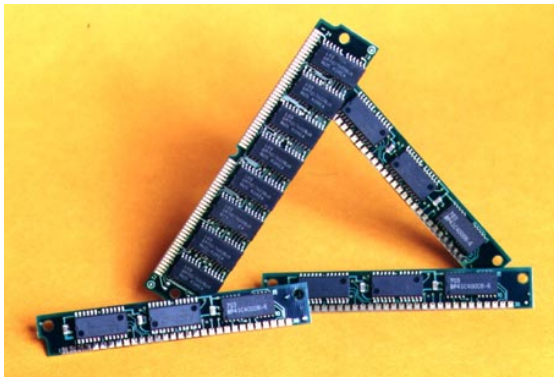
Programming I

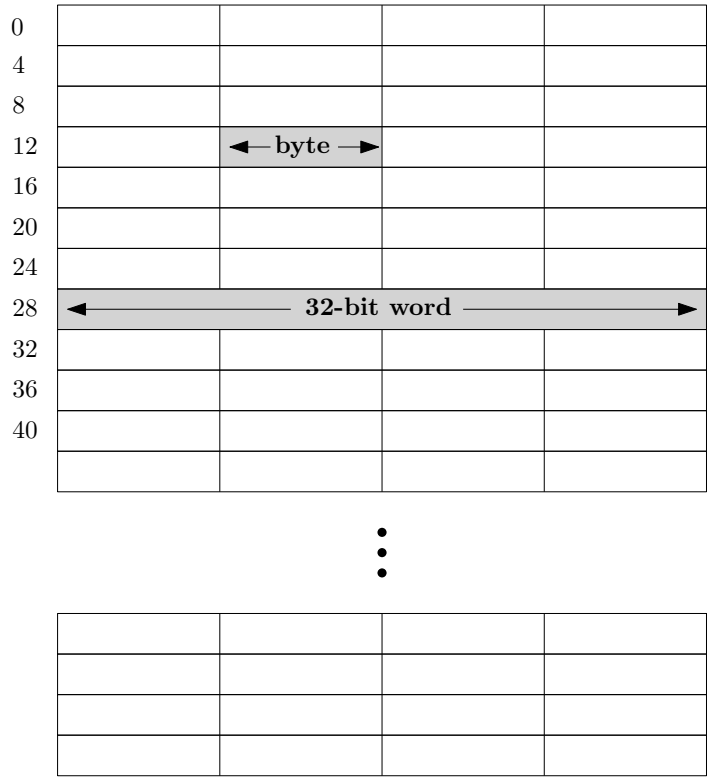
Variables, Types and Constants

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Memory





Memory behaves like an one dimensional array which we can address using an address which is either a 32-bit or 64-bit number depending on the architecture. The figure above presents the 32-bit version.

Variables in C

C language provides with an easy way to address the memory, without the need to memorize memory addresses.

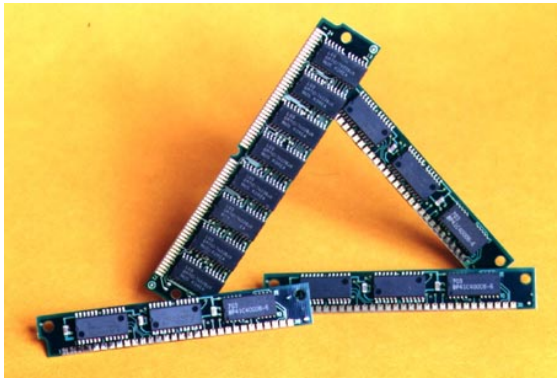
Variables in C

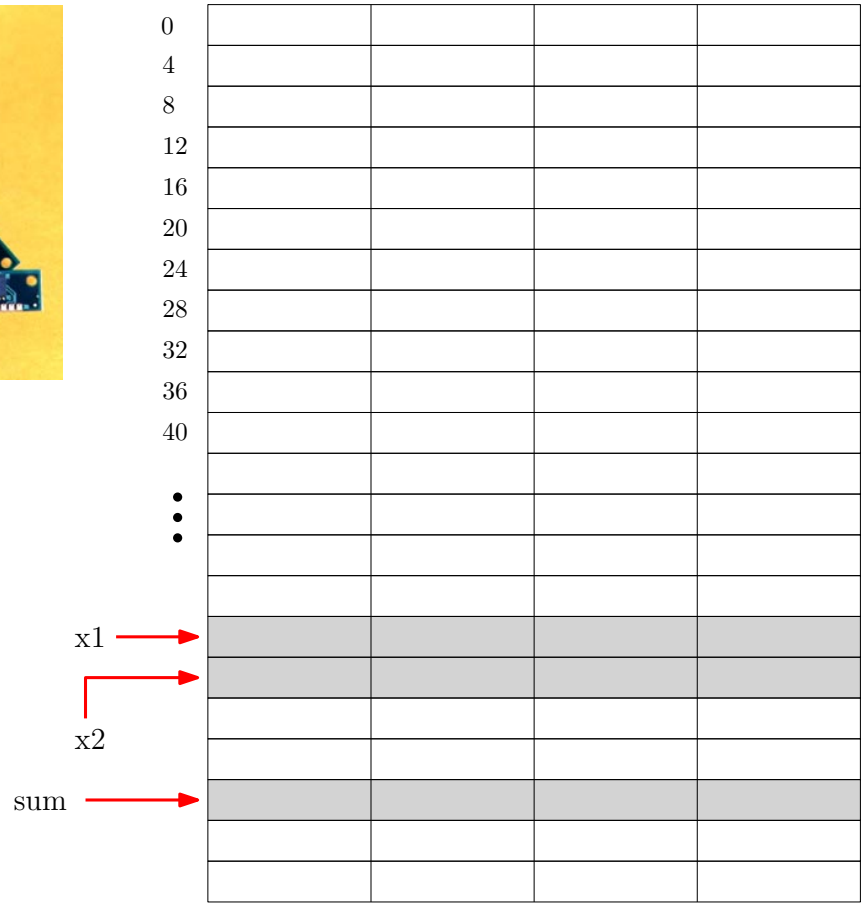
```
int main() {
    int x1, x2, sum;

    x1 = 1;
    x2 = 2;
    sum = x1 + x2;
}
```

In the program above we inform the compiler that we are going to require the use of three memory locations in order to store integer numbers. These 3 memory locations are going to be named `x1`, `x2`, and `sum` respectively.

The compiler remembers the mapping between names and addresses.

Memory





Variable names are shorthands for memory addresses.

Variables in C

```
int main() {
    int x1, x2, sum;

    x1 = 1;
    x2 = 2;
    sum = x1 + x2;
}
```

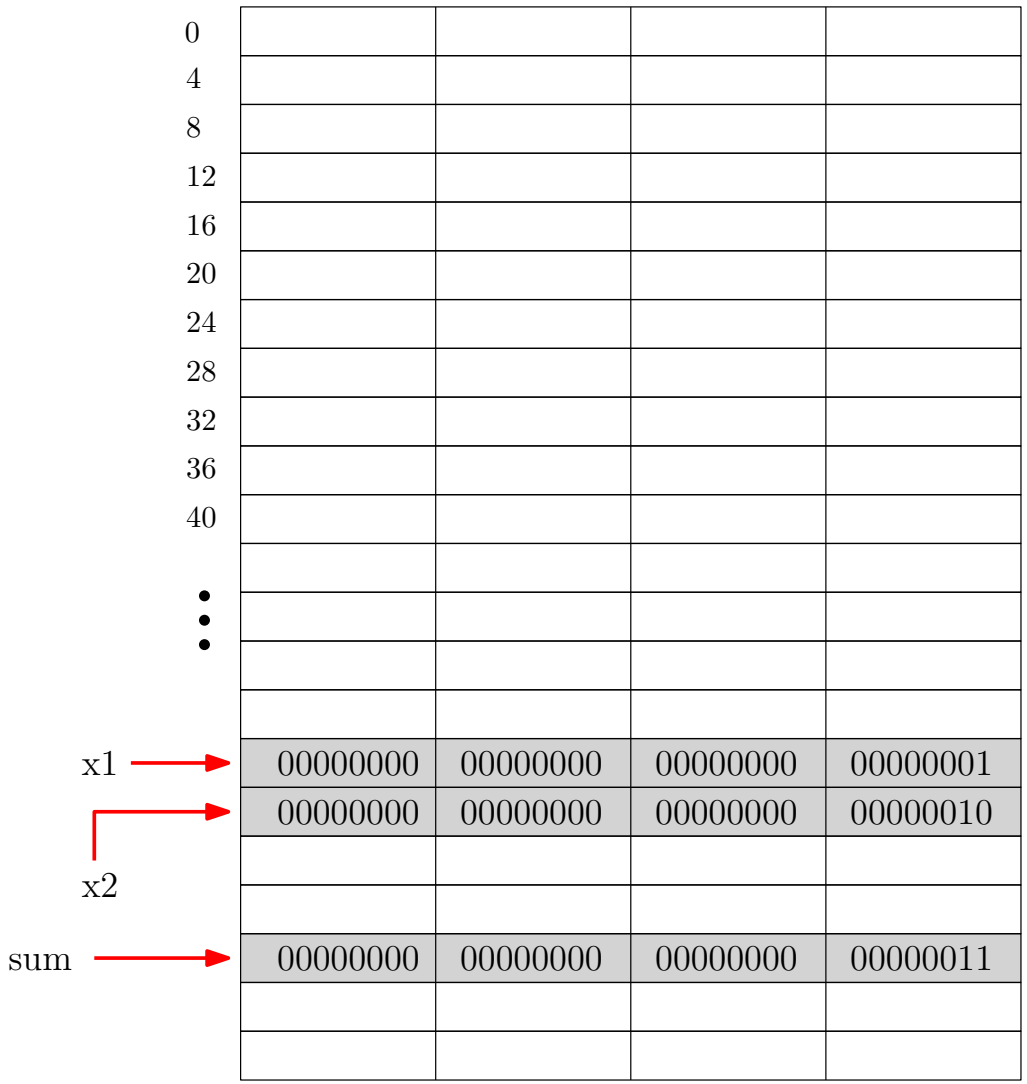
Memory locations of local variables are not initialized. It is the duty of the programmer to provide with initial values.

Variables in C

```c
#include <stdio.h>

int main() {
    int x1, x2, sum;

    x1 = 1;
    x2 = 2;
    sum = x1 + x2;
}
```

- Store value 1 at the memory location of variable $x1$ and 2 at the memory location of variable $x2$.
- Finally read these two values again, add them up and store the result at the memory location of variable *sum*.

Variables in C



Variables are shorthands for memory addresses.

Variables in C

Every variable has:

1. name
2. type
3. value

Variables in C

Name

A valid variable name in C must begin with a letter (not a digit), it should not contain any spaces and should be equal to certain reserved keywords such as `main`.

wrong variable names

```
int main;
int 3x;
int hello world;
```

correct variable names

```
int Main;
int x123456;
int hello_world;
```

Variables in C

Type

When we are declaring a variable, we explain to the compiler what kind of information we are going to store in that location.

Basic data types

1. integer: `int x;`
2. character: `char x;`
3. floating point (approximate representation for real numbers): `float x;`

There are more data types. We are going to see them in more detail after we learn a few things about number representation in a computer.

Variables in C

C language requires all variable sto be declared at the beginning of a block. For example:

```
int main() {
    int x, y, z; /* declare all variables */

    x = 1;
    y = 2;
    z = x + y;

    int i;       /* NO! */
}
```

Variables in C

From C99 and afterwards, we can also declare variable in other locations.

```
int main() {
    int x, y, z; /* declare some variables */

    x = 1;
    y = 2;
    z = x + y;

    int i;       /* YES! */

}
```

It is nowadays considered best practice to declare variables as close to their use as possible.

Type `int`

Type `int` stores integers and its size depends on the architecture of the computer that we are compiling our program.

- it must be able to take any value $\in [-32767, 32767]$
- in recent architectures, it can take any value $\in [-2147483648, 2147483647]$

This means that it needs 32-bits.

Type `char`

Type `char` is used to store characters.

```
int main() {
    char c;

    c = 'x';
}
```

In C characters are representation by numbers using the ASCII encoding.

Type `char`

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]

The 95 printable characters from 32 to 126 (decimal). Characted from 0 to 31 are special characters for controlling the output devices, e.g. 8 is BACKSPACE, 13 is ENTER and 27 is ESC.

printf and characters

```c
#include <stdio.h>

int main() {
   char c = 'a';

   printf("character %c is number %d in ASCII encoding\n", c, c);
}
```

The above program prints:

character a is number 97 in ASCII encoding

Type `float`

`float` is a type for approximately representing real numbers.

```
#include <stdio.h>

int main() {
    float pi = 3.14159265;

    printf("pi is approximately %f\n", pi);
}
```

Maximum and Minimum Values for Integer Types

The table below is taked from the C specification.

type	minimum value	maximum value
`char`	≤ -127	$\geq +127$
`unsigned char`	≤ 0	$\geq +255$
`short int`	≤ -32767	$\geq +32767$
`unsigned short int`	≤ 0	$\geq +65535$
`int`	≤ -32767	$\geq +32767$
`unsigned int`	≤ 0	$\geq +65535$
`long`	≤ 2147483647	$\geq +2147483647$
`unsigned long`	≤ 0	$\geq +4294967295$

Recent compilers have larger bounds. For example type `int` usually has the bounds that are shown for `long` in the table above.

Types and Limits of Floating Point Numbers

type	minimum value	maximum value
`float`	$\leq -1E-37$	$\geq 1E+37$
`double`	$\leq -1E-37$	$\geq 1E+37$

Modern compilers have larger limits from the ones mentioned in the ANSI C specification. For example, **gcc** on my computer has:

$$-1.175494E-38 \leq x \leq 3.402823E+38$$

for type `float` and:

$$-2.225074E-308 \leq x \leq 1.797693E+308$$

for type `double`.

Function `printf`

The general form of `printf()` is:

```
int printf(const char * format, ...);
```

The parameter *format* contains the text to be printed along with special character sequences which help to print the expressions that follow. These special sequences have the general form:

`%[flags][width][.precision][length]specifier`

Except for the specifier everything else is optional.

Specifiers of `printf()`

Most specifiers are shown in the following table:

specifier	Output	Example
c	character	a
d i	decimal with sign	392
e	scientific notation using e	3.9265e+2
E	scientific notation using E	3.9265E+2
f	decimal floating point number	392.65
o	octal with sign	610
s	string	sample
u	decimal without sign	7235
x	hexadecimal without sign	7fa
X	hexadecimal without sign with capitals	7FA

Line

```
printf("%d, %o and %x\n", 27, 27, 27);
```

prints 27, 33 and 1b.

printf() and padding

The width is a number which tells `printf` how many spaces to add to the output in order for the result to have the specified characters.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a = 100;
    int b = 1000;
    int c = 10000;

    printf("%7d\n", a);
    printf("%7d\n", b);
    printf("%7d\n", c);
    printf("%7d\n", 1000000);
}
```

prints

```
    100
   1000
  10000
1000000
```

`printf()` and precision

Precision has different meaning based on each type:

- for integers it denotes the least amount of digits to be printed (possible by including additional zeros a the beginning)
- for floats is represents the number of digits after the decimal
- for strings it denotes the maximum number of characters to be printed

`int` and precision

Denotes the least amount of digits to be printed (possible by including additional zeros a the beginning)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a = 100;
    int b = 10000;
    int c = 1000000;

    printf( "%10.7d\n", a );
    printf( "%.7d\n", b );
    printf( "%.7d\n", c );

    return 0;
}
```

prints

```
   0000100
0010000
1000000
```

float and precision

Denotes the number of digits after the decimal.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    float pi = 3.14159265;

    printf( "%.3f\n", pi );
    printf( "%.10f\n", pi );

    return 0;
}
```

prints

```
3.142
3.1415927410
```

printf() and Special Characters

The special characters in C can be found in the following table:

character	ASCII code	special character in C
newline	10	'\n'
tab	9	'\t'
carriage return	13	'\r'
backspace	8	'\b'
form feed	12	'\f'
backslash	92	'\\'
single quotation mark	39	'\''
double quotation mark	34	'\"'
null character	0	'\0'

for example the following code

```
printf("Very\tSimple\nExample");
```

prints

```
Very        Simple
Example
```

Constants

There are several different kind of constants:

- literals
- symbolic constants
- declared constants using `const`

Constants

There are several different kind of constants:

- literals: types directly in the code

```
int count;

count = 3;
```

- symbolic constants
- declared constants using `const`

Constants

There are several different kind of constants:

- literals
- symbolic constants: the pre-processor allows us to define symbols which during the pre-processing phase are replaced with corresponding values.

```
#include <stdio.h>

#define MAX 100

int main() {
    int x = MAX;
    printf( "%d\n", x );

    return 0;
}
```

- declared constants using `const`

Constants

There are several different kind of constants:

- literals
- symbolic constants
- declared constants using `const`:

  used in variable declarations and informs the compiler that a variable is not supposed to change value.

```
int main() {
    const int x = 3;

    x = 5; /* Compilation error! */
}
```

  If anyone tries to change the value, the compiler reports an error.