Languages for Informatics

2 – Getting Started with C Programming

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Progamming
  2. Variables, Data-types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. Structures
  6. Input and Output
- Basic system programming in Linux (10h)

Overview

Motivation for/against C

+

- Most common programming language before Java and Python (TIOBE 9/2020)
- C is a middle-level and procedural language, closing the gap between machine- and high-level languages.
- C works efficiently in embedded applications with very limited time and memory resources.

Motivation for/against C

+

- Most common programming language before Java and Python (TIOBE 9/2020)
- C is a middle-level and procedural language, closing the gap between machine- and high-level languages.
- C works efficiently in embedded applications with very limited time and memory resources.

-

- Limited **data abstraction** capabilities.
- Code has to be written carefully to maintain **portability** to other environments. Caution with data-types, byte ordering, size of pointers, etc.

History of C

- developed at Bell Labs by Dennis Ritchie (1941-2011) in 1972/1973, to reimplement the Kernel of UNIX.
- same syntax as B but, supports user-defined types, lets manipulate bits in memory, suitable for cross-platform programming.
- Initial standard was defined by Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 1978.
- Standards
  - ANSI-C by the American National Standards Institute in 1989 (=ISO C90). This is the most widely used and supported version.
  - C95: major improvement such as digraph support.
  - C99: several new library headers and data types, but still not support by all compilers.
  - C18 Is the current standard.

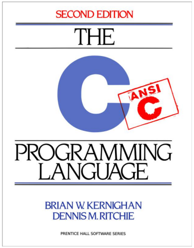
Typical Applications

Systems programming

- in **Operating Systems** (Linux, MAC OS)
- in **embedded microcontrollers**: Typical ’computer-on-achip applications are in consumer electronics products,instrumentation and process control, medical instruments, office equipment, multimedia applications, automobiles, etc....
- in **embedded (real-time) DSPs**: digital audio, TV, flight control in airplanes,

Reference book

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 2nd edition.



Getting Started

- **GNU Compiler Collection (GCC)** is a collection of compilers and libraries for C, C++, Objective-C, Fortran, Ada, Go, and D programming languages.
- Many open-source projects, including the GNU tools and the Linux kernel, are compiled with GCC.
- Installation instructions

```
$ sudo apt install build-essential
$ gcc --version
gcc (Ubuntu 9.2.1~17ubuntu1) 9.2.1 20191102
```

My first program

Use any text editor to create a file with `.c` extension

```
$ nano helloworld.c
```

```
#include <stdio.h>       /* C standard library */

int main()               /* mandatory function */
{
  printf("Hello world!\n");
  return 0;
}
```

```
CTRL O, CTRL X
```

Compile and Run

```
$ gcc -o helloworld helloworld.c
```

- Creates an executable called `helloworld`.

```
$ ls -l helloworld
-rwxrwxr-x 1 NyName MyGroup 8608 set 29 19:41
helloworld
```

- Run program with **`./helloworld`**.
- Here you go –

```
Hello World!
```

Structure of C Program

Pre-Processor directives

```
#include <stdio.h>
#define MYCONSTANT 0.1
```

Global Declarations

```
int count = 0;
int fun2(int a, int b);
```

Functions

```
int fun1(int a) { ... }
int fun2(int a,int b) { ... }
int main(void) { ... } /* obligatory */
```

From Source to Executable

Before it can be executed on a processor, the program needs to pass four stages of processing

1. **Preprocessing.** This first pass prepossess include-files, conditional compilation instructions and macros.
2. **Compilation** is the second pass. From output of the preprocessor + source code, an assembler source code `.s` is generated.
3. **Assembly.** In this third stage, an assembly listing with offsets is generated and stored in an object file `.o`.
4. **Linking.** One or more object files or libraries are used to produce a single executable by resolving references to external symbols and assigning final addresses to procedures/functions and variables. Code is relocated in memory.

Preprocessing

- happens before compilation
- It replaces symbolic information (text) in the source code with a content specified by the program using directives for the pre-processor
- Directives for the pre-processor are specified at the beginning of a C file and are identified by the character #
  - Inclusion of a file: `#include`
  - Macro: `#define`
  - Conditional compilation: `#ifdef...`

Don't be scared! It is just a complex Search and Replace

Include directive

`#include PATH_TO_FILE`

Instructs the pre-processor to insert the content of the file specified by `PATH_TO_FILE` in the C program at that particular line of code

Two ways to specify the file path:

- `#include <file>` - The file is looked-up in the C standard library path, e.g., `/usr/include` on Linux
- `#include "file"` - The file is looked up in the current directory

Example

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

Macro

```
#define NAME (<arg>) <expansion>
```

Replaces each occurrence of `NAME` with arguments `arg` with the text/function in `expansion`

Macro

```
#define NAME (<arg>) <expansion>
```

Replaces each occurrence of `NAME` with arguments `arg` with the text/function in `expansion`

Example 1: Defining a constant

```
# define MAX_INT 32767
```

Macro

```
#define NAME (<arg>) <expansion>
```

Replaces each occurrence of `NAME` with arguments `arg` with the text/function in `expansion`

Example 1: Defining a constant

```
# define MAX_INT 32767
```

It is even possible to specify parametric text

Example 2: Stringify a macro-expanded constant

```
# define BEER(z) "There are " str(z) " bottles
of beer on the shelf"
# define str(z) #z
```

Hence, **BEER(MAX_INT)** will be?

Macro (2)

```
#include <stdio.h>

#define BEER(z) "There are " str(z) " bottles of beer on
     the shelf"
#define str(z) #z
#define MAX_INT 32767

int main() {
  printf("%s \n",BEER(MAX_INT));
  return 0;
}
```

Shell

```
gcc -o mymacro mymacro.c
$ ./mymacro
There are 32767 bottles of beer on the shelf
```

Macro (3)

Macros can also contain **functions**.

Example

```
#define div(x,y) x/y
```

Macro (3)

Macros can also contain **functions**.

Example

```
#define div(x,y) x/y
```

Let us call this macro from the main-function by

```
int main() {
  printf("%.2f \n",div(2.0,3.0));
  return 0;
}
```

Macro (3)

Macros can also contain **functions**.

Example

```
#define div(x,y) x/y
```

Let us call this macro from the main-function by

```
int main() {
  printf("%.2f \n",div(2.0,3.0));
  return 0;
}
```

The result is

Shell

```
0.67
```

Conditional Compilation

```
#ifdef MACRO
     TEXT1
#elif
     TEXT2
#else
     TEXT3
#endif
```

Check whether `MACRO` is defined: if yes, it executes directives specified in `TEXT1`; otherwise, it runs the directives in `TEXT2`

For instance, it is useful to include a file only once (just the first time when this include directive is executed)

There exist other conditional directives: `#IF`, `#IFNDEF`,...

Conditional Compilation (2)

Typical example for architecture dependent files.

```
#ifdef _WIN32  /* 32/64 bit, _WIN64 for 64bit only */
     //do windows-specific stuff
#elif __linux__
     //do LINUX-specific stuff
#elif __APPLE__
     //do MAC-specific stuff
#else
     //do something else
#endif
```

Conditional Compilation (2)

Typical example for architecture dependent files.

```
#ifdef _WIN32  /* 32/64 bit, _WIN64 for 64bit only */
     //do windows-specific stuff
#elif __linux__
     //do LINUX-specific stuff
#elif __APPLE__
     //do MAC-specific stuff
#else
     //do something else
#endif
```

We have used **predefined macros.**

gcc Preprocessor (1)

You can check the result of the pre-processor, and convince yourself that is just a sophisticated *search and replace* tool.

helloworld

```
$ gcc -E helloworld.c
```

pre-processes `helloworld.c` and redirects the result to standard-out.
To store the result in a file,

shell

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

Note

`cpp helloworld.c helloworld.i` does the same.

gcc Preprocessor (2)

-> Live demonstration using ”helloworld.c”

gcc Preprocessor (2)

-> Live demonstration using ”helloworld.c”

The output is of the form

`# <linenum> <filename> <flags>`.

- These are called **linemarkers**, stating that the current line originated in file `filename` at line `linenum`.
- After the file name come zero or more flags.
  - ’1’ start of a new file.
  - ’2’ return to a file (after having included another file).
  - ’3’ text comes from a system header file, warnings should be suppressed (see Module 4).
  - ’4’ treated as being wrapped in an implicit `extern` ”C” block (see Module 4).

gcc Compiler

- The pre-processed file (**without** `#include` and `#define`) is transformed into a assembly code.
- The output is plain-text and (somewhat) human read-able source code comprising direct machine instructions.
- Can be used to optimize performance manually.

gcc Compiler

- The pre-processed file (**without** `#include` and `#define`) is transformed into a assembly code.
- The output is plain-text and (somewhat) human read-able source code comprising direct machine instructions.
- Can be used to optimize performance manually.
- C compiler executes correctness checks
  - Syntax: e.g., termination of each statement with ";", parenthesis balance, etc.
  - Coherence of data types: e.g., parameters of the functions, ...
  - Linear processing: a piece of code can only use variables and functions defined before

gcc Compiler (2)

Note

The gcc produces AT&T assembly syntax by default. Intel syntax can be produces, though, by option **-masm=intel**.

At this stage, the compiler generates an assembly code. For our helloworld-example, we get

shell

```
gcc -S helloworld.c
```

generating the file `helloworld.s`. It has the form

gcc Compiler (3)

```
.file   "helloworld.c"
        .section        .rodata    ; read-only data, pre-init. constants
.LC0:
        .string "Hello world!"     ; init string
        .text
        .globl  main               ; declare externally visible
        .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
        movl    $.LC0, %edi
        call    puts                ; put string
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret                         ; return from loop
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

gcc Assembler

- The end result of the first three stages is an **object code** that is understood by a computer at the lowest hardware level.
- The code is translated in corresponding machine language (i.e. binary)
- Extension is `.o`
- syntax is `gcc -c <source_file>` The source file can be the source code (.c) or the assembly code (.s).
- The underlying assembly code can be seen by the simple utility `objdump -dS <object_file>.o` (**D**isassemble, display **S**ource code intermixed with disassembly)

gcc Linker

The link creates an executable file from

- one or more object (**.o**) files,
- standard or self-made static libraries (**.a**) [Lesson 4], and
- dynamic libraries (**.so**) [Lesson 4].

Usage:

```
gcc <file>.o -o <exec>.out
```

runs the linker on the object file `file.o` and produces the executable `exec.out`.

All together

Command

```
gcc <file>.c -o <exec>.out
```

starts the GCC pre-processing, the compilation and the linking of code in `file.c` generating the executable `exec.out`.

All together

Command

```
gcc <file>.c -o <exec>.out
```

starts the GCC pre-processing, the compilation and the linking of code in `file.c` generating the executable `exec.out`.

```
gcc -Wall -pedantic <file>.c -o <exec>.out
```

- `-Wall -pedantic` options to increase the number of checks and displayed warning messages
- Use `gcc -v --help` to get info on the available options for GCC

Typical compilation errors

```
file.c: In function ’main’:
file.c:9: warning: implicit declaration of
function ’max’
/tmp/ccp8kHh0.o: In function ‘main’:
file.c:(.text+0x26): undefined reference to ‘max’
collect2: error: ld returned 1 exit status
```

Typical compilation errors

```
file.c: In function 'main':
file.c:9: warning: implicit declaration of
function 'max'
/tmp/ccp8kHh0.o: In function `main':
file.c:(.text+0x26): undefined reference to `max'
collect2: error: ld returned 1 exit status
```

(Compiler) max function is unknown: assuming it will be defined later

Typical compilation errors

```
file.c: In function 'main':
file.c:9: warning: implicit declaration of
function 'max'
/tmp/ccp8kHh0.o: In function `main':
file.c:(.text+0x26): undefined reference to `max'
collect2: error: ld returned 1 exit status
```

(Linker) I searched all possible objects' files, but I did not find max: error!

Finally, let us execute the program ...

Once all compilation errors are gone ...

Finally, let us execute the program ...

Once all compilation errors are gone ...

```
Segmentation fault (core dumped)
```

Finally, let us execute the program ...

Once all compilation errors are gone ...

```
Segmentation fault (core dumped)
```

Possible reasons:

- **Overflow** Numeric calculations not supported by type.
- **Divide** by Zero Dividing a numeric value by zero.
- **Invalid Shift Shifting** might lead to undefined result.
- **Memory Errors** by accessing an array outside its bounds or accessing heap-allocated memory after the memory has been freed.
- **Uninitialized Data Access** when data is used before the memory has been initialized, ...

Comment your code (I)

Programming nicely means also writing code that has useful comments and that is readable

```
//Single line comment

/* You can also have
   comments on more lines  */
```

ALWAYS comment your code (with useful explanations)!!!

Comment your code (II)

Why?

- Describe how to use your code
- Describe how the routine works
- Explain difficult passages in your code

Comment your code (II)

Why?

- Describe how to use your code
- Describe how the routine works
- Explain difficult passages in your code

For whom?

- Anybody that will modify your code....
- ....including you after weeks, months or years

Helloworld revisited

```
#include <stdio.h>  /* Standard C library for IO */

// main defines the starting point for our program.
// void -> no input parameters (in this case)
// int -> returns an integer */
int main(void) {

  /* Prints to standard output (screen) the string
   passed as argument */
  printf("Hello World!\n");

  /* Value returned from main to OS (0 -> OK) */
  return  0;
}
```

GNU Debugger gdb

GNU Project debugger,

- allows you to see what is going on ‘inside’ another program **while it executes** – or what another program was **doing before it crashed.**

GNU Debugger gdb

GNU Project debugger,

- allows you to see what is going on ‘inside’ another program **while it executes** – or what another program was **doing before it crashed.**
- GDB not only for C language but also supports Ada, Assembly, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, and Rust

GNU Debugger gdb

GNU Project debugger,

- allows you to see what is going on ‘inside’ another program **while it executes** – or what another program was **doing before it crashed.**
- GDB not only for C language but also supports Ada, Assembly, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, and Rust
- http://sourceware.org/gdb/onlinedocs/gdb - online manual

Adding debugging information

Let us compile our program one more time, but this time we add the −g option,

```
gcc -Wall -pedantic -g <file>.c -o <exec>.out
```

- The option −g adds built-in debugging support.

Adding debugging information

Let us compile our program one more time, but this time we add the −g option,

```
gcc -Wall -pedantic -g <file>.c -o <exec>.out
```

- The option −g adds built-in debugging support.

Example

```
gcc -Wall -pedantic -g myfile.c -o myfile
```

Launching gdb

To start up gdb, type `gdb` or `gdb myfile` in the shell. The resulting prompt looks like this:

```
(gdb)
```

If you started gdb without arguments, you need to load the program now.

```
(gdb) file myfile
```

In gdb-mode, the command **`file`** loads an executable file to execute under debugger control.

The Interactive Shell gdb

- To recall **history**, use the arrow up/down keys
- To **auto-complete** commands, use the TAB key
- To get more information on any command or on a specific, type

Hint

`(gdb) help [comamand]`

You can always ask GDB itself for information on its commands, using the command **`help`** (abbreviated **`h`**).

Running the program

To run the program in the debugger, type

```
(gdb) run <arg1> ...  <argN>
```

- If it is needed to supply any command-line arguments for the execution of the program, simply include them after the run command.
- If the program contains only logical errors, no error message will appear.
- If the program produces a core dump, you (should) get information on the line number in the source and parameters of the function that caused the error.

Core dump

Typical Core dump

```
Program received signal SIGSEGV, Segmentation
fault.
0x0000000000400545 in main () at myfile.c:10
10 temp[3]='F';
```

Strategy to investigate the cause of the crash:

- Set **breakpoints** in your code, to stop the program;
- Set **watchpoints** for a variable (in the current scope);
- Set **catchpoints** for system calls;
- Step through the code at a time, until you arrive upon the error.

Breakpoints (1)

Breakpoints can be used to stop your program at certain lines of code. If the program reaches this breakpoint, you can poke around in memory

Breakpoint in the current file

```
(gdb) break 9
Breakpoint 1 at 0x40053d:  file myfile.c, line
9.
```

When more files are loaded, you must specify a filename as well:

```
(gdb) break myfile.c:9
```

Breakpoints (2)

Suppose you have the function call **myfunc** in the program, determined by

```
int myfunc(int a, int b)
```

Then gdb can make a break point on that function by

```
(gdb) break myfunc
Breakpoint 2 at 0x4005f8:  file myfile.c, line
14.
```

Breakpoints (2)

Suppose you have the function call **myfunc** in the program, determined by

```
int myfunc(int a, int b)
```

Then gdb can make a break point on that function by

```
(gdb) break myfunc
Breakpoint 2 at 0x4005f8:  file myfile.c, line
14.
```

To break at a required condition in a particular thread and condition, you can use

```
(gdb) break thread THREADNUM if CONDITION
```

Parallel processing using threads we will tackle later on.

Breakpoints (3)

Example

```
(gdb) break if i==2
```

will only interrupt the program if `i` is equal `2`.

Breakpoints (3)

Example

```
(gdb) break if i==2
```

will only interrupt the program if `i` is equal `2`.

To get a **list of breakpoints**, use the command

```
(gdb) info breakpoints

Num    Type          Disp   Enb   Address     What
1      breakpoint    keep   y     0x40053d    in main at myfile.c:9
2      breakpoint    keep   y     0x4005f8    in myfunc at myfile.c:14
```

Breakpoints (3)

When not needed anymore, any breakpoints can be **disabled** by the number from above list of breakpoints.

```
(gdb) disable 1

Num   Type          Disp   Enb   Address     What
1     breakpoint    keep   n     0x40053d    in main at myfile.c:9
2     breakpoint    keep   y     0x4005f8    in myfunc at myfile.c:14
```

Breakpoints (3)

When not needed anymore, any breakpoints can be **disabled** by the number from above list of breakpoints.

```
(gdb) disable 1

Num    Type         Disp   Enb    Address     What
1      breakpoint   keep   n      0x40053d    in main at myfile.c:9
2      breakpoint   keep   y      0x4005f8    in myfunc at myfile.c:14
```

Breakpoints can also be **ignored** for a while to speed-up iterations inside a loop.

```
(gdb) ignore 1 5
```

The **`ignore`** takes two arguments: the breakpoint number to skip, and the number of times to skip it.

The debugging cycle (1)

- Now, try to run your program again. It will stop at the first breakpoint (or sooner due to a signal e.g. crash).
- To **proceed** to the next breakpoint, type

```
(gdb) continue
```

The debugging cycle (1)

- Now, try to run your program again. It will stop at the first breakpoint (or sooner due to a signal e.g. crash).
- To **proceed** to the next breakpoint, type

```
(gdb) continue
```

- To **step-in** a subroutine **n** single instruction (if there is line number information for the function), type

```
(gdb) step [n]
```

Skipping **n**, the default is **n=1**.

The debugging cycle (2)

To complete the current stack frame, which will normally complete the current subroutine and return to the caller, type

```
(gdb) finish
```

- The **next** command continues **n** source lines, and **steps-over** subroutines:

```
(gdb) next [n]
```

Skipping **n**, the default is **n=1**, as well.

The debugging cycle (2)

To complete the current stack frame, which will normally complete the current subroutine and return to the caller, type

```
(gdb) finish
```

- The **`next`** command continues **`n`** source lines, and **steps-over** subroutines:

```
(gdb) next [n]
```

Skipping **`n`**, the default is **`n=1`**, as well.

Watchpoints (1)

- So far, you have seen how to interrupt and continue the program flow at fixed, specified source lines.
- **Watchpoints**, in contrast, can be used to interrupt the program, when the value of a variable changes

```
(gdb) watch <variable>
```

- Whenever the value of `variable` is modified, gdb prints the old and the new values.
- Active watchpoints show up in the breakpoint list.

Note

The variable you want to watch must be in the current scope (i.e. accessible). Otherwise, the watchpoint will be deleted!

Watchpoints (2)

At any time you may print the current value of a variable in memory with

```
(gdb) print <variable>
```

Watchpoints (2)

At any time you may print the current value of a variable in memory with

```
(gdb) print <variable>
```

and to track the variable at each breakpoint by

```
(gdb) display <variable>
```

Watchpoints (2)

At any time you may print the current value of a variable in memory with

```
(gdb) print <variable>
```

and to track the variable at each breakpoint by

```
(gdb) display <variable>
```

Finally, we want to point out the possibility to assign a value to some variable **on the fly** with

```
(gdb) set $<variable>=<value>
```

Catchpoints

The third class of watchpoints, **catchpoints** can be used to stop the debugger at certain kinds of program events such as systemcalls. An entire module will be dedicated to system calls later on.

```
(gdb) catch syscall <name>
```

- If no argument is specified, calls to and returns from all system calls will be caught.
- You may also specify the system call numerically.

Example for checking the connection with clients

```
(gdb) catch syscall socket
(gdb) catch syscall 41
```

valgrind

What to do when the amount of available memory becomes less and less over time i.e., there is **memory leak**?

- The program incorrectly manages memory allocations in a way that memory is not released when it is no longer needed.
- To check whether your program has memory leaks, type

Valgrind

```
valgrind --tool-memcheck --leak-check=yes
./myexecutable
```

The **`valgrind`** core runs your program on a synthetic CPU.

Quiz

What is the output of the following program?

```
#include <stdio.h>

int main()
{
  printf("Hello World! %d \n", z);
  return 0;
}
```

1. Hello World! z;
2. Hello World! followed by some junk value
3. Compile time error
4. Hello World!