KUAS
KYOTO UNIVERSITY OF ADVANCED SCIENCE
京都発世界人財

京都先端科学大学

Information Processing 2　01

introduction to C programming

course information
installing the environment
compiling and running C programs

Ian Piumarta

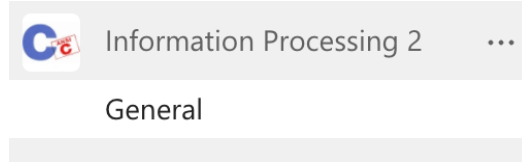
Faculty of Engineering, KUAS

course materials

distributed by MS Teams (please ensure you are subscribed)

in “General” channel, “Files” section:

- text book (PDF)
- reference books (PDF)
- class lecture slides (PDF)
- worksheets for weekly exercises and projects
- weekly class support material



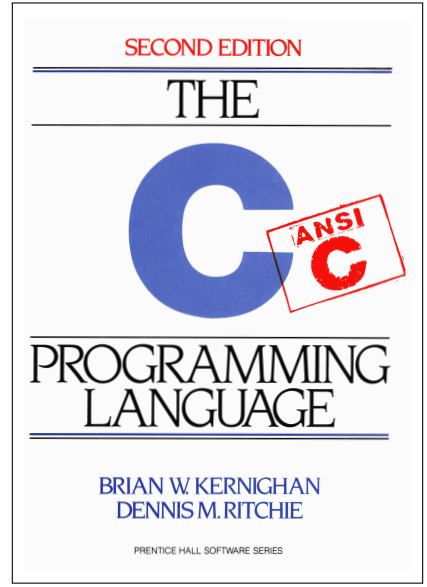
text book

Brian Kernighan and Dennis Ritchie,
The C Programming Language,
2nd edition, Prentice Hall, 1988.
ISBN 0-13-1110362-8

(free PDF in our Teams channel)

This is the definitive book on C, co-authored by the inventor of the language. If you already know the basics of programming then this is the only book you will ever need to read to learn C programming. It is short (190 pages) and dense. We will average about 13 pages of the book per week in approximately the same order. Try to read the relevant sections *before* class.

Our library has physical copies. English and Japanese versions are available on Amazon.



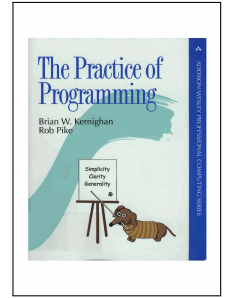
reference books

Brian Kernighan and Rob Pike,

The Practice of Programming,

(free PDF in our Teams channel)

How not to ‘screw up’ when programming: assessing trade-offs, designing programs, debugging them, improving them. Ideas not specific to a particular language. Examples presented using C, Java, and C++.

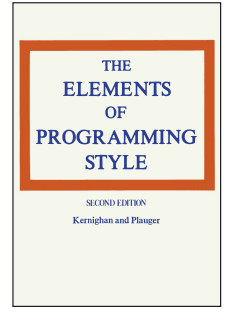


Brian Kernighan and Phillip Plauger,

The Elements of Programming Style,

(free PDF in our Teams channel)

How experts think and plan when writing code. Examples in FORTRAN but the ideas and philosophy are completely language-independent.



outline of the course (approximately)

1. intro and ‘hello world’
2. character I/O and arrays
3. functions, line input
4. types and operators
5. control flow
6. program structure
7. multiple source files
8. memory, arrays, pointers
9. arrays of pointers; pointer arithmetic
10. number representations and binary
11. structures and bit fields
12. recursive structures
13. standard I/O and file access
14. system calls and the file system
15. intro to C++ (just the useful bits)

grading

similar to IP1

the University insists you obtain the same score for lecture and exercises, therefore:

this class (lecture): instead of an exam, so treat these tests as seriously as an exam!

- 50% = weekly tests (weeks 3, 6, 9, 12, 15)

exercises class (next period):

- 25% = exercises (finished in-class, in principle)
- 25% = projects (can work on these between classes)

outline of today's class

why C?
history of C
interpreters vs. compilers

running the compiler
importance and reading of error messages
importance of writing lots of programs

`hello.c`, compiling, running what each line in `hello.c` means
`printf` and formatted output

integer and floating variables
F to C temperature table using a `while` loop

(installing the compiler, if you have not already)

exercises

history of C

once upon a time, operating systems were written entirely in machine code

- hard to maintain
- harder to port to new hardware
- slow to evolve

1969: Ken Thompson invents UNIX OS

- written entirely in machine language
- new computers (new machine code) being developed rapidly
- porting was painful

idea: invent a portable language for OS development?

LSI-11 Lookup Code

```
'              ************************************
'              * TABLE LOOK-UP (BINARY ALGORITHM) *
'              ************************************
'
'     TABLES SHOULD CONTAIN 4 BYTES PER ENTRY STARTING ON A 4-BYTE BOUNDARY
'     WITH THE ENTRIES ARRANGED IN INCREASING NUMERICAL ORDER.
'
'     REGISTERS DESTROYED:  R0,R1,R2
'
'     ENTRY (LOOKUP):       R1    BASE ADDRESS OF TABLE
'                           R2    ADDRESS OF LAST ENTRY IN TABLE
'                           R3    ADDRESS OF ITEM
'
'     EXIT:                 PSW   > 0  ITEM WAS NOT IN TABLE
'                                 = 0  R0 CONTAINS ADDRESS OF MATCHING ENTRY
'
LOOKUP MOV    R1,R0         'LOW
       ADD    R2,R0         'LOW + HIGH
       ASR    R0            '(LOW + HIGH) / 2
       BIC    #^H3,R0       'TRUNCATE TO 4-BYTE BOUNDARY
       CMPB   (R0),(R3)     'FIRST BYTES MATCH?
       BNE    1F            'NO
       CMPB   1(R0),1(R3)   'SECOND?
       BNE    1F            'NO
       CMPB   2(R0),2(R3)   'THIRD
       BNE    1F            'NO
       CMPB   3(R0),3(R3)   'FOURTH?
       BEQ    0F            'YES
1F     BHI    1F            'ITEM IS BELOW TEST POINT
       CMP    (R0)+,(R0)+   'BUMP OFFSET BY ONE ENTRY
       MOV    R0,R1         'MOVE "LOW" UP
       BR     2F
1F     CMP    -(R0),-(R0)   'DROP OFFSET
       MOV    R0,R2         'MOVE "HIGH" DOWN
2F     CMP    R1,R2         'DONE?
       BLOS   LOOKUP        'NO
0F     RTS    PC            'RETURN
```

Figure 3

history of C

PDP-11 machine code

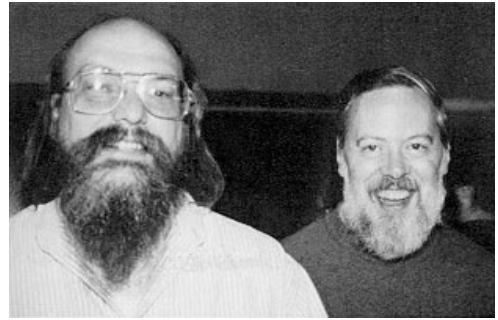
```
        .data
a:      .word
        .text
        mov   #0, a
loop:   cmp   a, #10
        bge   done
        add   #1, a
        jmp   loop
done:
```

equivalent C program

```
int a;

a = 0;
while (a < 10)
{
    a += 1;
}
```

↓ Ken Thompson



Dennis Ritchie ↑

C (invented by Dennis Ritchie in 1972) was used to re-implement UNIX portably

- only a very small amount of non-portable machine code remained
- Linux, Darwin (macOS), etc., are modern implementations of UNIX written in C
- Windows NT (7, 8, 10, 11) is heavily influenced by UNIX and written in C

why learn a 50-year old programming language?

C is easily the most widely-known programming language

most Internet servers run a software ‘stack’ written entirely in C

- ‘LAMP’ stack: Linux/FreeBSD OS, Apache web server, MySQL database, PHP

Android (smart ’phone OS) is written in C

most ‘system’ software is written in C

- OS, drivers, compilers, interpreters, 3D graphics toolkits, game engines, etc.

70% of embedded devices programmed in C (maybe with a little C++ added)

- mechatronic control systems, real-time applications, embedded OS, etc.

C is everywhere (especially if you have no interest in ‘apps’)

what is C like?

very small: if you try really hard you can learn it in **one day**

‘general-purpose’, ‘low-level’ programming language

- data types reflect the capabilities of the underlying computer hardware
- strings, lists, dictionaries, etc., **do not exist** in C (you have to make your own)

operations on data closely resemble machine code instructions

- C does not hide any of the capabilities (or complexity) of the machine from you
- you have immense power and control, once you can use C properly

no memory management

- you must explicitly allocate and deallocate memory for your data, yourself

no input/output facilities

- library functions connect your program directly to OS services and HW devices

the importance of writing lots of programs

no matter how many books you read, you will never learn how to

- play the guitar
- ride a bicycle
- paint pictures
- speak English
- write C programs

you have to *practice* these things to become good at them

programming is 10% knowledge, 20% instinct/intuition, 30% judgement, and 40% art

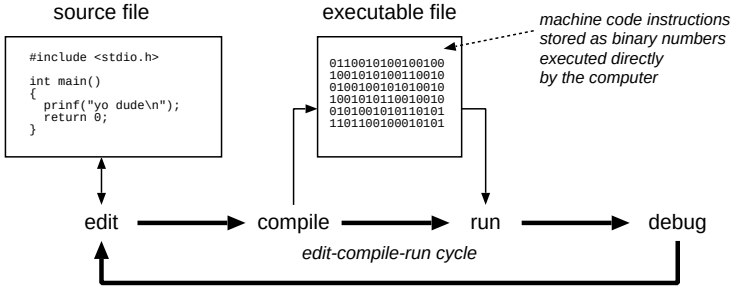
the *knowledge* that you have to acquire is presented in this course
the practical *experience* you need, to acquire all the others, is your responsibility

great programmers practice programming every day

$\Rightarrow$ find a way to use programming in one of your hobbies or interests

interpreters (e.g., Python) vs. compilers (e.g., C)

you have to *compile* your (text) program into an executable (binary) file



advantage: speed, plus some kinds of error are detected at compile time

- undefined variables, missing return values, wrong number of arguments, ...

disadvantage: development is non-interactive

- cannot modify, or otherwise repair, a program while running/debugging it

everyone's first C program: hello world

open a terminal, `cd` to a convenient directory, and create a file called `hello.c`

enter the following program, exactly as shown (including spaces, newlines)

```
#include <stdio.h>
int main()
{
  printf("hello, world\n");
  return 0;
}
```

when ready, quit the editor and in the same terminal run these commands:

1. `gcc hello.c` ← compiles `hello.c` into `a.out` (Unix) or `a.exe` (Windows)
2. `./a.out` ← everywhere except, of course, on...
   `a.exe` ← Windows

everyone's first C program: hello world

```
   #include <stdio.h>
→  int main()
   {
     printf("hello, world\n");
     return 0;
   }
```

C programs consist of variables and functions
variables store values used by the program
functions contain the statements of the program

functions can have (almost) any name, but the name `main` is special

- the program begins execution by the OS calling the `main` function

everyone's first C program: hello world

```
    #include <stdio.h>
    int main()
→   {
      printf("hello, world\n"); ←
      return 0; ←
→   }
```

the statements of a function are enclosed in braces '{ ... }'

each statement is *terminated* with a semicolon ';'

our `main` function contains two statements, each on its own line, however...

newlines and spaces have almost no significance in C

indentation is ignored: the compiler knows where `main` ends because of the closing '}'

everyone's first C program: hello world

```
→ #include <stdio.h>
  int main()
  {
→   printf("hello, world\n");
    return 0;
  }
```

our `main` function uses a library function called `printf`

- `printf` prints things on the terminal
- it is part of the 'standard I/O library'

the first line of our program, `#include <stdio.h>`

- tells the C compiler to read the contents of the file `stdio.h`

the `stdio.h` file contains information about the standard I/O library

- including how to call the `printf` function
  (without that information, `printf` cannot be used safely)

everyone's first C program: hello world

```
    #include <stdio.h>
    int main()
    {
→     printf("hello, world\n");
      return 0;
    }
```

a function is called by writing its name then zero or more *arguments* in parentheses

our program calls `printf` with one argument, the *string* `"hello, world\n"`

the two-character sequence `\n` represents a single *newline* character

putting a literal newline inside the string does not work; try this if you like:

```
printf("hello, world
")
```

unlike Python's `print`, C's `printf` only prints a newline if you ask for one explicitly

everyone's first C program: hello world

```
   #include <stdio.h>
→  int main()
   {
     printf("hello, world\n");
→    return 0;
   }
```

every function has to indicate what kind of value it returns as its result

conventionally, the `main` function returns an integer result

- the `int` before `main` says the result from calling `main` is an integer

the result returned from `main` tells the OS whether the program succeeded or failed

the second statement in the function body is `return 0;`

- returning `0` means 'success'
- returning any other value means 'failure'

compiler option: choosing an output filename

almost nobody uses the default name `a.exe` or `a.out`

the C compiler has a command line *option* to change it, like this: '`-o` *filename*'

- the compiled program will be called *filename* (instead of `a.out`)

assuming your program is called `hello.c`, try compiling like this:

```
gcc hello.c -o hello      ← tell compiler the name of the output file
```

run the program like this:

```
./hello          ← every shell on Earth except, of course...
hello            ← Windows cmd shell
```

choose any name you like for your compiled program (except the input filename!)

(why '`./`' ???)

the importance of reading error messages

the compiler prints *error* and *warning* messages

- a warning will not stop the compiler from producing an output file
- an error will prevent the compiler from producing an output file

in *both cases* you should **read** the message *very carefully* and then **fix** the problem

here is an example error message from `gcc`

```
      line  column
     number  number    error message
          ↓ ↓         ↓
program.c:5:27: error: expected ';' after expression
  printf("hello, world\n")    ← problematic line of source code
                          ^   ← arrow showing where the error occurred
                          ;   ← possible hint(s) about how to fix it
```

English translations of some common error messages

```
program.c:3:3: warning: implicitly declaring library function 'printf'
```

- you forgot to `#include` the file declaring `printf`
  - in this case, the file you forgot to include was `stdio.h`
  - the `man` (manual) command says which include file describes a function

```
program.c:3:1: warning: type specifier missing, defaults to 'int'
```

- you forgot to indicate the type of a variable or function result

```
program.c:5:27: error: expected ';' after expression
```

- you forgot to terminate a statement with a semicolon

```
program.c:5:3: warning: implicit declaration of function 'print' is invalid
```

- you are trying to call a function that you did not declare
  - maybe because you misspelled a library function name?

```
program.c:6:1: warning: control reaches end of non-void function
```

- you forgot to `return` a value at the end of a function

beware of secondary errors

one error in a program can cause lots of error messages to be printed

- the first error message indicates the real problem
- the rest are *secondary errors*, all **caused by the first error**

```
f()                    ← I forgot to specify that this function returns no result
{
}
```

which causes a relevant warning message

```
program.c:1:1: warning: type specifier missing, defaults to 'int'
f()
^
```

and a secondary warning message for a problem that really *does not exist*

```
program.c:3:1: warning: control reaches end of non-void function
}
^
```

some ‘pro tips’

turn on all warning messages

```
gcc -Wall sourcefile.c -o outputfile
```

enable debugging information (we will talk about debugging later)

```
gcc -Wall -g sourcefile.c -o outputfile
```

when the program is finished, and works perfectly, *optimise* it to run faster

```
gcc -Wall -O sourcefile.c -o outputfile
```

or use

- `-O3` to optimise even more
- `-Os` to optimise for space (smallest code size) instead of speed

`printf` and formatted output

`printf` stands for '**print f**ormatted'

the first argument to `printf` is a string that is printed, except that

- if a '%' appears in the string, it will be replaced by something else
- usually that 'something else' is a printed representation of the next argument

for example, '%d' means 'print the next argument, which must be an integer, in decimal'

since reading about this is only 10% effective, let's practice...

your second C program: temperature chart

```c
#include <stdio.h>

int main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower =   0;   // lower limit of temperature
    upper = 300;   // upper limit of temperature
    step  =  20;   // step between temperatures

    fahr = lower;
    while (fahr <= upper) {
        celsius = (fahr - 32) * 5 / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }

    return 0;
}
```

include information about `printf`

function has no parameters, returns integer result
open brace { marks the start of the function body
create five `int` *variables (to hold integers)*

'//' indicates that the rest of the line is a comment

set up initial conditions for the loop
***test** whether the loop should continue*
{ and } braces delimit the body of the loop
print two decimal integers separated by a tab ('\t')
update the variable that controls the loop
*end of the loop: go back to the 'while' **test***

exit the program with status zero ⇒ 'no error'
close brace } marks the end of the function body

`printf` and formatted output

the function call

```
printf("%d\t%d\n", fahr, celsius);
```

passes three arguments to `printf`

the first is a *format string* to be printed, in which

- each '%' prints the next argument at that position in the output

breaking the format string apart:

`%d`	prints the next argument, which is `fahr`
`\t`	is a *tab* character (moves the output forward to a multiple of 8 columns)
`%d`	prints the next argument, which is `celsius`
`\n`	is a newline character (moves the output to the start of the next line)

integer and floating variables

variables must be *declared* before they are used

a *declaration* specifies the properties of a variable

```
int fahr, celsius;
```
← declares that these two variables contain integers (only)

```
float f, g;
```
← declares variables containing floating point numbers (only)

the range of values that `int` or `float` can store *depends on the machine*

- ranges can differ for the exact same program compiled on two different computers

other integer and floating types are provided; a typical modern 64-bit machine has:

type	*size*	*range*
`char`	one byte character	-128 to 127, or 0 to 255
`short`	two byte short integer	-32,768 to 32,767
`int`	four byte normal integer	-2,147,483,648 to 2,147,483,647
`long`	eight byte integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	four byte floating point	$\pm$ 3.4$\times 10^{-38}$ to $\pm$ 3.4$\times 10^{+38}$
`double`	eight byte floating point	$\pm$ 1.7$\times 10^{-308}$ to $\pm$ 1.7$\times 10^{+308}$

more formatting options

in a format string, the *conversion specifier* '%d' prints an integer in decimal

```
printf("%d\n", 6*7);                    ⇒ 42
```

to print a floating-point number, use '%f'

```
printf("%f\n", 4.0/3.0);                ⇒ 1.333333
```

a conversion specifier can also specify a *field width* and *precision*

```
printf("%4.1f\n", 4.0/3.0);             ⇒ ␣1.3
```

- field width 4 is the number of characters set aside for the number
- if the field width is negative the number is left-justified, otherwise right-justified
- precision .1 is the number of fractional digits to be printed
- the '␣' represents the (invisible) space character needed to fill 4 characters

exercises

please download assignment from
MS Team “Information Processing 2”, “General” channel

installing the compiler

run "`gcc -v`" in a terminal to check whether you have the compiler

Linux

- you probably have everything you need installed by default
- if not, try this (Debian or Ubuntu): `sudo apt install build-essential`

Mac

- open a Terminal then run this command: `xcode-select --install`
- if it asks to install "command line tools", click the "Install" button

Windows

- use Windows Subsystem for Linux (WSL2) ← *highly recommended!!*
  - see `https://learn.microsoft.com/en-us/windows/wsl/install`
  - reboot (if needed) and after choosing a Linux username and password, run:
    `sudo apt install build-essential`
- **or** use the Chocolatey package manager to install a Windows C compiler
  - see next page →

installing a compiler on Windows using Chocolatey

install Chocolatey package manager

1. in search box, type “cmd” to search for ‘command prompt’
2. right-click and choose “Run as administrator” (and confirm it in the pop-up)
3. open a web browser here: `https://community.chocolatey.org/courses/installation/installing`
4. set “Choose installation method” to “Basic Chocolatey Install”
5. scroll down to “Install with cmd.exe”, copy the code, paste it into your `cmd` window, press return

close the `cmd` window

run PowerShell as administrator

- `choco install unzip mingw make -y`

optional: install a better editor than Notepad or Notepad++

- `choco install nano-win -y` (as used in Information Literacy)

(to check which packages are installed, you can run: `choco list --local`)

installing a shell and terminal on Windows using Chocolatey

optional: install a standard shell, command-line utilities, and terminal

- run: `choco install git.install microsoft-windows-terminal -y`
- in the search box, type “terminal” and run the Terminal app
- under Settings, select “+ Add a new profile” and fill in:
  - Name: `bash`
  - Command line: `C:\Program Files\Git\bin\bash.exe`
  - Starting directory: `C:\Users\`*your-username*
- click “Save”
- under Startup, set:
  - Default profile: `bash`
- click “Save”

close and re-open the Terminal app and you should now be using the bash shell

(‘serious’ text editors such as Emacs, remote access with `ssh`, etc., will all work perfectly in this terminal and shell)

exercises

using `printf` to print messages

changing `int` variables to `float`

using `printf` to print numbers

different conversion specifiers in `printf`

using field width and precision to align columns

12:40	15	intro
12:55	5	outline, grading
13:00	5	history
13:05	5	why learn C?
13:10	5	what is C like?
13:15	5	compilers vs. interpreters
13:20	5	installing the compiler
13:25	20	installing via Chocolatey
13:45	5	the importance of writing programs
13:50	5	hello world programming
13:55	10	hello world explaining
14:05	5	-o option
14:10	10	*break*
14:20	5	reading error messages
14:25	5	English translations of some error messages
14:30	5	secondary errors
14:35	5	pro tips
14:40	5	printf and formatted output
14:45	10	temperature programming
14:55	5	formatted output
15:00	5	integer and floating variables
15:05	5	more formatting options
15:10	40	exercises
15:50	00	end