Operating systems

Elements of C for operating systems

Alberto Castellini
alberto.castellini@univr.it

Florenc Demrozi
florenc.demrozi@univr.it

University of Verona
Department of Computer Science

2021/2022

Table of Contents

1. Command line arguments
2. ASCII coding
3. Bitwise operators
4. Macros
5. Structures
6. Unions
7. Pointers
8. References

Command line arguments

Command line arguments (1/2)

The `main()` method can be used without arguments

```
#include <stdio.h>
int main() {
  printf("Hello world!");
  return 0;
}
```

or with two parameters argc, and argv (called *command line arguments*):

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

Command line arguments (2/2)

```c
int main(int argc, char * argv[])
```

In the latter case:

- `argc`: gets the *number* of parameters in the command line
- `argv`: is an array of char pointers (i.e., strings) that correspond to command line *arguments*
  - argv[0]: program name
  - argv[i], $i > 0$: program arguments

```
./ex1_commandLineArgs myArg1 myArg2 myArg3
argc = 4
argv[0] = ./ex1_commandLineArgs
argv[1] = myArg1
argv[2] = myArg2
argv[3] = myArg3
```

ASCII coding

ASCII coding (1/2)

- Character in C are represented by integers
- Constants 'a' and '+', for instance, have type `int`
- Several systems use the *American Standard Code for Information Interchange* (ASCII) for representing characters
- Example 1: character 'A' is represented by the integer 65

```
putchar(65); // Prints character 'A'
putchar('A'); // Prints character 'A'
```

- Example 2: obtain the ASCII code of a given "character"

```
char value;
scanf("%c", &value); // Input 'A'
printf("%c\n",value); // Prints character 'A'
printf("%d\n",value); // Prints 65 the ASCII code of character 'A'
```

ASCII coding (2/2)

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Bitwise operators

Bitwise operators

- Bitwise operators work on integer expressions represented as strings of bits
- These operators are system dependent
- In the following we analyze operators for systems having
  - bytes of 8 bits
  - integers of 4 bytes
  - two's complement notation for integers
  - ASCII coding for chars
- **Logical operators:**

  ~ : unary complement (bitwise)

  & : and (bitwise)

  ^ : xor (bitwise)

  | : or (bitwise)
- **Shift operators:**

  << : shift to the left

  >> : shift to the right

Unary complement (bitwise)

- The unary complement inverts every bit in the binary representation of the operand
- Example 1:
  - Integer representation of the operand:

    ```
    int a = 70707;
    ```
  - Binary representation:

    ```
    00000000 00000001 00010100 00110011
    ```
  - Its unary complement (˜a):

    ```
    11111111 11111110 11101011 11001100
    ```
  - Integer representation of ˜a:

    ```
    -70708
    ```

Two's complement

- The two's complement of an integer n is:
  - If $n \geq 0$: the standard binary representation (in base 2) of n
  - If $n < 0$: the unary complement of the standard binary representation of $-n$, summed to one,
- Example 2:
  - Integer number:

    ```
    int n = 7;
    ```
  - Binary representation of `n`:

    ```
    00000000 00000111
    ```
- Example 3:
  - Integer number:

    ```
    int n = -7;
    ```
  - Binary representation of `-n`:

    ```
    00000000 00000111
    ```
  - Unary complement of `-n` (`~-n`):

    ```
    11111111 11111000
    ```
  - Two's complement of `n` (`~-n` + 1):

    ```
    11111111 11111001
    ```

And, xor, or (bitwise)

- And (&), xor (^), or (|) are binary operators having integer arguments.
- Truth tables

AND		
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

exclusive-OR

- Example 4:

a	00000000 00000000 10000010 00110101	(33333)
b	11111111 11111110 11010000 00101111	(-77777)
a & b	00000000 00000000 10000000 00100101	(32805)
a ^ b	11111111 11111110 01010010 00011010	(-110054)
a | b	11111111 11111110 11010010 00111111	(-77249)
~(a | b)	00000000 00000001 00101101 11000000	(77248)
(~a & ~b)	00000000 00000001 00101101 11000000	(77248)

Left shift

- `expr1` $<<$ `expr2`: shifts the binary representation of `expr1`, of `expr2` positions to the left. It inserts zeros on the right.
- Example 5:
  - Let us take this as example:
    `int c='Z';`
  - which in ASCII representation corresponds to 90
  - Let us now apply the left shift operation:

c	00000000 00000000 00000000 01011010
c $<<$ 1	00000000 00000000 00000000 10110100
c $<<$ 4	00000000 00000000 00000101 10100000
c $<<$ 31	00000000 00000000 00000000 00000000

- **Notice:** even if `c` is a character (1 byte), it is cast to `int`. Both arguments of the shift operator are always cast to `int`.

Right shift

- `expr1 >> expr2`: shifts the binary representation of `expr1`, of `expr2` positions to the right. If `expr1` is an `unsigned` then the shift operator inserts `zeros` on the left, while if `expr1` is a `signed` number then it may insert `zeros` or `ones` (i.e., the *sign bit*), depending on the specific machine.
- Examples 6:
  - `int a = 1 << 31;`

a	10000000 00000000 00000000 00000000
a >> 3	11110000 00000000 00000000 00000000

  - To preserve the sign bit, it inserts **ones**.
- Examples 7:
  - `unsigned b = 1 << 31;`

b	10000000 00000000 00000000 00000000
b >> 3	00010000 00000000 00000000 00000000

  - We are working with an **unsigned**, thus it fills with **zeros**.

Masks

- A **mask** is a constant or a variable used to extract some bits from another variable or expression.
- Since constant 1 has binary representation

  00000000 00000000 00000000 00000001

  it can be used to determine the less significant bit of another expression.
- What does this code print? (Example 8)

```
int i, mask = 1;
for (i = 0; i < 10; ++i)
  printf("%d", i & mask)
```

- Expression 1 << 2 may be used instead as a mask to extract the third bit from the right (less-significant).
- The value of expression `(v & (1 << 2)) ? 1 : 0` is 1 if the third less-significant bit of `v` is 1, otherwise it is 0 (Example 9).

Macros

The `#define` directive

- The *C preprocessor* enables the inclusion of header files, macro expansions, conditional compilation, and line control in C programs.
- The `#define` directive allows the definition of *macros* within the source code.
- This directive may have two forms:
  1. `#define identifier tokenString`
  2. `#define identifier(param1,..., paramN) tokenString`

  where `tokenString` is optional.
- Macros are often used to *substitute* function calls with *inline code* which improves efficiency.

The `#define` directive: Form 1

- When the preprocessor finds a #define of the first form

  `#define identifier tokenString`

  it substitutes every occurrence of identifier in the rest of the code with `tokenString`, except for the occurrences in quotes.
- Examples:

```
#define SECONDS_PER_DAY (60 * 60 * 24)
#define PI 3.14159
#define C 299792.458 // Light speed in Km/sec
#define EOF (-1)
#define MAXINT 2147483647
#define ITERS 50
```

- Symbolic constants improve the readability of the code
- Syntactic sugar: it is also possible to modify the C syntax using these kind of constants

  Example: `#define EQ ==`

The #define directive: Form 2 (1/2)

- The general syntax is

```
#define identifier(param1,..., paramN) tokenString
```

- There must be no space between the first identifier and the first bracket
- The list of parameters may contain between 0 and several identifiers
- Example:

```
#define SQ(x) ((x) * (x))
```

the x identifier is a parameter which is substituted in the subsequent text (i.e., `((x) * (x))`)

The `#define` directive: Form 2 (2/2)

- String substitution is performed by the preprocessor, for instance:

```
SQ(7 + w)
is substituted by
((7 + w) * (7 + w))
```

and

```
SQ(SQ(*p))
is substituted by
((((*p) * (*p))) * (((*p) * (*p))))
```

The `#define` directive: Brackets (1/2)

- Notice: brackets are important to avoid undesired expansions
- Example 1:

```
// Macro definition:
#define SQ(x)  x * x

// Macro usage:
SQ(a + b)

// Macro expansion:
a + b * a + b // ERROR! Different from ((a + b) * (a + b))
```

- Notice: macro definitions do not end with a semicolon

The #define directive: Brackets (2/2)

- Example 2:

```
// Macro definition:
#define SQ(x) (x) * (x)

// Macro usage:
4 / SQ(2)

// Macro expansion:
4 / (2) * (2) // ERROR! Different from 4 / ((2) * (2))
```

Macros: advanced concepts

- Macro definitions may use both functions and other macros
- Example:

```
#define SQ(x)   ((x) * (x))
#define CUBE(x) (SQ(x) * (x))
```

- The preprocessor directive

```
#undef identifier
```

deletes a macro definition.

Structures

Structures: definition and variable declaration (1/2)

- Structures are *derived* data structures for *heterogeneous* data
- The structure components are said *members*. Each member has a name
- Structure definition (example)

```
struct card {
  int pips; // 1,...,13
  char suit; // 'c'(clubs), 'd'(diamonds), 'h'(hearts), 's'(spades)
};
```

Structures: definition and variable declaration (2/2)

- Struct *variable declaration* (example 1):

```
struct card {
  int pips; // 1,...,13
  char suit; // 'c'(clubs), 'd'(diamonds), 'h'(hearts), 's'(spades)
};

struct card c1, c2;
```

- Struct *variable declaration* (example 2):

```
struct card {
  int pips; // 1,...,13
  char suit; // 'c'(clubs), 'd'(diamonds), 'h'(hearts), 's'(spades)
} c1, c2;
```

Typedef

- To simplify the declaration of struct variables, it is a good practice to define a new type using the operator `typedef`.
- Syntax:

```
typedef data_type new_name;
```

- Example with structures:

```
// Definition of new type name "card" from type "struct card"
typedef struct card card;
// Usage of the new type
card c3, c4, c5;
```

Struct members (1/4)

- Struct members can be accessed by the dot "." operator.
- Example:

```
c1.pips = 3;
c1.suit = 's';
```

Struct members (2/4)

- Member names must be unique within a structure but the same names may be used in different structures.

```c
struct fruit {
  char * name;
  int calories;
} a;

struct vegetable {
  char * name;
  int calories;
} b;

a.name = "apple";
b.name = "salad";
```

Struct members (3/4)

- When we deal with struct pointer variables, members are accessed by the "->" operator.
- Example:

```
struct complex {
  double re;
  double im;
}

typedef struct complex complex; // Typedef of complex

void add(complex *a, complex *b, complex *c) { // a = b + c
  a->re = b->re + c->re;
  a->im = b->im + c->im;
}
```

- Notice that a, b and c are pointers to structures.

Struct members (4/4)

- The -> operator (example):

```c
struct student {
  char * last_name;
  int student_id;
  char grade;
}

struct student tmp, *p = &tmp;

tmp.grade='A';
tmp.student_id=342;
tmp.last_name="Rossi";

printf("%c", tmp.grade); // Prints: A
printf("%c", p->grade);  // Prints: A
```

Unions

Unions: definition and variable declaration (1/2)

- *Unions* are *derived* data structures for *heterogeneous* data (as structures) but their members share the *same memory*.
- An union type defines a series of *alternative values* that can be contained in the same portion of shared memory.
- Union definition (example):

```
union int_or_float { // Union definition
  int   i;
  float f;
}
typedef union int_or_float number; // Typedef of number

number a, b, c; // Union variable definition
```

- The compiler allocates memory for the larger member.

Unions: definition and variable declaration (2/2)

- *Access* (example):

```
number n;
n.i=4444;
printf("i: %10d f: %16.10e\n", n.i, n.f);
// Prints: i:          4444     f: 6.227370375e-41

n.f=4444;
printf("i: %10d f: %16.10e\n", n.i, n.f);
// Prints: i: 1166729216   f: 4.4440000000e+03
```

Pointers

Pointers

- Variables are stored in memory using a certain number of bytes (dependent on variable type) and from a specific location (address)
- *Pointers* are used to store memory addresses and to access memory
- & operator: if v is a variable, then &v is the location (address) where *v* is stored in memory
- Pointer declaration (example): `int * p;`
- Usage of pointers (example):

```c
int a = 1, b = 2, * p;
p = &a; // Pointer p contains the address of variable a
b = *p; // Variable b contains the content of the variable pointed by p
// Now b = a
```

Pointers: Arrays

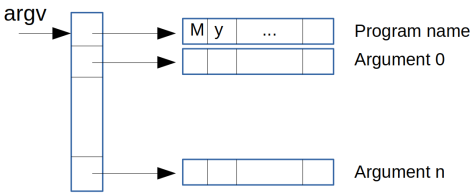
- Pointers and **arrays**

```
int a[3];
a[0] = 5;
a[1] = 7;
a[2] = 9;
// a[i] is equivalent to *(a + i)
printf("%d == %d\n", a[1], *(a + 1)); // Prints: 7 == 7
```

- It is possible to use pointers notation with arrays and array notation with pointers

Multidimensional arrays: pointers to pointers

- Example: the `argv` argument of method `main` is an array of strings, and it can be seen as a pointer to pointers to char or a bi-dimensional array (`char ** argv`):



Function pointers (1/2)

- Example

```c
int addInt(int n, int m) {
  return n + m;
}
int main() {
  // Definition of funct pointer
  int (*functionPtr)(int,int);

  // Let functionPtr point to addInt
  functionPtr = &addInt;

  // Use the pointer sum == 5
  int sum = (*functionPtr)(2, 3);

  return 0;
}
```

Function pointers (2/2)

- Example

```
void fun(int a) {
  printf("Value of a is %d\n", a);
}
int main() {
  // fun_ptr is a pointer to function fun()
  void (*fun_ptr)(int) = &fun;

  // Invoking fun() using fun_ptr
  (*fun_ptr)(10);

  return 0;
}
```

References

References

- Al Kelley, Ira Pohl. *C – Didattica e Programmazione.* Quarta edizione.Pearson. 2004.