

# Address-of and dereference operators

---

Copyright (c) 2024 - 2010 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto: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)*

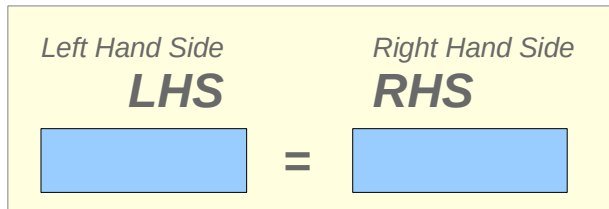
**\* address :**  
*returns the value at the address*

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

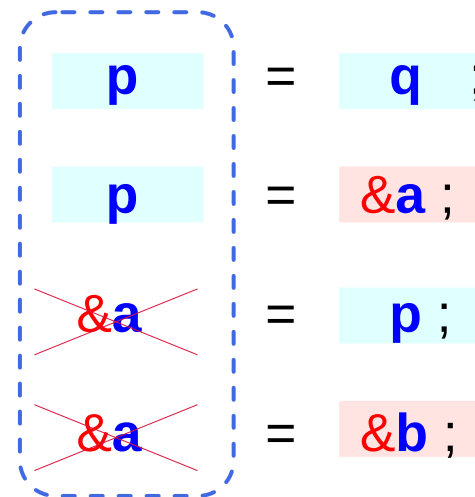
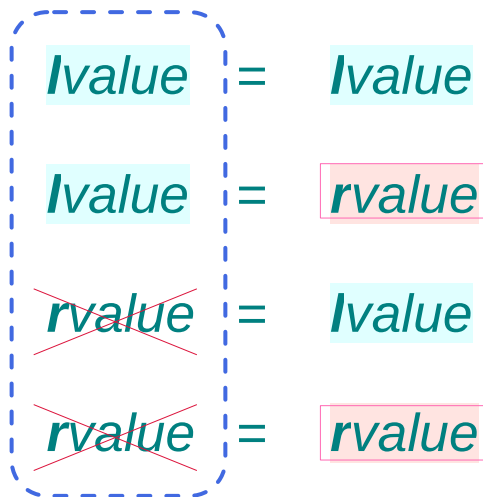
**(\* address** *is an lvalue)*

# Ivalue and rvalue in assignments

an assignment statement



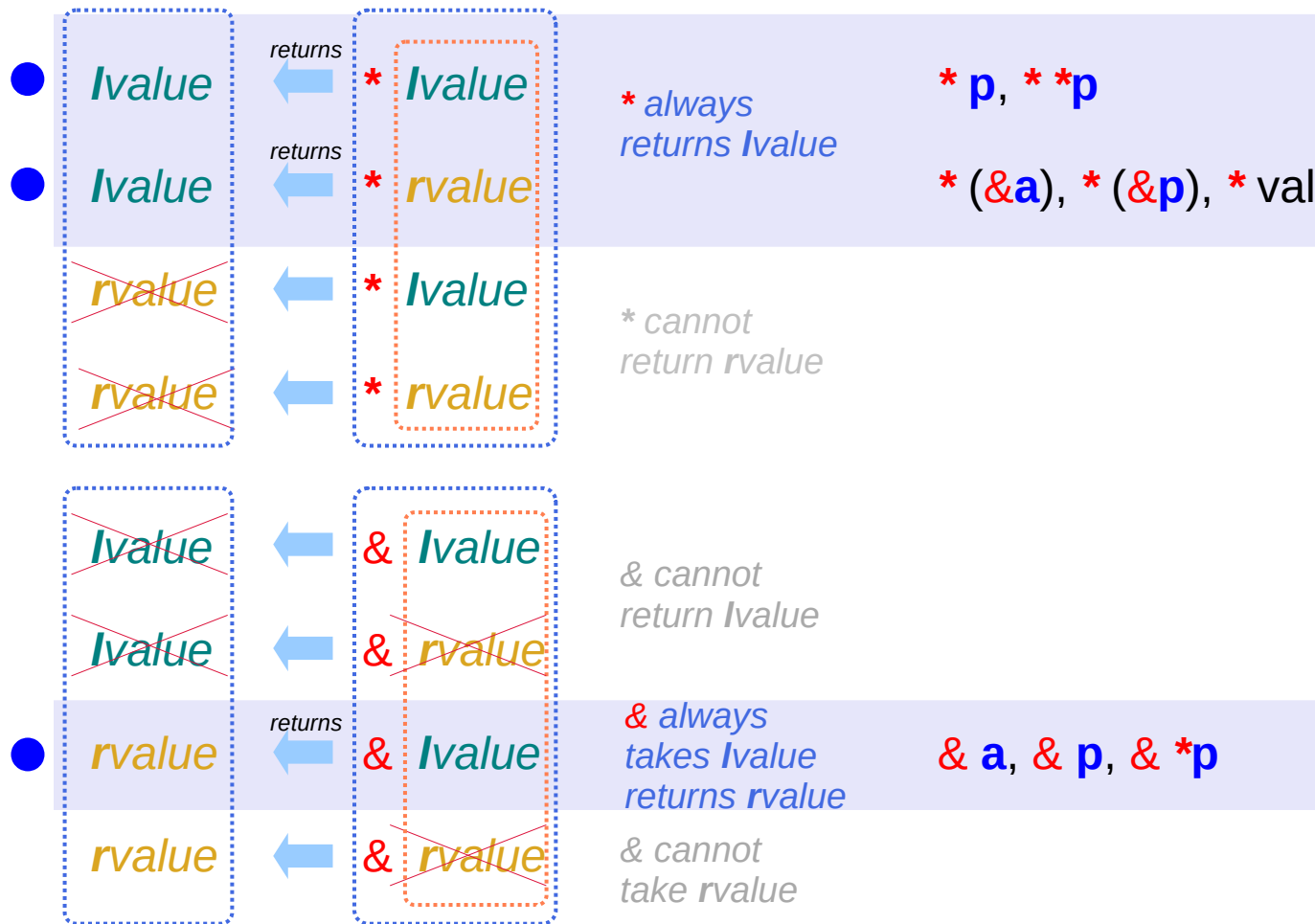
```
int a, b = 10 ;  
int * p, q = &a ;
```



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

<b>a, b, p, q</b>	: Ivalues	... variables	... RW
<b>*p, *q</b>	: Ivalues	... variables	... RW
<b>&amp;a, &amp;b</b>	: rvalues	... constants	... RO

# Ivalue and rvalue with \* and & operators



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

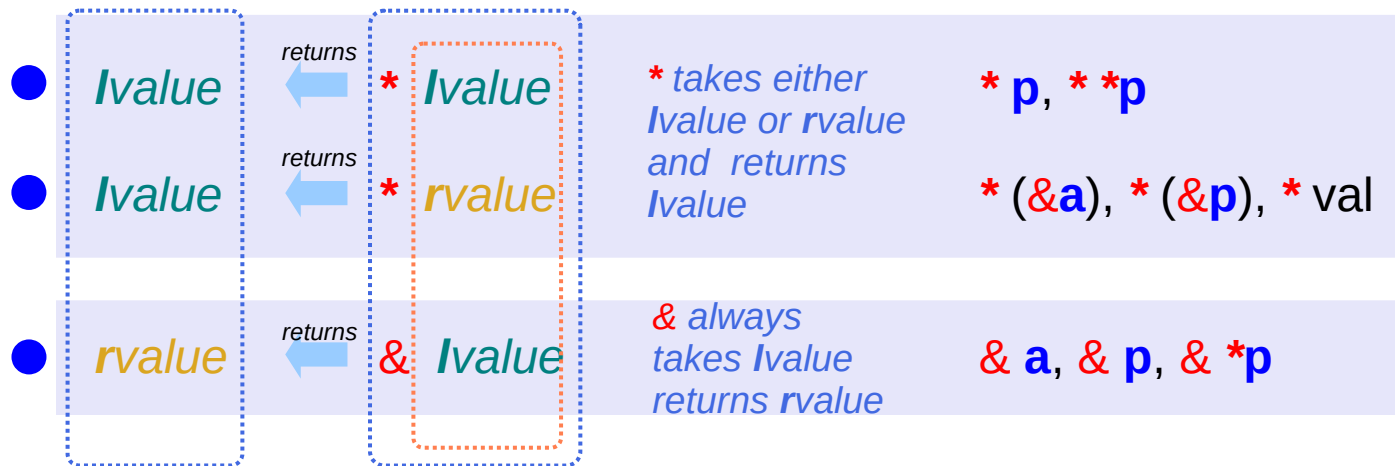
\* can be applied to either an **Ivalue** variable or a **rvalue** address

\* **operand** becomes an **Ivalue** variable thus can be applied successively.

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

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

# Ivalue and rvalue with \* and & operators



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

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

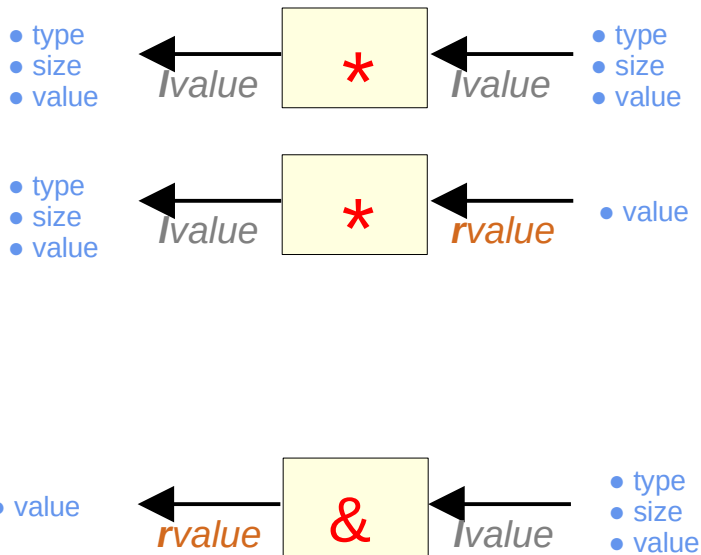
\* **operand** acts as an **lvalue** variable

\* can be applied successively.

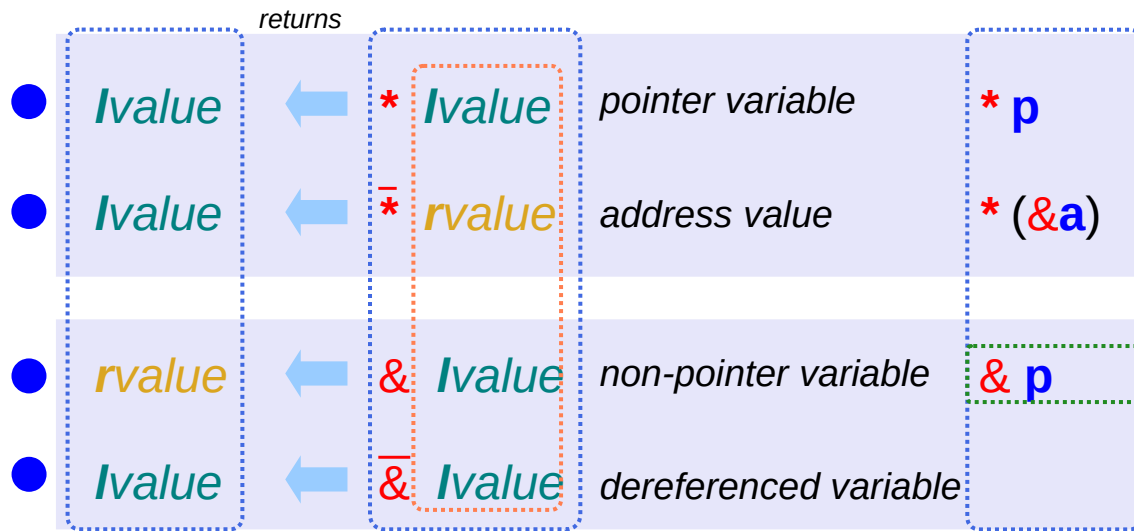
& can take only an **lvalue** variable and returns only an **rvalue** address

& cannot be applied successively.

<b>a, p</b>	: lvalues	... variables	... RW
<b>*p</b>	: lvalues	... variables	... RW
<b>&amp;a</b>	: rvalues	... constants	... RO



# Ivalue and rvalue with \* and & operators

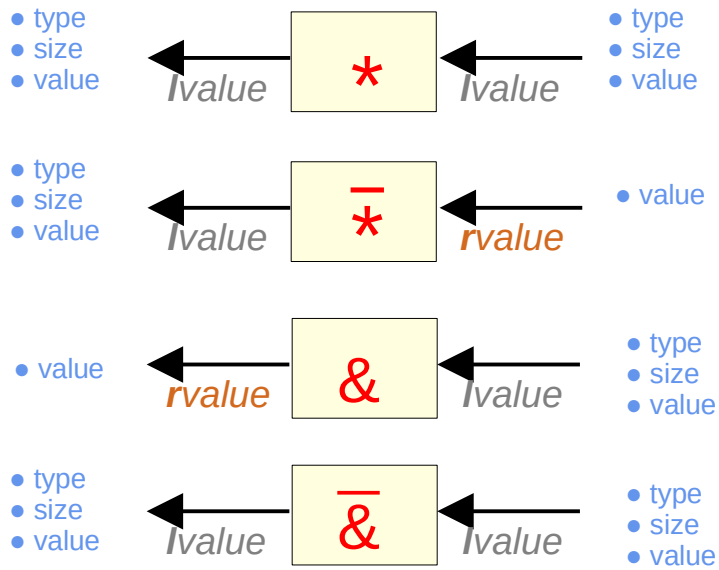


```
int a = 10 ;
int * p = &a ;
```

\* 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
```

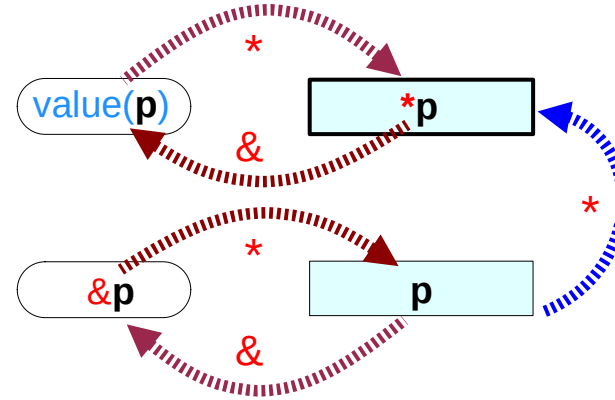
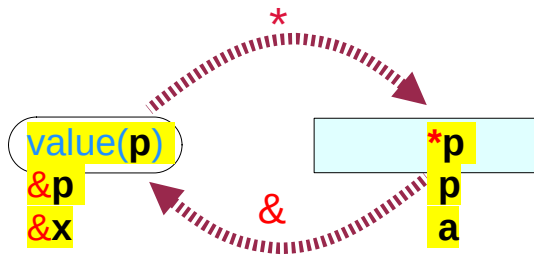


# Variable types

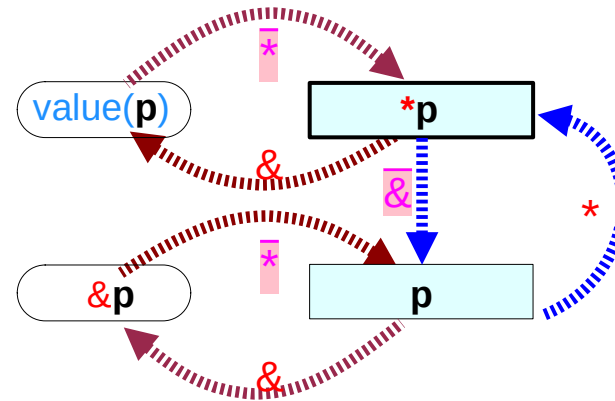
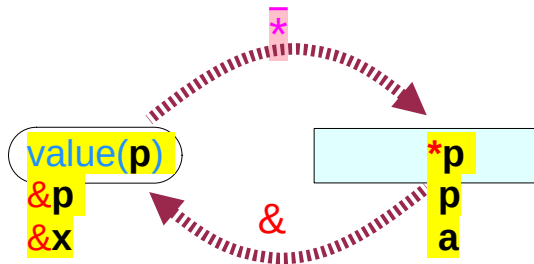
<b>a</b>	non-pointer variables	<i>lvalue</i>
<b>p</b>	pointer variables	<i>lvalue</i>
<b>*p</b>	dereferenced variables	<i>lvalue</i>
<b>val</b>	address values	<i>rvalue</i>

# Comparisons (1)

## C operators : \*, &



## augmented operators : $\bar{*}$ , $\bar{\&}$



# Comparisons (2)

## C operators : \*, &

	×	a	<i>Ivalue</i>
<i>Ivalue</i> ←	*	p	<i>Ivalue</i>
<i>Ivalue</i> ←	*	*p	<i>Ivalue</i>
<i>Ivalue</i> ←	*	val	<i>rvalue</i>

<i>rvalue</i> ←	&	a	<i>Ivalue</i>
<i>rvalue</i> ←	&	p	<i>Ivalue</i>
<i>rvalue</i> ←	&	*p	<i>Ivalue</i>
	×	val	<i>rvalue</i>

## augmented operators : $\bar{*}$ , $\bar{\&}$

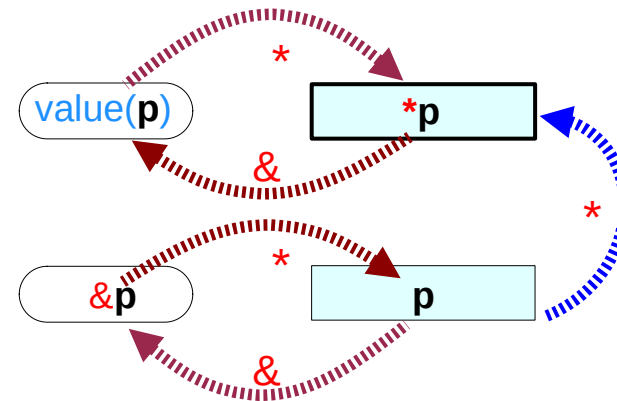
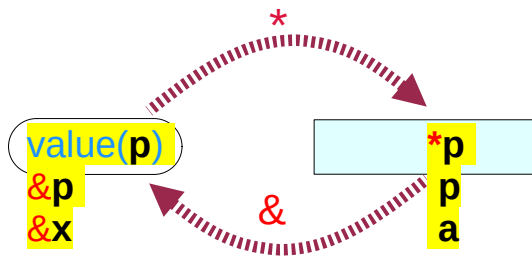
	×	a	<i>Ivalue</i>
<i>Ivalue</i> ←	*	p	<i>Ivalue</i>
<i>Ivalue</i> ←	*	*p	<i>Ivalue</i>
<i>Ivalue</i> ←	$\bar{*}$	val	<i>rvalue</i>

<i>rvalue</i> ←	&	a	<i>Ivalue</i>
<i>rvalue</i> ←	&	p	<i>Ivalue</i>
<i>rvalue</i> ←	&	*p	<i>Ivalue</i>
<i>Ivalue</i> ←	$\bar{\&}$	*p	<i>Ivalue</i>
	×	val	<i>rvalue</i>

# C operators : \*, &

	×	a	Ivalue
Ivalue ←	*	p	Ivalue
Ivalue ←	*	*p	Ivalue
Ivalue ←	*	val	rvalue

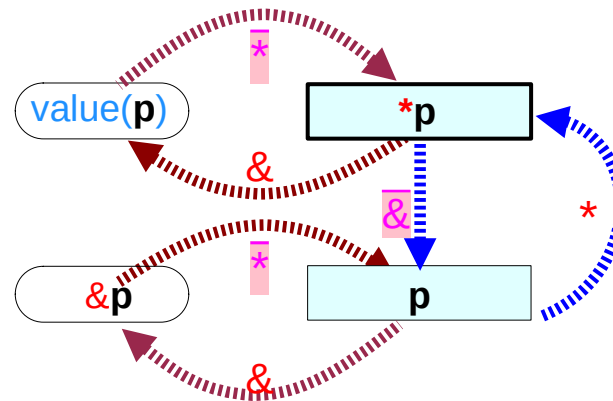
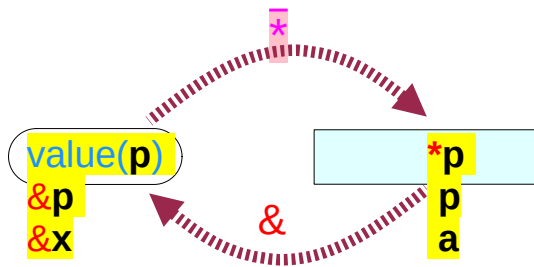
rvalue ←	&	a	Ivalue
rvalue ←	&	p	Ivalue
rvalue ←	&	*p	Ivalue
	×	val	rvalue



# Augmented operators : $\bar{*}$ , $\bar{\&}$

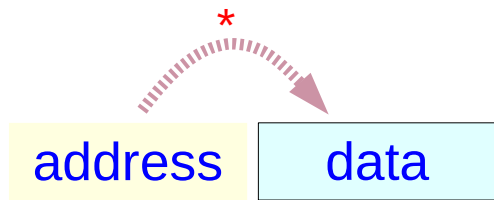
	×	a	Ivalue
Ivalue ←	*	p	Ivalue
Ivalue ←	*	*p	Ivalue
Ivalue ←	$\bar{*}$	val	rvalue

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

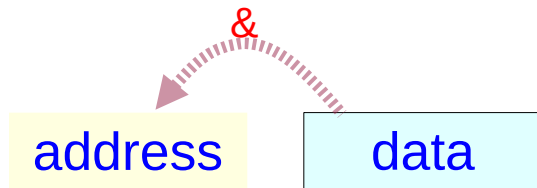


# Address-of & and dereference \* C operators (1)

## Primitive Data Type

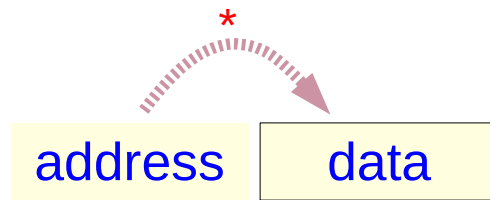


*rvalue* &a  $\xrightarrow{*}$  *Ivalue* a  
 constant variable  
*Ivalue* p  $\xrightarrow{*}$  *Ivalue* \*p  
 pointer variable

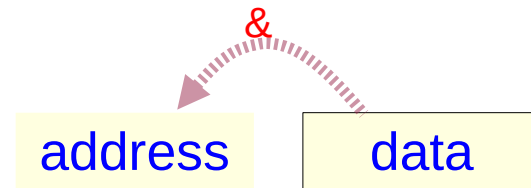


*rvalue* &a  $\xleftarrow{\&}$  *Ivalue* a  
 constant variable  
~~*Ivalue* p  $\xleftarrow{\&}$  *Ivalue* \*p~~  
~~pointer variable~~

## Pointer Data Type

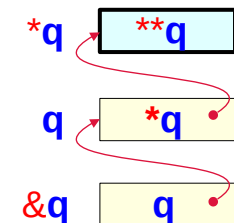
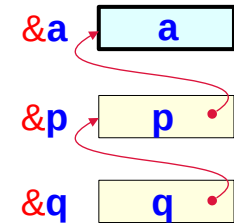


*rvalue* &q  $\xrightarrow{*}$  *Ivalue* q  
 constant double pointer  
*Ivalue* q  $\xrightarrow{*}$  *Ivalue* \*q  
 double pointer pointer



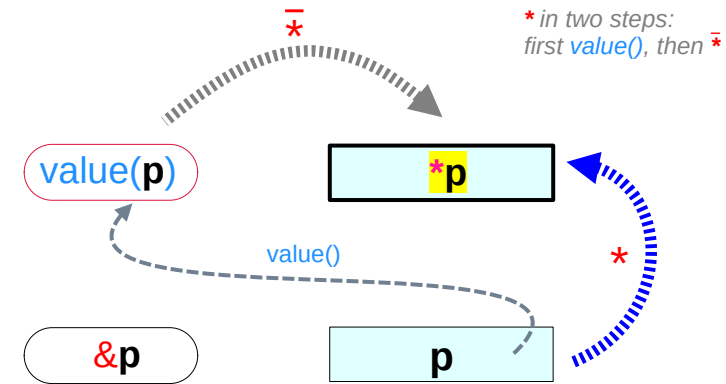
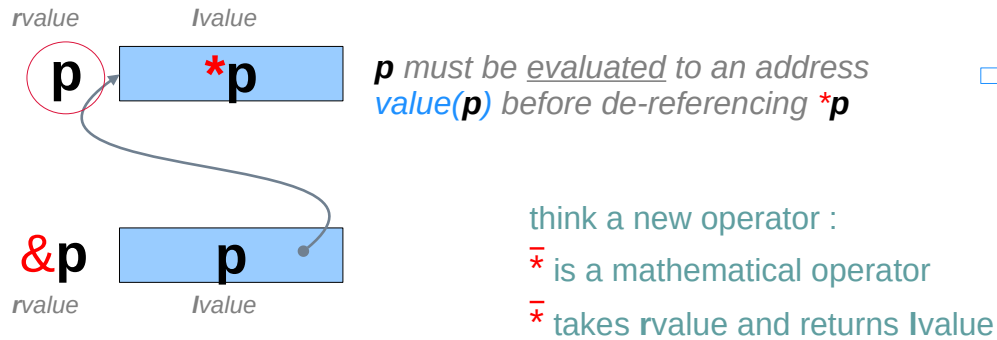
*rvalue* &q  $\xleftarrow{\&}$  *Ivalue* q  
 constant double pointer  
~~*Ivalue* q  $\xleftarrow{\&}$  *Ivalue* \*q~~  
~~double pointer pointer~~

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

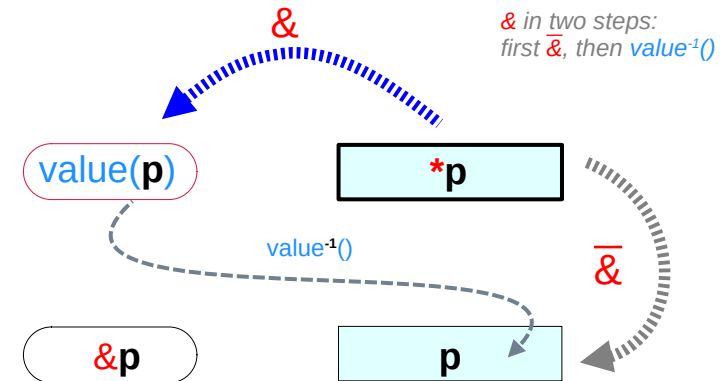
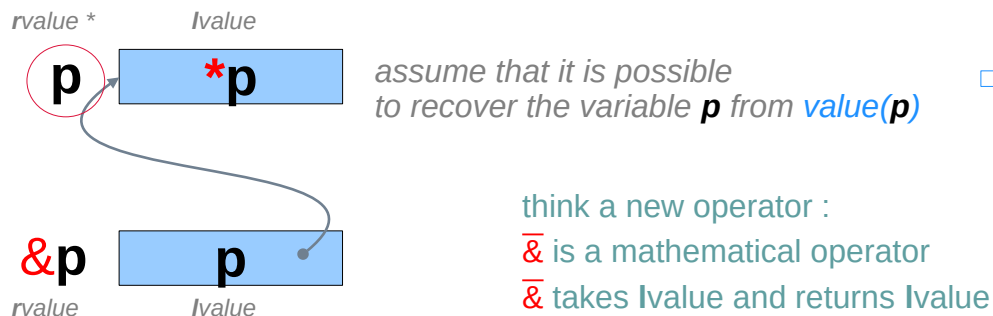


# & and \* operators in pointer de-referencing

## Two step De-reference \* operation



## Two step Address-of & operation



# Recursive application of the address-of operator

~~$\&(\&(\&(c[i])[j])[k])$~~

## $\&$ 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 contrast, **\*p** becomes a lvalue variable

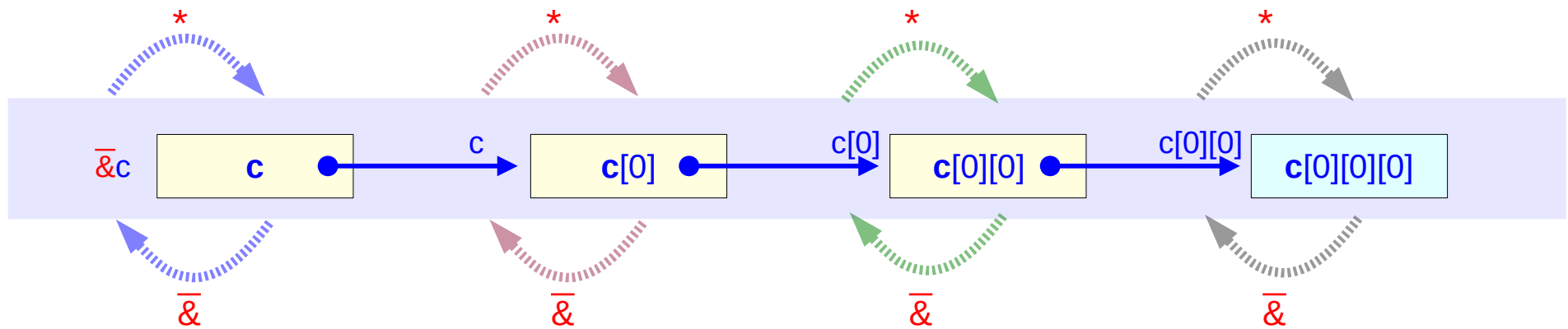
**\*** operator can be applied successively.

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

## $\bar{\&}$ mathematical operator

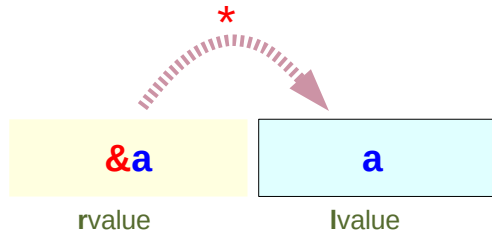


# Two step deferencing in type II (1) – without skipping

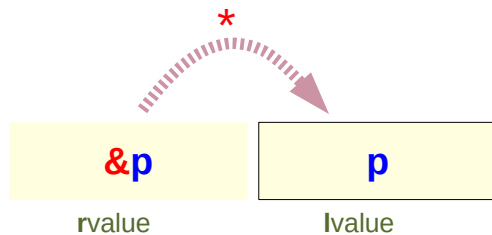


$\bar{\&}$  : mathematical  $\&$  operator

# Type, Size, and Value attributes of an **Ivalue**



*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

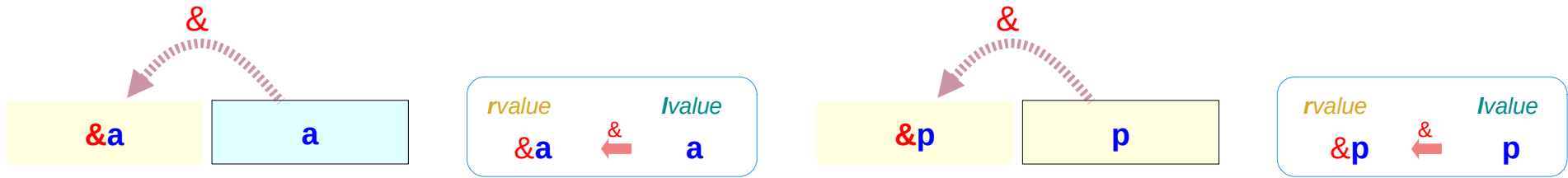
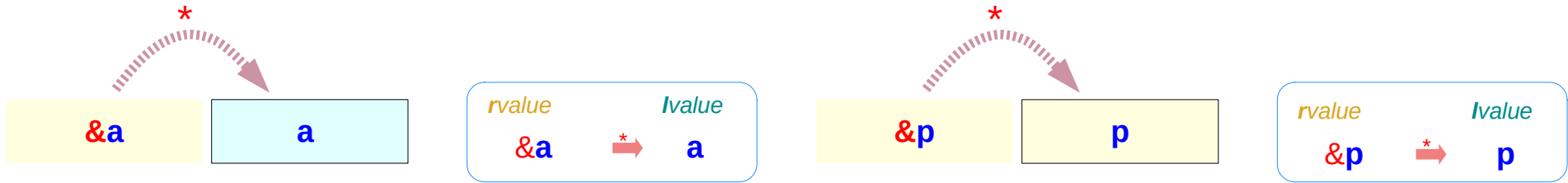
- Value

assume the function `value()`

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

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

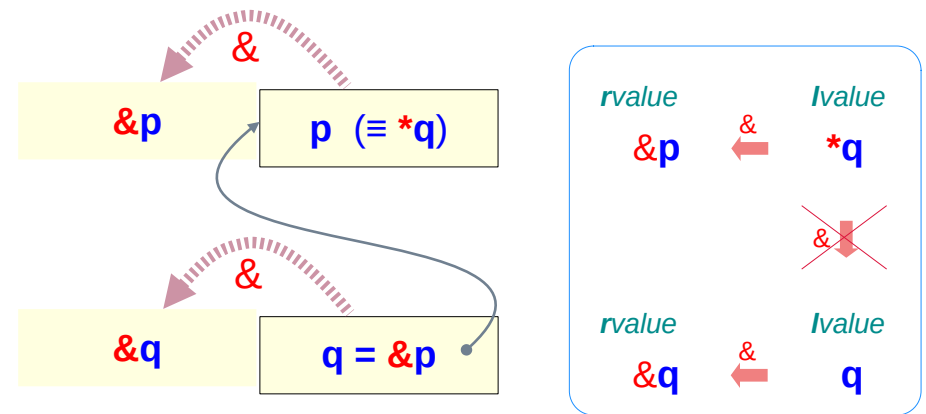
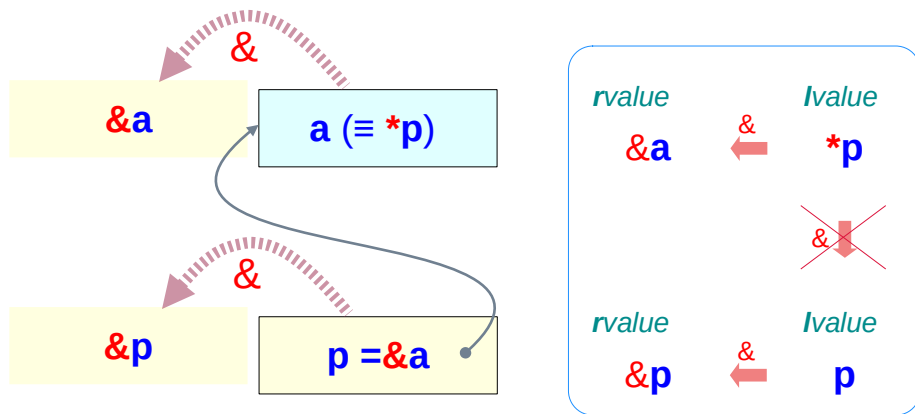
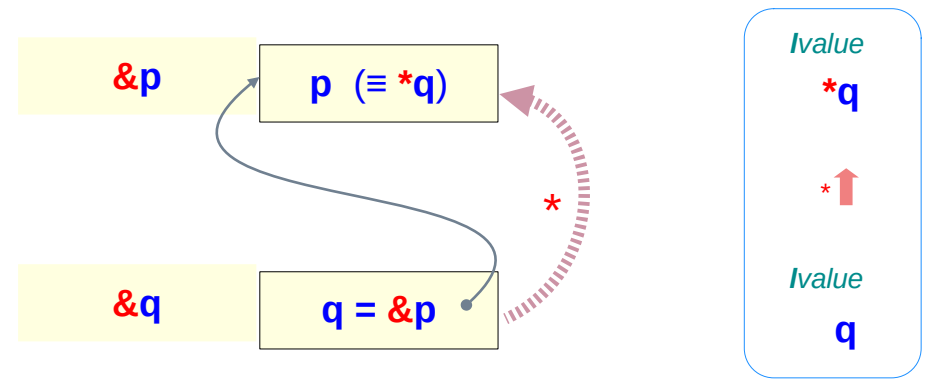
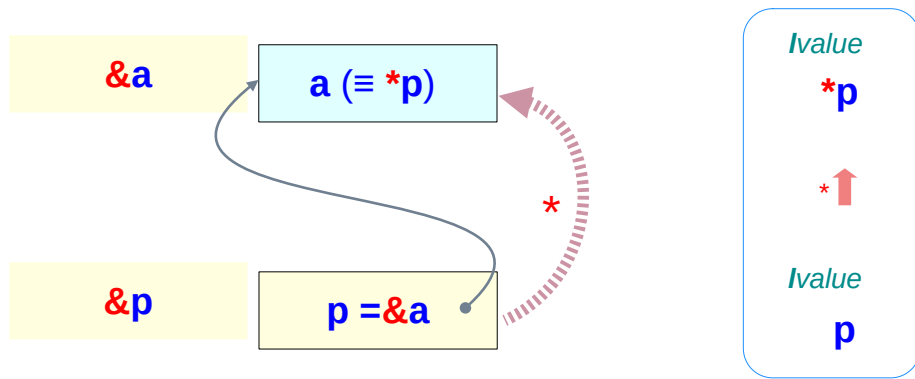
# Address-of & and dereference \* C operators (1)



$$*\&a = a$$

$$*\&p = p$$

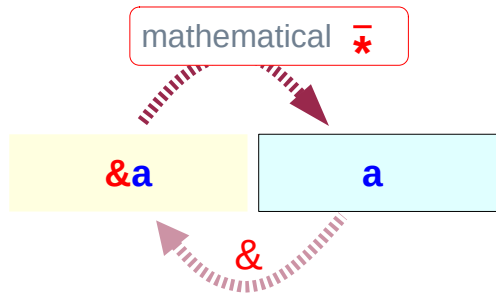
# Address-of & and dereference \* C operators (2)



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

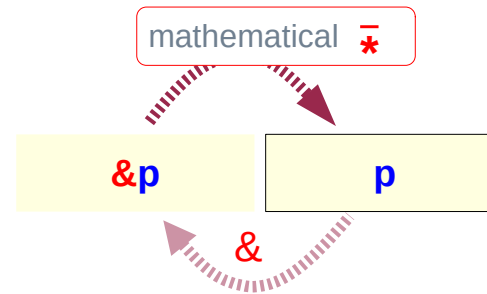
$*\&q = q$   
 ~~$\&*q = q$~~      *value(q)*

# Introducing mathematical operators : $\bar{\&}$ and $\bar{*}$



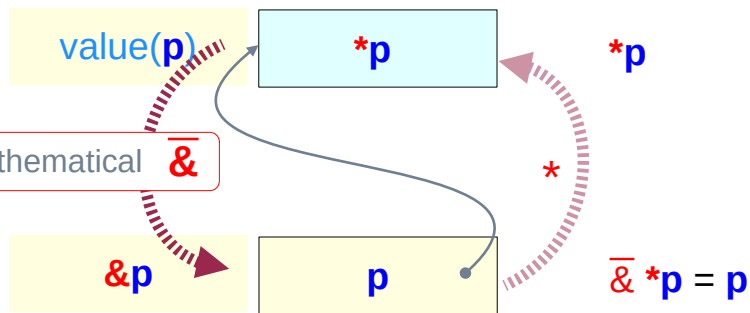
$$\bar{*} \&a = a$$

$\&a$

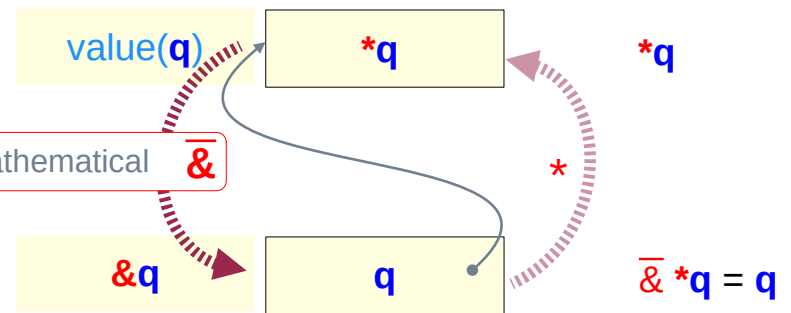


$$\bar{*} \&p = p$$

$\&p$

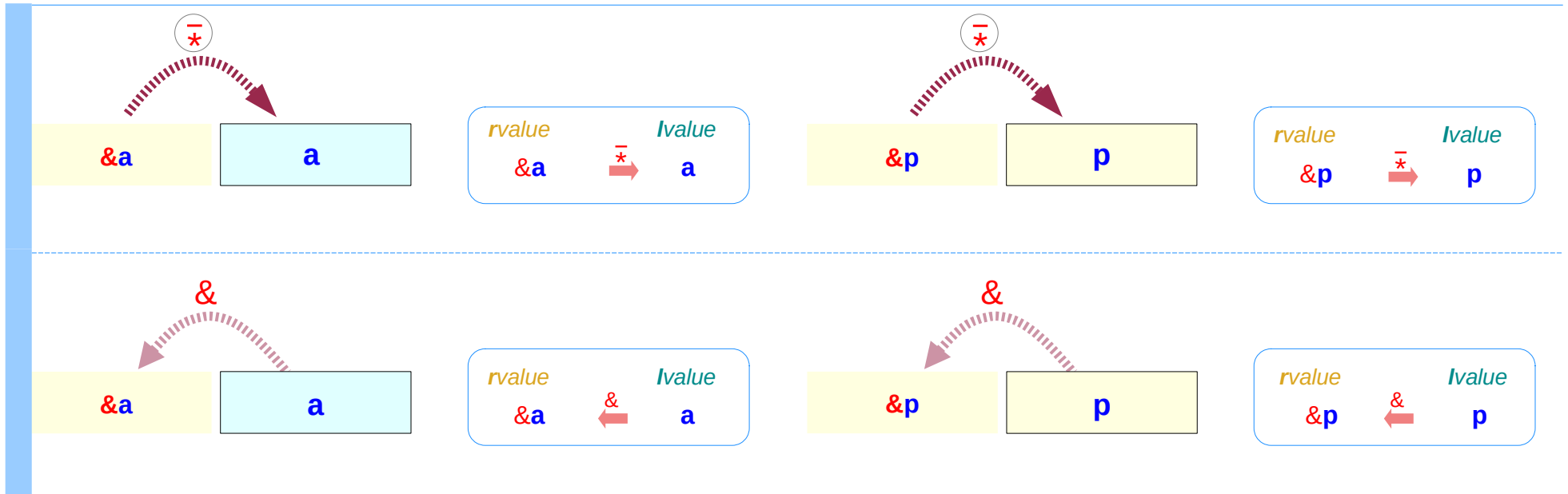


$$\bar{\&} *p = p$$



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

# Inverse operators $\bar{*}$ and $\&$



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

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

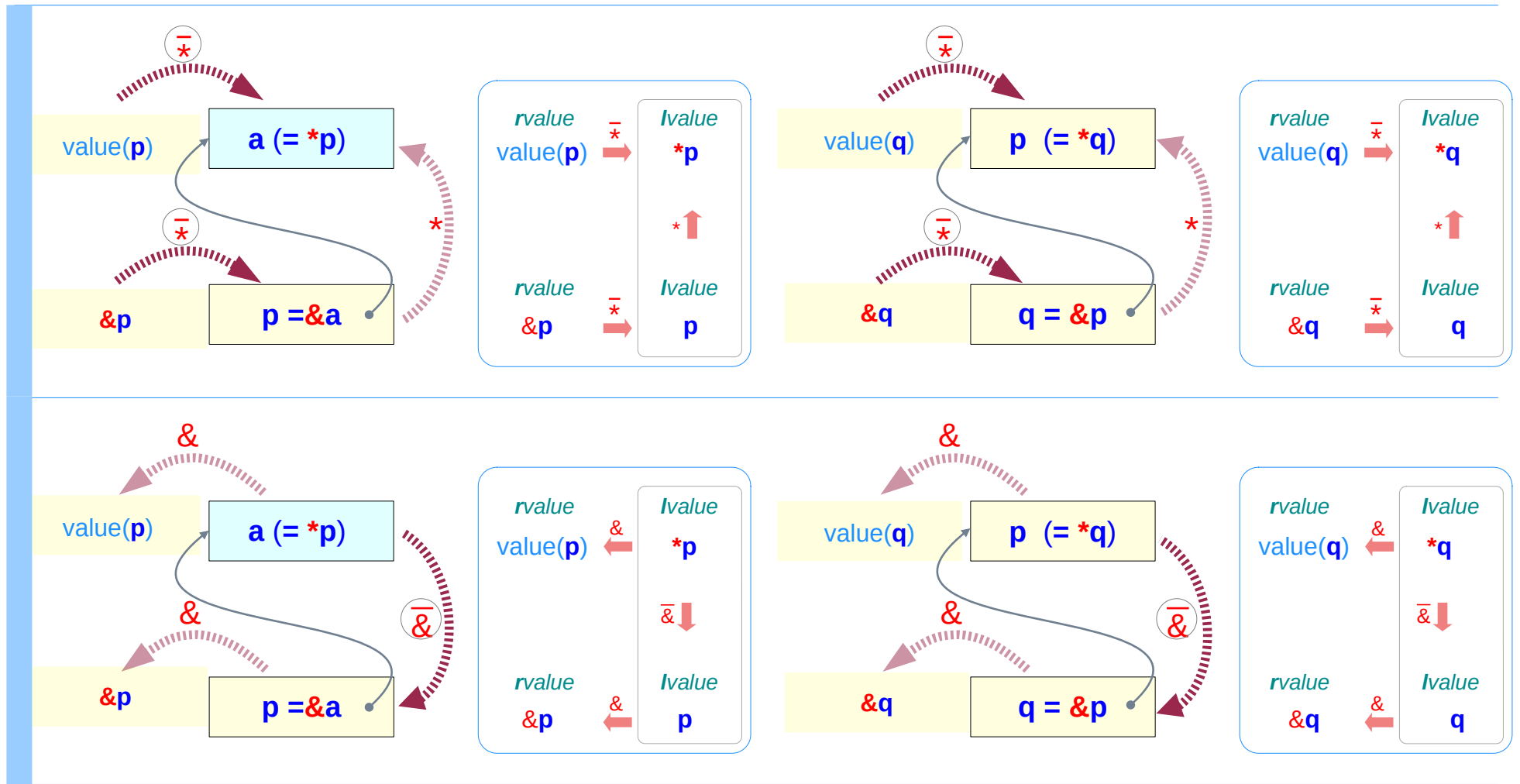
$\bar{*}$  and  $\&$  are  
inverse operators  
to each other

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

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

$\bar{*}$  and  $\&$  are  
inverse operators  
to each other

# Inverse operators $\bar{\&}$ and $\bar{*}$

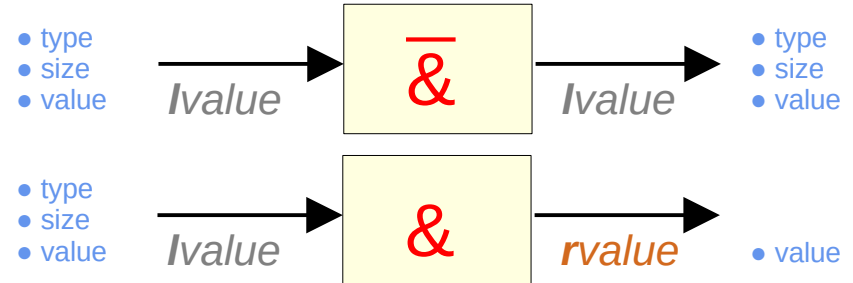


$$\begin{aligned} \bar{*}\&p &= p \\ \bar{\&}*p &= p \end{aligned}$$

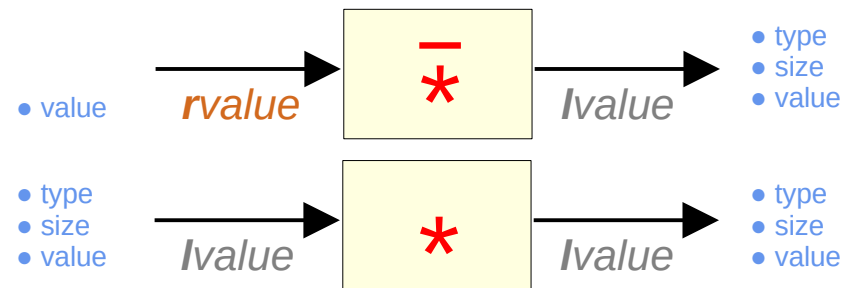
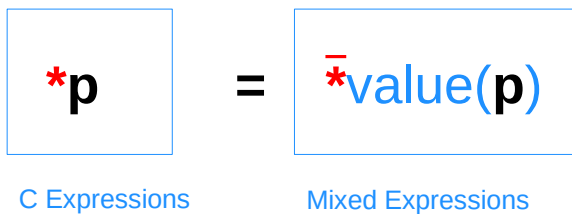
$$\begin{aligned} \bar{*}\&q &= q \\ \bar{\&}*q &= q \end{aligned}$$

# C operators and mathematical operators

## Address-of operation

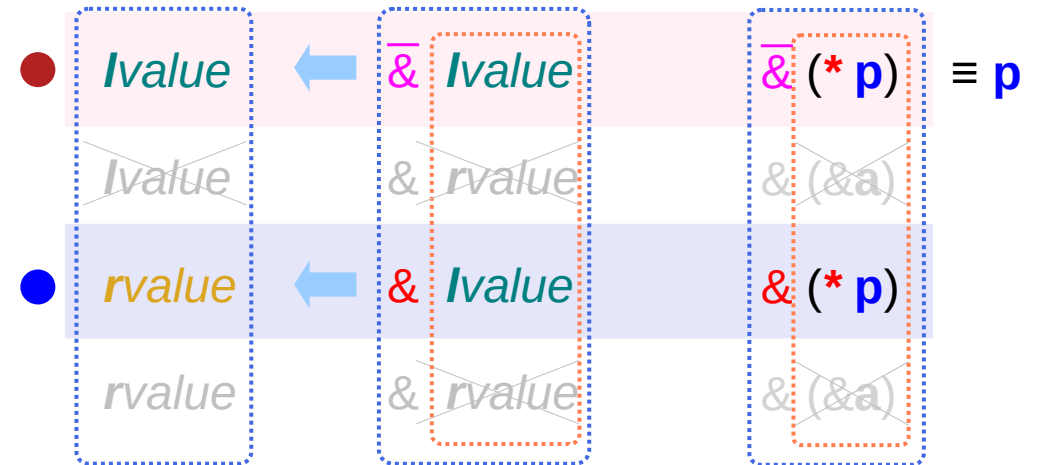
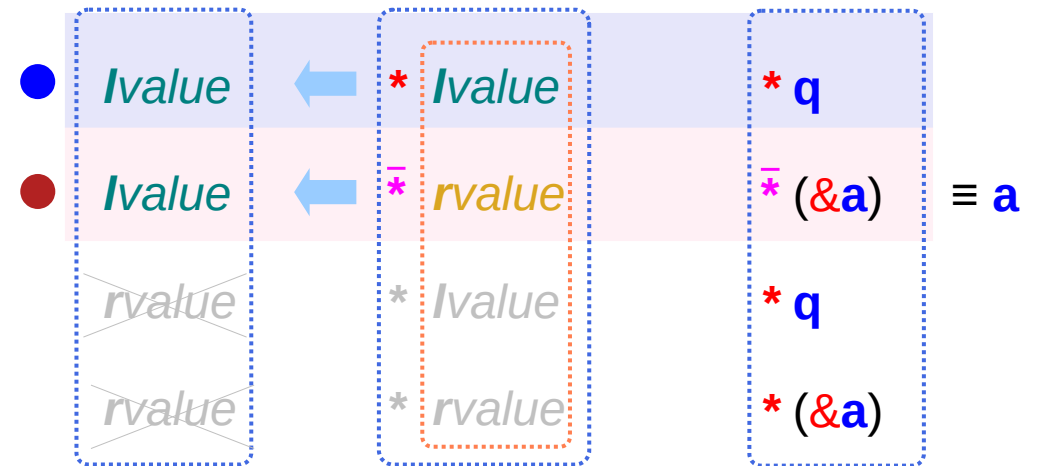
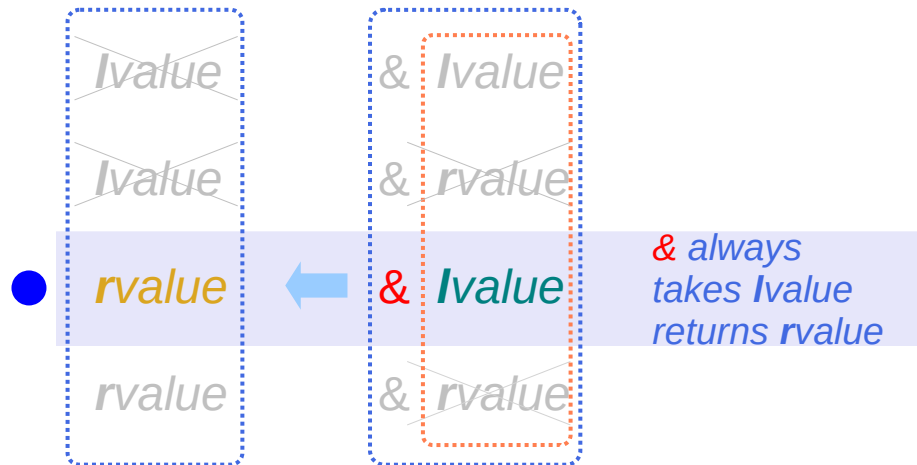
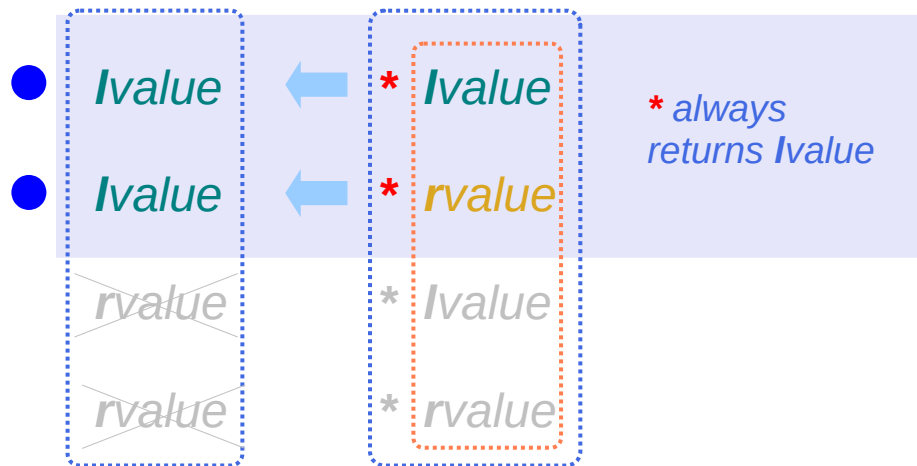


## De-reference operation





# Ivalue and rvalue with \* and & operators



**a, p, q** : Ivalues ... variables ... RW  
**\*p, \*q, \*\*q** : Ivalues ... variables ... RW  
**&a, &p, &q** : rvalues ... constants ... RO

# Examples of inverse operators

$\overline{*}&a \rightarrow a$

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

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

$\overline{*}value(p) \rightarrow *p \rightarrow a$

$\overline{*}value(q) \rightarrow *q \rightarrow p$

$\overline{*}value(*q) \rightarrow **q \rightarrow *p \rightarrow a$

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

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

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

Extended Operators

$*\&a \rightarrow a$

$*\&p \rightarrow p$

$*\&q \rightarrow q$

$*p \rightarrow a$

$*q \rightarrow p$

$**q \rightarrow *p \rightarrow a$

$\&*p \rightarrow \&a$

$\&*q \rightarrow \&p$

$\&**q \rightarrow \&a$

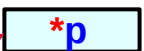
C Operators

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

$\&a$  

$\&p$  

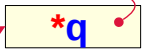
$\&q$  

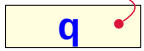
$value(p)$  

$\&p$  

$\&q$  

$value(*q)$  

$value(q)$  

$\&q$  

# Operands of mathematical operators : $\bar{\&}$ and $\bar{*}$

$\bar{*}$  address value

$\bar{\&} \bar{*} \text{value}(\mathbf{P}) \longrightarrow \text{value}(\mathbf{P})$

$\bar{*}$   $\bar{\&}$  variable

$\bar{*} \bar{\&} \mathbf{X} \longrightarrow \mathbf{X}$

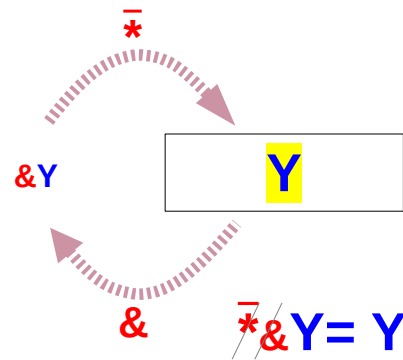
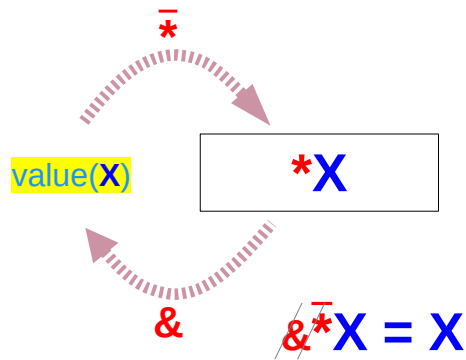
$\bar{\&}$   $\bar{*}$  pointer  
dereferenced pointer

$\bar{\&} \bar{*} \mathbf{P} \longrightarrow \mathbf{P}$

$\bar{\&}$  ~~pointer~~  
must be a dereferenced pointer  
not considering  
simultaneous pointing

$\bar{*} \bar{\&} \mathbf{Q} \longrightarrow \mathbf{Q}$   
Q must be a dereferenced pointer  
i.e,  $\mathbf{Q} = \bar{*} \mathbf{p} \longrightarrow \bar{\&} \mathbf{Q} = \mathbf{p}$

# & and mathematical $\bar{*}$



$$\begin{aligned} \& \bar{*} \text{value}(\mathbf{a}) &= \text{value}(\mathbf{a}) \\ \& \bar{*} \text{value}(\mathbf{p}) &= \text{value}(\mathbf{p}) \\ \& \bar{*} \text{value}(\mathbf{q}) &= \text{value}(\mathbf{q}) \end{aligned}$$

$$\begin{aligned} \& * \mathbf{a} &= \text{value}(\mathbf{a}) \\ \& * \mathbf{p} &= \text{value}(\mathbf{p}) \\ \& * \mathbf{q} &= \text{value}(\mathbf{q}) \end{aligned}$$

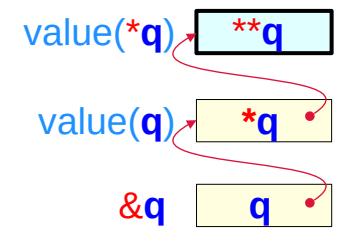
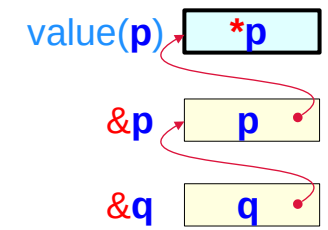
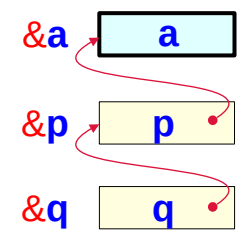
$$\begin{aligned} \bar{*} \&\mathbf{a} &= \mathbf{a} \\ \bar{*} \&\mathbf{p} &= \mathbf{p} \\ \bar{*} \&\mathbf{q} &= \mathbf{q} \end{aligned}$$

$$\begin{aligned} \bar{*} \&\mathbf{a} &= \mathbf{a} \\ \bar{*} \&\mathbf{p} &= \mathbf{p} \\ \bar{*} \&\mathbf{q} &= \mathbf{q} \end{aligned}$$

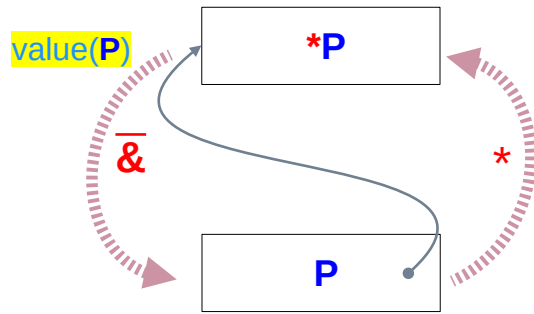
$$\begin{aligned} * \&\mathbf{a} &= \mathbf{a} \\ * \&\mathbf{p} &= \mathbf{p} \\ * \&\mathbf{q} &= \mathbf{q} \end{aligned}$$

C expressions

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



# \* and mathematical $\bar{\&}$



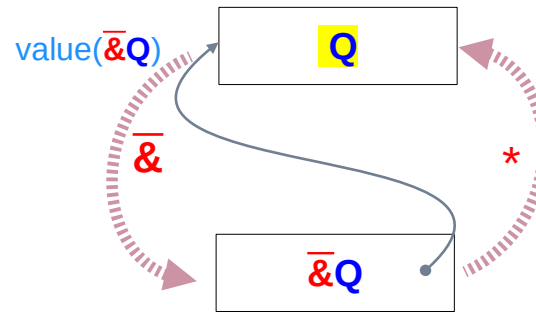
$$\bar{\&} * P = P$$

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

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

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

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



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

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

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

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

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

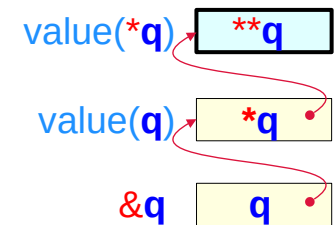
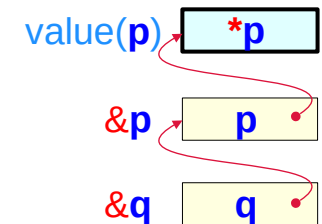
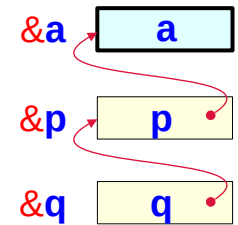
if not considering simultaneous pointing

$$* \& * p = * p$$

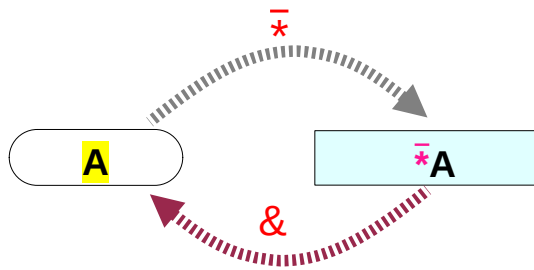
$$* \& * q = * q$$

C operators

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

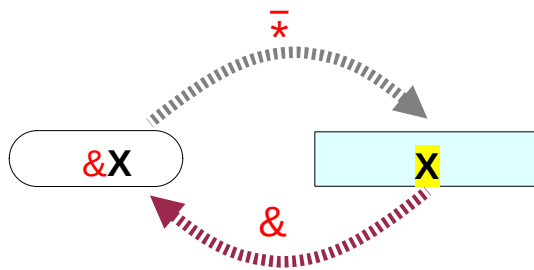


# Requirements of address and data



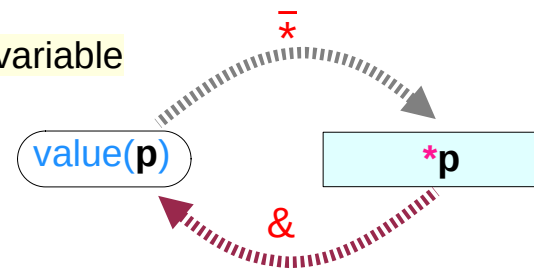
A : an address value - a number

`value(p), &p, &x`



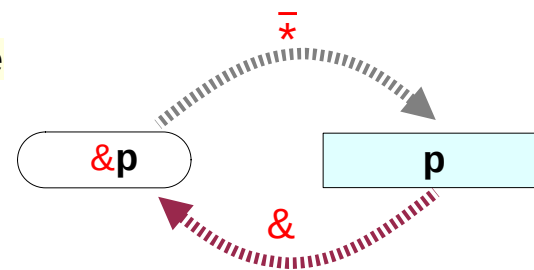
X : a variable    `*p, p, a`

`*p` : a dereferenced variable



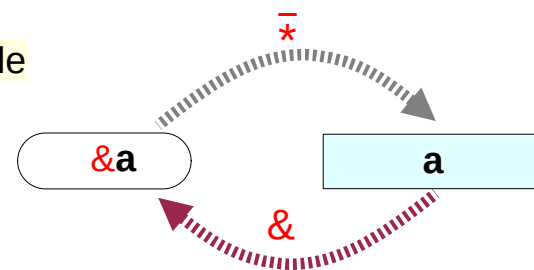
Pointer Dereferencing

`p` : a pointer variable



Pointer Data

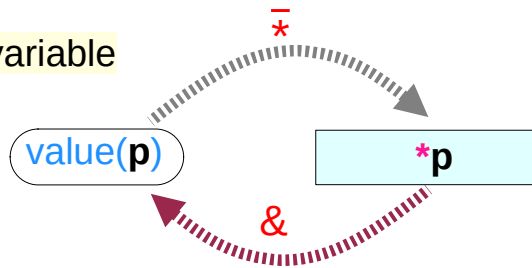
`a` : a primitive variable



Non-pointer Data

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

$*p$  : a dereferenced variable

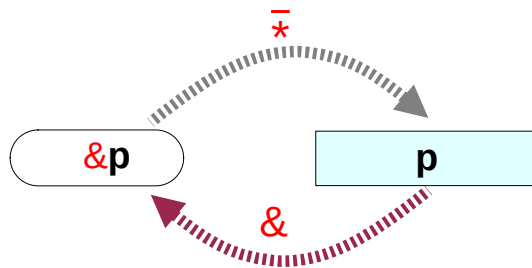


$$\bar{*} \text{value}(p) = *p$$

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

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

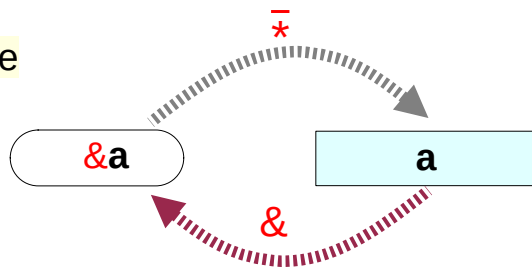
$p$  : a pointer variable



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

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

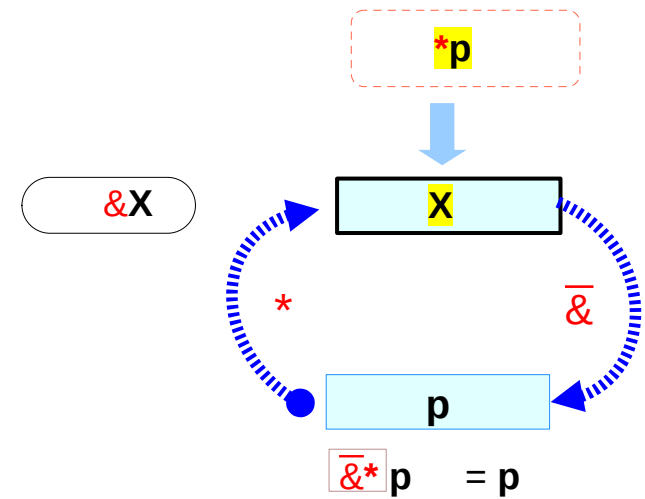
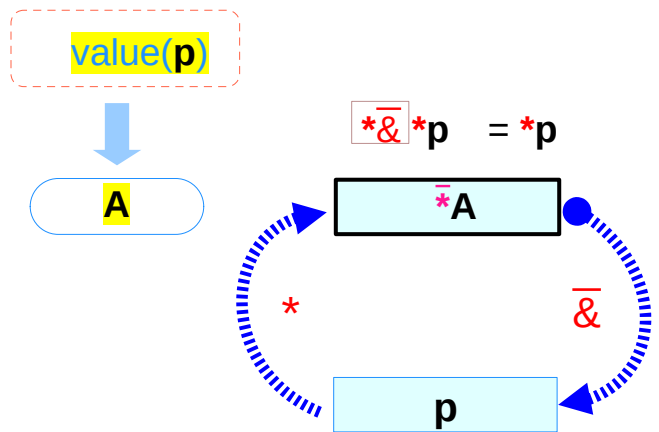
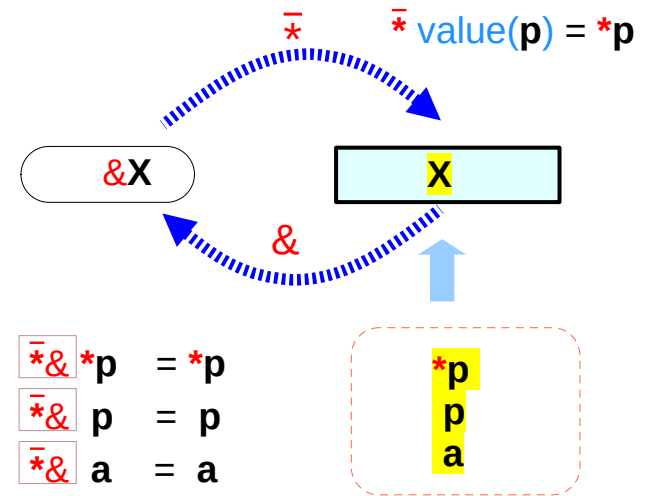
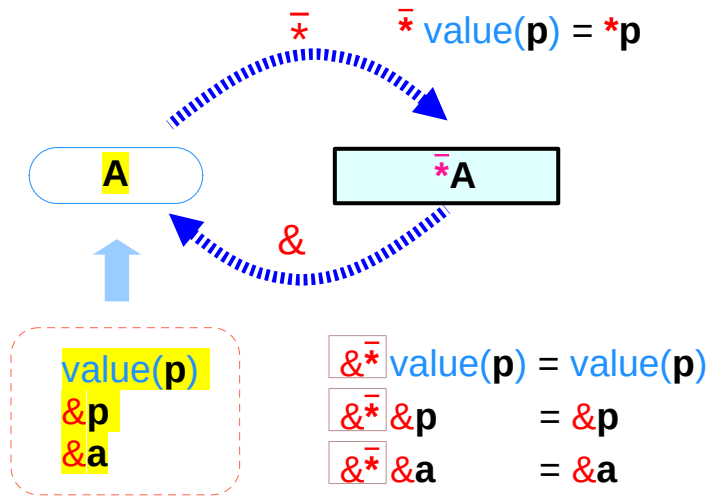
$a$  : a primitive variable



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

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

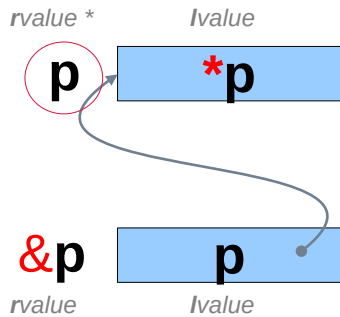
# Requirements of pointed address and data



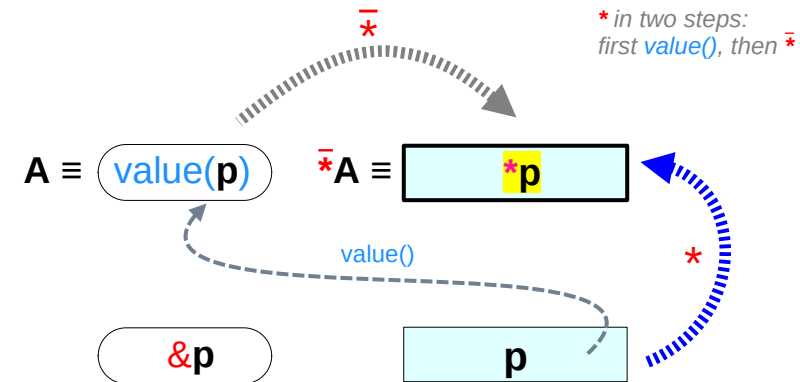


# & and \* operators in pointer de-referencing (1)

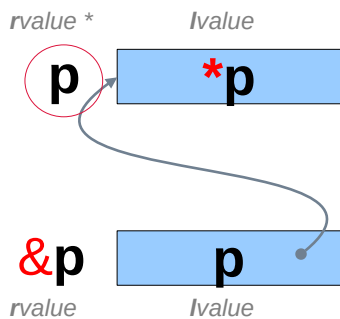
## Two step De-reference \* operation



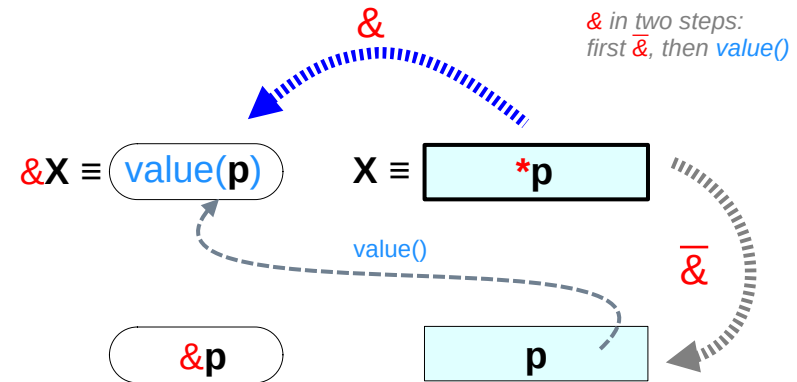
$$\begin{array}{l} \boxed{*p} = \boxed{\bar{*}\text{value}(p)} \\ \text{C Expressions} \quad \text{Mixed Expressions} \\ \boxed{\bar{*}A} = \boxed{* \text{value}^{-1}(A)} \end{array}$$



## Two step Address-of & operation

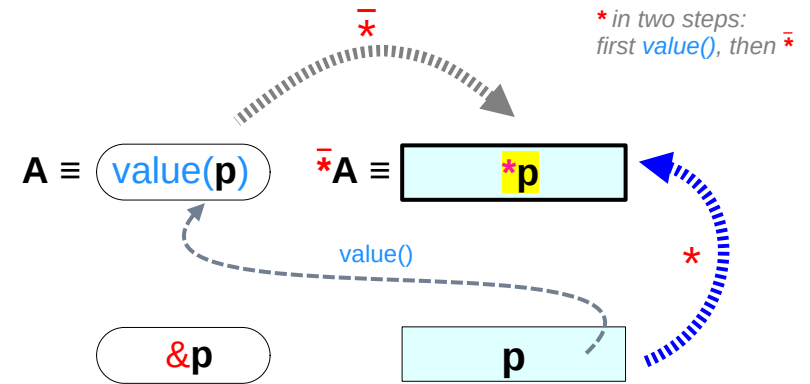
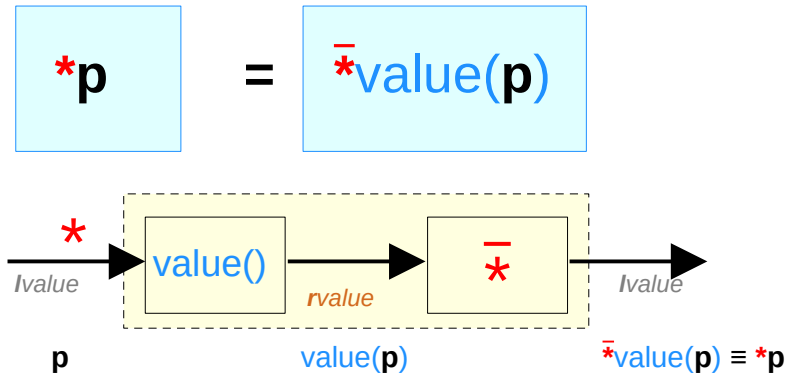


$$\begin{array}{l} \boxed{\&X} = \boxed{\text{value}(\bar{\&}X)} \\ \text{C Expressions} \quad \text{Mixed Expressions} \\ \boxed{\bar{\&}X} = \boxed{\text{value}^{-1}(\&X)} \end{array}$$

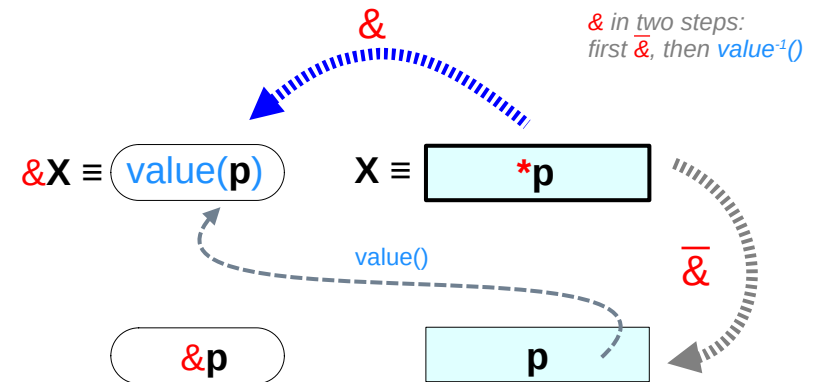
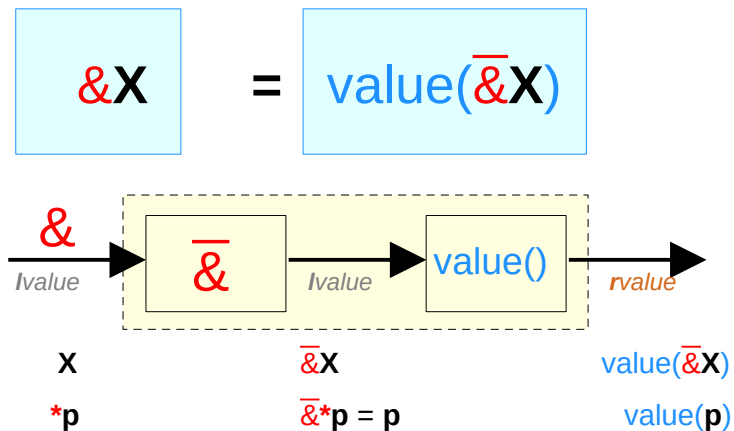


# & and \* operators in pointer de-referencing (2)

## Two step De-reference \* operation

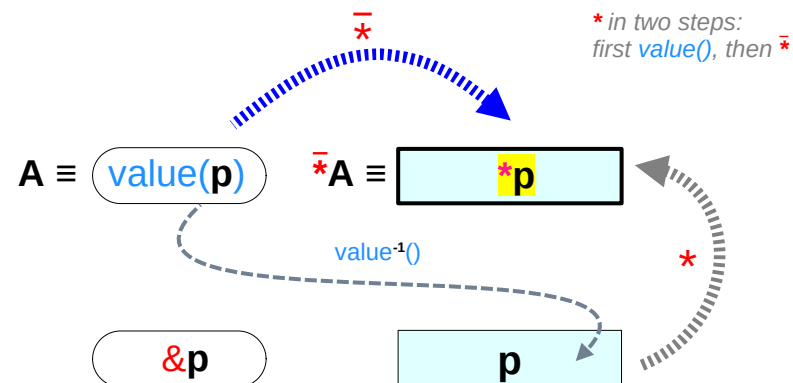
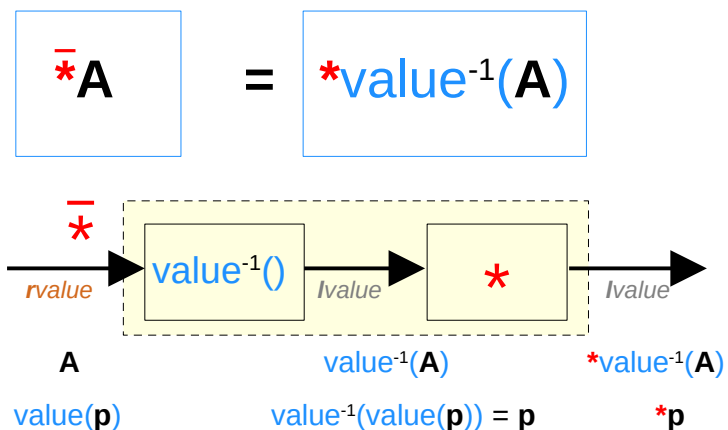


## Two step Address-of & operation

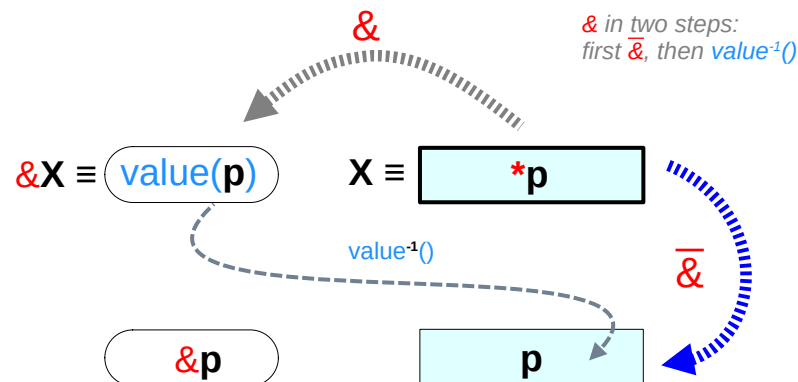
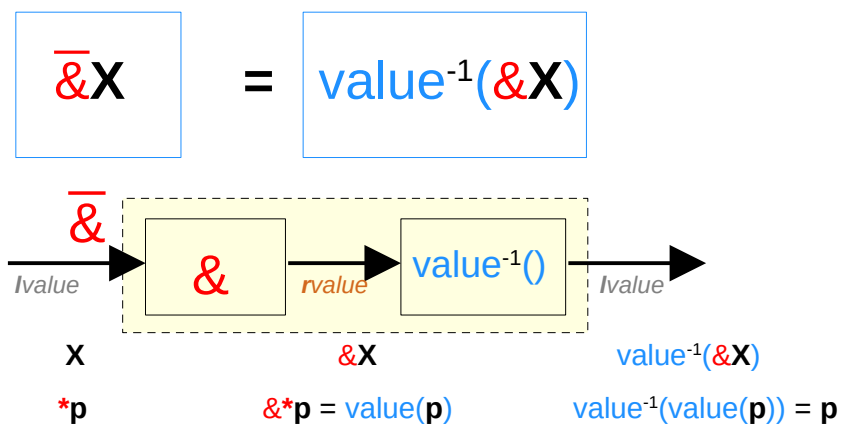


# $\bar{\&}$ and $\bar{*}$ operators in pointer de-referencing (3)

## Two step De-reference $\bar{*}$ operation



## Two step Address-of $\bar{\&}$ operation



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

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

$$\&X = \text{value}(\bar{\&X})$$

C Expressions

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

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

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

$\&X$

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

$\&p$

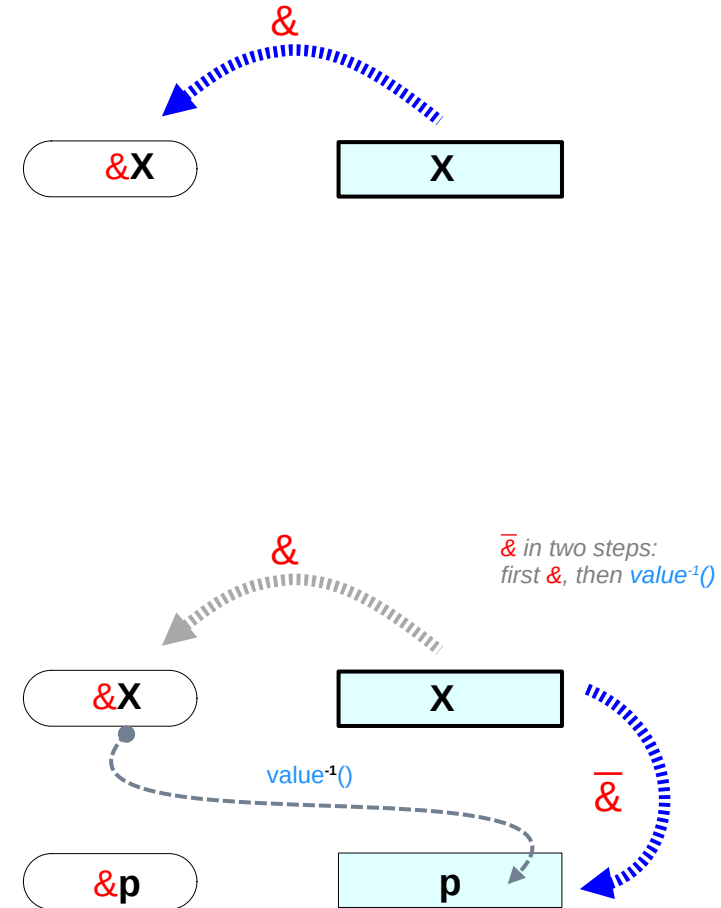
$\&a$

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

$\bar{\&X} = p$

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

~~$\bar{\&p}$~~   
 ~~$\bar{\&a}$~~



# Two step de-reference $\bar{*}$ and $*$ operators

for  $\bar{*}A$ , A can be  $\text{value}(p)$ ,  $\&p$ ,  $\&a$

$$\bar{*}A = *value^{-1}(A)$$

$\text{value}(p)$  value of a pointer variable  
 $\&p$  address of a pointer variable  
 $\&a$  address of a primitive variable

$\bar{*}A$

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

$*\&p = p$

$*\&a = a$

for  $\bar{*}A = *p$ , A must be  $\text{value}(p)$

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

C Expressions

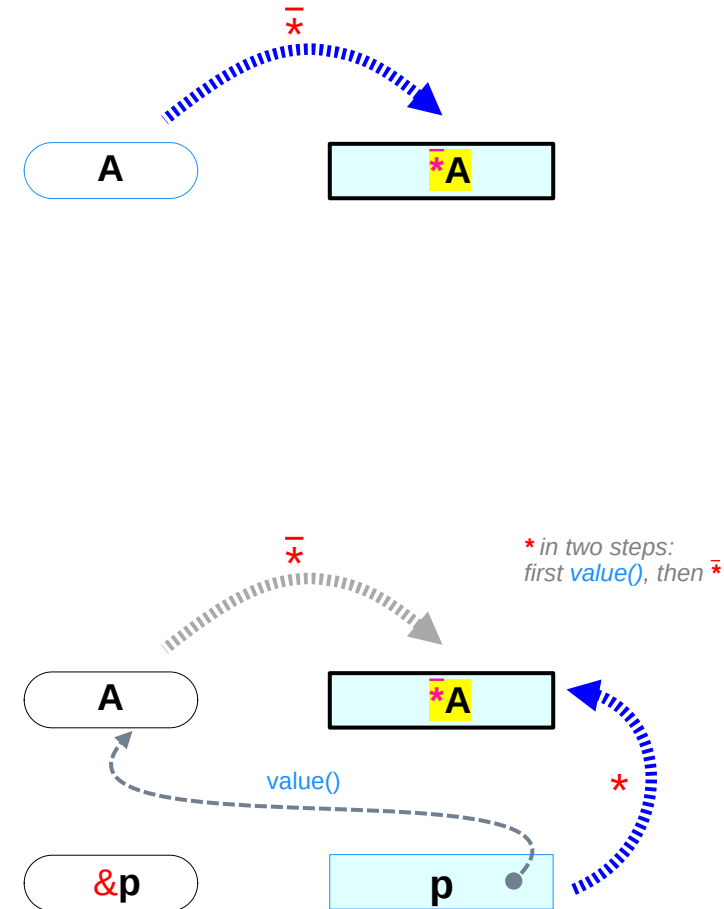
$\text{value}(p)$  value of a pointer variable  
 ~~$\&p$  address of a pointer variable~~  
 ~~$\&a$  address of a primitive variable~~

$*p$

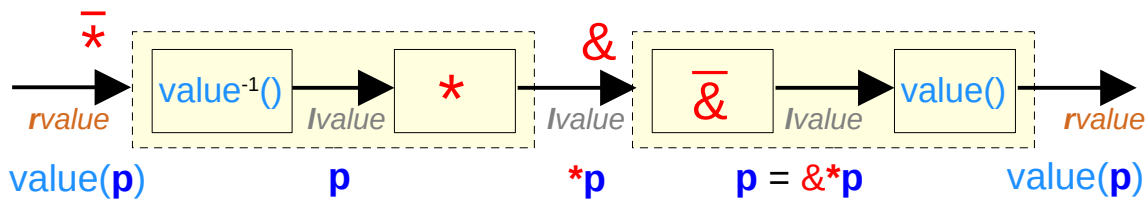
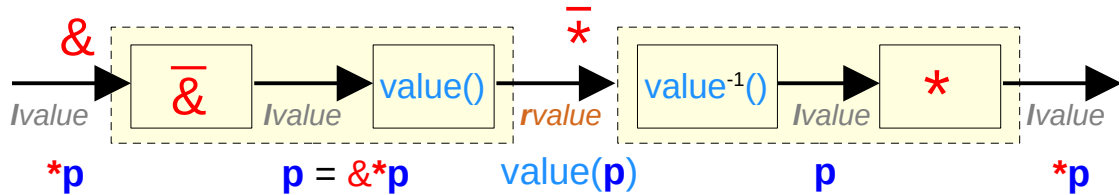
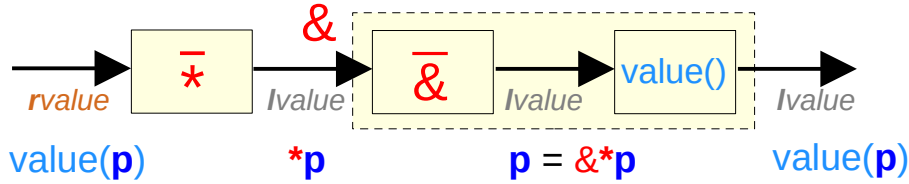
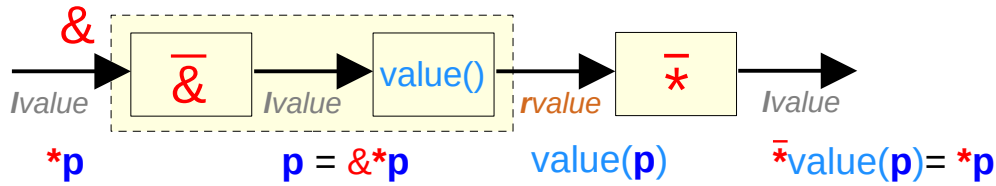
$*p \equiv \bar{*} \text{value}(p)$

~~$*\&p = p$~~

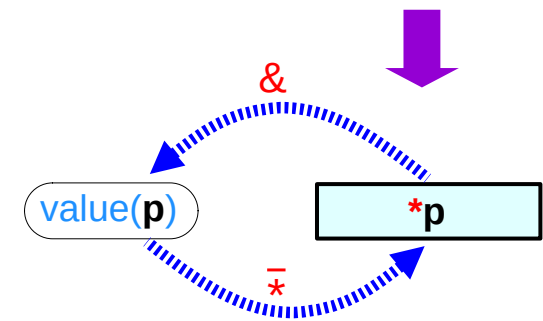
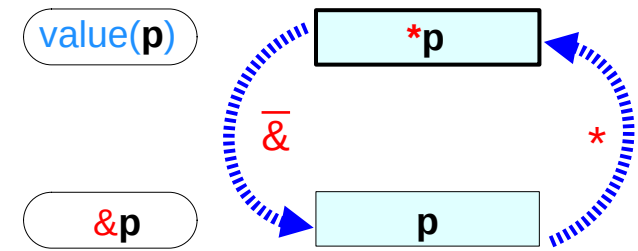
~~$*\&a = a$~~



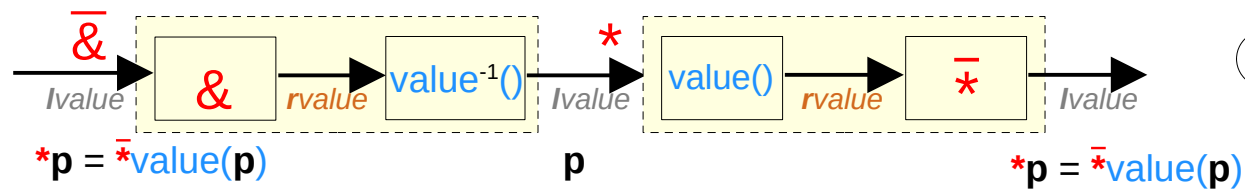
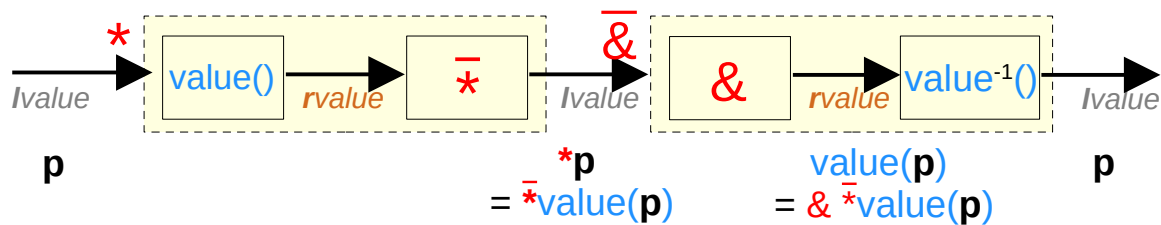
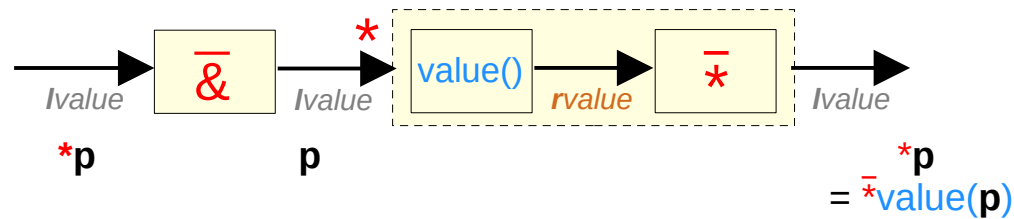
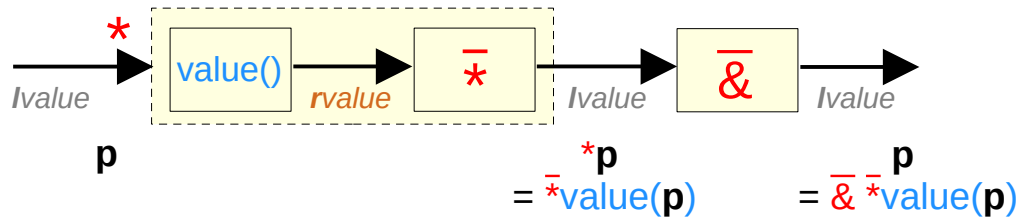
# Inverse operators : $\&$ and $\bar{*}$ in pointer de-referencing



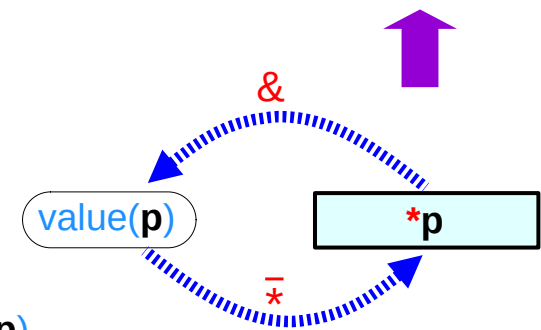
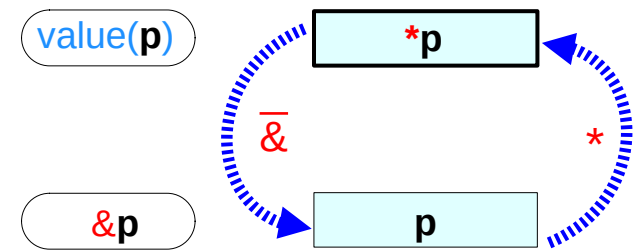
$\bar{*} value(p) = *p$   
 $\&\bar{*} value(p) = \&*p = value(p)$   
 $\bar{*}\&\bar{*} value(p) = \bar{*}\&*p = \bar{*}value(p) = *p$



# Inverse operators of & and \*



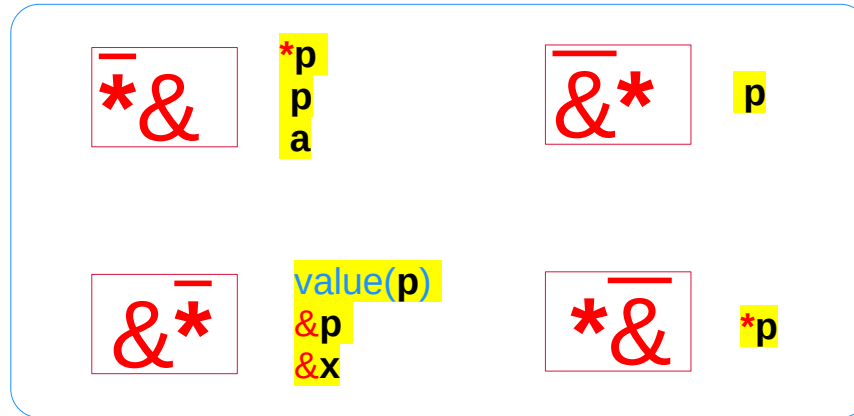
$\bar{*} value(p) = *p$   
 $\bar{\&} * p = p$   $p = \bar{\&} \bar{*} value(p)$   
 $* \bar{\&} * p = *p$   $value(p) = \bar{\&} \bar{*} value(p)$



# Inverse operators in pointer referencing (1)

$$\begin{aligned} \overline{*} \& *p &= *p \\ \overline{*} \& p &= p \\ \overline{*} \& a &= a \end{aligned}$$

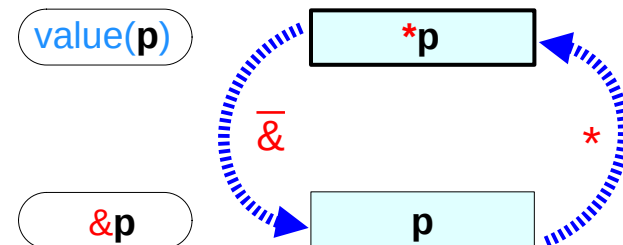
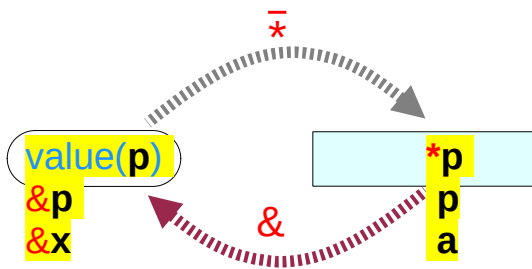
$$\begin{aligned} \& \overline{*} \text{value}(p) &= \text{value}(p) \\ \& \overline{*} \&p &= \&p \\ \& \overline{*} \&a &= \&a \end{aligned}$$



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

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

Math expressions

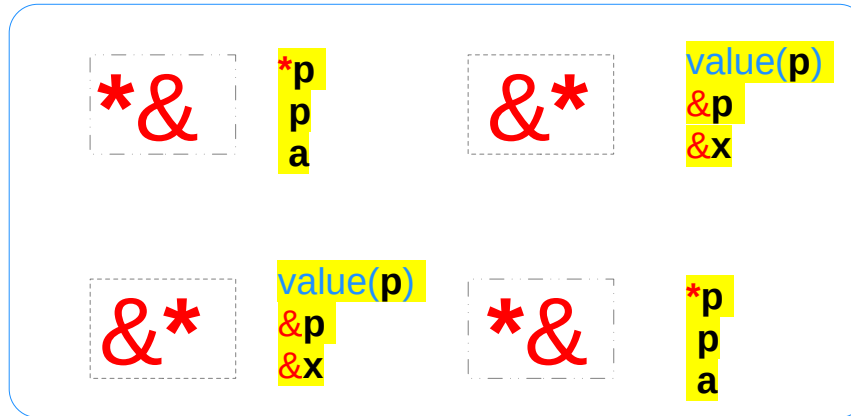




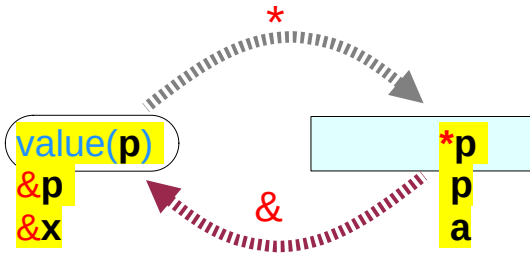
# Inverse operators in pointer referencing (2)

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

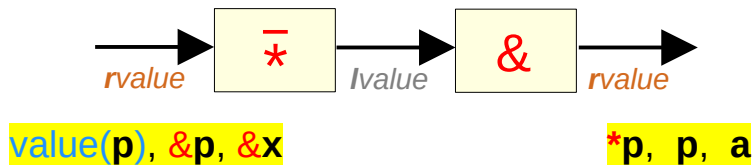
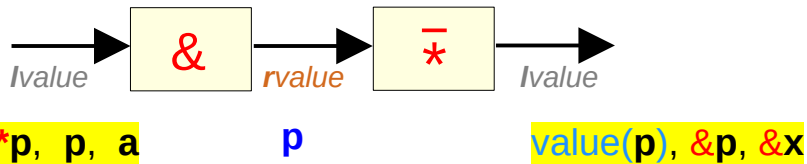
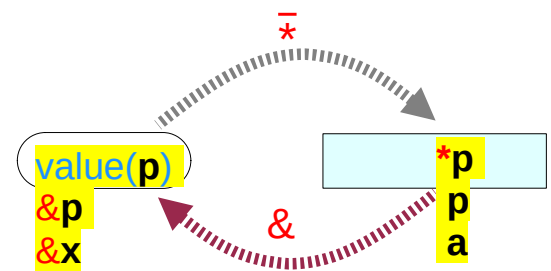
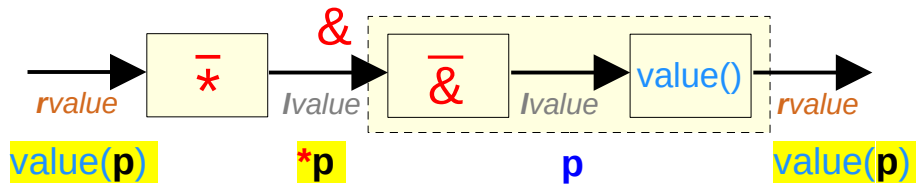
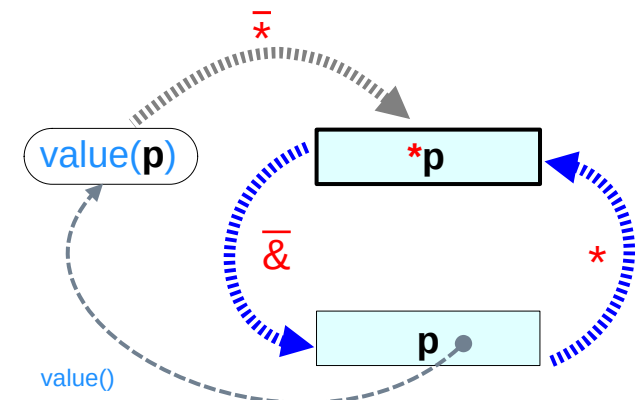
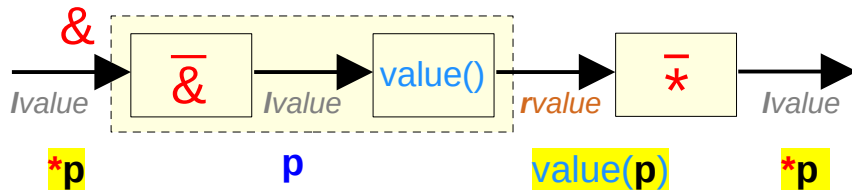
`&*value(p) = value(p)`  
`&*&p = &p`  
`&*&a = &a`



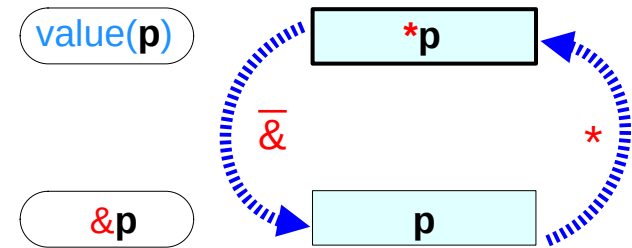
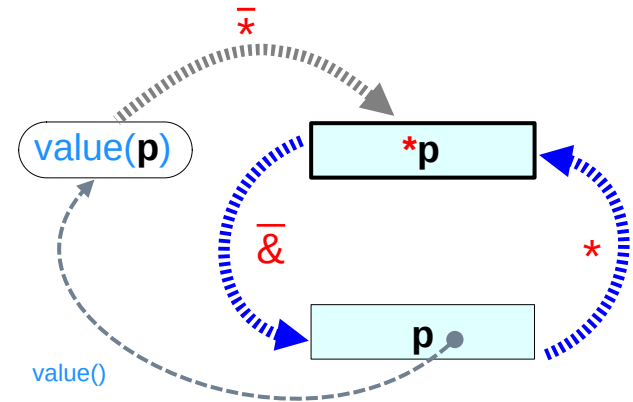
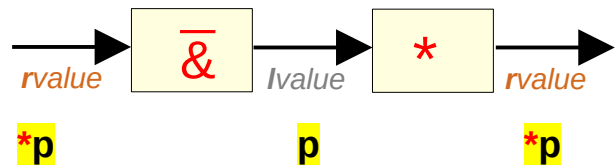
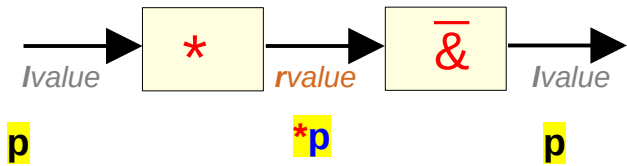
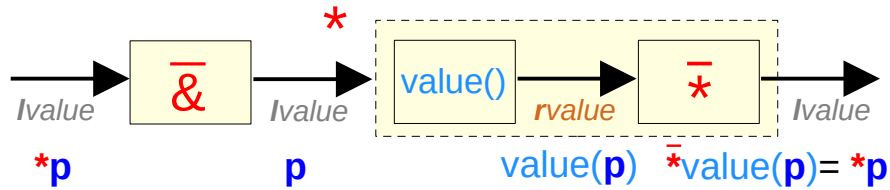
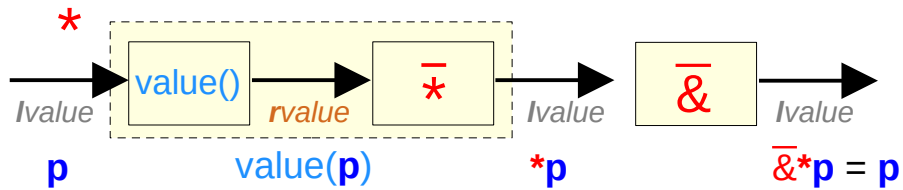
C expressions



# 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