Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Outline

C vs. Java

Founding principles of C programming:

1. Trust the programmer.
2. Don't prevent the programmer from doing what needs to be done.
3. Keep the language small and simple.
4. Make it fast, even at risk of portability.

Efficiency is favoured over abstraction
(no objects or fancy stuff)

- C is the standard language for: device drivers, kernel. Widely used in embedded systems (all contexts where high efficiency is a must)

How to write a C program

1. verify the presence of the C compiler `gcc` by

   `gcc -v`

   If not installed, then

   `sudo apt-get install gcc`

2. Edit a program by a text editor (nano, vim, emacs, gedit on GNOME, kate on KDE, ...)
   - you should know what the editor writes into the saved file
   - sophisticated development environment "helps" you to write the code. Sometime they take decisions for you and you don't know about it
3. Compile the program by `gcc <filename>`
   - If no compilation error, execute the program
   - If errors, try to understand the errors, fix them and recompile

My first C program

1. Create and edit the following program
   hello.c
2. Compile it by `gcc hello.c`

- By default the executable is `a.out`
- Launch it by `./a.out` (why not just `a.out`?)
- Usually we want the executable to have a name similar to the program. We do it by the "-o" option
  `gcc hello.c -o hello`

Basic structure of a C program

1. Pre-processor directives (`#include <stdio.h>`)
   - `#include ...` is used to add libraries
   - in the example `#include <stdio.h>` is needed to use the function `printf()`
2. Declaration of types (not in `hello.c`)
3. Declaration of global variables (not in `hello.c`)
4. Declaration of functions (not present in `hello.c`)
5. `main` function: the first function invoked at execution
6. Declaration of local variable `v` (array of characters)
7. Body of function `main`

Coding style

- C is powerful. C programs must be clean and understandable
- It is highly recommended to adopt a coding style
- It is suggested: the Linux kernel coding style
  (may be useful if one day you'll write kernel code)
  `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`
- In short:
  - indentation made with TAB (8 characters long).
    - ⋆ TAB is one byte only (ASCII character number 9). Not 8 spaces (8 bytes!!). C programmers like to be efficient and not to waste bytes, energy, ...
  - no new line before "{" (unless first brace of a function)
  - new line after "}", unless there is a continuation of the previous statement as in "`} else {`"
  - do your best to stay in 80 columns
- Check example at the website

The opposite of coding style: Obfuscated C Code

- some guys enjoy write "obfuscated code"

```
      #define/*__Int3rn^ti[]n/l_()I3fusc^t3|]_C_C<>I7E_C[]nt3st__*/L/*__MMXVIII__*/for
   #include/*!"'()*+,-./12357:;<=>?CEFGHIJKLMNSTUVWXYZ[]^_`cfhijklmnrstuvwxyz{|}*/<stdio.h>
  char*r,F[1<<21]="~T/}3(|+G{>/zUhy;Jx+5wG<v>>u55t.?sIZrC]n.;m+:l+Hk]WjNJi/Sh+2f1>c2H`)(_2(^L\
 -]=([1/Z<2Y7/X12W:.VFFU1,T77S+;N?;M/>L..K1+JCCI<<H:(G*5F--E11C=5?.(>+(=3)Z-;*(:*.Y/5(-=)2*-U,\
/+-?5'(,+++***''EE>T,215IEUF:N`2`:?GK;+^`+?>)5?>U>_)5GxG).2K.2};}_235(]:5,S7E1(vTSS,-SSTvU(<-HG\
-2E2/2L2/EE->E:?EE,2XMMMM1Hy`)5rHK;+.T+?[n2/_2{LKN2/_|cK2+.2`;}:?{KL57?|cK:2{NrHKtMMMK2nrH;rH[n"
"CkM_E21-E,-1->E(_:mSE/LhLE/mm:2Ul;2M>,2KW-+.-u).5Lm?fM`2`2nZXjj?[n<YcK?2}yC}H[^7N7LX^7N7UN</:-\
ZWXI<^I2K?>T+?KH~-?f<;G_x2;;2XT7LXIuuVF2X(G(GVV-:-:KjJ]HKLyN7UjJ3.WXjNI2KN<l|cKt2~[IsHfI2w{[<VV"
"GIfZG>x#&#&&$#$;ZXIc###$&$$#>7[LMv{&&&&#&##L,l2TY.&$#$#&&$,(iiii,#&&&#$#$?TY2.$#$1(x###;2EE[t,\
SSEz.SW-k,T&&jC?E-.$##      &#&57+$$#      &&&W1-&$$7W  -J$#$kEN&#&      $##C^+$##W,h###n/+L2YE"
"2nJk/H;YNs#$[,:TU(#$   ,:   &&~H>&#   Y;   &&G_x&#2;   ,mT&$YE-#&   5G   $#VVF$#&zNs$$&Ej]HELy\
CN/U^Jk71<(#&:G7E+^&#  l|?1  $$Y.2$$  7lzs  WzZw>&$E    -<V-wE(2$$  G>x;  2zsW/$$#HKt&$$v>+t1(>"
"7>S7S,;TT,&$;S7S>7&#>E_::U  $$'",op  ,*G=  F,*I=957+F  ;int*t,k,O,  i,   j,T[+060<<+020];int M(
int m,int nop){;;;return+   m%(0+nop  );;}  int*tOo,w,  h,z,W;void(C)  (int n){n=putchar(n);}int
f,c,H=11,Y=64<<2,Z,pq,X   ;void(E/*d  */)(  int/*RP*/n  ){L(Z=k+00;  Z;  Z/=+2+000)G[000]=*G*!!f
|M(n,2)<<f,pq=2,f=+06   <f?++pq,++pq  ,G++  ,z:f+001,n  /=2;;}void  (V)(  int/*opqrstabd*/n){C(n
%Y);;C(n/Y+00);;}void  J(){L(pq--,pq   =j   =O=-1+0;++  j<240;I[6+   (h   +6+j/12/2*2+M(j/2,2))*
W+M(j/2/2,+06)*2+w*014      +00+M(00+      000+j,002      +00)]=000      +00+k)k=M(G[j/2/2+(*r-+
32)**"<nopqabdeg"],/*4649&96#*/3);/*&oaogoqo*/;}/*xD%P$Q#Rq*/int/*dbqpdbqpxyzzyboo3570OQ*/main()
{L(X=Y-1;i<21*3;i++,I++)L(r=G,G+=2;*G++;)*G>=13*3?*G-*r?*I++=*G:(*I++=r[1],*I++=r[2]):1;L(j=12,r
=I;(*I=i=getchar())>-1;I++)i-7-3?I-=i<32||127<=i,j+=12:(H+=17+3,W=W<j?j:W,j=12);L(;*r>-1;r++)*r-
7-3?J(),w++:(w=z,h+=17+3);C(71);C(73);V('*'*'1'*7);C(57);C(32*3+1);V(W);V(H);C(122*2);L(V(i=z);i
<32*3;)C(i++/3*X/31);C(33);C(X);C(11);L(G="SJYXHFUJ735";*G;)C(*G++-5);C(3);V(1);L(V(j=z);j<21*3;
 j++){k=257;V(63777);V(k<<2);V(M(j,32)?11:511);V(z);C(22*2);V(i=f=z);V(z);V(W);V(H);V(1<<11);r=
  G=I+W*H;L(t=T;i<1<<21;i++)T[i]=i<Y?i:-1;E(Y);L(i=-1;++i<W*H;t=T+Z*Y+Y)c=I[i]?I[i]*31-31:(31<
    j?j-31:31-j),Z=c[t[c]<z?E(Z),k<(1<<12)-2?t[c]=++k,T:T:t];E(Z);E(257);L(G++;k=G-r>X?X:G-r
        ,C(k),k;)L(;k--;C(*r++/*---#$%&04689@ABDOPQRabdegopq---*/));}C(53+6);return(z);}
```

- by Yusuke Endoh at 2018 International Obfuscated C Code Contest
- download `2018.c` and `smily.txt`
- compile by `gcc 2018.c -o 2018`
- run by `./2018 < smily.txt > smily.gif`
- open `smily.gif` with any image viewer

Outline

What is a variable?

- What is a variable?

What is a variable?

- What is a variable?
- In C, a view closer to implementation is taken
- In C, the best way to think of a variable is as
  a **portion of memory**
- In some sense, variables "do not exist". Only the memory exists!
  "Variables" are just a convenient way to refer to pieces of memory.
- In C, we care only about:
  - the **amount** of memory taken by a variable
  - **where** in the memory is a variable allocated
  - the **content** of the variable: the bytes in memory
- For this memory-centric view, C has operators for
  - "give me the memory address of this variable" and
  - "give me the variable at this memory address"
- The **type** of a variable describes how to interpret the bytes in memory
  - 2-complement integers,
  - floating-point, ...

the C is a **weakly typed** language

no strict control is made on the type of operands

Memory: the abstraction

- In C the memory is abstracted as a loooong sequence of **bytes**
- Memory is **byte-addressable**: at every memory address, there is only one byte

address	content
...	...
7FFF0040671A8107	A7
7FFF0040671A8108	E8
7FFF0040671A8109	03
7FFF0040671A810A	00
7FFF0040671A810B	00
7FFF0040671A810C	50
...	...

- addresses in memory are represented by a machine-dependent number of bytes: in the slide by 8 bytes (16 hex digits)
  - by 8-bytes-long addresses, it is possible to address up to $2^{8\times 8} = 2^{64} \approx 16 \times 10^{18}$ bytes (16 billions of GB)
- about a billion times larger than current size of large RAMs
- the address space is not used only to store data
- files or I/O devices (video) may be mapped onto a portion of the address space

Memory: endianness

- How to store variables needing more bytes?
  - by using contiguous memory locations
- Example
  - an `int` variable `var` is represented over 4 bytes
  - if stored at address `7FFF0040671A8108`, then it occupies the cells at:
    1. `7FFF0040671A8108`
    2. `7FFF0040671A8109`
    3. `7FFF0040671A810A`
    4. `7FFF0040671A810B`
  - its value is `var = 1000` (1000 decimal $=$ 4 bytes 00 00 03 $\texttt{E8}_{16}$)

little-endian: starts from **least** significant byte

...	...	
7FFF0040671A8108	E8	
7FFF0040671A8109	03	
7FFF0040671A810A	00	var
7FFF0040671A810B	00	

big-endian: starts from **most** significant byte

...	...	
7FFF0040671A8108	00	
7FFF0040671A8109	00	
7FFF0040671A810A	03	var
7FFF0040671A810B	E8	

x86 processors: multi-byte data is stored as **little-endian**

test-endian.c (the code is for experts, still questions are welcome)

Variables in C

1. the **declaration** of a variable informs the compiler of the size of the variable and its type (Example: `int a;` informs that a is an integer)
2. the **identifier** is the "name" of the variable.
   - identifiers may be composed by alphanumeric characters and underscore "_". Cannot start with a number. Cannot be a reserved C keyword (`for`, `while`, etc.)
3. An optional **initialization** by a constant

```
int a;                    /* not initialized */
int b = 91;               /* initialized */
int c = my_function();    /* wrong init: not a constant */
```

4. The **value** of the variable is the interpretation of the bytes in memory according the variable type
5. a variable is a portion of memory. The amount of memory used depends on the type of the variable.
   - A variable is **never** empty: it always has the value of the bytes in the memory
   - **Do not** assume that the initial content of a variable is zero (or else). Always **initialize it**.

Variable types

- Possible types of C variables are:
  - integer types of incresing length: `(char)`, `(short)`, `(int)`, `(long)`
  - floating point types: `(float)`, `(double)` are
  - addresses of memory, aka pointers: `(char *)`, `(int *)`, `(double *)`, `(void *)`
  - No other standard C types (example: no `boolean`)
- the size (in bytes) of these types is highly machine dependent
- the operator `sizeof()` returns the number of bytes of the type
  - `sizeof(int)`, number of byte of any variable of type `int`
  - `sizeof(a)`, number of byte of the variable `a`
- Check the size of the type of variables on your machine
  `test-sizeof.c`

Outline

Printing to the terminal

- The classic function fo print is `printf`
- It needs the directive `#include <stdio.h>` to be used
- `printf` can print strings, the value of variables and special characters
- The format is

  `printf(<format-string>, <list-of-expressions>)`

  `test-printf.c`
- For each expression in the list, the format string must specify how this expresion should be printed.
- Format specificators **must** be as many as the expressions

`%d`	print integer, base 10
`%o`	print integer, base 8
`%X`	print integer, base 16
`%e`	print floating point, notation `1.23e1`
`%f`	print floating point, notation `12.3`
`%s`	string of characters
`%c`	the ASCII character

- `man 3 printf` for full reference

Printing: escape character

- The <format-string> may contain escape characters to print non ASCII standard characters

\n	new line
\t	tab
\"	character "
\'	character '
\\	character \
%%	character %
\uXXXX	Unicode character coded by the 4 hex digits XXXX
\UXXXXXXXX	Unicode character coded by the 8 hex digits XXXXXXXX

test-printf.c