\n", iptr);
    printf("string: %s\n", string);
```

See **types.c**.

## 3.3 Control flow

### Selection statements

The syntax of the `if`, `if ... else ...` and `switch` statements should be familiar from other programming languages. In C the expression in the `switch` statement has to have an integral value (`int`, `char`, or anything else that evaluates to an integer).

```
   if (...) {
       ...
   }
------------------------
   if (...)
       ...
   else if
       ...
   else
       ...
------------------------
   switch (c) {
       case 'a' :
           x = 'A';
           break;
       case 'b' :
           x = 'B';
           break;
       ...
       default:
           break;
  }
```

Joanna Klukowska
joannakl@cs.nyu.edu

**Repetition/loops**

C also has three different loops: `for` loop, `while` loop, and `do ... while` loop. One thing to remember about the **`for`** loop is that the control variable has to be declared before the loop (in some C standards).

```
  int i;
  for (i = 0; i < 10; i++) {
      print("i=%d\n",i);
  }
------------------------

  int i = 10;
  while (i > 0) {
      print("i=%d\n",i);
      i--;
  }
------------------------

  int i = 10
  do {
      print("i=%d\n",i);
      i--;
  } while (i > 0)
```

**Jump statements**

C provides break and continue statements. Both of them can be used with loops. break can also be used in a switch statement.

**`break`** causes execution flow to jump immediately to the next statement after the loop or after the switch statement

**`continue`** causes execution flow to jump immediately to the next iteration of the loop

C also has another way of "jumping" in the code. It is the **`goto`** statment. One can specify labels in the code as in the example below and the goto statement immediately moves to the line following the label.

```
LABEL_0:
  for (...) {
      for (...) {
          ...
          if (disater)
              goto LABEL_1;      //exit the loops
          else if (another_problem)
              goto LABEL_0;      //restart the loops
      }
  }
LABEL_1:
  ...
```

**Do not use `goto` statements in your code. They are considered to be bad programming style at this point and result in code that is hard to debug and trace.**

## 3.4 Functions

The file that contains `main` function can also contain other functions.

In C functions cannot be defined within other functions. A function has to be declared before it is used.

Joanna Klukowska
joannakl@cs.nyu.edu

```c
#include <stdio.h>
int modulo(int x, int divisor);   // Function declaration

int main()
{
    int x = 10;
    int divisor = 2;
    printf("%d mod %d = %d\n", x, divisor,
              modulo(x,divisor, 0 ));
    divisor = 3;
    printf("%d mod %d = %d\n", x, divisor,
              modulo(x,divisor, 1));
    return 0;
}

/*computes the remainder from dividing x by divisor */
int modulo(int x, int divisor, int rec)
{
  if ( !rec ) {
    /* iterative version */
    while (x >= divisor) {
        x -= divisor;
    }
    return x;
  }
  else {
    /* recursive version */
    if (x < divisor)
        return x;
    else
        return modulo(x - divisor, divisor, rec);
  }
}
```

We can also create functions in multiple files. If **functionA** uses **functionB** which is defined in another file, then one of the following has to be true:

- **functionB** is declared in a file that defines **functionA** (before **functionB** is used)
- the file that contains **functionA** includes a header file that declares **functionB** (we will look at header files soon) - this is the solution of choice

## 3.5 Variable scope

Variable Scope: global/external variables

**Scope rules:**

- The scope of a variable/function name is the part of the program within which the name can be used
- A global (external) variable or function's scope lasts from where it is declared to end of the file
- A local (automatic) variable's scope is within the function or block

```c
int x;

void foo(int y) {
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
  y++;
  x++;   /* x is accessible because it is in global scope*/
}

void bar() {
  y = 1; /* wrong - there is no y in this scope*/
  x = 1; /* x is accessible because it is in global scope*/
}
```

Demo (`scope.c`): What if we change all `y`'s to `x`'s?

```
void foo(int y) {
  if (y > 10) {
    int i;
    for (i = 0; i < y; i++) {
    ...
    }
  }
  else {
    y++;
    // 'i' is out of scope here
  }
}
```

**Global/Local Variable Initialization** In the absence of explicit initialization

- global and static variables are guaranteed to be initialized to zero
- local variables have undefined initial value **!!!**

Definition vs. declaration of a global (external) variable

- Declaration specifies the type of a variable.
- Definition also set aside storage for the variable and initializes its value.

The following example uses an external/global variable and a function both defined in a file different than `main` function.

```
add.c
---------------------
int counter = 0;

void add_one() {
  counter++;
}
```

```
program.c
---------------------
#include <stdio.h>

extern int counter;
void add_one();

int main() {
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    printf("counter is %d\n", counter);
    add_one();
    add_one();
    printf("counter is %d\n", counter);
    return 0;
  }
```

The two file program can be compiled using

```
gcc add.c program.c  -o program
```

or using separate compilation (each file compiled individually)

```
gcc -c -Wall -g add.c

gcc -c -Wall -g program.c

gcc add.o program.o -o program
```

WARNING: Avoid global/external variables (constants are fine)! They are considered bad programming style. Use of globals results in programs that are hard to debug and trace: just imagine how many different functions can possibly be modifying the value of a global variable.

**`static` keyword** The keyword `static` limits a scope of a global variable to within the rest of the source file in which it is declared. If we add the keyword `static` to the declaration of `counter` in `add.c` above, the `counter` variable will no longer be accessible from `program.c`.

The other use of keyword static allows the variable (global or local) to preserve its value between function calls. If the variable is declared with keyword static the initialization is performed only once.

```
add_2.c
--------------------
#include <stdio.h>

void add_one() {
    static int counter = 0;
    counter++;
    printf("counter is %d\n", counter);
}
```

```
program_2.c
--------------------
void add_one();

int main() {
    add_one();
    add_one();
    return 0;
}
```

## 3.6 Header files

Header files contain declarations of functions and globals that can be included in other source code files. The header file's name usually matches the name of the file that provides the source code (the definitions) and ends with `.h` extension.

We could rewrite the example from the previous section by adding a header file as follows:

Joanna Klukowska
joannakl@cs.nyu.edu

```
add.c
--------------------
int counter = 0;

void add_one() {
        counter++;
}
```

```
add.h
--------------------
extern int counter;
void add_one();
```

```
program.c
--------------------
#include <stdio.h>
#include "add.h"

int main() {
    printf("counter is %d\n", counter);
    add_one();
    add_one();
    printf("counter is %d\n", counter);
    return 0;
}
```

Notice that the file `program.c` no longer needs to repeat all the declarations, it simply includes the file `add.h`. The name of that file is included in **quotes** - this tells gcc to look for that file in a local directory, rather than in the locations of standard header files on your machine.

You should never use the **`#include`** directive to include one **`.c`** file inside another **`.c`** file. This *trick* may work to *solve/hide* some of your compilation/linking problems, but will lead to even greater issues with larger programs.