Address-of and dereference operators

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

& address-of operator

***** dereference operator

Address-of operator and dereferencing operator

the address of a variable :
address-of operator ***&***

the content at an address :
dereferencing operator *******

& variable *:*
returns the address of a variable

variable *has memory locations*
whose value can be changed
by an assignment

(***variable*** *must be an lvalue*)

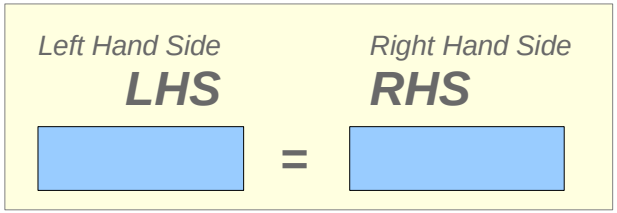
**** address*** *:*
returns the value at the address

**** address*** *has memory locations*
whose value can be changed
by an assignment

(**** address*** *is an lvalue*)

lvalue and rvalue in assignments

an assignment statement




```
int    a,  b = 10 ;
int  * p, q = &a ;
```

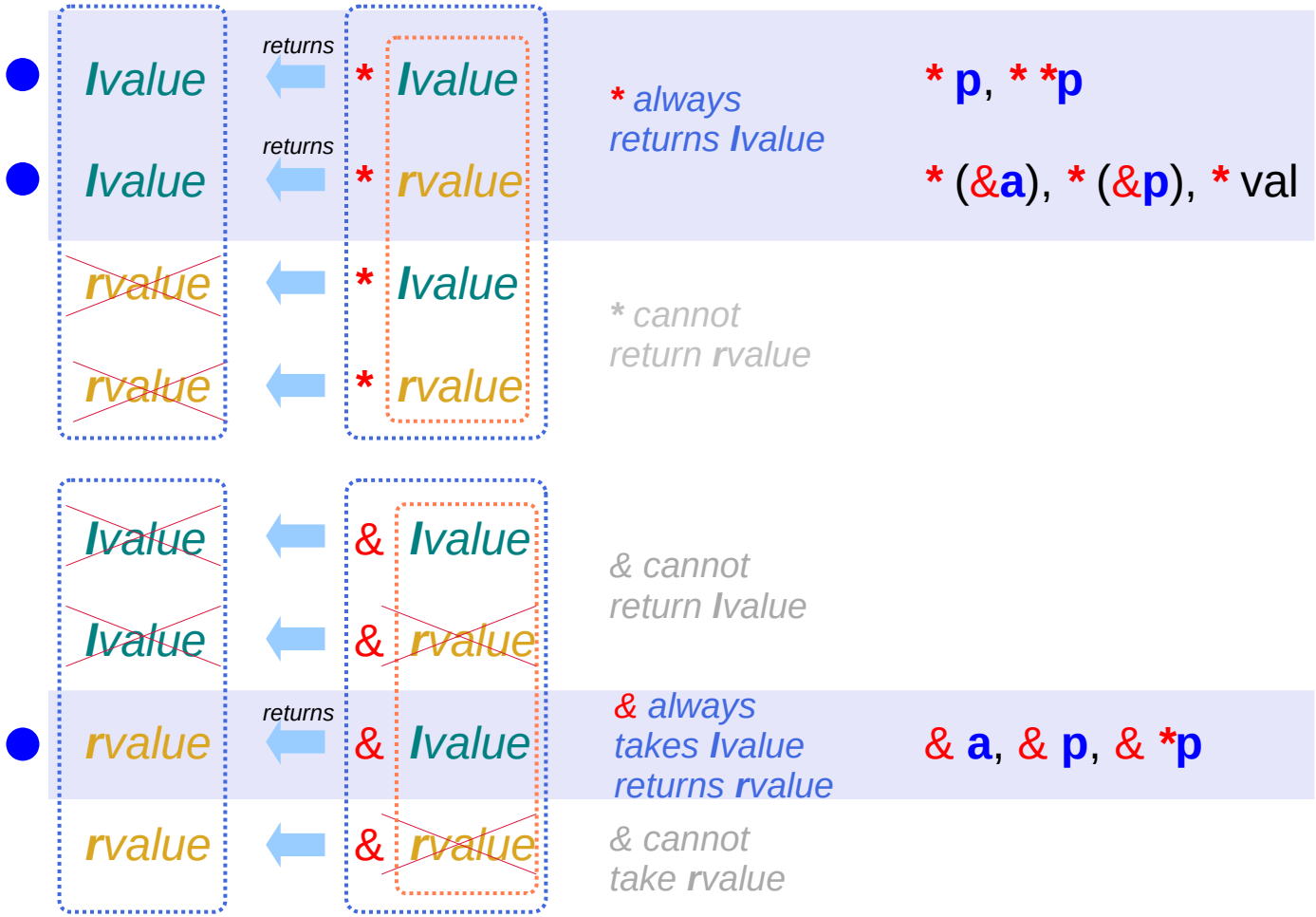
lvalue	=	lvalue
lvalue	=	rvalue
~~rvalue~~	=	lvalue
~~rvalue~~	=	rvalue

p	=	q ;
p	=	&a ;
~~&a~~	=	p ;
~~&a~~	=	&b ;

- *in the **LHS,** only lvalue can exist*
- *rvalue can exist only in the **RHS***

a, **b**, **p**, **q** : lvalues … variables … RW
***p**, ***q** : lvalues … variables … RW
&a, **&b** : rvalues … constants … RO

lvalue and rvalue with * and & operators




```
int    a = 10 ;
int  *  p = &a ;
```

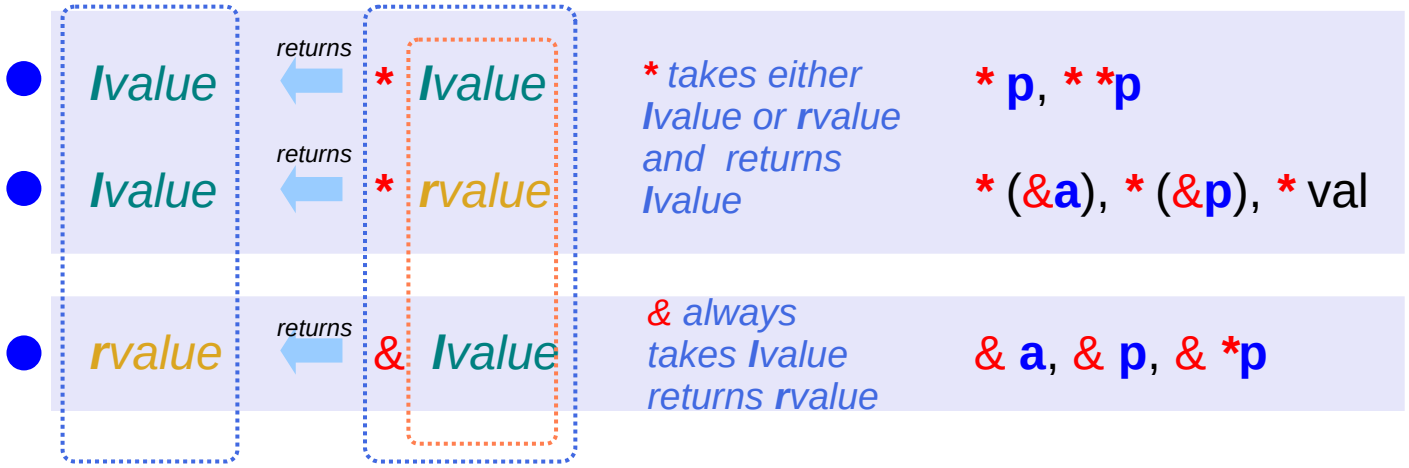
* can be applied to either an **lvalue** variable or a **rvalue** address

* **operand** becomes an **lvalue** variable thus can be applied successively.

& can be applied to only an **lvalue** variable and returns only an **rvalue** address

a, **p** : lvalues … variables … RW
***p** : lvalues … variables … RW
&a : rvalues … constants … RO

lvalue and rvalue with * and & operators




```
int     a = 10 ;
int  *  p = &a ;
```

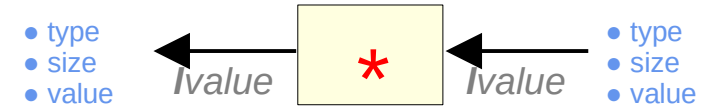
* can can take either an **lvalue** variable or a **rvalue** address

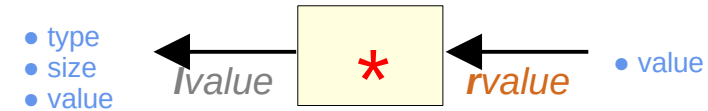
* **operand** acts as an **lvalue** variable

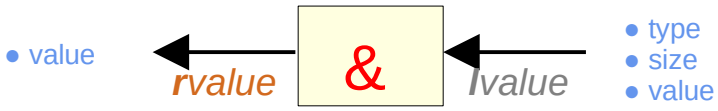
* can be applied successively.

& can take only an **lvalue** variable and returns only an **rvalue** address

& cannot be applied successively.

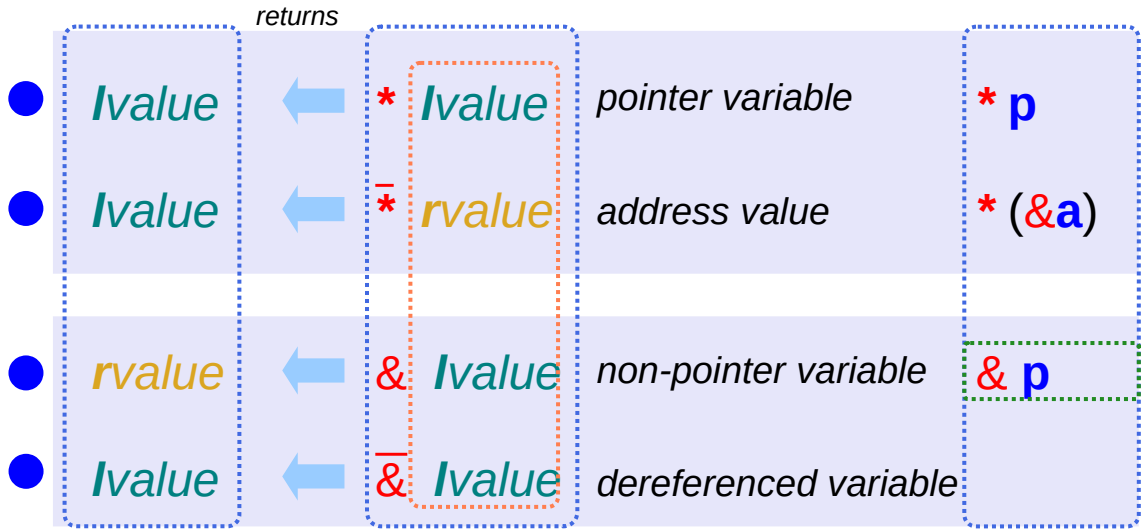


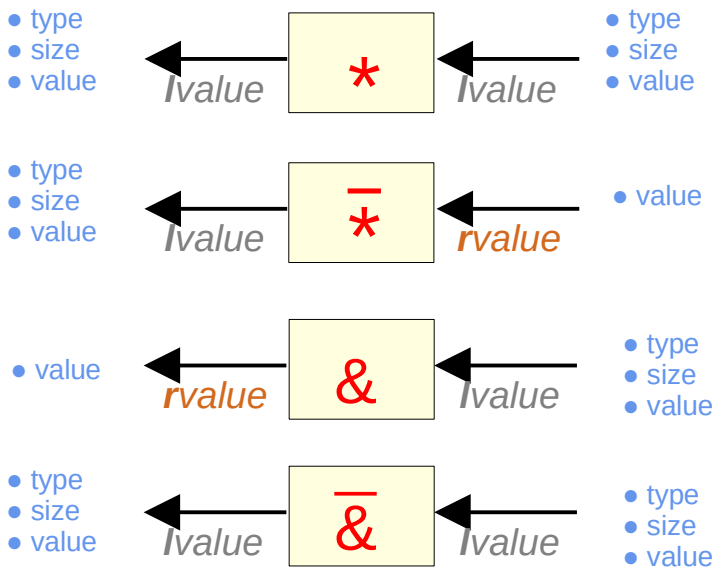




a, **p** : lvalues … variables … RW
***p** : lvalues … variables … RW
&a : rvalues … constants … RO

lvalue and rvalue with * and & operators






```
int    a = 10 ;
int  *  p = &a ;
```

* can be applied
to either an **lvalue** variable
or a **rvalue** address

* **operand** becomes
an **lvalue** variable
thus can be applied
successively.

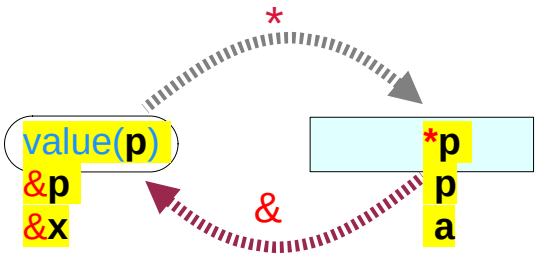
& can be applied
to only an **lvalue** variable and
returns only an **rvalue** address

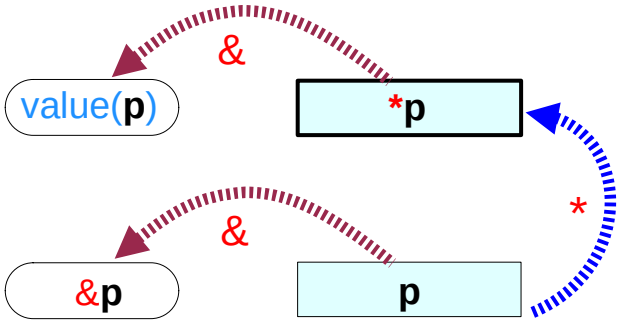
a, **p** : lvalues … variables … RW
***p** : lvalues … variables … RW
&a : rvalues … constants … RO

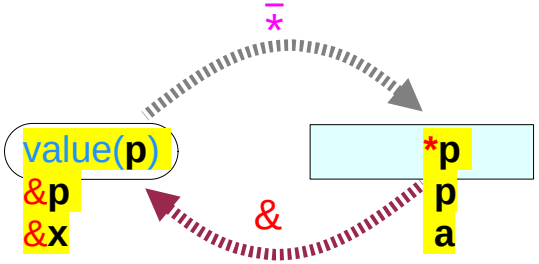
* and & c operators

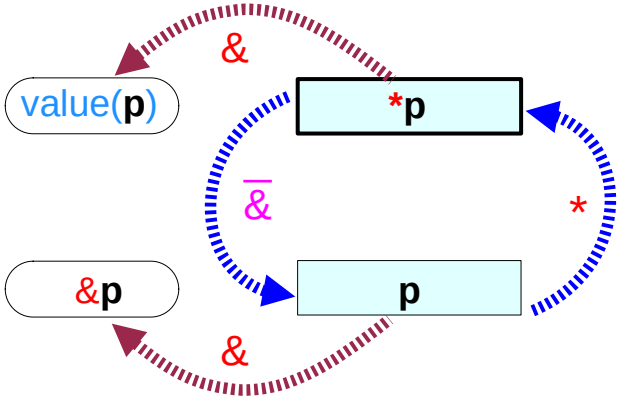
a	non-pointer variables	*lvalue*
p	pointer variables	*lvalue*
***p**	dereferenced variables	*lvalue*
val	address values	*rvalue*

* and & c operators





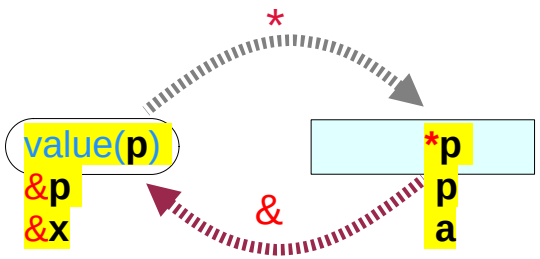


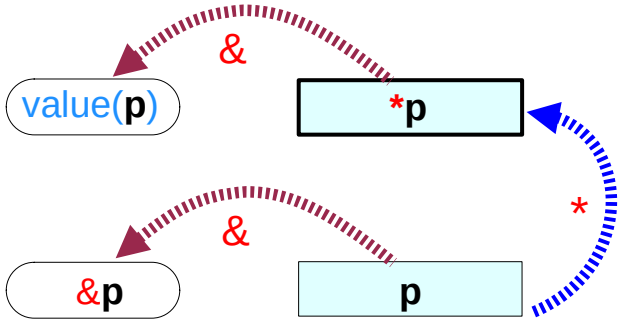


* and & c operators

	×	a	lvalue
lvalue ←	*	p	lvalue
lvalue ←	*	* p	lvalue
lvalue ←	*	val	rvalue

rvalue ←	&	a	lvalue
rvalue ←	&	p	lvalue
rvalue ←	&	* p	lvalue
	×	val	rvalue

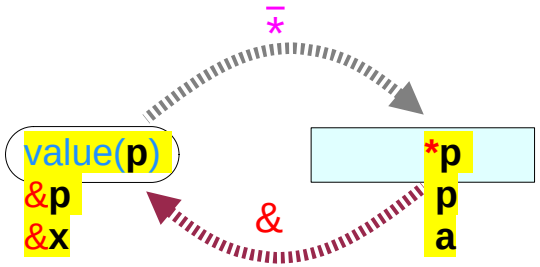


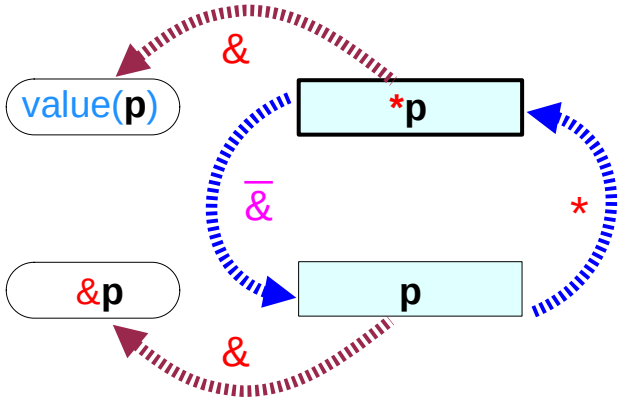


Augmented $\bar{*}$ and $\bar{\&}$ math operators

	×	a	lvalue
lvalue ←	*	p	lvalue
lvalue ←	*	* p	lvalue
lvalue ←	$\bar{*}$	val	rvalue

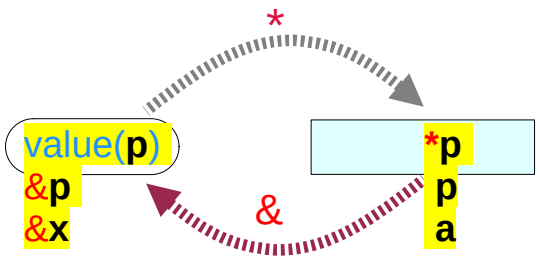
rvalue ←	&	a	lvalue
rvalue ←	&	p	lvalue
rvalue ←	&	* p	lvalue
lvalue ←	$\bar{\&}$	* p	lvalue
	×	val	rvalue

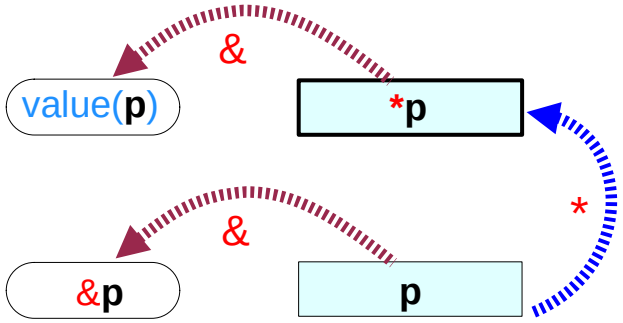


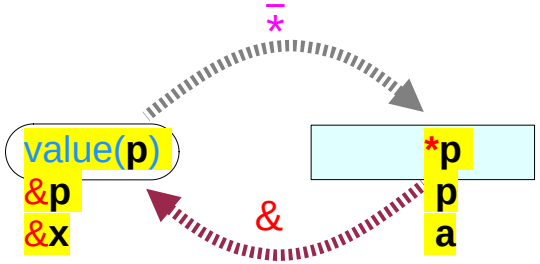


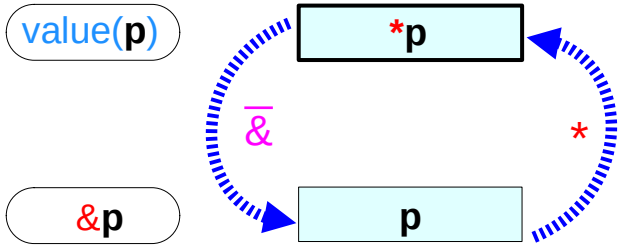
Comparisons (1)

a	non-pointer variables	*lvalue*
p	pointer variables	*lvalue*
***p**	dereferenced variables	*lvalue*
val	address values	*rvalue*









Comparisons (2)

	×	a	lvalue
lvalue ←	*	p	lvalue
lvalue ←	*	* p	lvalue
lvalue ←	*	val	rvalue

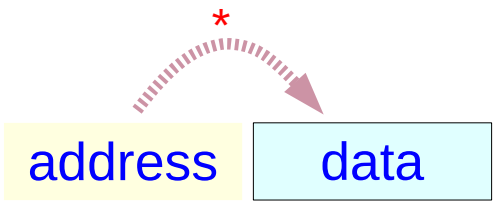
rvalue ←	&	a	lvalue
rvalue ←	&	p	lvalue
rvalue ←	&	* p	lvalue
	×	val	rvalue

	×	a	lvalue
lvalue ←	*	p	lvalue
lvalue ←	*	* p	lvalue
lvalue ←	$\overline{*}$	val	rvalue

			lvalue
rvalue ←	&	a	lvalue
rvalue ←	&	p	lvalue
lvalue ←	$\overline{\&}$	* p	rvalue
	×	val	

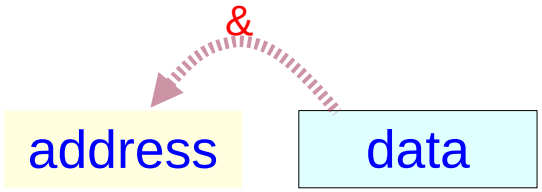
Address-of & and dereference * C operators (1)

Primitive Data Type



rvalue &a → *lvalue* a
constant *variable*

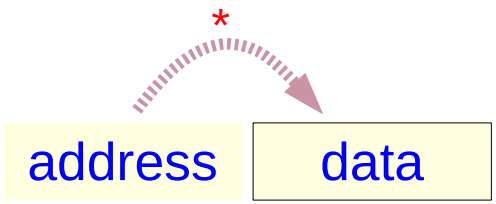
lvalue p → *lvalue* *p
pointer *variable*



rvalue &a ← *lvalue* a
constant *variable*

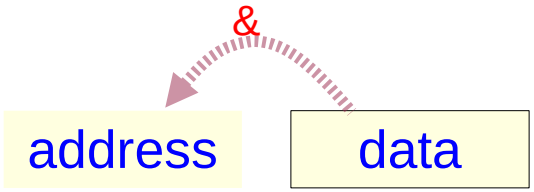
~~*lvalue*~~ p ← *lvalue* *p
pointer *variable*

Pointer Data Type



rvalue &q → *lvalue* q
constant *double pointer*

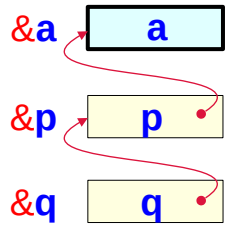
lvalue q → *lvalue* *q
double pointer *pointer*

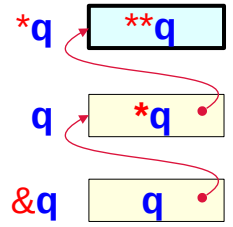


rvalue &q ← *lvalue* q
constant *double pointer*

~~*lvalue*~~ q ← *lvalue* *q
double pointer *pointer*

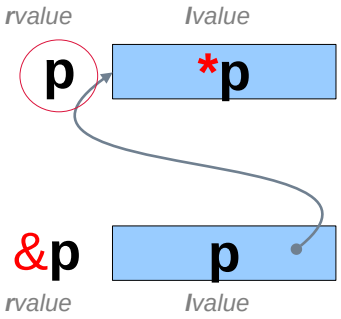
```
int   a ;
int *  p ;
int ** q ;
```





& and * operators in pointer de-referecing

Two step De-reference * operation

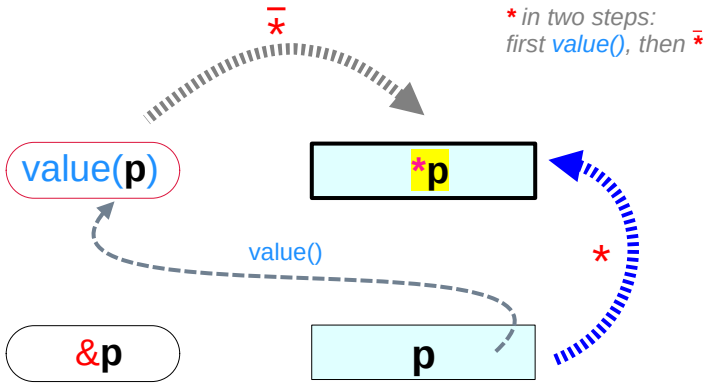


p *must be evaluated to an address* *value(**p**) before de-referencing* ***p***

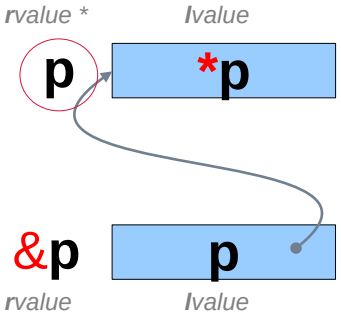
think a new operator :

$\overline{*}$ is a mathematical operator

$\overline{*}$ takes rvalue and returns lvalue



Two step Address-of & operation

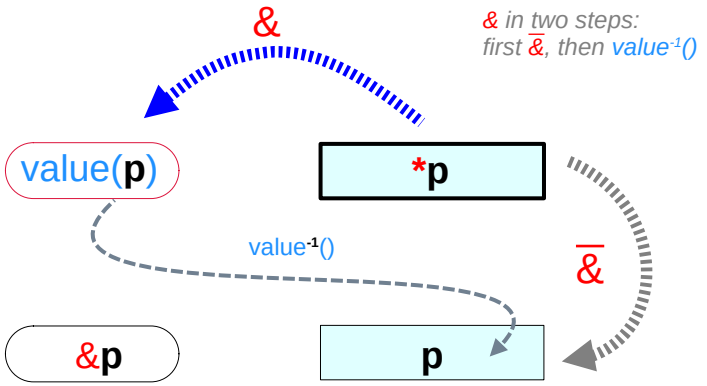


assume that it is possible
to recover the variable ***p*** *from value(**p**)*

think a new operator :

$\overline{\&}$ is a mathematical operator

$\overline{\&}$ takes lvalue and returns lvalue



Recursive application of the address-of operator

&(&(&(c[i])[j])[k])

$\overline{\&}(\overline{\&}(\overline{\&}(c[i])[j])[k])$

& C operator

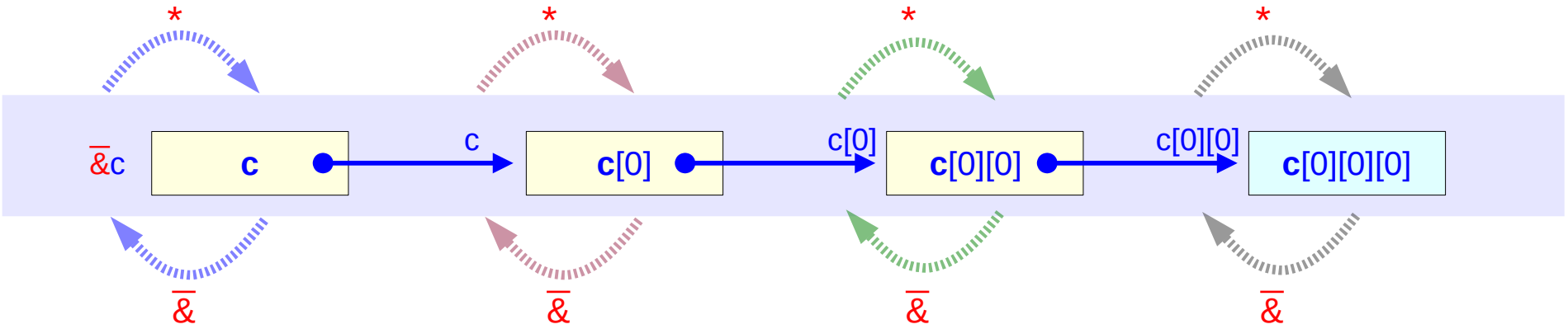
can be applied to only **lvalue** variable
returns address value
thus, the above expression is not possible

successive application of & is not possible

In constrast, ***p** becomes a lvalue variable
* operator can be applied successively.

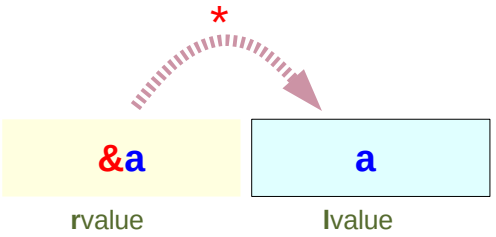
$\overline{\&}$ mathematical operator

Two step deferencing in type **II** (1) – without skipping



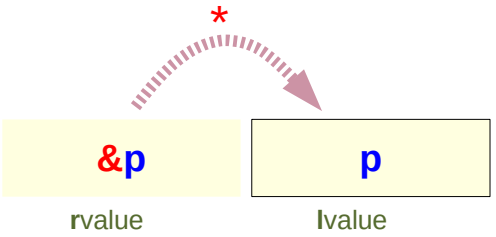
& : **mathematical & operator**

Type, Size, and Value attributes of an **lvalue**



lvalue ⟵ ***rvalue***

a ****&a***



lvalue ⟵ ***rvalue***

p ****&p***

lvalue is associated with a memory location

lvalue has the following attributes
- Type
- Size
- Value

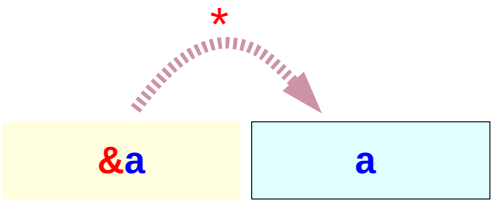
rvalue has the only attribute
- Value

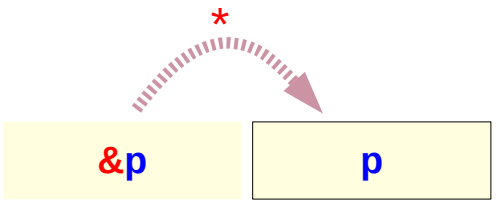
assume the function value()

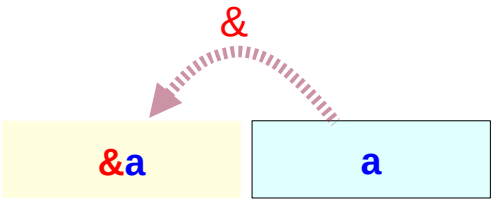
value(**lvalue**) returns
the Value attribute of **lvalue**

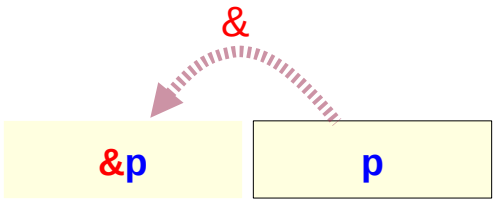
value(**rvalue**) returns
the Value attribute of **rvalue**

Address-of & and dereference * C operators (1)





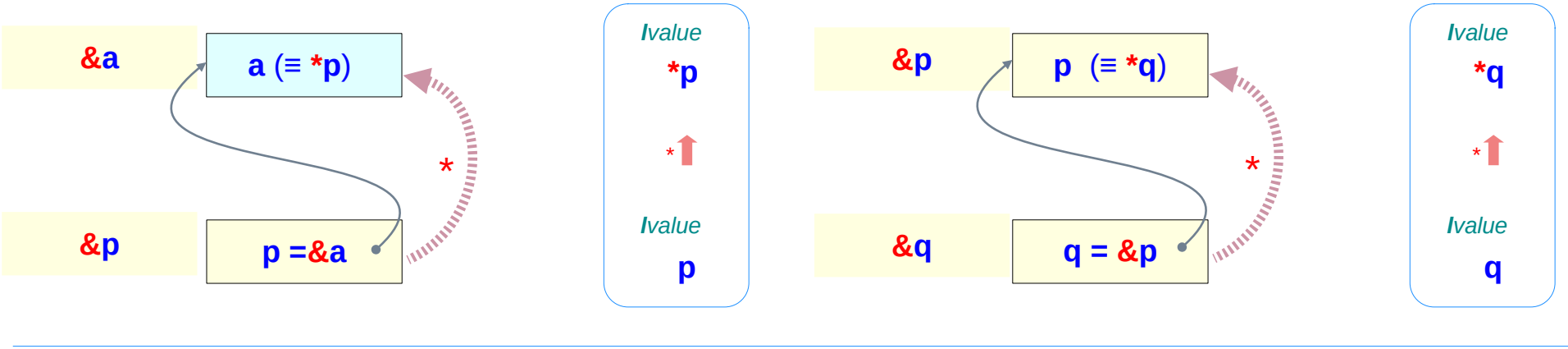


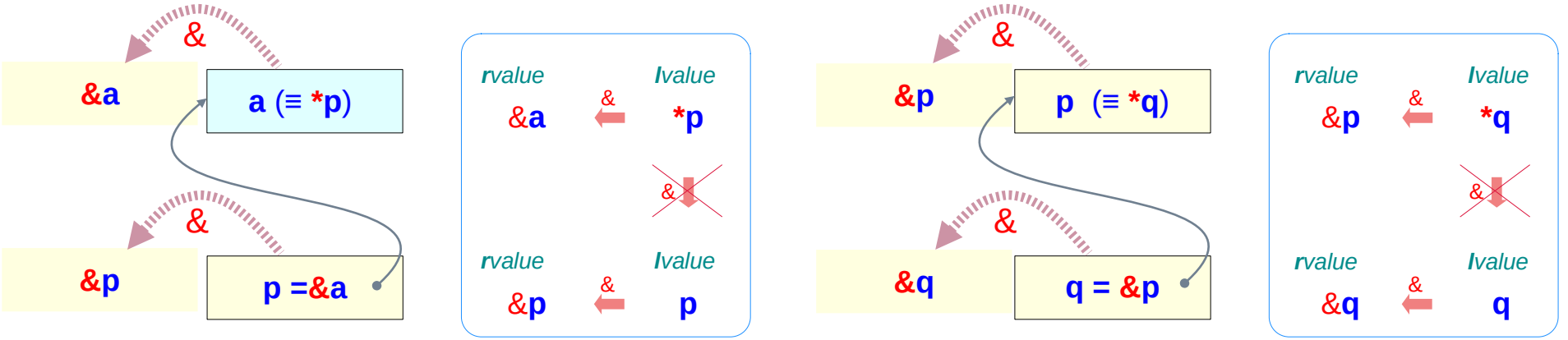


*&a = a

*&p = p

Address-of & and dereference * C operators (2)





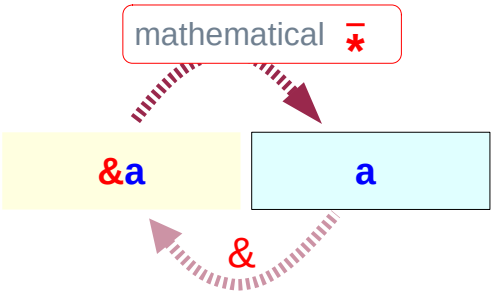
*&p = p

~~&*p = p~~ value(p)

*&q = q

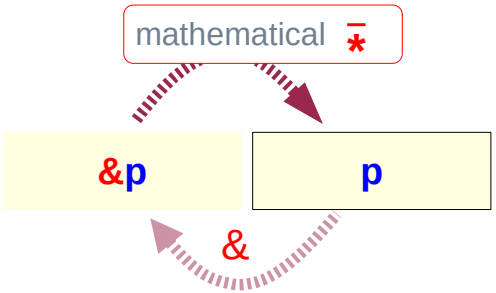
~~&*q = q~~ value(q)

Introducing mathematical operators : $\overline{\&}$ and $\overline{*}$



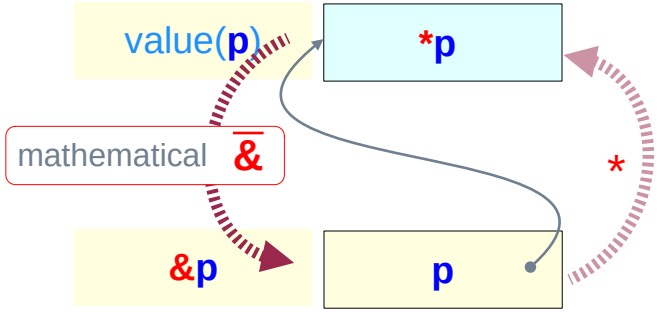
$\overline{*}$ &a = a

&a



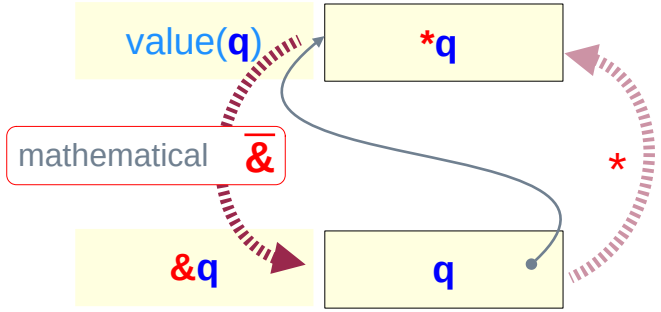
$\overline{*}$ &p = p

&p



*p

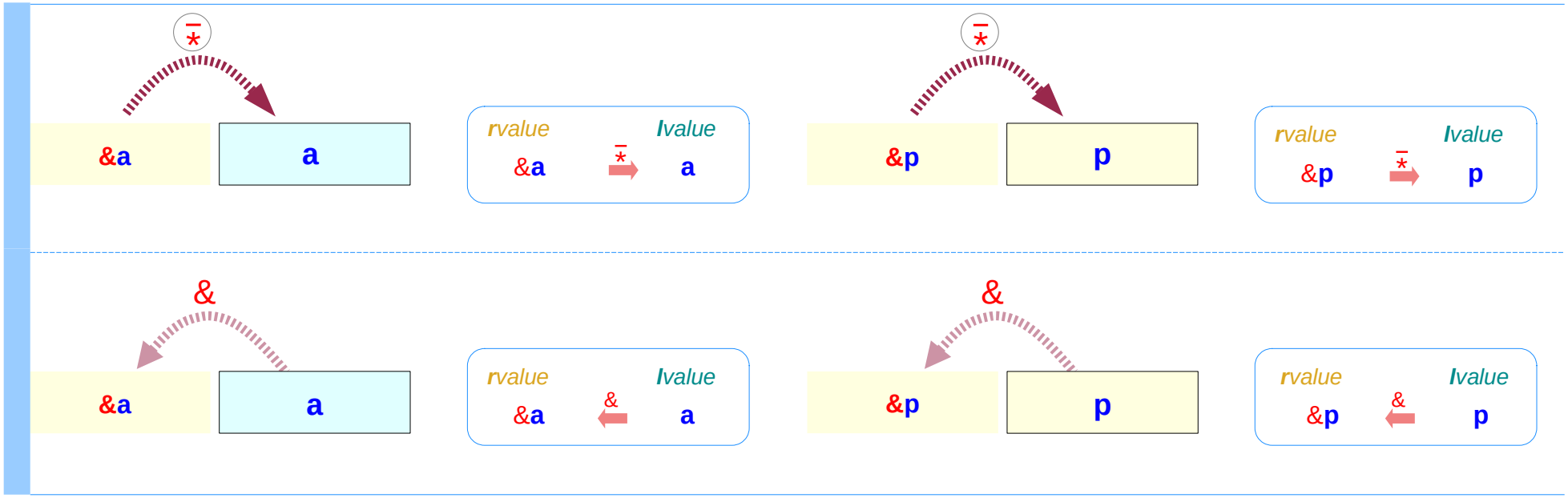
$\overline{\&}$ *p = p



*q

$\overline{\&}$ *q = q

Inverse operators *̄ and &



*̄&a = a

&*̄&a = &a

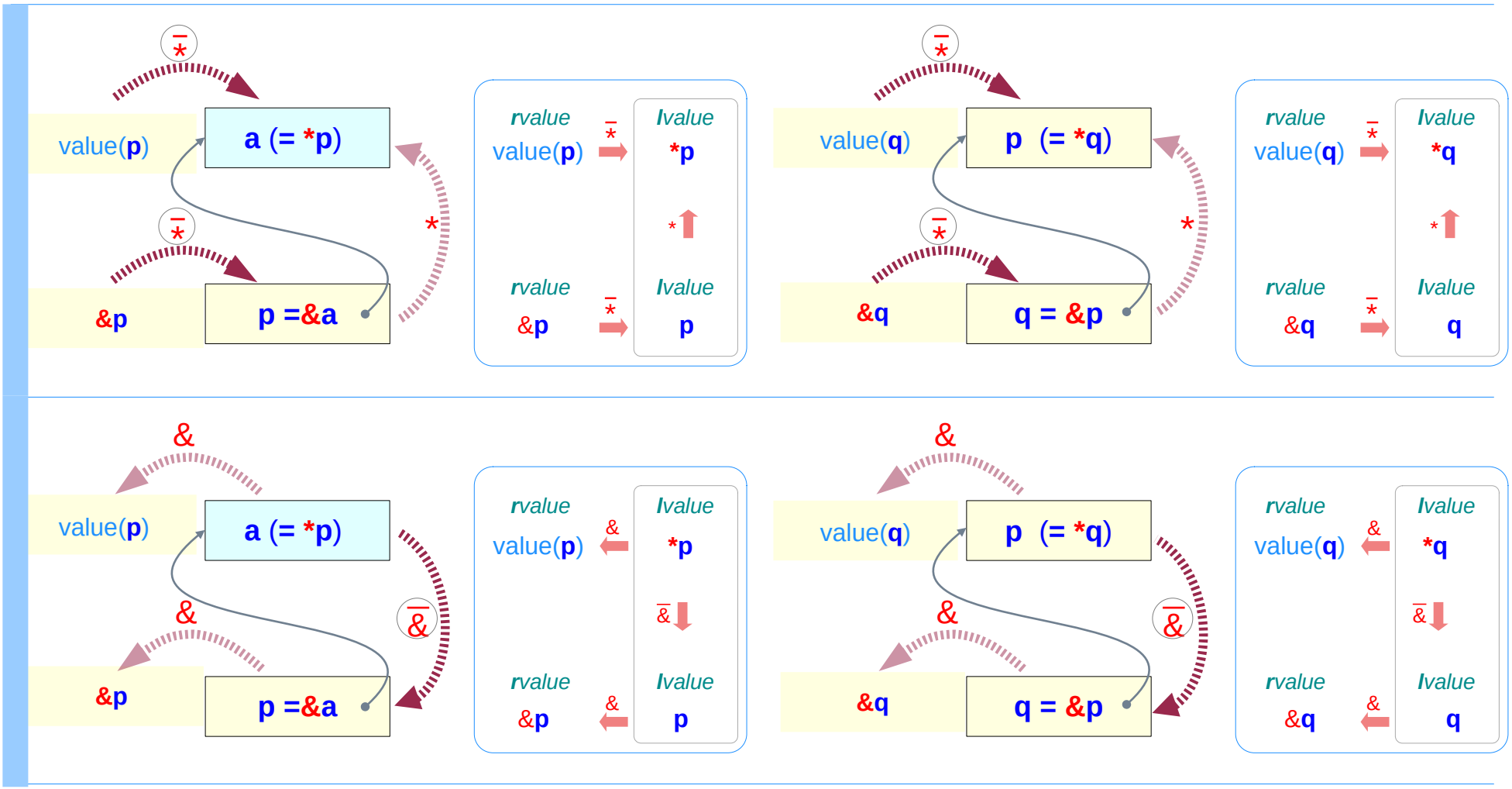
*̄ and & are inverse operators to each other

*̄&p = p

&*̄&p = &p

*̄ and & are inverse operators to each other

Inverse operators $\overline{\&}$ and *



$\overline{*}\&p = p$

$\&\overline{*}p = p$

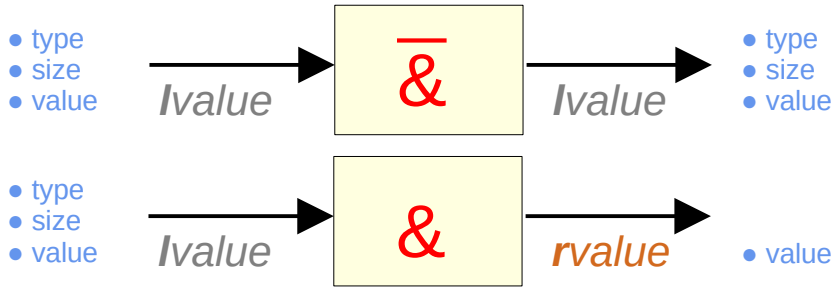
$\overline{*}\&q = q$

$\&\overline{*}q = q$

C operators and mathematical operators

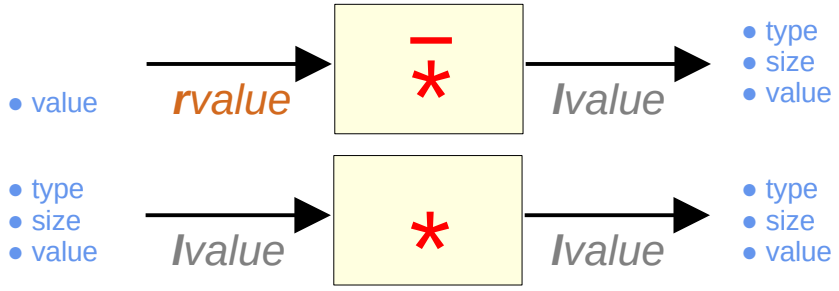
Address-of operation

&x	=	value(&x)
C Expressions		Mixed Expressions

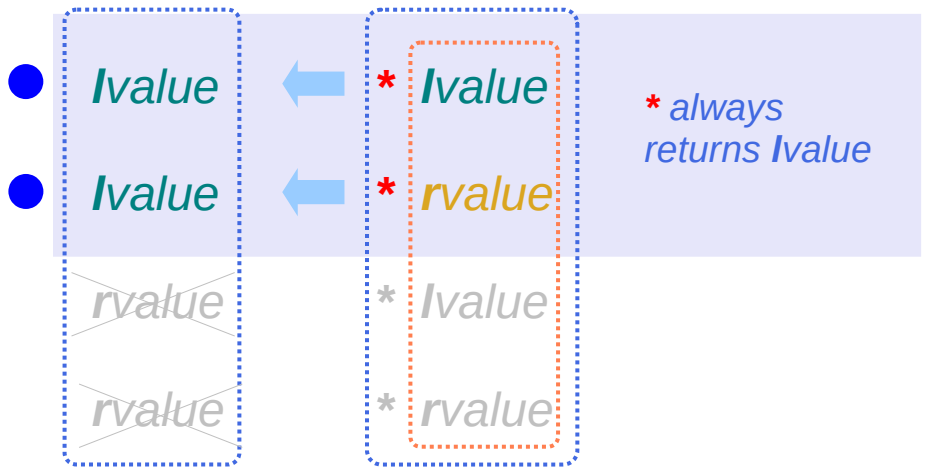


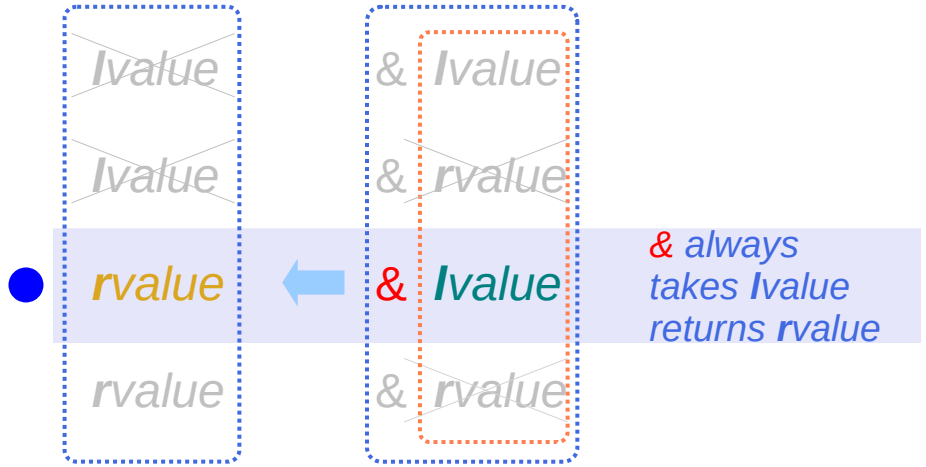
De-reference operation

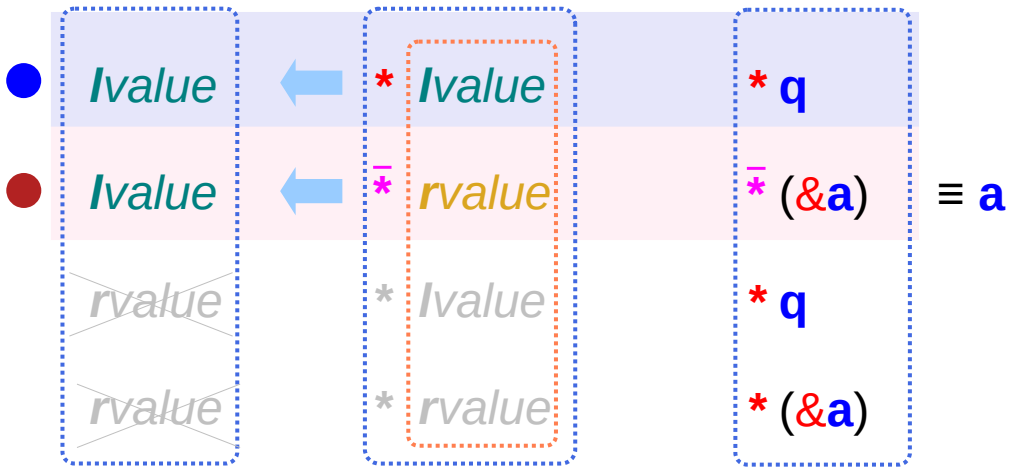
*p	=	*value(p)
C Expressions		Mixed Expressions

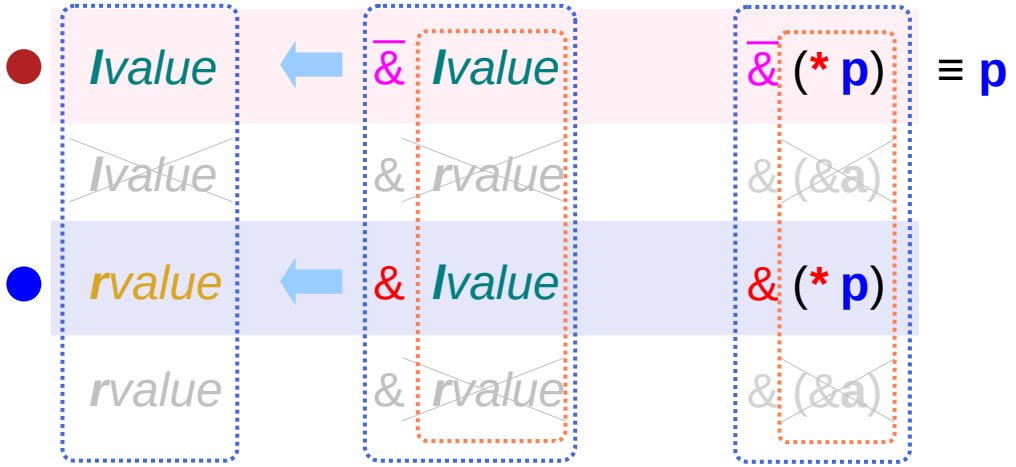


lvalue and rvalue with * and & operators









a, **p**, **q** : lvalues … variables … RW
***p**, ***q**, ****q** : lvalues … variables … RW
&a, **&p**, **&q** : rvalues … constants … RO

Examples of inverse operators

$\bar{*}\&a \Rightarrow a$

$\bar{*}\&p \Rightarrow p$

$\bar{*}\&q \Rightarrow q$

$\bar{*}value(p) \Rightarrow *p \Rightarrow a$

$\bar{*}value(q) \Rightarrow *q \Rightarrow p$

$\bar{*}value(*q) \Rightarrow **q \Rightarrow *p \Rightarrow a$

$\bar{\&}*p \Rightarrow p$

$\bar{\&}*q \Rightarrow q$

$\bar{\&}**q \Rightarrow *q$

Extended Operators

$*\&a \Rightarrow a$

$*\&p \Rightarrow p$

$*\&q \Rightarrow q$

$*p \Rightarrow a$

$*q \Rightarrow p$

$**q \Rightarrow *p \Rightarrow a$

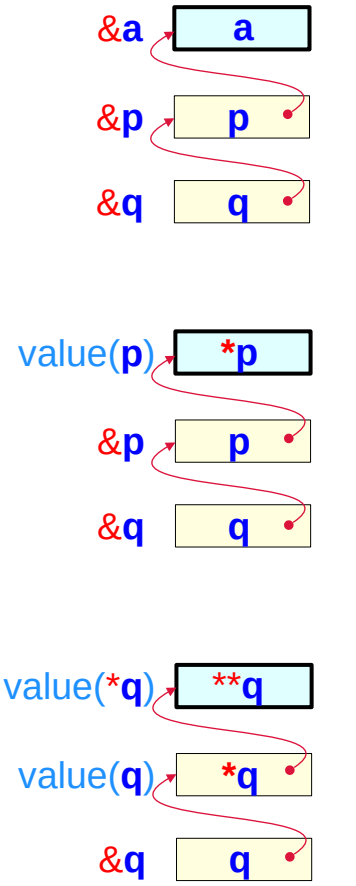
$\&*p \Rightarrow \&a$

$\&*q \Rightarrow \&p$

$\&**q \Rightarrow \&a$

C Operators

```
int    a ;
int *  p = & a ;
int ** q = & p ;
```



Operands of mathematical operators : & and *

* address value

& * value(**P**) ➡ value(**P**)

* & variable

* &**X** ➡ **X**

& * pointer

dereferenced pointer

& *P ➡ **P**

& ~~pointer~~

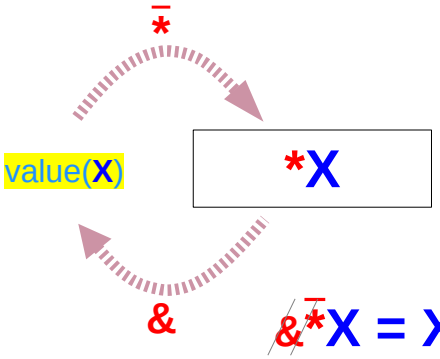
must be a dereferenced pointer

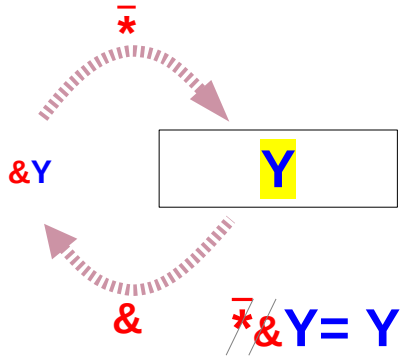
not considering
simultaneous pointing

* &**Q** ➡ **Q**

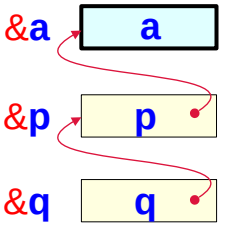
Q must be a dereferenced pointer
i.e, **Q = *p** ➡ &**Q = p**

& and mathematical $\bar{*}$






```
int    a ;
int *  p = & a ;
int ** q = & p ;
```



& *value(**a**) = value(**a**)

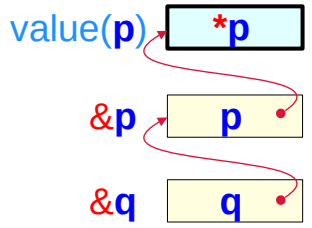
& *value(**p**) = value(**p**)

& *value(**q**) = value(**q**)

* &**a** = **a**

* &**p** = **p**

* &**q** = **q**



& ***a** = value(**a**)

& ***p** = value(**p**)

& ***q** = value(**q**)

* &**a** = **a**

* &**p** = **p**

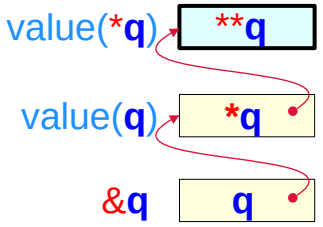
* &**q** = **q**

* &**a** = **a**

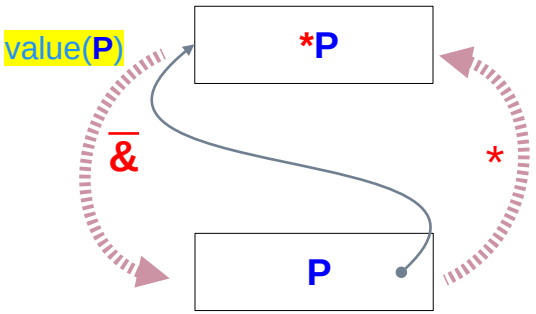
* &**p** = **p**

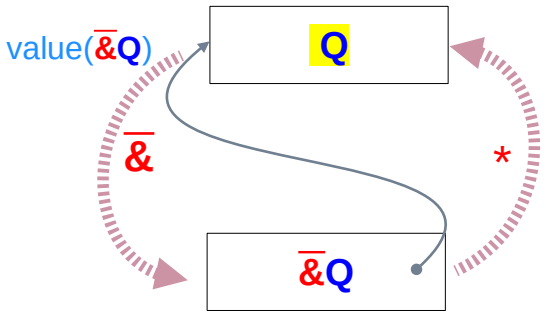
* &**q** = **q**

C expressions



* and mathematical &





$\overline{\&}{}^{*}P = P$

if not considering simultaneous pointing

${}^{*}\overline{\&}Q = Q$

$\overline{\&}\ {}^{*}p = p$

$\overline{\&}\ {}^{*}q = q$

$\&\ {}^{*}p = value(p)$

$\&\ {}^{*}q = value(q)$

${}^{*}\ \overline{\&}{}^{*}p = {}^{*}p$

${}^{*}\ \overline{\&}{}^{*}q = {}^{*}q$

$\overline{{}^{*}}\ \&{}^{*}p = {}^{*}p$

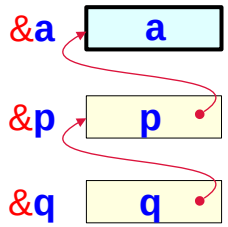
$\overline{{}^{*}}\ \&{}^{*}q = {}^{*}q$

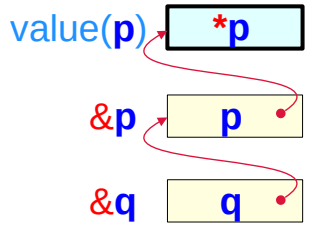
$* \ \&{}^{*}p = {}^{*}p$

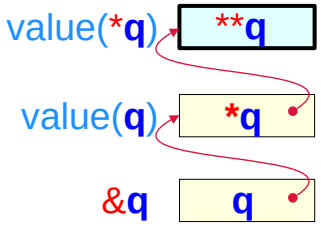
$* \ \&{}^{*}q = {}^{*}q$

C operators

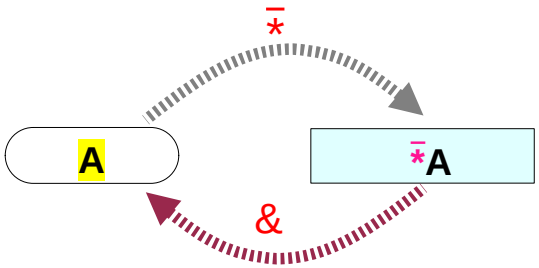
```
int    a ;
int *  p = & a ;
int ** q = & p ;
```





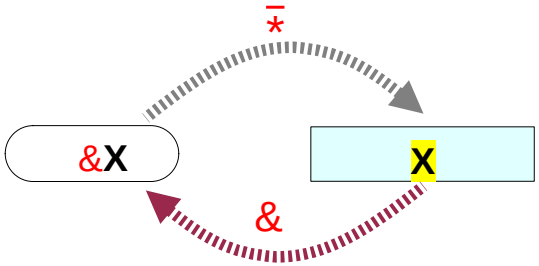


Requirements of address and data



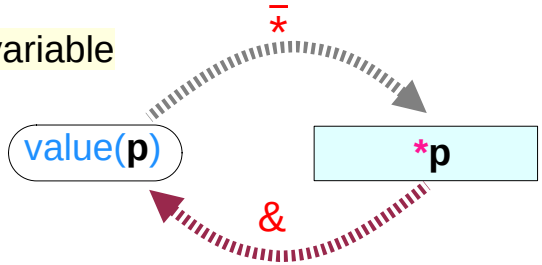
A : an address value - a number

value(**p**), &**p**, &**x**



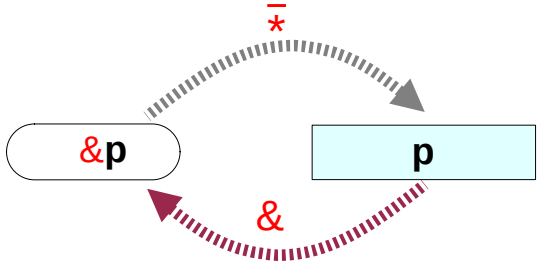
X : a variable ***p**, **p**, **a**

***p :** a dereferenced variable



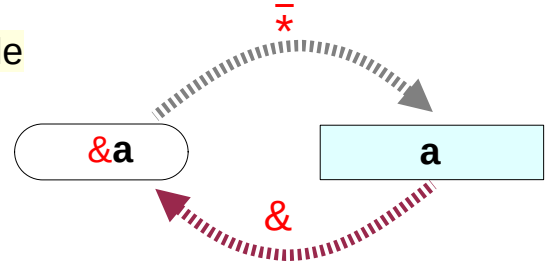
Pointer Dereferencing

p : a pointer variable



Pointer Data

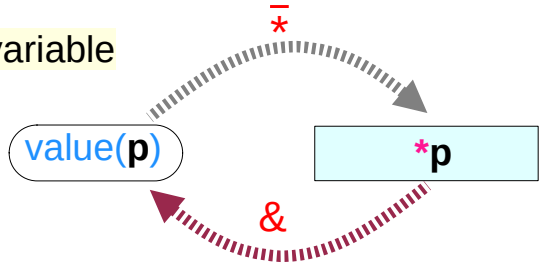
a : a primitive variable



Non-pointer Data

Inverse relations of & and $\bar{*}$ operators

***p :** a dereferenced variable

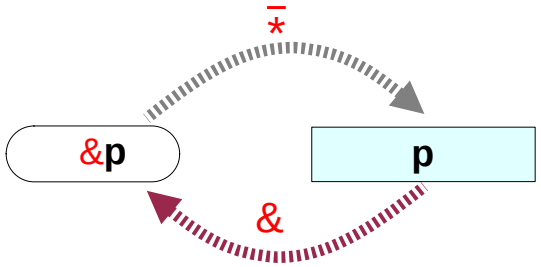


$\bar{*}$ value(**p**) = ***p**

$\&\bar{*}$ value(**p**) = & ***p** = value(**p**)

$\bar{*}\&$ $\bar{*}$value(**p**) = $\bar{*}\&$ ***p** = $\bar{*}$value(**p**) = ***p**

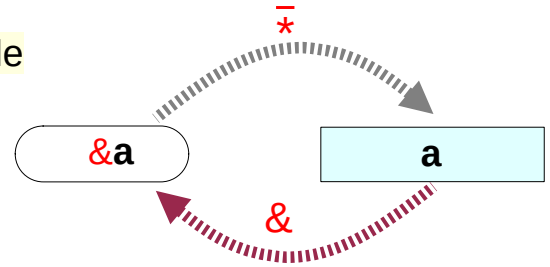
p : a pointer variable



$\bar{*}$ &**p** = **p**

$\&\bar{*}$ &**p** = &**p**

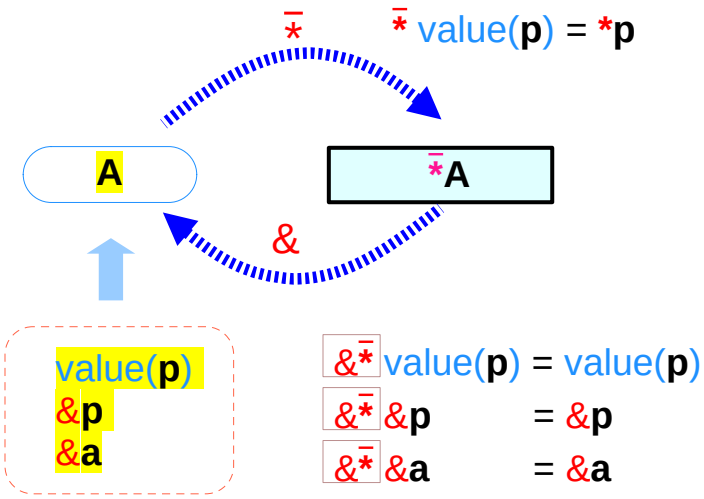
a : a primitive variable

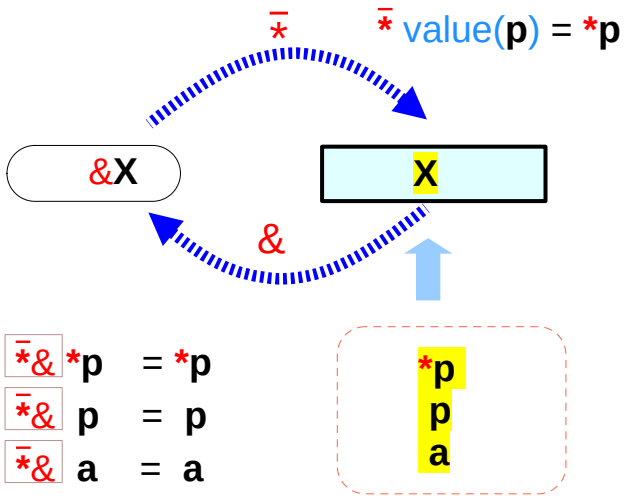


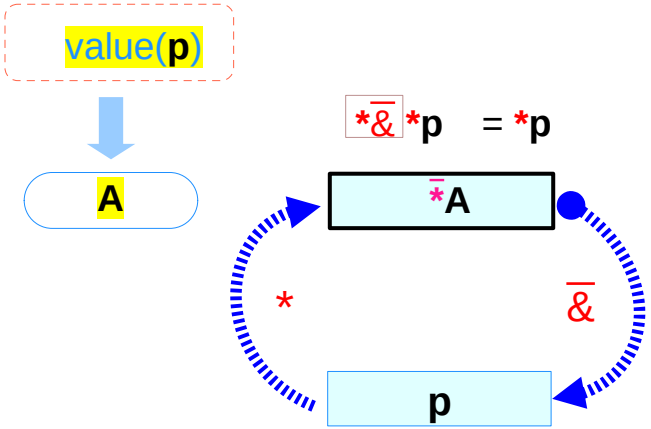
$\bar{*}$ &**a** = **a**

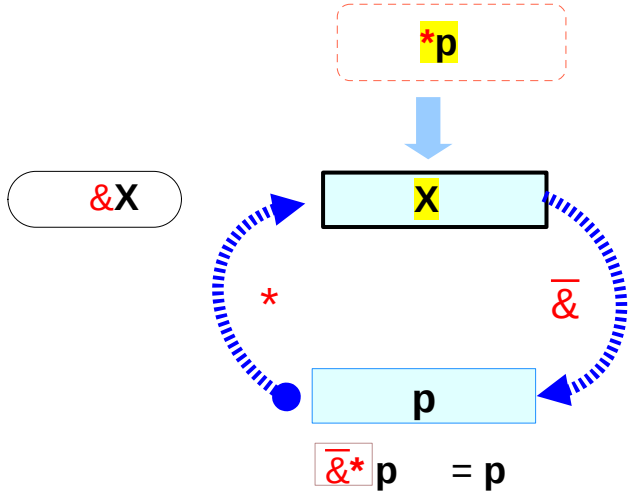
$\&\bar{*}$ &**a** = &**a**

Requirements of pointed address and data



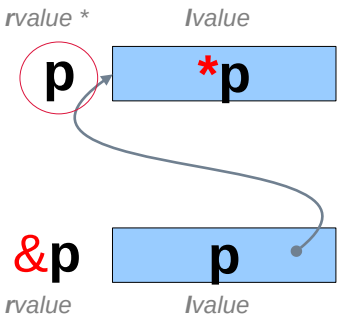




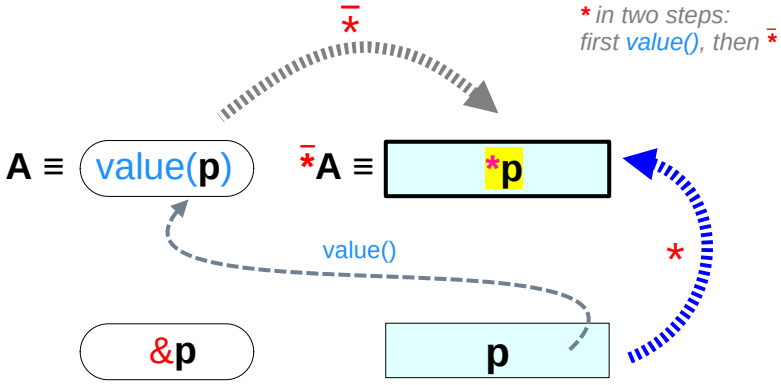


& and * operators in pointer de-referecing (1)

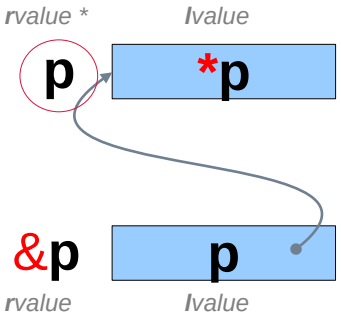
Two step De-reference * operation



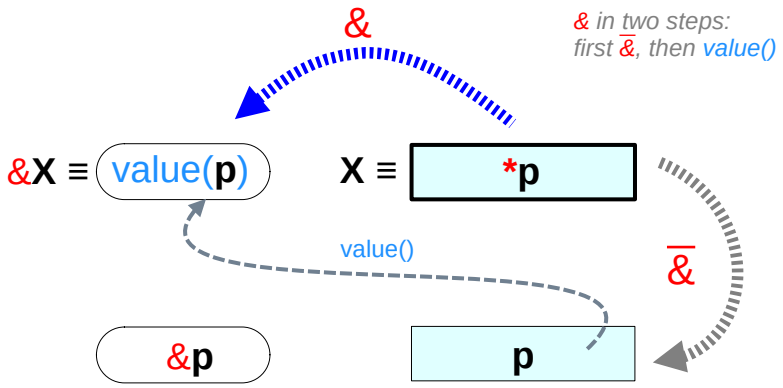
*p	=	$\bar{*}$value(p)
C Expressions		Mixed Expressions
$\bar{*}$A	=	*value^{-1}(A)



Two step Address-of & operation

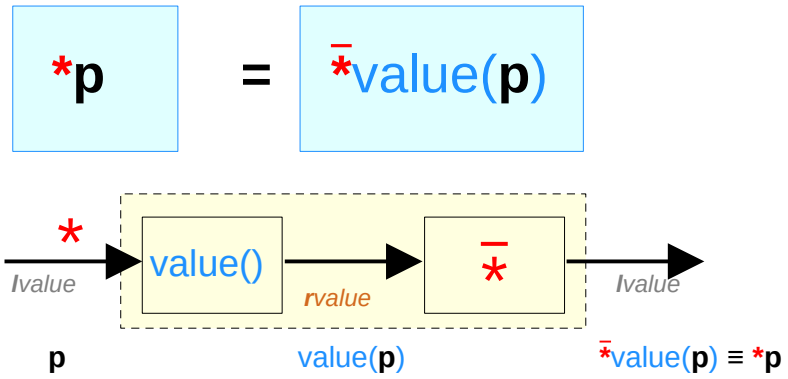


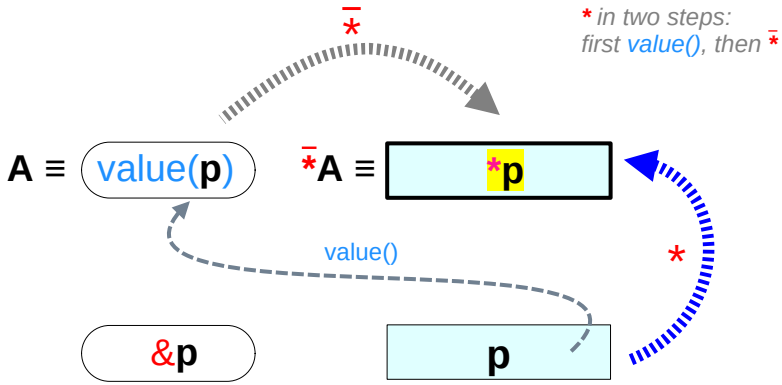
&X	=	value($\bar{\&}$X)
C Expressions		Mixed Expressions
$\bar{\&}$X	=	value^{-1}(&X)



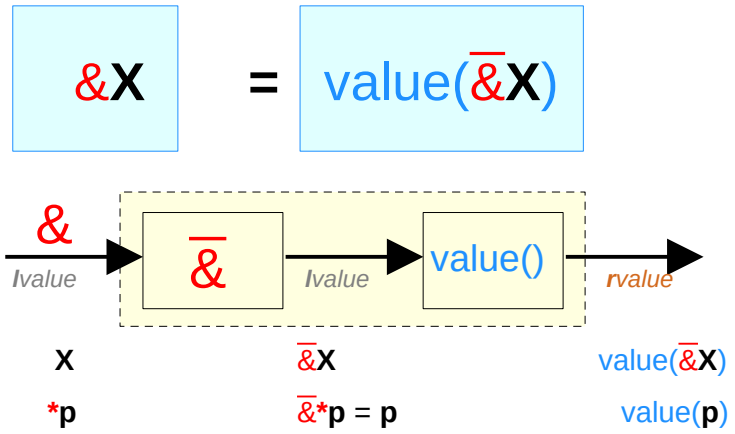
& and * operators in pointer de-referecing (2)

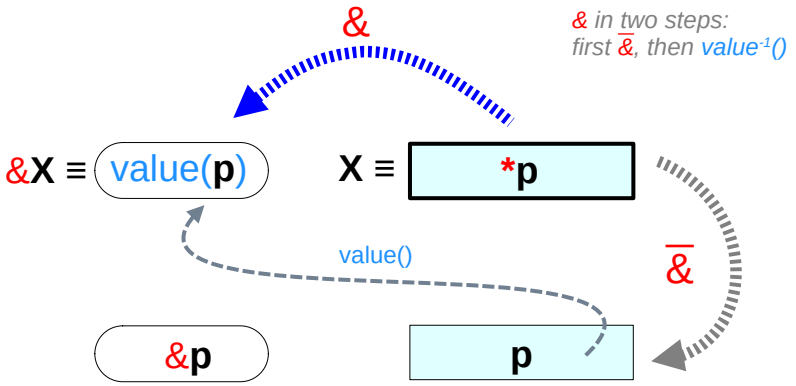
Two step De-reference * operation





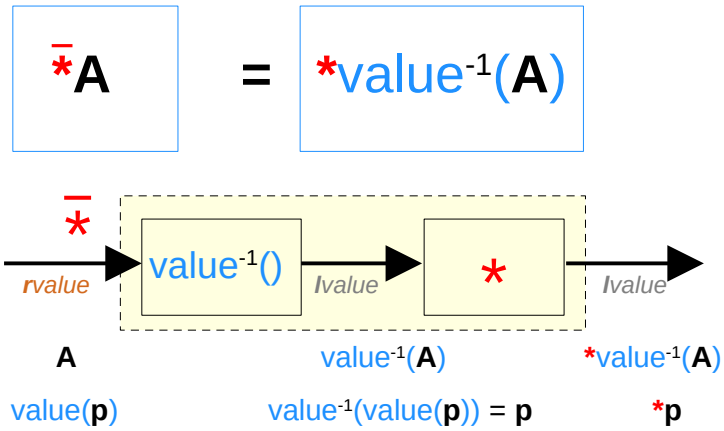
Two step Address-of & operation

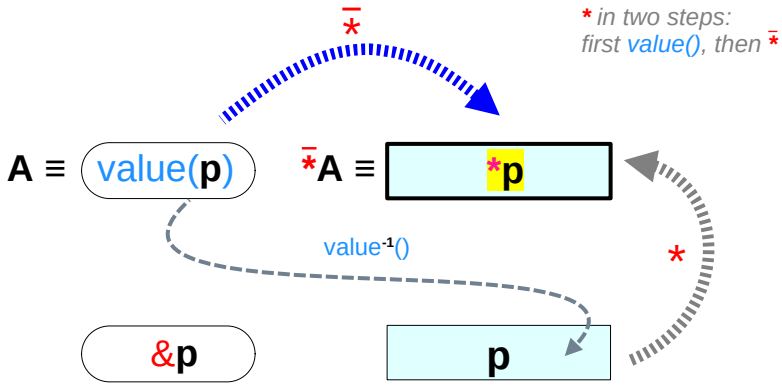




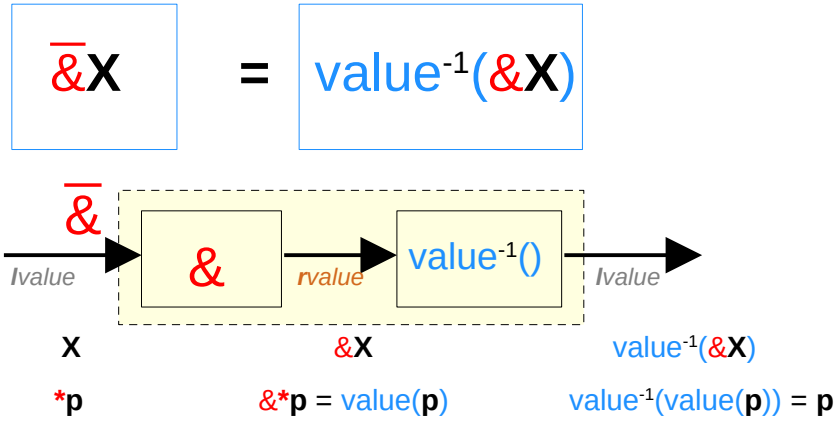
& and * operators in pointer de-referecing (3)

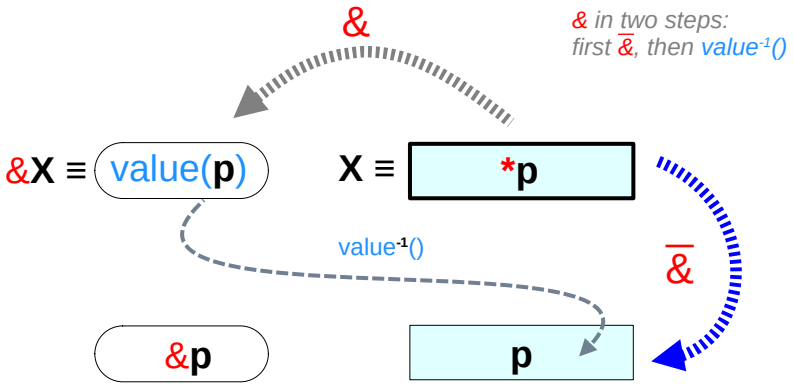
Two step De-reference * operation





Two step Address-of & operation





Two step address-of & and $\overline{\&}$ operators

for &X, X can be *p, p, a

$$\&X = value(\overline{\&}X)$$

C Expressions

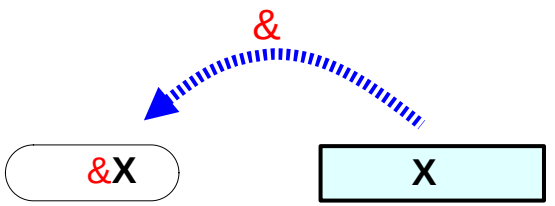
*p a dereferenced variable
p a pointer variable
a a primitive variable

& X

& *p = $\overline{\&}$ * value(p)
= value(p)

& p

& a



for $\overline{\&}$X = p, X must be *p

$$\overline{\&}X = value^{-1}(\&X)$$

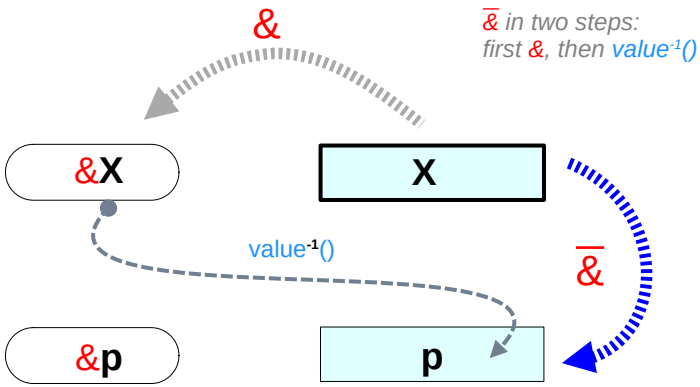
*p a dereferenced variable
~~p a pointer variable~~
~~a a primitive variable~~

$\overline{\&}$ X = p

$\overline{\&}$ *p ≡ p

~~$\overline{\&}$ p~~

~~$\overline{\&}$ a~~



Two step de-referece *̄ and * operators

for *̄**A**, **A** can be value(**p**), &**p**, &**a**

*value(**p**)* value of a pointer variable
*&**p*** address of a pointer variable
*&**a*** address of a primitive variable

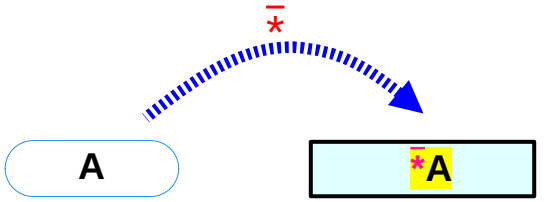
*̄**A**	=	***value**$^{-1}$(**A**)

*̄ **A**

*̄ value(**p**) = ***p**

*̄ &**p** = **p**

*̄ &**a** = **a**



for *̄**A** = ***p**, **A** must be value(**p**)

*value(**p**)* value of a pointer variable
~~*&**p*** address of a pointer variable~~
~~*&**a*** address of a primitive variable~~

***p**	=	*̄**value**(**p**)

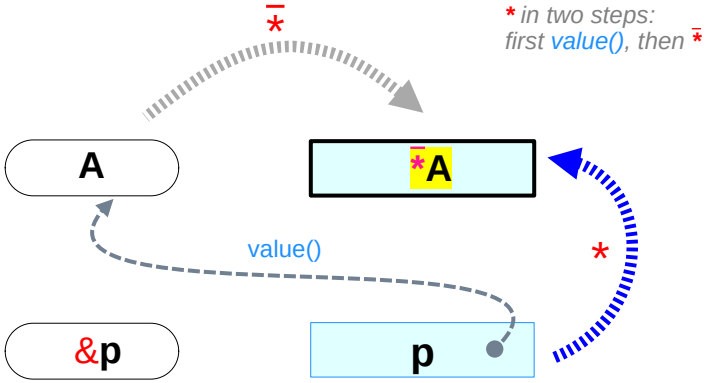
C Expressions

* **p**

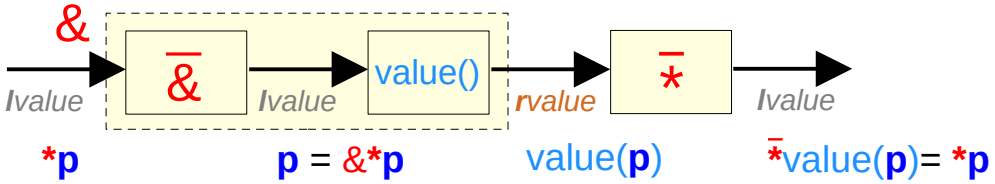
* **p** ≡ *̄**value**(**p**)

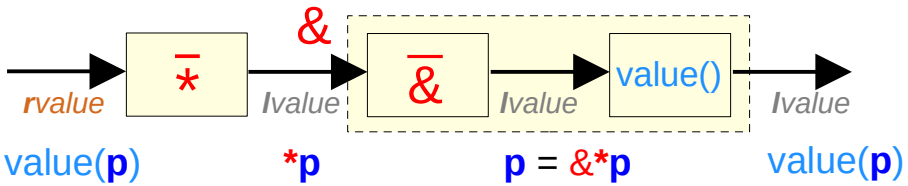
~~* &**p** = **p**~~

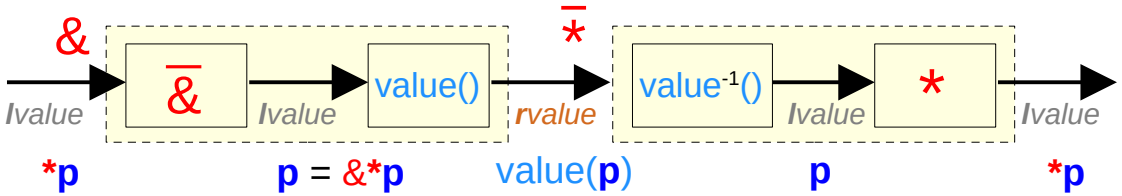
~~* &**a** = **a**~~

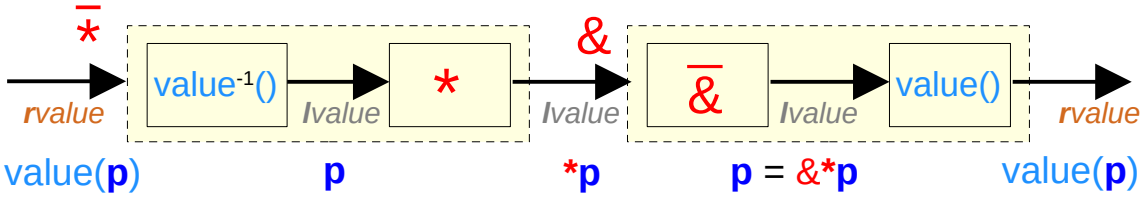


Inverse operators : & and $\bar{*}$ in pointer de-referencing





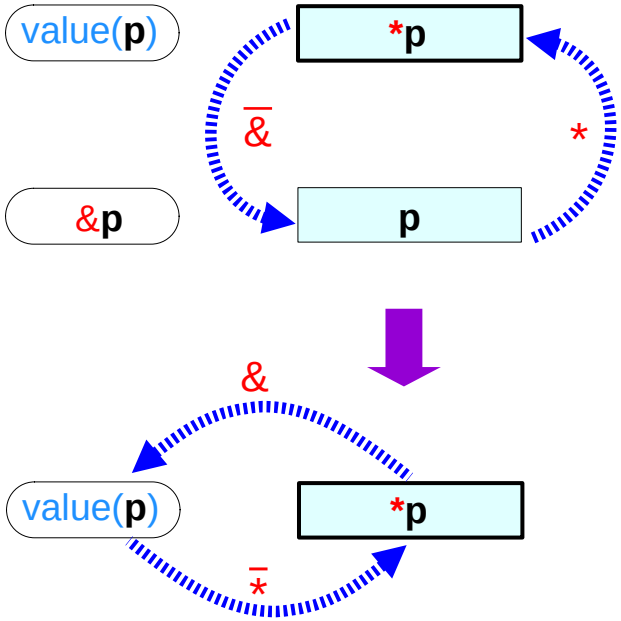




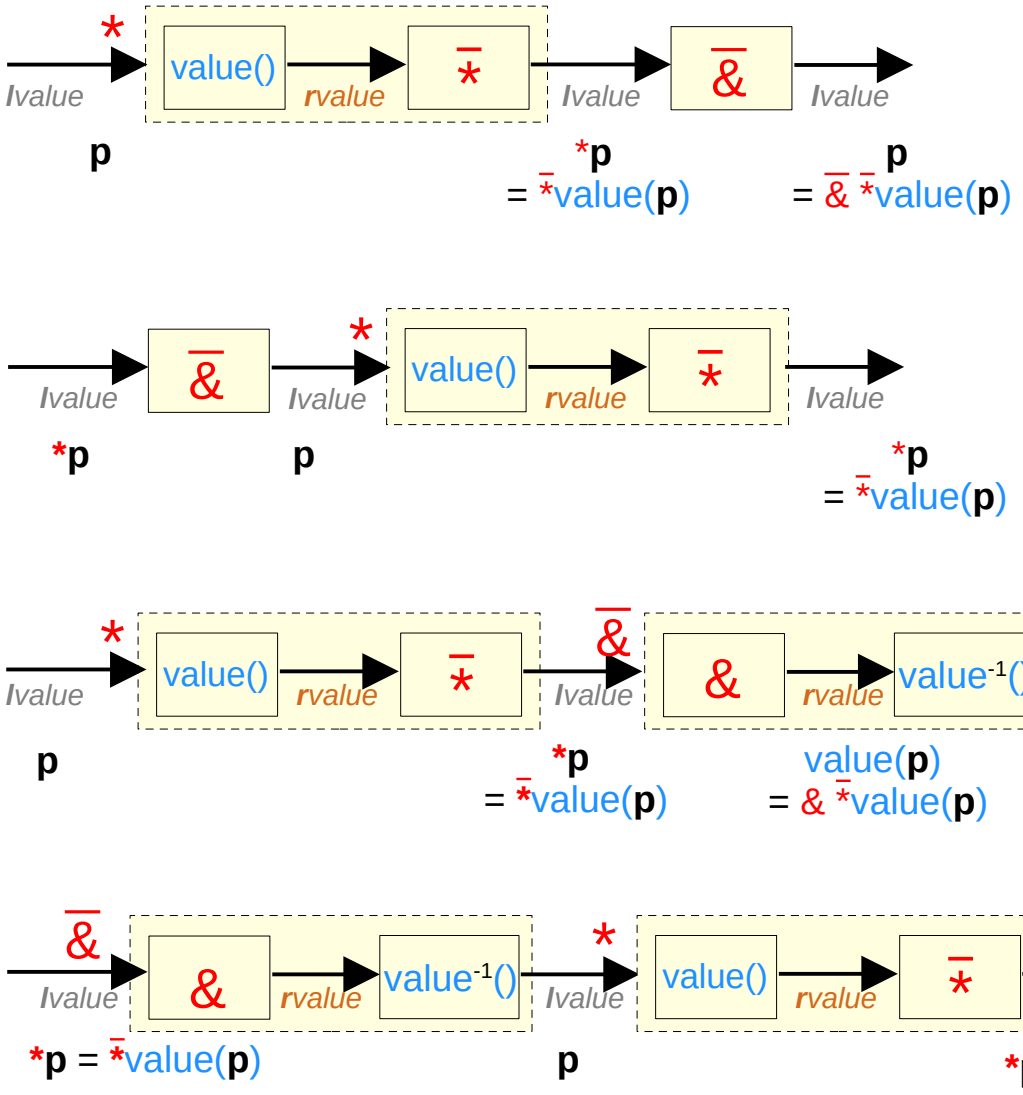
$\bar{*}$ value(**p**) = ***p**

&$\bar{*}$ value(**p**) = & ***p** = value(**p**)

$\bar{*}$&$\bar{*}$value(**p**) = $\bar{*}$&***p** = $\bar{*}$value(**p**) = ***p**



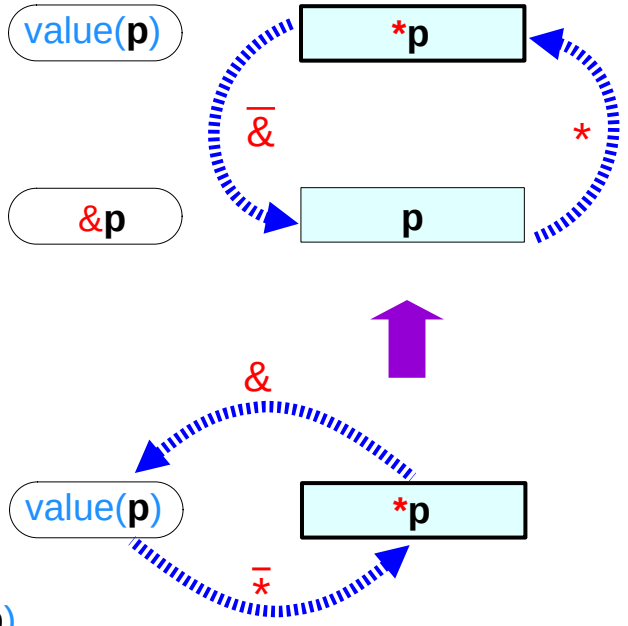
Inverse operators of & and *



*̄ value(**p**) = ***p**

&̄ * **p** = **p** **p** = &̄ *̄value(**p**)

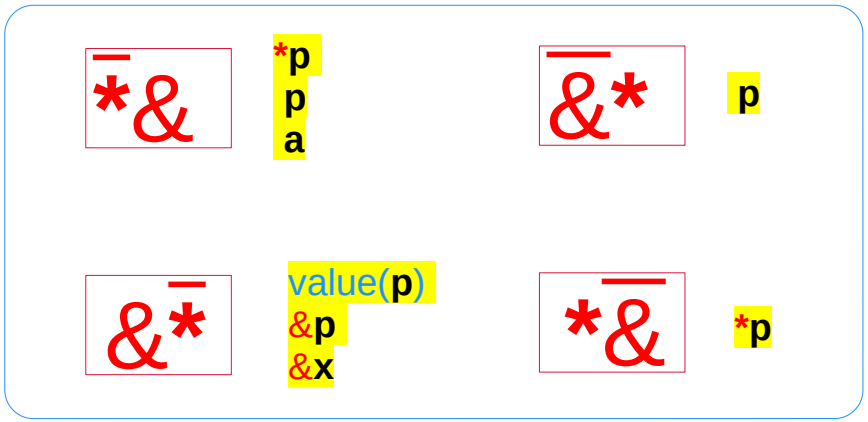
* &̄ ***p** = ***p** value(**p**) = & *̄value(**p**)



Inverse operators in pointer referencing (1)

*& *p = *p
*& p = p
*& a = a

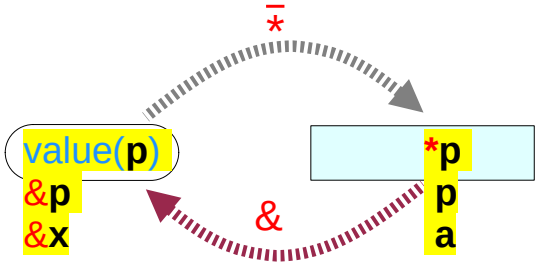
&* value(p) = value(p)
&* &p = &p
&* &a = &a

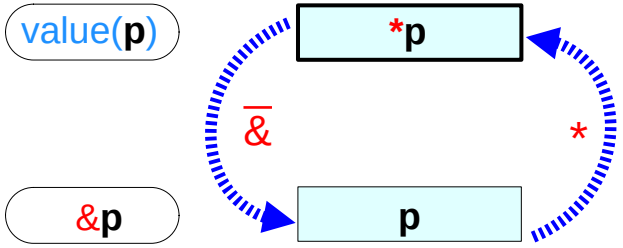


&* p = p

*& *p = *p

Math expressions

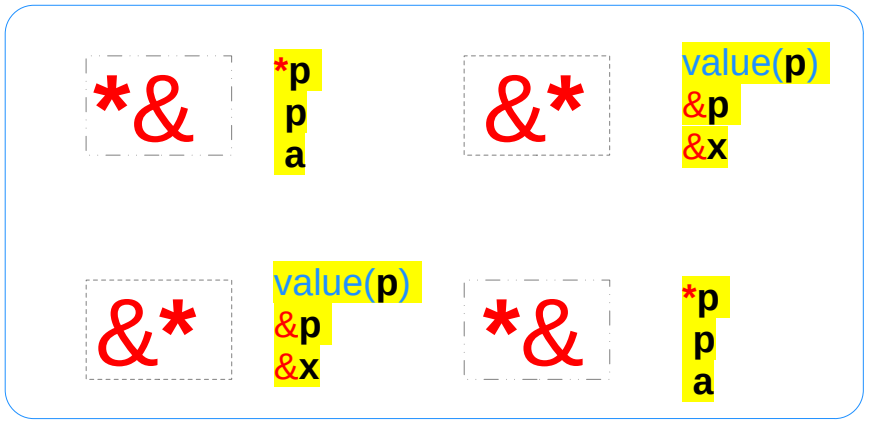




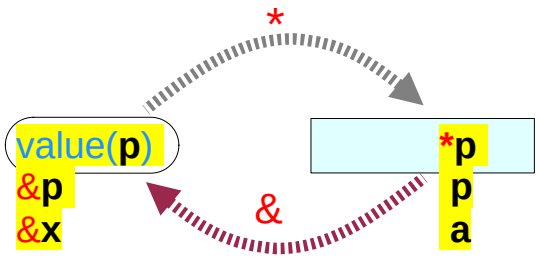
Inverse operators in pointer referencing (2)

*& *p = *p
*& p = p
*& a = a

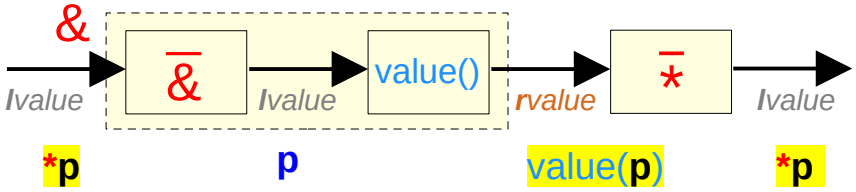
&* value(p) = value(p)
&* &p = &p
&* &a = &a

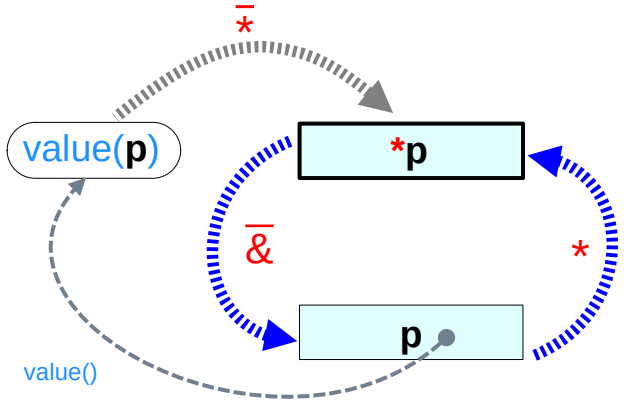


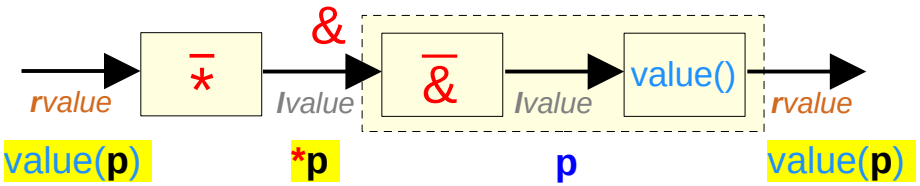
C expressions

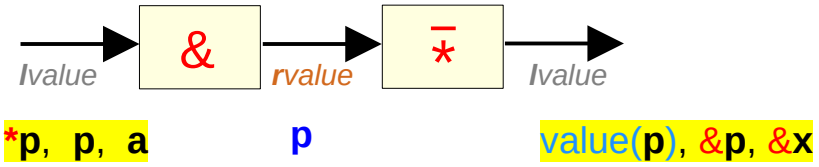


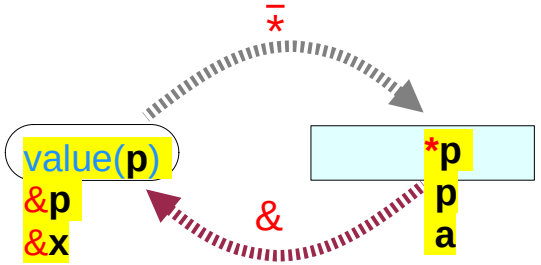
Inverse operators in pointer referencing (3)

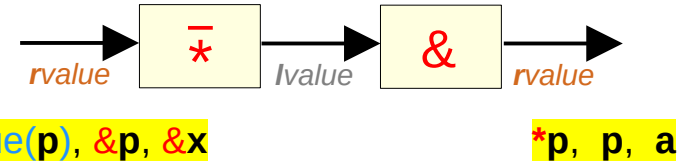




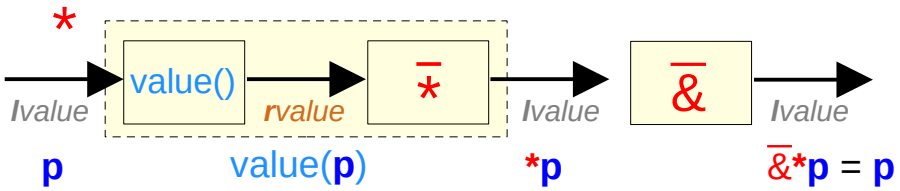


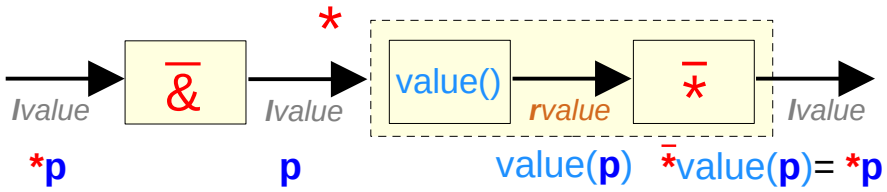


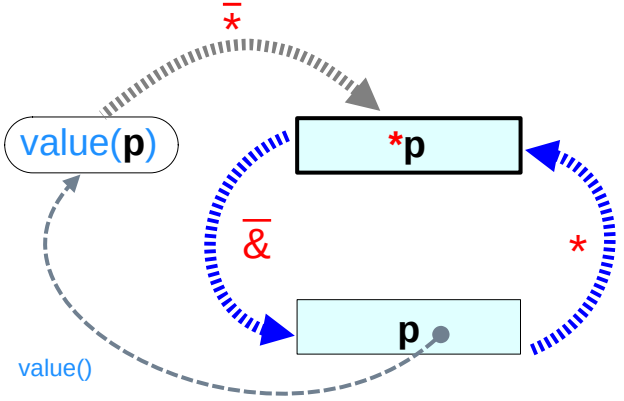


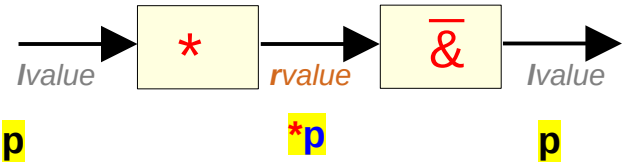


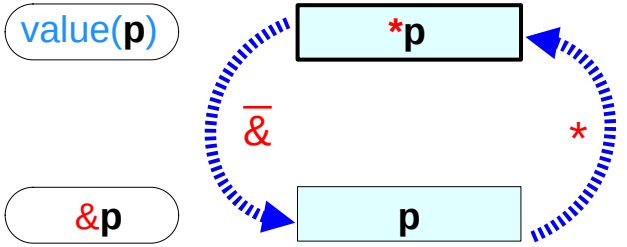
Inverse operators in pointer referencing (3)

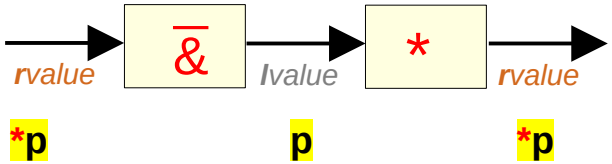












References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun