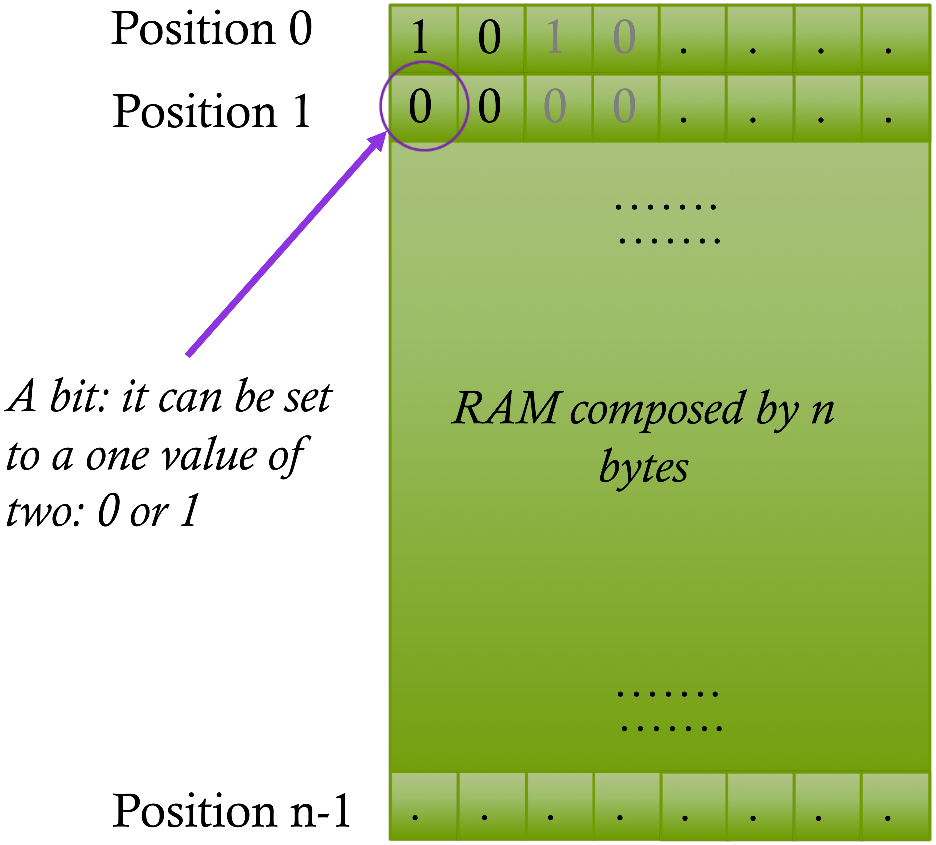
Variables and Objects

How to complicate a simple thing

massimo.marchi@unimi.it

Computer Memory

- *The RAM can be viewed as a ribbon of bytes, each one composed by 8 bit*
- *Any data (sound, table, image, etc.) have to be translated into a sequence of bytes*
- *Any cell can be reached, for read or write operations, by its address which is its position on the ribbon*



Position 0	1	0	1	0	.	.	.	.
Position 1	0	0	0	0	.	.	.	.

....
....

RAM composed by n bytes

....
....

Position n-1	.	.	.	.	.	.	.	.

A bit: it can be set to a one value of two: 0 or 1

Format Representation

- Any data have to be expressed by a sequence of bytes
- For ex. *Unsigned Short Integer*: a number from 0 to 255 can be express as a sequence of 8 bit:

$0_{10} = 00000000_0$, $1_{10} = 00000001_0$, $2_{10} = 00000010_0$,

…. , …. , ….

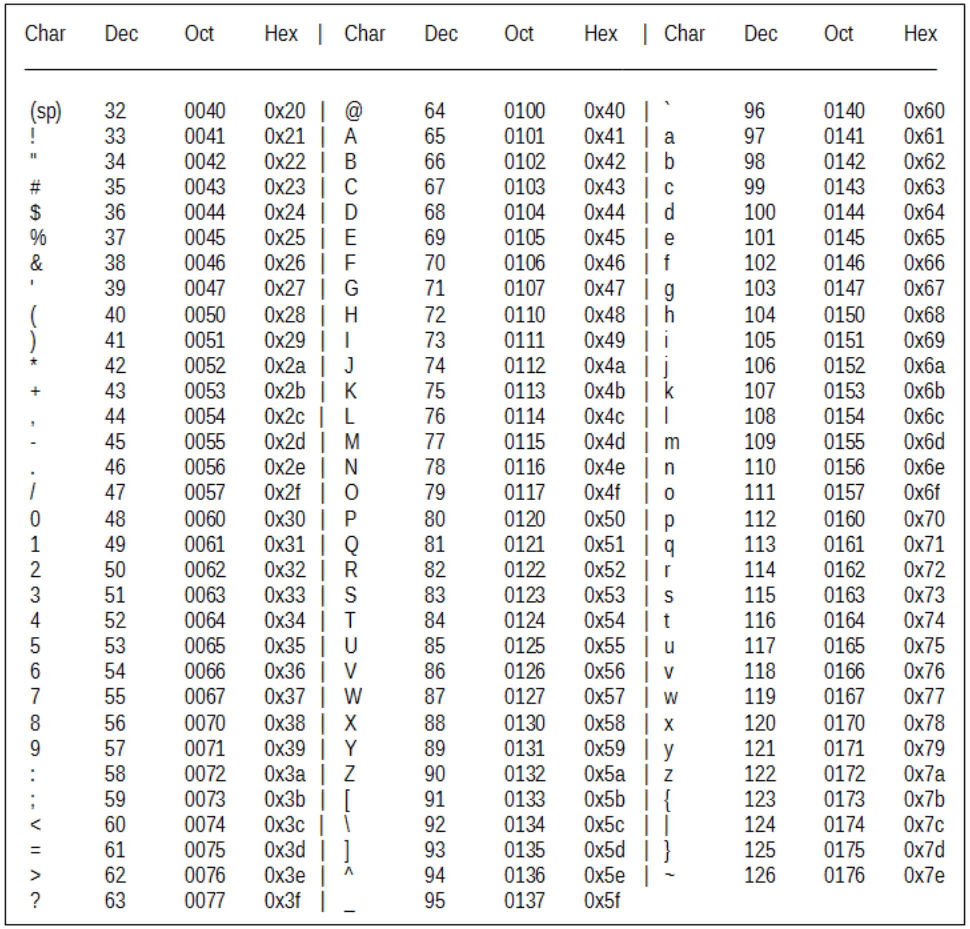
…. , $254_{10} = 11111110_2$, $255_{10} = 11111111_2$

Base 10: Decimal notation; number are expressed with digits from 0 to 9

Base 2: Binary notation; number are expressed with digits from 0 to 1

Format Representation$_2$

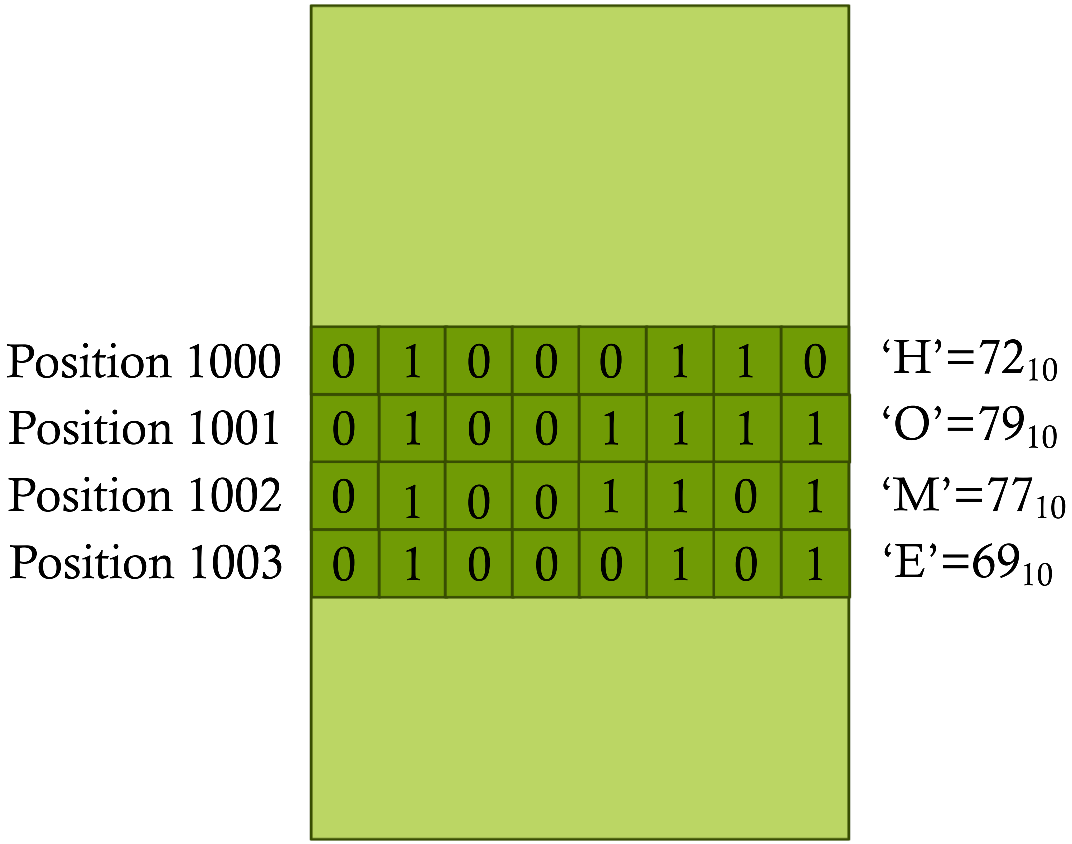
- For ex. *ASCII char*: a limited set of char can be memorized as a single byte; the meaning of any value is defined by a *lookup* table:
  - ‘A’=65
  - ‘a’=97
  - ’0’=48
  - ‘8’=56



Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

A data in memory

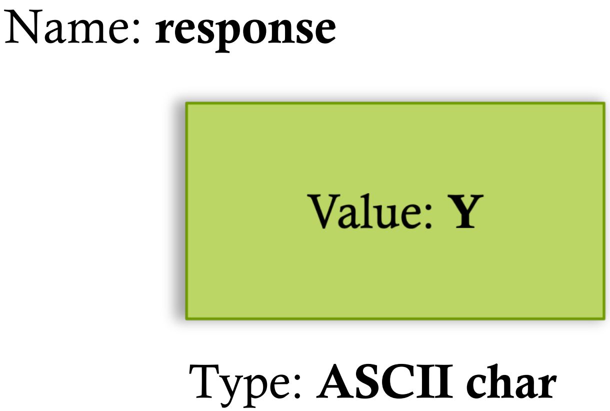
- *Ex.: we can memorize the sequence of char ‘HOME’ inside memory from position 1000 in this way*



Position 1000	0	1	0	0	0	1	1	0	‘H’=72_{10}
Position 1001	0	1	0	0	1	1	1	1	‘O’=79_{10}
Position 1002	0	1	0	0	1	1	0	1	‘M’=77_{10}
Position 1003	0	1	0	0	0	1	0	1	‘E’=69_{10}

Variables in programming

- Variables are «box» with these properties:
  - A **name** which is used to «address» it
  - A **type** which express the set of valid values you can store in it
  - A **value** which is the current value.



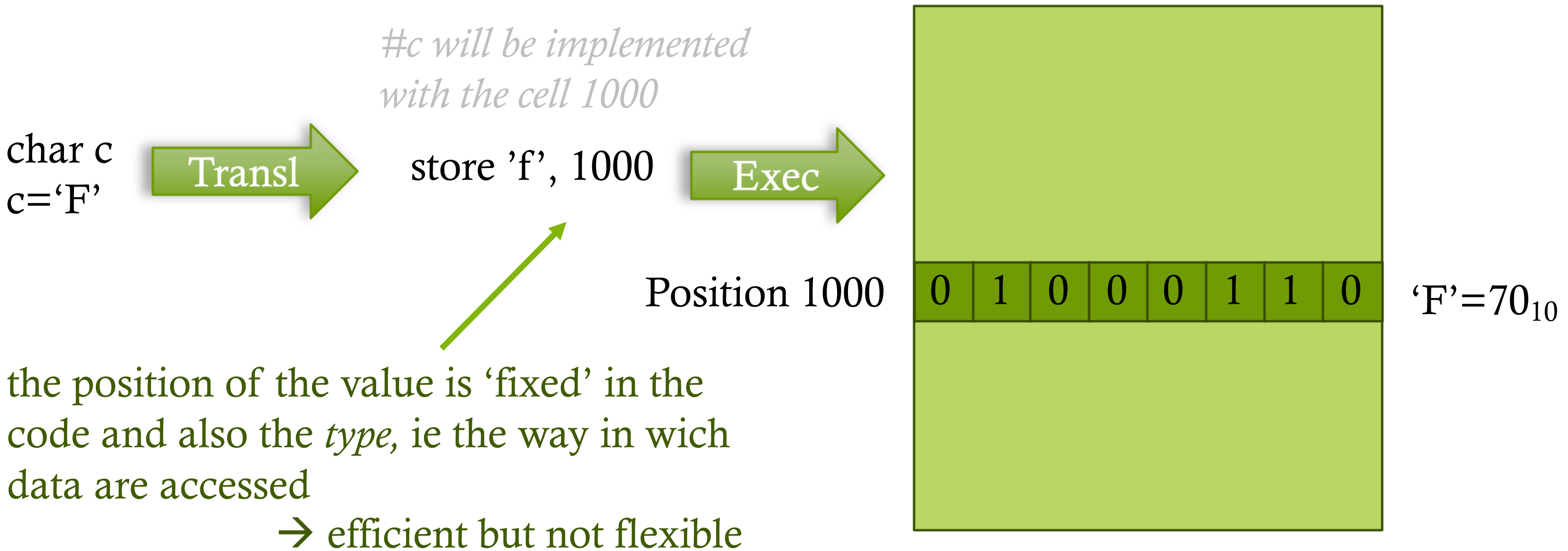
Name: **response**

Value: **Y**

Type: **ASCII char**

Variables as Variables

- During the translation from a not-object oriented Hi-Level language (for ex. C) to Low-level language (Assembly) the reference of a variable became an address:



char c
c=‘F’

Transl

#c will be implemented with the cell 1000

store ’f’, 1000

Exec

Position 1000

0	1	0	0	0	1	1	0

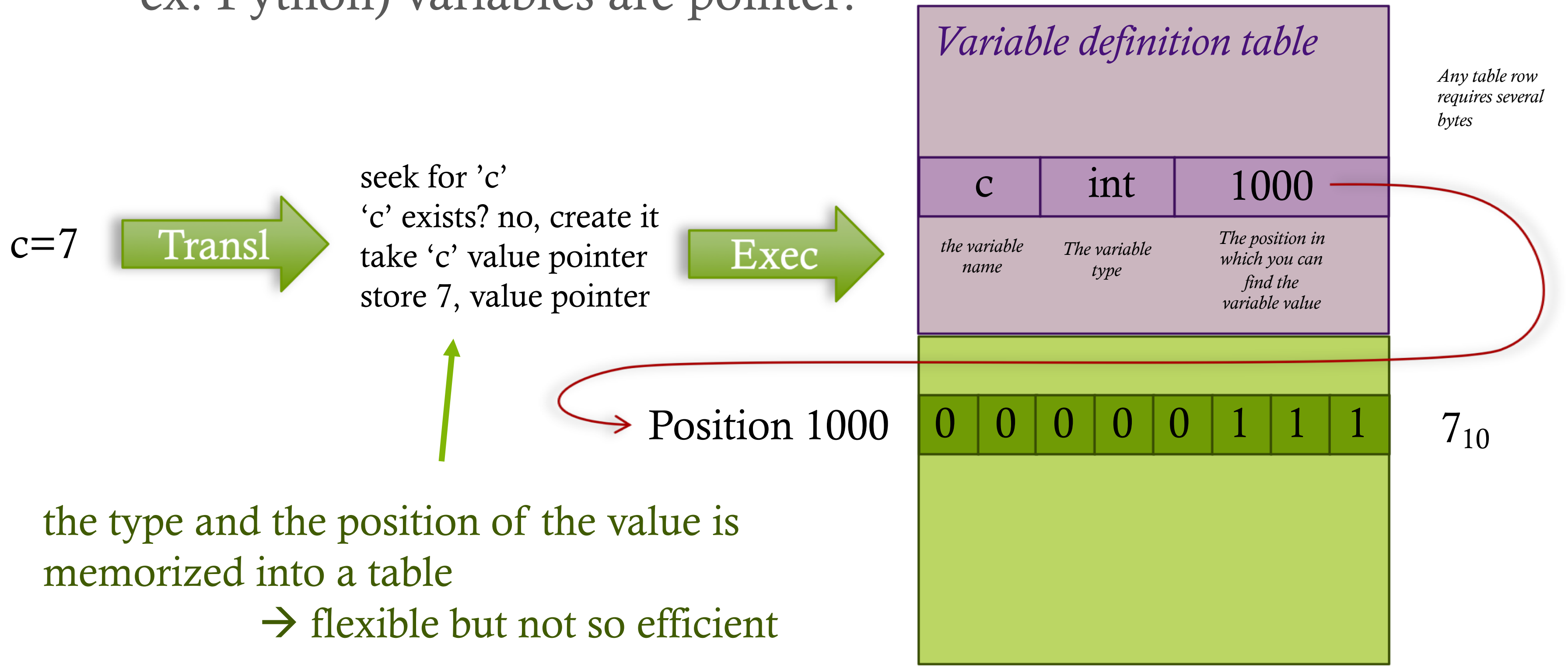
‘F’=70_{10}

the position of the value is ‘fixed’ in the code and also the *type,* ie the way in wich data are accessed

→ efficient but not flexible

Variables as References

- In a dynamic typed object oriented Hi-Level language (for ex. Python) variables are pointer:



c=7

Transl

seek for ’c’
‘c’ exists? no, create it
take ‘c’ value pointer
store 7, value pointer

Exec

Variable definition table

c	int	1000
the variable name	*The variable type*	*The position in which you can find the variable value*

Any table row requires several bytes

Position 1000

0	0	0	0	0	1	1	1

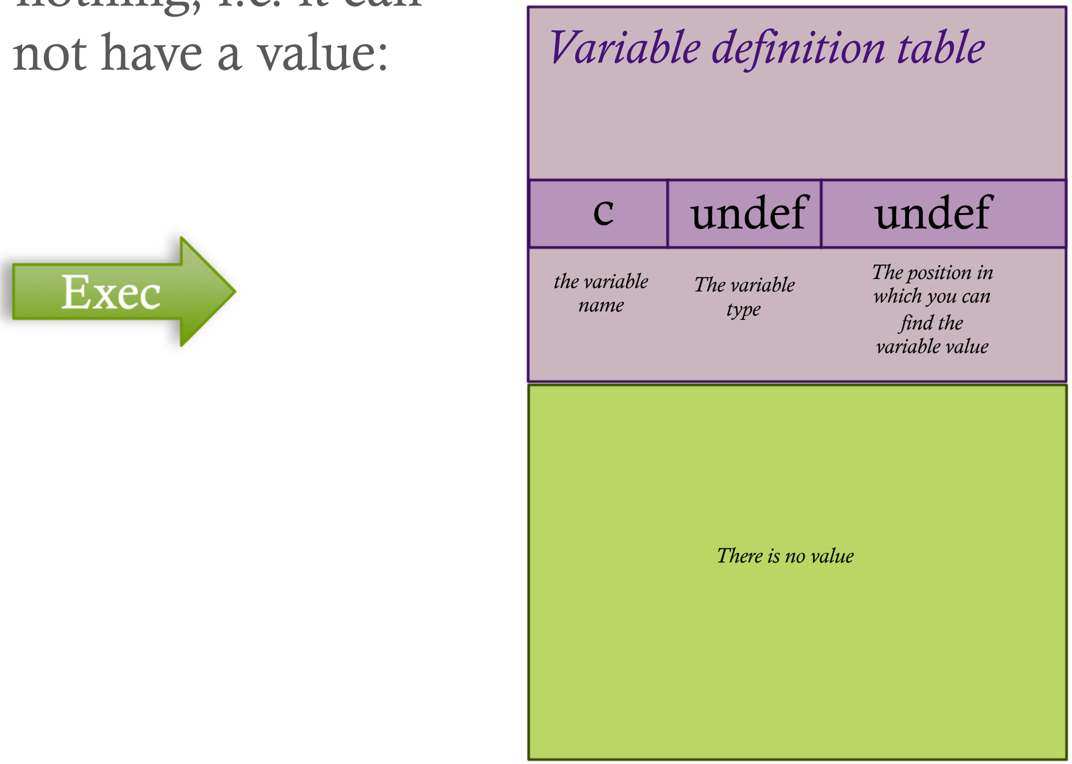
7_{10}

the type and the position of the value is memorized into a table
→ flexible but not so efficient

Undefined values

- If a variable is a reference, then it can also point to nothing, i.e. it can defined but does not have a value:

```
c=undef
```



Variable definition table

c	undef	undef
the variable name	*The variable type*	*The position in which you can find the variable value*

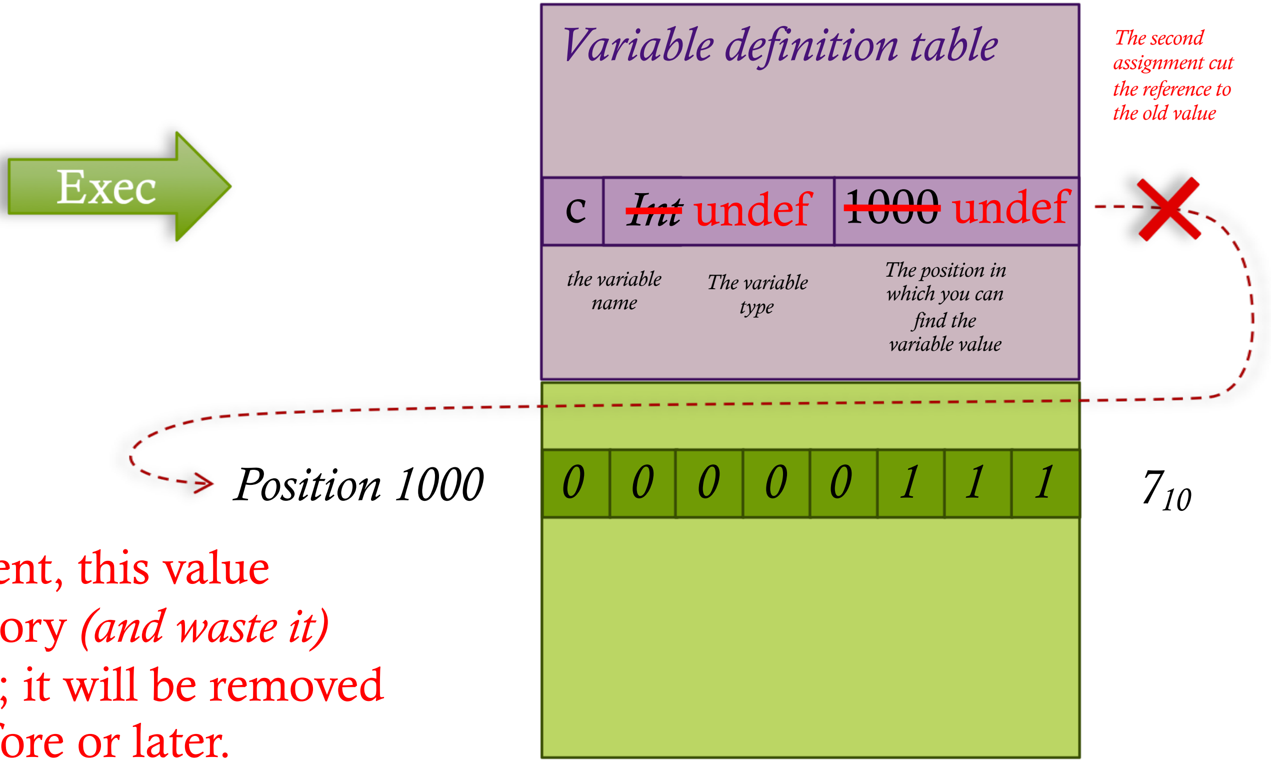
There is no value

Often undef value is coded using the value 0

Orphan Values

- A variable can *lost its* value:

```
c=7
c=undef
```



Exec

Variable definition table

c	~~*Int*~~ undef	~~1000~~ undef
the variable name	*The variable type*	*The position in which you can find the variable value*

The second assignment cut the reference to the old value

Position 1000

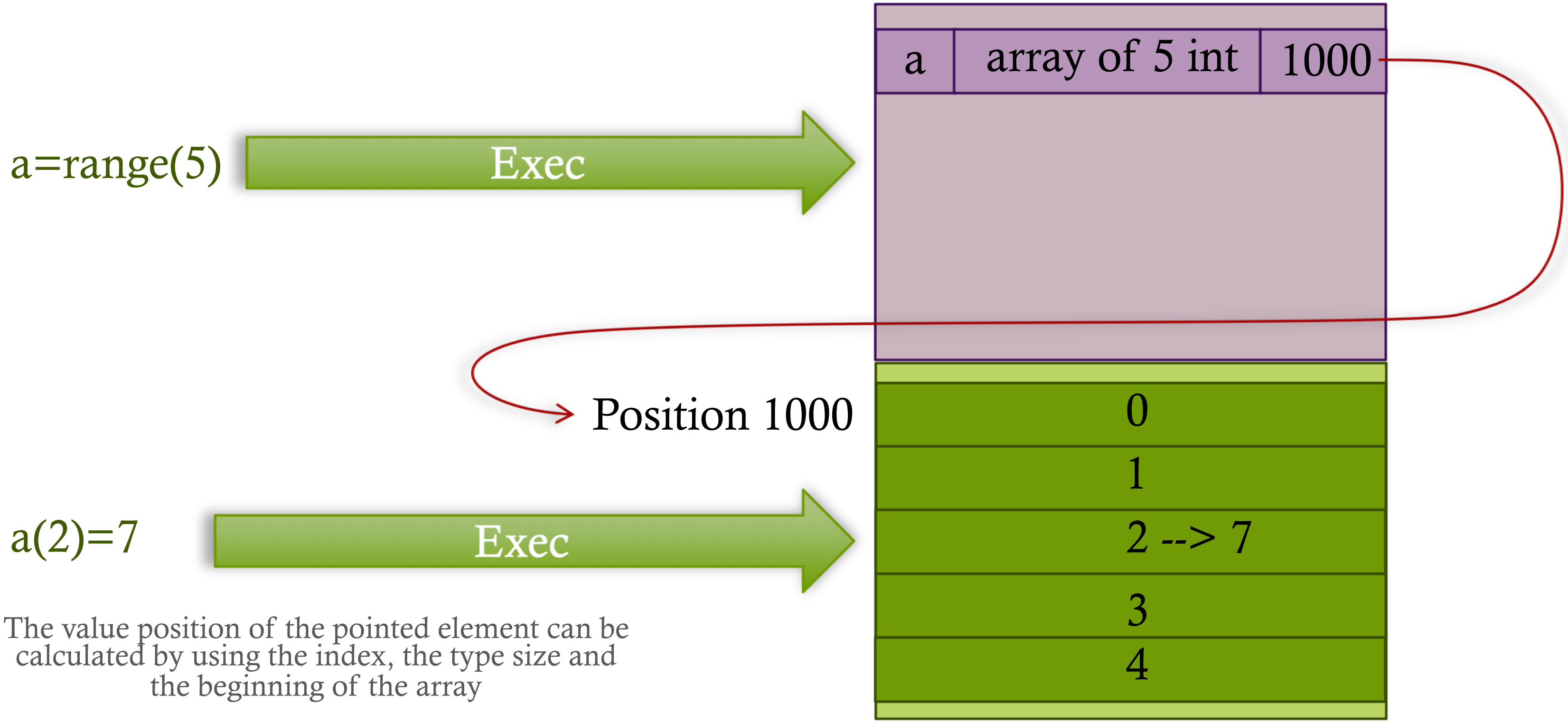
0	*0*	*0*	*0*	*0*	*1*	*1*	*1*

7_{10}

After the second assignment, this value continue to exists in memory *(and waste it)* but its no more accessible; it will be removed by *Garbage Collector*…. before or later.

Structured Variables

- A variable can be composed by several inner variables indexed by a *key* instead of a number

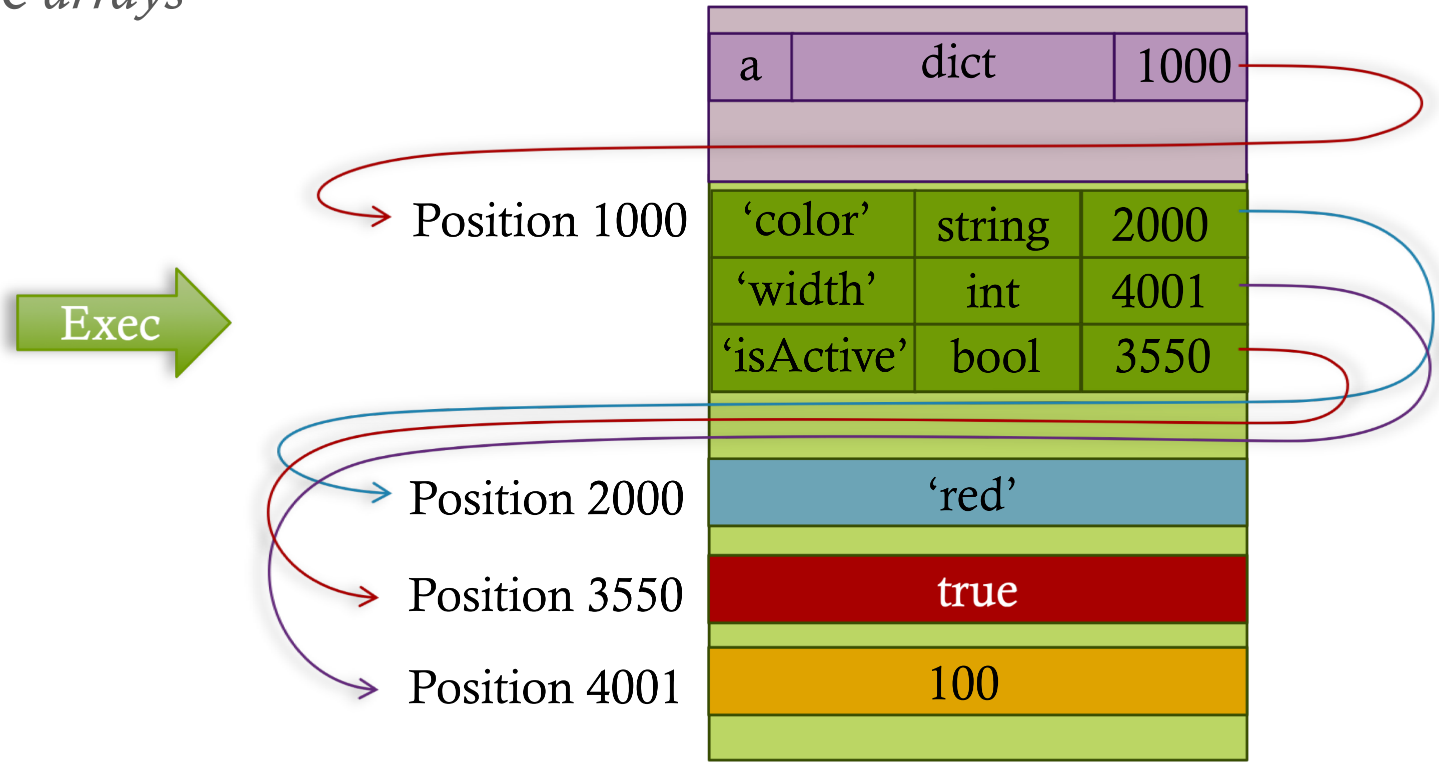


The value position of the pointed element can be calculated by using the index, the type size and the beginning of the array

Dictionaries

- A variable can be composed by several inner variables, for example *arrays*

```
a={
 'color' : 'red',
 'width': 100,
 'isActive': true
 }
```



a	dict	1000

Position 1000

'color'	string	2000
'width'	int	4001
'isActive'	bool	3550

Position 2000 'red'

Position 3550 true

Position 4001 100

Objects

- A Object can be viewed as a structured variable that bring also *actions* other then values.

- Objects are *instances* of a given *class* which defines the internal structures and the exposed values and actions.

The class IntList define an internal variable ,the_list, whioch is an array of int

The class IntList defines two method: append and show

```
# a brief pseudo-code
class IntList {
            the_list: array of int,
            append(x){  append_to_the_end(the_list,x) },
            show(){ if (the_list is empty) print "Empty"
                    else foreach e in the_list{ print e} }
            }

a_obj=new IntList()
a_obj.show()
```

a_obj is an instance of the class IntList

```
Empty
```

```
a_obj.append(10)
a_obj.append(3)
a_obj.append(4)
a_obj.show()
```

```
10
 3
 4
```

The method defined by the class IntList can be viewed as properties of the object a_obj: when a method is invoked, the execution context is the connected object, in this case the variable the_list used by the method is the one defined inside a_obj

Variables exemples in Python

```
i=4         #the type integer can contains any integer
x=2**200    #integers have no limits, the real memory occupation change follows whats needs
s=‘home’    #a sequence of char
x=[1,2,3]   #array of int
x[1]=10     #assignment of 10 to the second element of x: after this statement, it values [1,10,3]
```

!!!!!! Mutable Object !!!!!!!

In Python variables are objects. If a variable points to a **mutable object** Python only copy the pointer not the entire structure:

```
>>> x=[1, 2, 3]
>>> type(x)                    #x is pointer a mutable object of type «list»
<class 'list'>
>>> y=x                        #y is a copy of the pointer x
>>> type(y)
<class 'list'>
>>> x.append(4)                # the action append is applied to the object
>>> print(x)                   # the effect is visible both from x and y
[1, 2, 3, 4]                   # because they point to the same objec
>>> print(y)
[1, 2, 3, 4]
```

!!!!!! Mutable Object !!!!!!$_2$

```
>>> x=[1, 2, 3]
>>> type(x)
<class 'list'>
>>> y=x.copy()                #the method copy duplicate the object
>>> type(y)
<class 'list'>
>>> print(x)                  #x and y now point to different objects
[1, 2, 3]
>>> print(y)
[1, 2, 3]
>>> x.append(4)               #the method append change the first object
>>> print(x)
[1, 2, 3, 4]
>>> print(y)                  #the second object remain untouched
[1, 2, 3]
```