Address-of and dereference operators

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

Young Won Lim
11/6/25

& address-of operator

***** dereference operator

Address-of operator and dereferencing operator

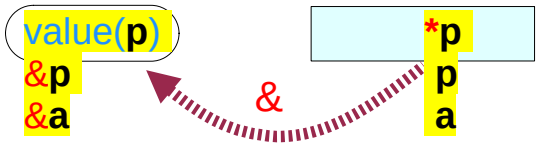
*address-of operator **&** : the address of a variable*

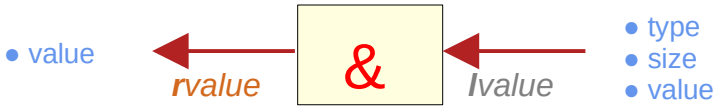
*a **variable** has a memory location whose value can be changed by an assignment*

*a **variable** must be an **l**value*

__&variable__ returns the address of a variable

*__&variable__ returns an **r**value*





Address-of operator and dereferencing operator

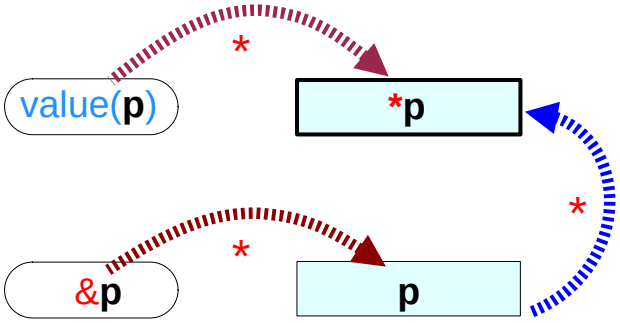
dereferencing operator ***** *: the content at an address*

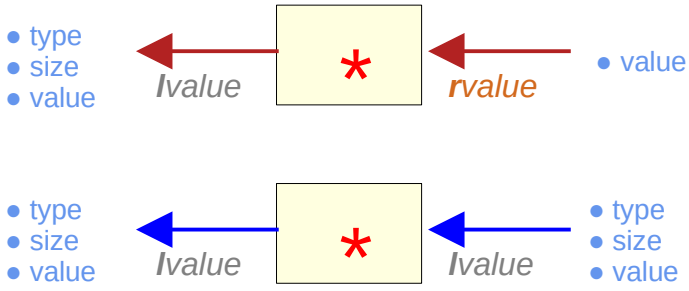
an ***address*** *can be an address value or a pointer variable that holds an address value*

an ***address*** *must be an* ***r****value, or evaluated to be an* ***r****value*

***** ***address*** *returns the content value at the address*

***** ***address*** *returns an* ***l****value*





***p** ≡ ***value(p)**

lvalue and rvalue in assignments

an assignment statement

Left Hand Side *Right Hand Side*

LHS = ***RHS***

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

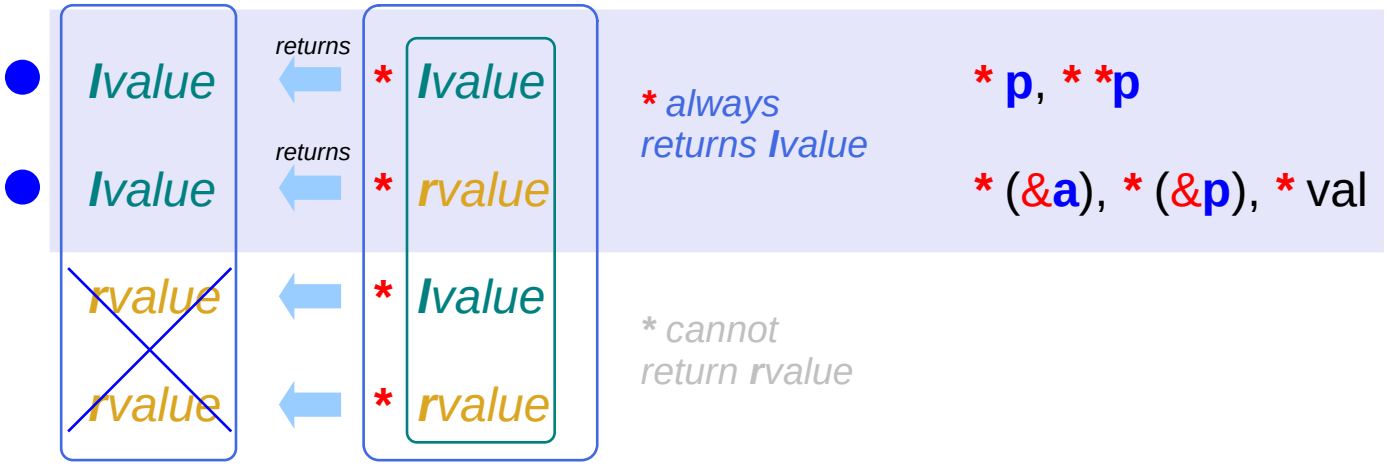
```
int   a,  b = 10 ;
int  *p, *q = &a ;
```

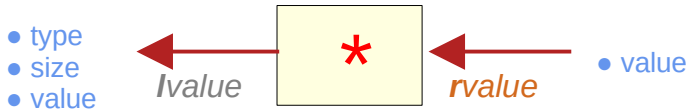
```
 a, b, p, q : lvalues   ... variables  ... RW
 *p, *q     : lvalues   ... variables  ... RW
&a, &b      : rvalues   ... constants  ... RO
```

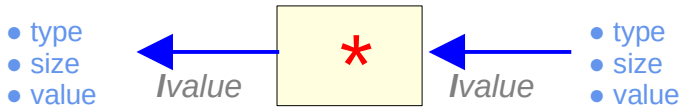
```
lvalue = lvalue
lvalue = rvalue
rvalue = lvalue
rvalue = rvalue
```

```
 p  =  q  ;
 p  = &a  ;
&a  =  p  ;
&a  = &b  ;
```

lvalue and rvalue with * and & operators





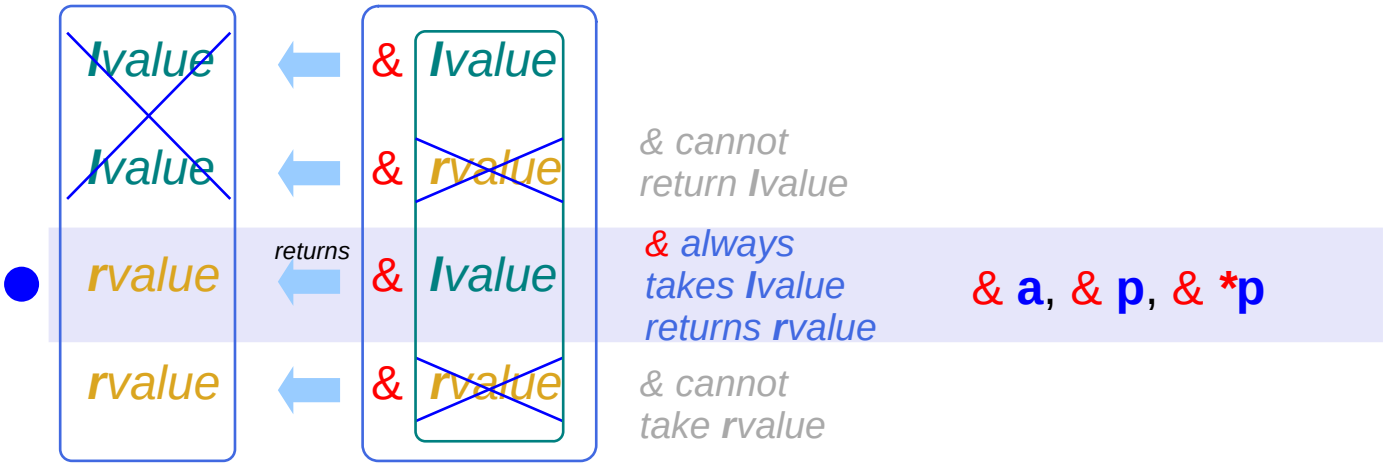


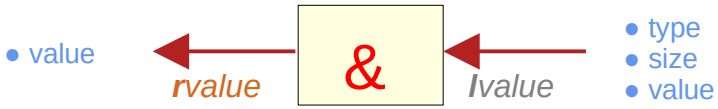
*p ≡ *value(p)

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

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

lvalue and rvalue with * and & operators





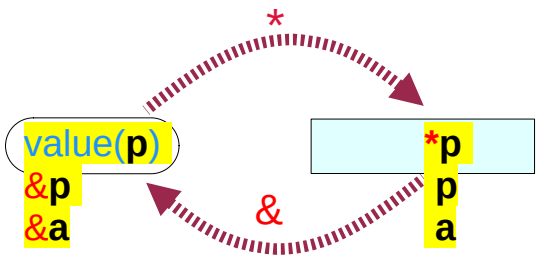
& can be applied
to only an **lvalue** variable and
returns only an **rvalue** address

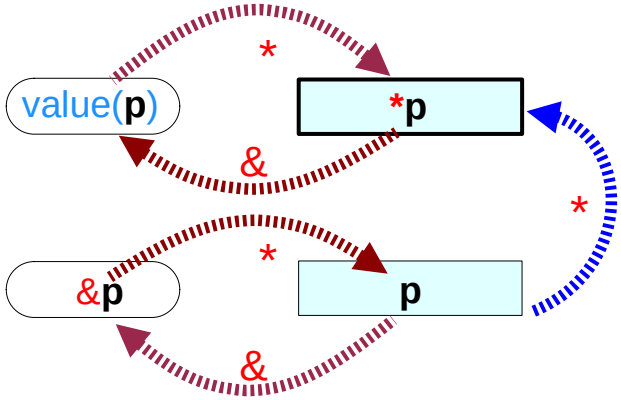
Variable types

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

C operators : *, &

C operators : *, &



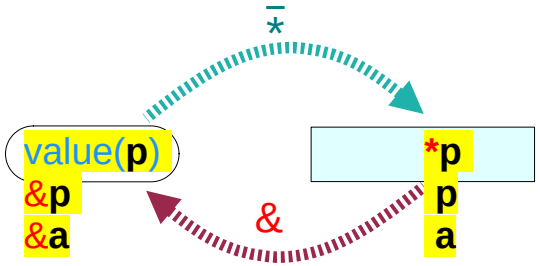


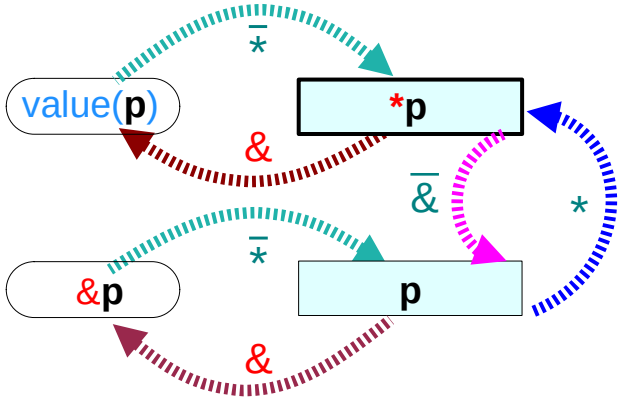
		*	~~&~~	lvalue
lvalue	p	*	p	lvalue
lvalue	* p	*	* p	lvalue
lvalue	* val	*	val	rvalue

rvalue	& p	&	a	lvalue
rvalue	& p	&	p	lvalue
rvalue	& * p	&	* p	lvalue
		&	~~val~~	rvalue

Extending C operators with $\bar{*}$, $\bar{\&}$ (1)

Extending C operators with Mathematical operators ($\bar{*}$, $\bar{\&}$)



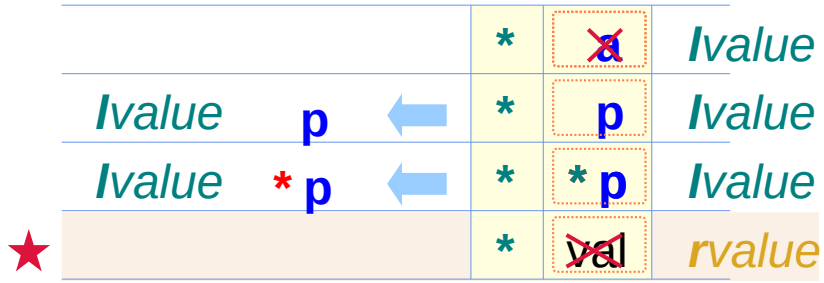


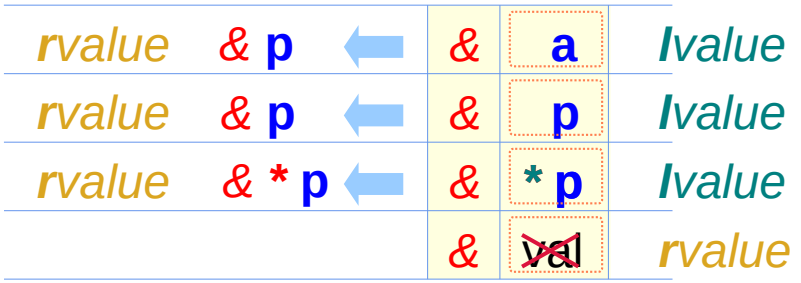
		*	~~&~~ a	lvalue
lvalue	p	*	p	lvalue
lvalue	* p	*	* p	lvalue
★ lvalue	* val	*	val	rvalue

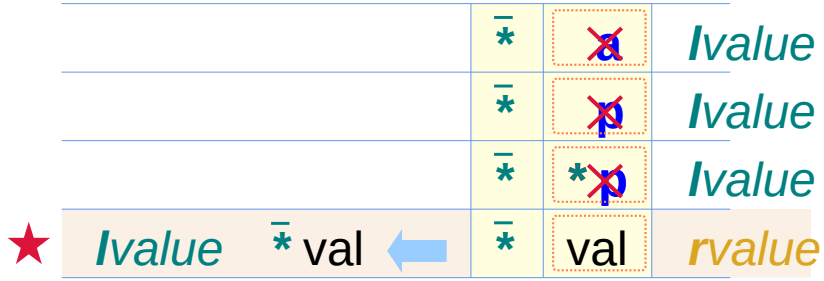
rvalue	& p	&	a	lvalue
rvalue	& p	&	p	lvalue
rvalue	& * p	&	* p	lvalue
★ lvalue	p	&	* p	lvalue

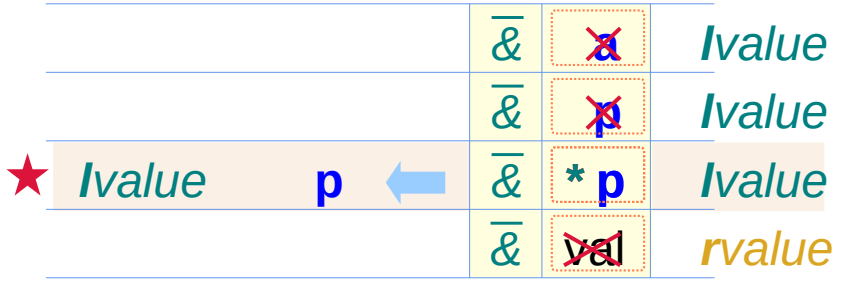
Extending C operators with $\bar{*}$, $\bar{\&}$ (2)

Extending C operators with Mathematical operators ($\bar{*}$, $\bar{\&}$)



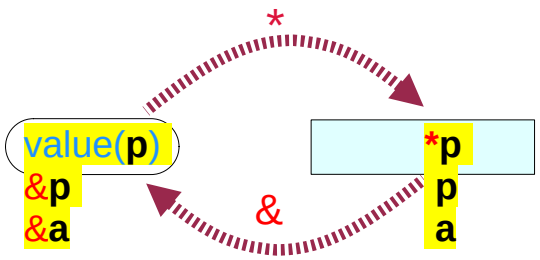


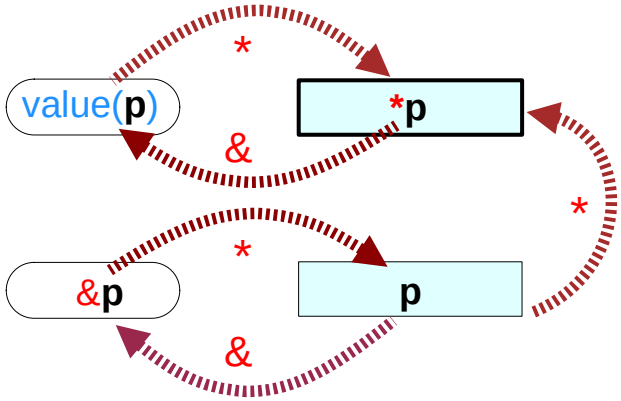




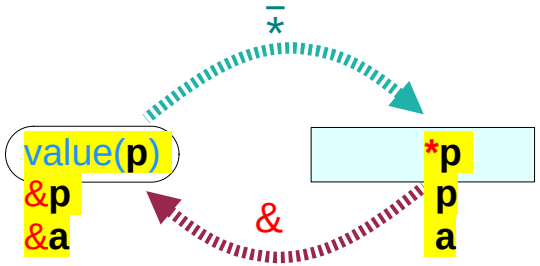
Comparisons (1)

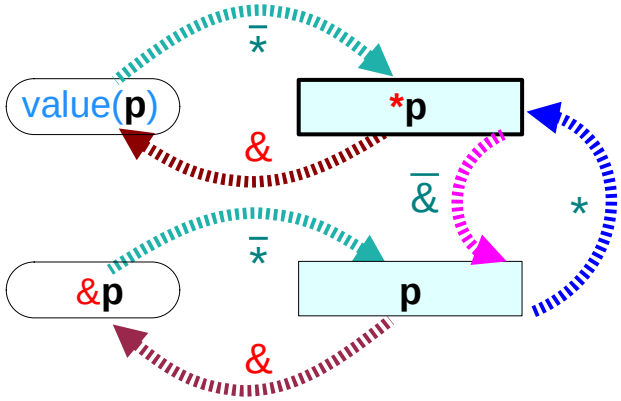
C operators : *, &





Extending C operators with Mathematical operators ($\bar{*}$, $\bar{\&}$)





Comparisons (2)

C operators : *, &

		*	~~a~~	lvalue		rvalue	& a	⟵	&	a	lvalue
lvalue	p	*	p	lvalue		rvalue	& p	⟵	&	p	lvalue
lvalue	* p	*	* p	lvalue		rvalue	& * p	⟵	&	* p	lvalue
lvalue	* val	*	val	rvalue					&	~~val~~	rvalue

Extending C operators with Mathematical operators ($\bar{*}$, $\bar{\&}$)

		*	~~a~~	lvalue		rvalue	& a	⟵	&	a	lvalue
lvalue	p	*	p	lvalue		rvalue	& p	⟵	&	p	lvalue
lvalue	* p	*	* p	lvalue		rvalue	& * p	⟵	&	* p	lvalue
★ lvalue	$\bar{*}$ val	$\bar{*}$	val	rvalue	★	lvalue	p	⟵	$\bar{\&}$	* p	lvalue

Comparisons (3)

C operators : *, &

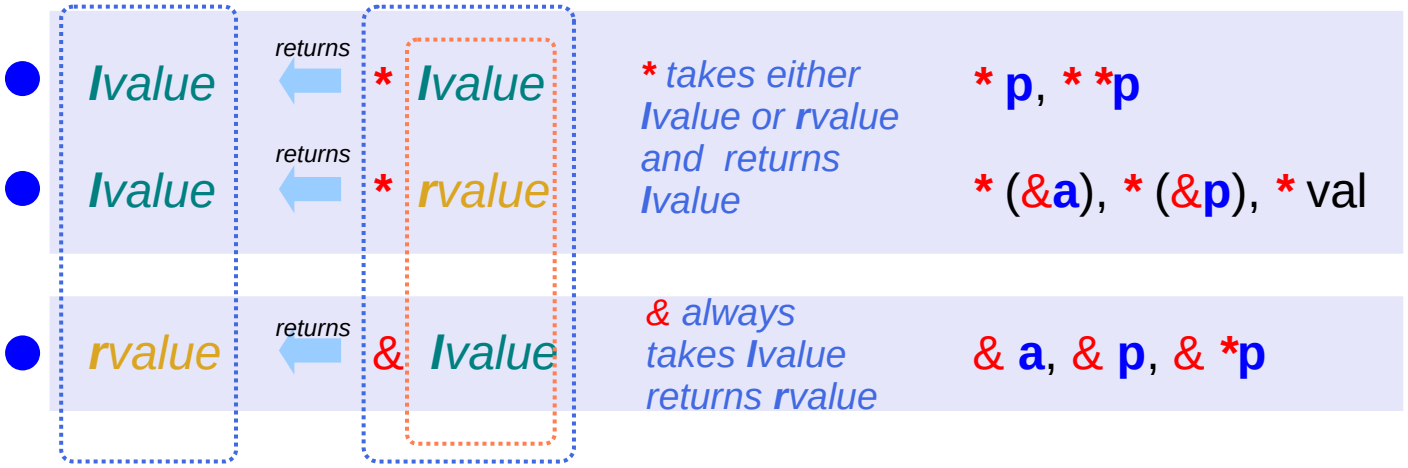
		*	~~a~~	lvalue	rvalue	& p	&	a	lvalue
lvalue	p	*	p	lvalue	rvalue	& p	&	p	lvalue
lvalue	* p	*	* p	lvalue	rvalue	& * p	&	* p	lvalue
lvalue	* val	*	val	rvalue			&	~~val~~	rvalue

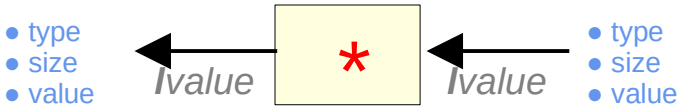
Extending C operators with Mathematical operators (*̄, &̄)

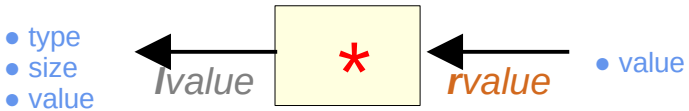
		*	~~a~~	lvalue	rvalue	& p	&	a	lvalue
lvalue	p	*	p	lvalue	rvalue	& p	&	p	lvalue
lvalue	* p	*	* p	lvalue	rvalue	& * p	&	* p	lvalue
★		*	~~val~~	rvalue			&	~~val~~	rvalue

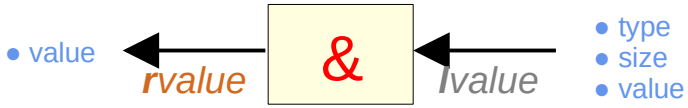
		*̄	~~a~~	lvalue			&̄	~~a~~	lvalue
		*̄	~~p~~	lvalue			&̄	~~p~~	lvalue
		*̄	~~*p~~	lvalue	★	lvalue p	&̄	* p	lvalue
★	lvalue *̄ val	*̄	val	rvalue			&̄	~~val~~	rvalue

lvalue and rvalue with * and & operators





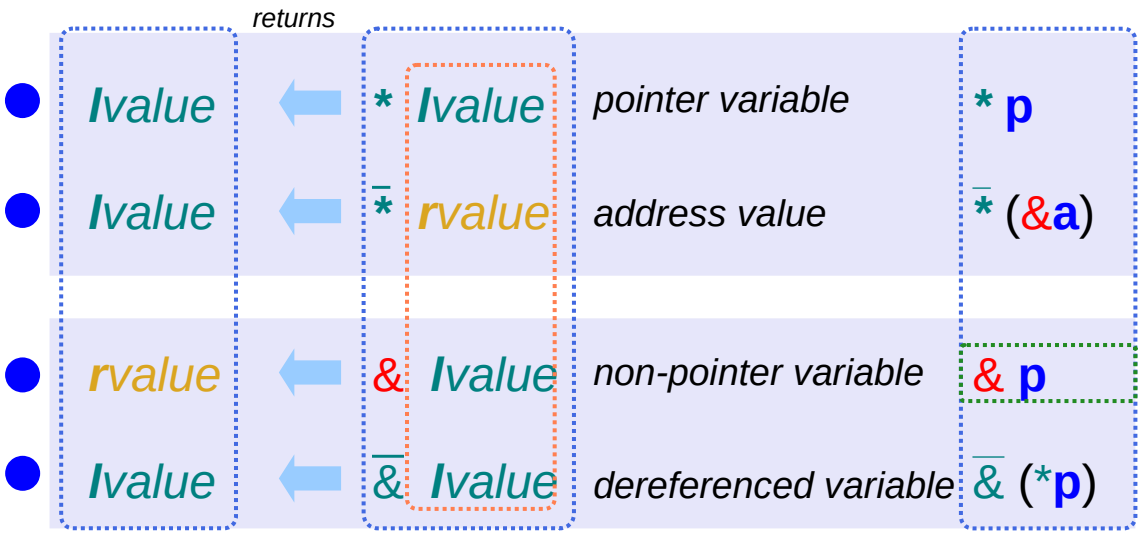




* (an **lvalue** variable)
* (an **rvalue** address)
* **operand** : an **lvalue** variable
* can be applied successively.

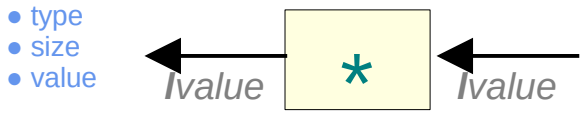
& (an **lvalue** variable)
& **operand** : an **rvalue** address,
& cannot be applied successively.

lvalue and rvalue with * and & operators

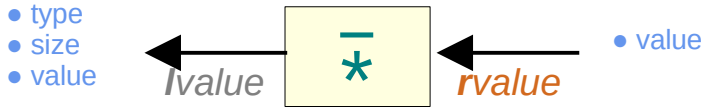


* can be applied successively.
& can be applied successively.

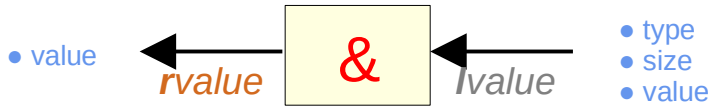
**p
&**p = *p
&&**p = &*p = p



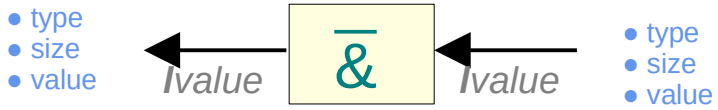
* (an **lvalue** variable)
* **operand**: an **lvalue** variable



* (an **rvalue** address)
* **operand**: an **lvalue** variable



& (an **lvalue** variable)
& **operand**: an **rvalue** address



& (an dereferenced **lvalue** variable)
& **operand**: an **lvalue** variable

Recursive application of the address-of operator

$* = \bar{*}v$

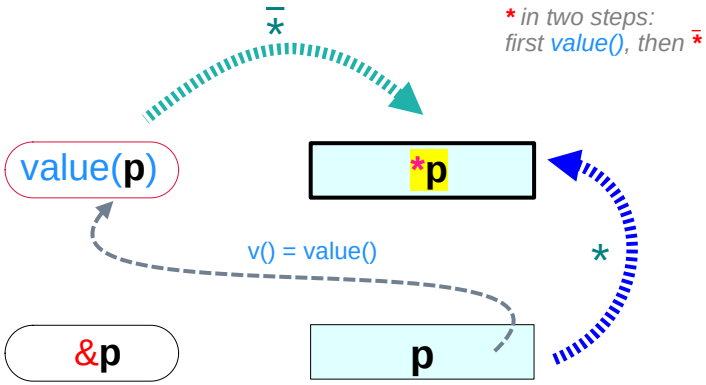
$*v^{-1} = \bar{*}vv^{-1}$

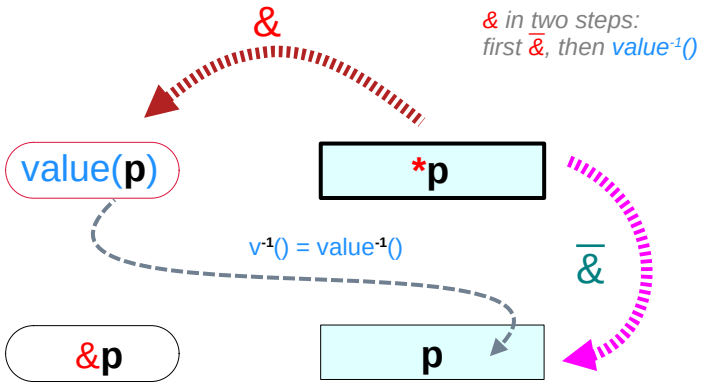
$*v^{-1} = \bar{*}$

$\bar{\&} = v^{-1}\&$

$v\bar{\&} = vv^{-1}\&$

$v\bar{\&} = \&$



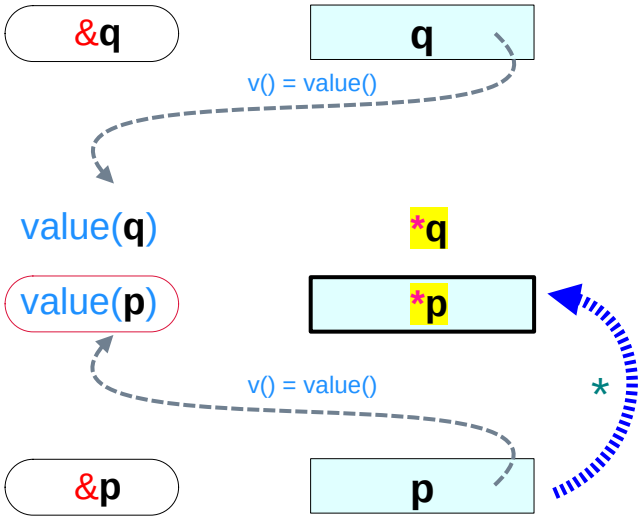


$v() = value()$

$v^{-1}() = value^{-1}()$

When two pointers point to the same location

When two pointers **p** and **q**
point to the same memory location
value(**p**) = value(**q**)
value^{-1}(***p**) = **p** or
value^{-1}(***q**) = **q**
Cannot determine



v() = value()

value(**p**) = value(**q**) = **x**
then value^{-1}(**x**) = **p** or **q** ?

When value^{-1}() is prohibited (1)

$* = \bar{*}v$

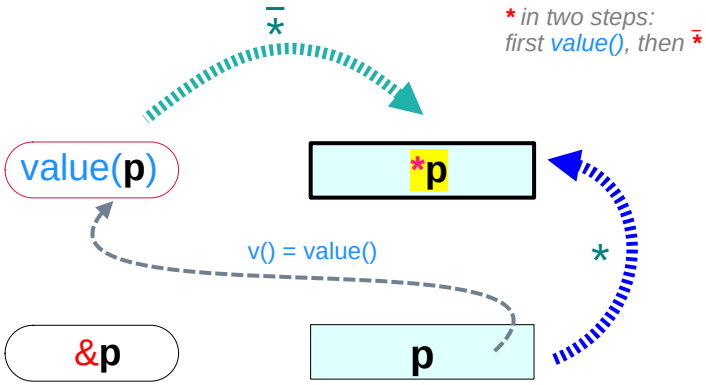
$*v^{-1} = \bar{*}vv^{-1}$

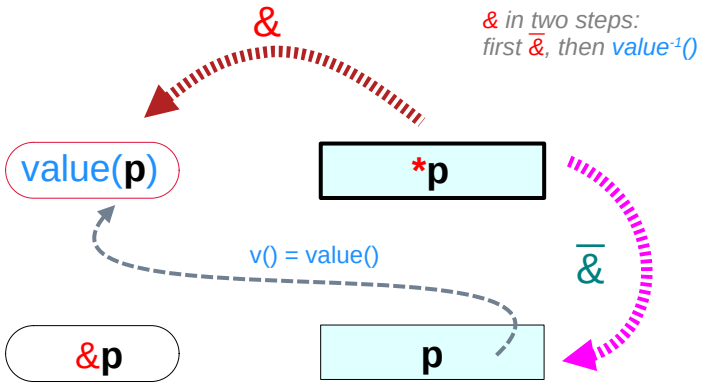
$*v^{-1} = \bar{*}$

$\bar{\&} = v^{-1}\&$

$v\bar{\&} = vv^{-1}\&$

$v\bar{\&} = \&$





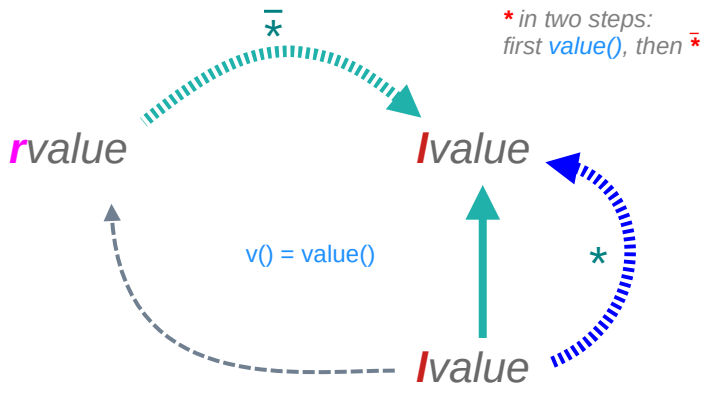
$v() = value()$

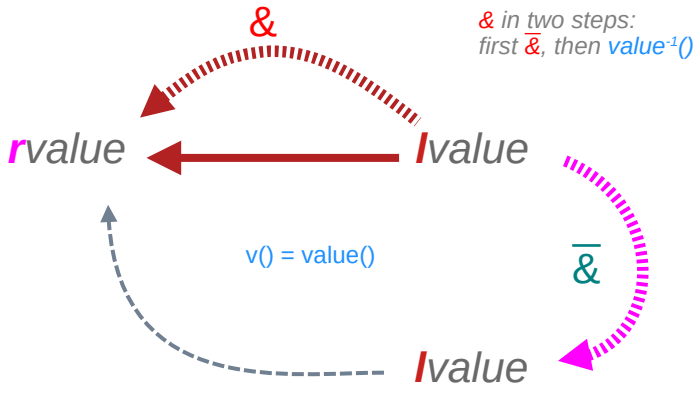
$v^{-1}() = value^{-1}()$

When value^{-1}() is prohibited (2)

$$* = \overline{*}v$$

$$v\overline{\&} = \&$$

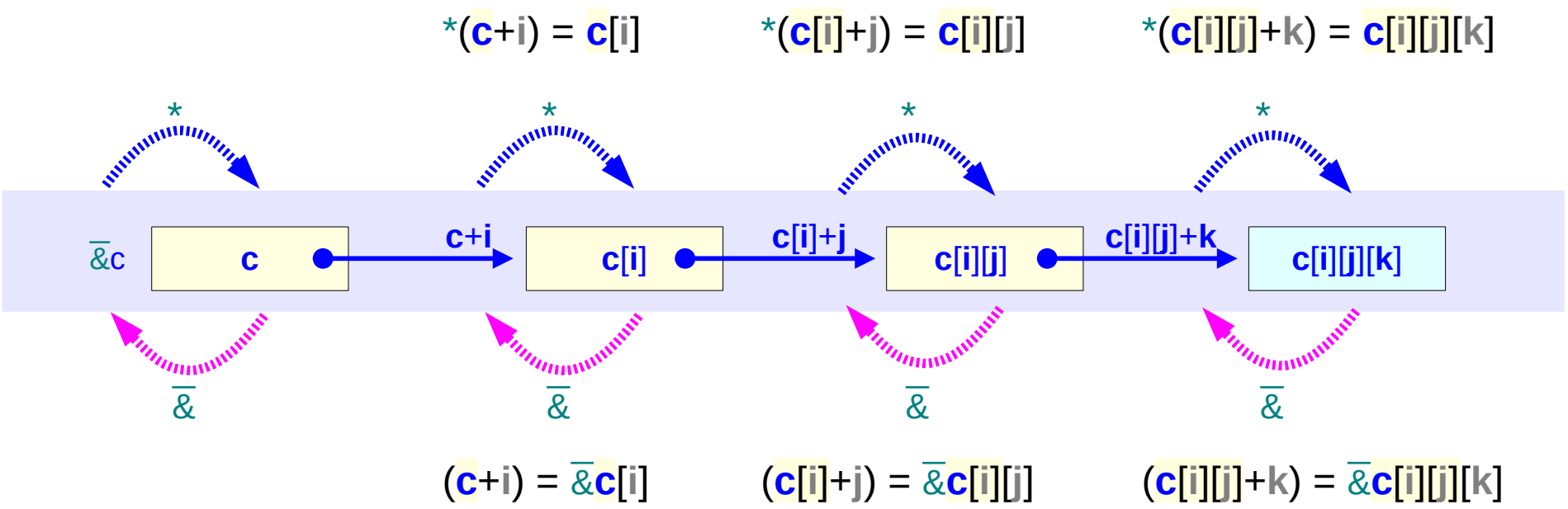




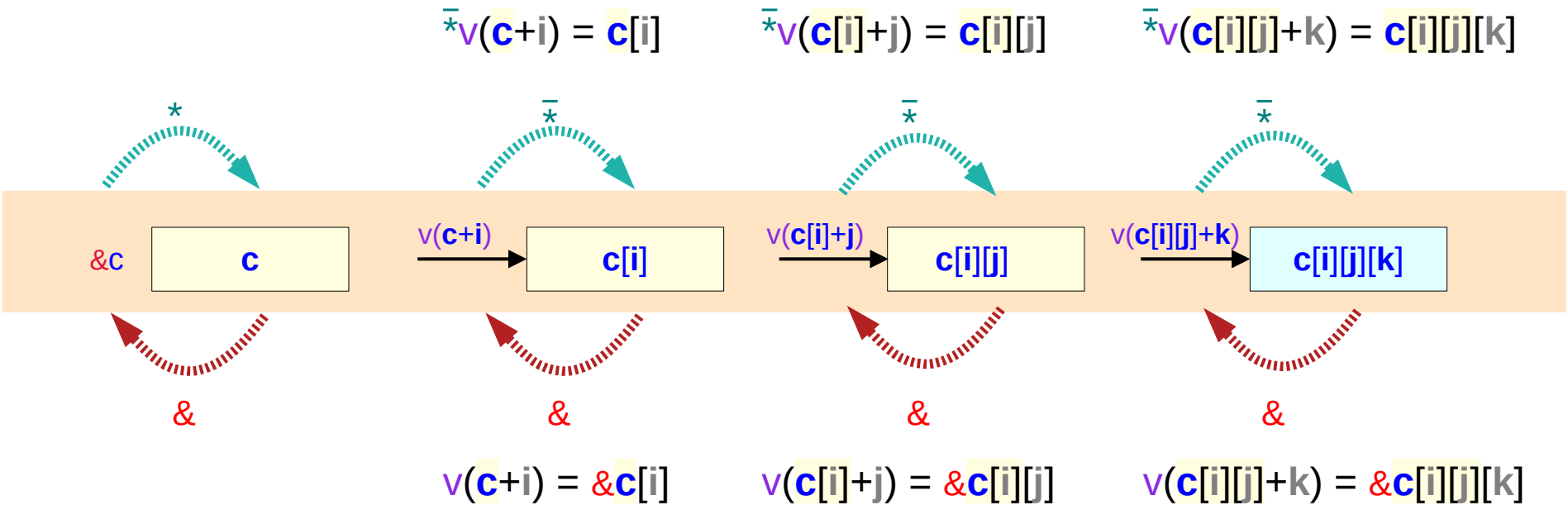
v() = value()

v() = value()

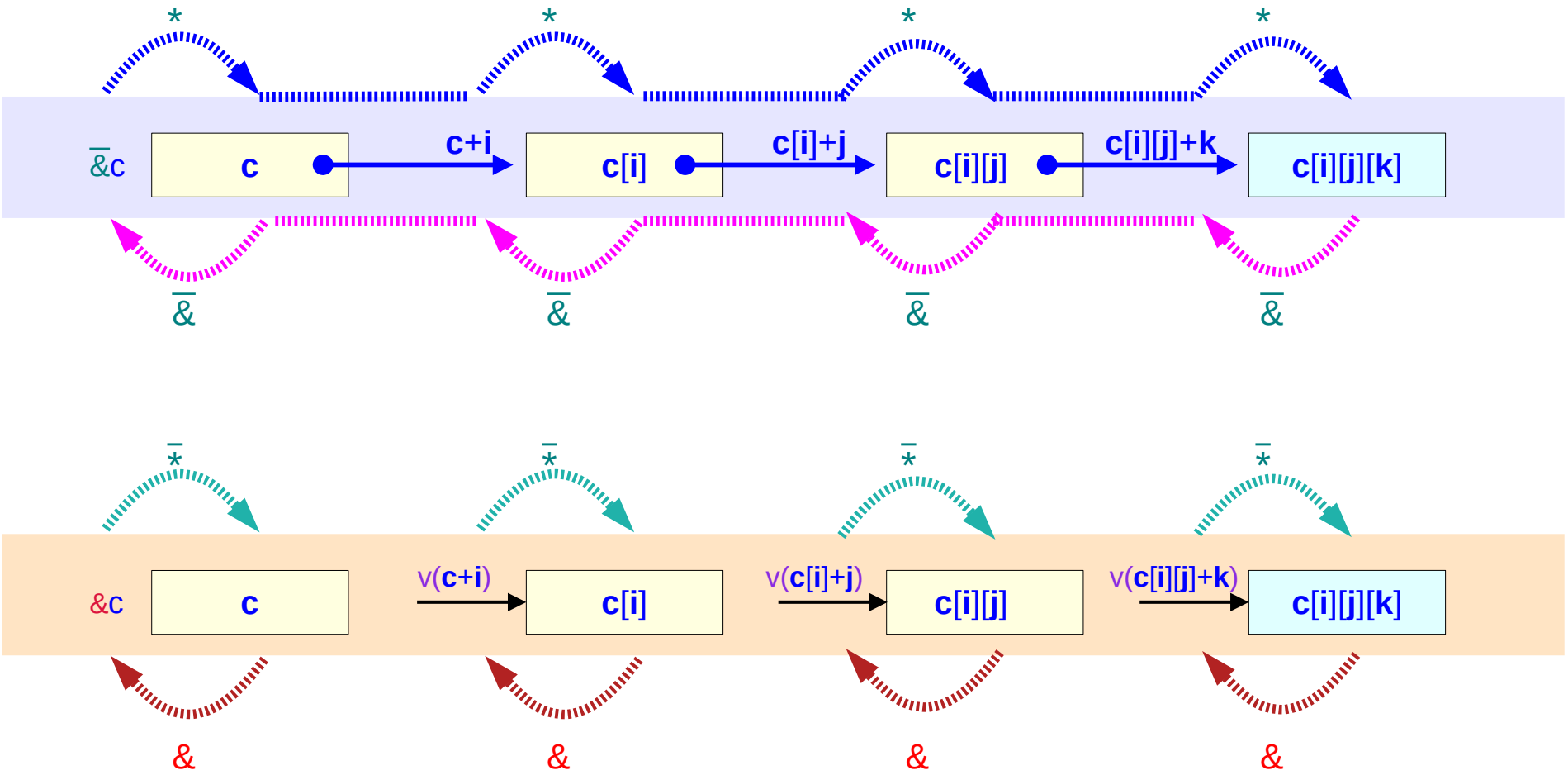
Inverse relationship between * and & operators



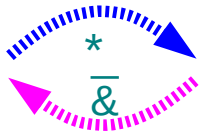
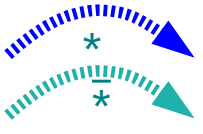
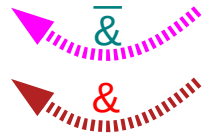
Inverse relationship between *̄ and & operators



Inverse relationship (1)



Inverse relationship (2)

lvalue  *lvalue*	*(c+i) = c[i] (c+i) = &c[i]	*(c[i]+j) = c[i][j] (c[i]+j) = &c[i][j]	*(c[i][j]+k) = c[i][j][k] (c[i][j]+k) = &c[i][j][k]
rvalue  *lvalue*	*v(c+i) = c[i] v(c+i) = &c[i]	*v(c[i]+j) = c[i][j] v(c[i]+j) = &c[i][j]	*v(c[i][j]+k) = c[i][j][k] v(c[i][j]+k) = &c[i][j][k]
lvalue *rvalue*  *lvalue* *lvalue*	*(c+i) = c[i] *v(c+i) = c[i]	*(c[i]+j) = c[i][j] *v(c[i]+j) = c[i][j]	*(c[i][j]+k) = c[i][j][k] *v(c[i][j]+k) = c[i][j][k]
lvalue *rvalue*  *lvalue* *lvalue*	(c+i) = &c[i] v(c+i) = &c[i]	(c[i]+j) = &c[i][j] v(c[i]+j) = &c[i][j]	(c[i][j]+k) = &c[i][j][k] v(c[i][j]+k) = &c[i][j][k]

Types of sub-arrays (1)

```
s2.c: In function ‘main’:
s2.c:7:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[3][4]’ [-Wformat=]
    7 |   printf("%d \n", a);
      |           ~^      ~
      |            |      |
      |            int    int (*)[3][4]
s2.c:8:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[4]’ [-Wformat=]
    8 |   printf("%d \n", a[0]);
      |           ~^      ~~~~
      |            |      |
      |            int    int (*)[4]
s2.c:9:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int *’ [-Wformat=]
    9 |   printf("%d \n", a[0][0]);
      |           ~^      ~~~~~~~
      |            |        |
      |            int       int *
      |           %ls
```

```
s2.c:11:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[3][4]’ [-Wformat=]
   11 |   printf("%d \n", a+1);
      |           ~^      ~~~
      |            |      |
      |            int    int (*)[3][4]
s2.c:12:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[4]’ [-Wformat=]
   12 |   printf("%d \n", a[0]+1);
      |           ~^      ~~~~~~
      |            |        |
      |            int       int (*)[4]
s2.c:13:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int *’ [-Wformat=]
   13 |   printf("%d \n", a[0][0]+1);
      |           ~^      ~~~~~~~~
      |            |          |
      |            int         int *
      |           %ls
```

```
#include <stdio.h>

int main(void) {

  int a[2][3][4];

  printf("%d \n", a);
  printf("%d \n", a[0]);
  printf("%d \n", a[0][0]);

  printf("%d \n", a+1);
  printf("%d \n", a[0]+1);
  printf("%d \n", a[0][0]+1);

  printf("%d \n", &a[0]);
  printf("%d \n", &a[0][0]);
  printf("%d \n", &a[0][0][0]);


}
```

Types of sub-arrays (2)

```
s2.c:15:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[3][4]’ [-Wformat=]
   15 |   printf("%d \n", &a[0]);
      |          ~^      ~~~~~
      |           |     |
      |           int   int (*)[3][4]
s2.c:16:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int (*)[4]’ [-Wformat=]
   16 |   printf("%d \n", &a[0][0]);
      |          ~^      ~~~~~~~~
      |           |     |
      |           int   int (*)[4]
s2.c:17:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int *’ [-Wformat=]
   17 |   printf("%d \n", &a[0][0][0]);
      |          ~^      ~~~~~~~~~~~
      |           |     |
      |           int   int *
      |          %ls
y
```

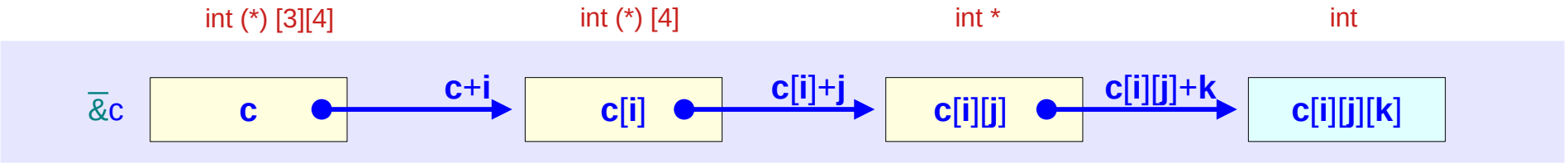
```
#include <stdio.h>

int main(void) {

  int a[2][3][4];

  printf("%d \n", a);
  printf("%d \n", a[0]);
  printf("%d \n", a[0][0]);

  printf("%d \n", a+1);
  printf("%d \n", a[0]+1);
  printf("%d \n", a[0][0]+1);

  printf("%d \n", &a[0]);
  printf("%d \n", &a[0][0]);
  printf("%d \n", &a[0][0][0]);


}
```

Types of sub-arrays (3)

```
int c[2][3][4];
```

c		:: int (*) [3][4]
c[i]	= *(**c**+i)	:: int (*) [4]
c[i][j]	= *(**c**[i]+j)	:: int *
c[i][j][k]	= *(**c**[i][j]+k)	:: int

&**c**[i]	= (**c**+i)	:: int (*) [3][4]
&**c**[i][j]	= (**c**[i]+j)	:: int (*) [4]
&**c**[i][j][k]	= (**c**[i][j]+k)	:: int (*)



Dimensional Parentheses (1)

int c[2][3][4];

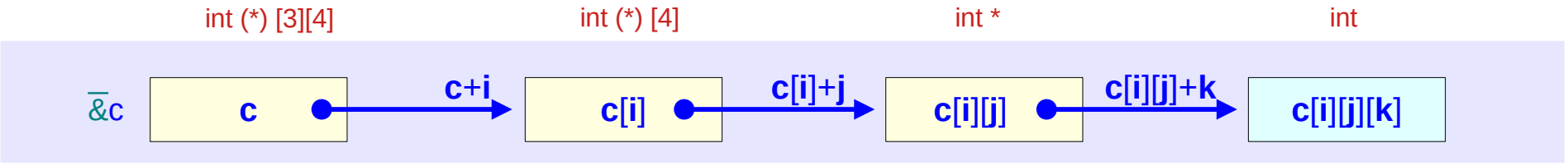
(c+i)	= ?
((c+i)+j)	= ?
(((c+i)+j)+k)	= ?

≠ c+i
≠ c+i+j
≠ c+i+j+k

(c+i)	= $(c+i)_{[3][4]}$
((c+i)+j)	= $((c+i)_{[3][4]}+j)_{[4]}$
(((c+i)+j)+k)	= $(((c+i)_{[3][4]}+j)_{[4]}+k)$

dimensional parentheses
not a simple parenthesis, but associated with the dimensions of an array defined

mathematical expressions obtained by mathematical operator &



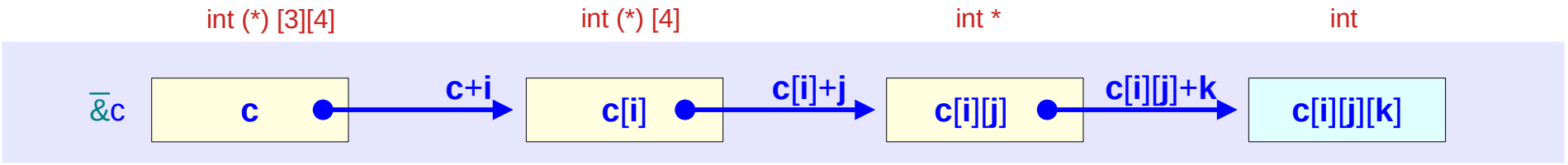
Dimensional Parentheses (2)

int c[2][3][4];

(c+i)	= $(c+i)_{[3][4]}$
((c+i)+j)	= $((c+i)_{[3][4]}+j)_{[4]}$
(((c+i)+j)+k)	= $(((c+i)_{[3][4]}+j)_{[4]}+k)$

type(c+i)	= int (*) [3][4]
type((c+i)+j))	= int (*) [4]
type(((c+i)+j)+k)	= int *

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k



Dimensional Parentheses (3)

int c[2][3][4];

type(c+i)	= int (*) [3][4]
type((c+i)+j))	= int (*) [4]
type(((c+i)+j)+k)	= int *

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

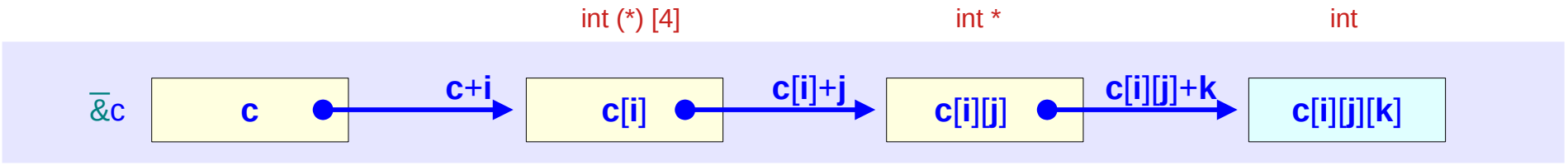
equivalences

c[i][j]	≡ &c[i][j][0]
c[i]	≡ &c[i][0]
c	≡ &c[0]

address replication

value(c[i][j])	= value(&c[i][j])
value(c[i])	= value(&c[i])
value(c)	= value(&c)

v(c+i)	= v(c[i])
v(c[i]+j)	= v(c[i][j])
v(c[i][j]+k)	= v(c[i][j][k])



Address replications in a multi-dimensional array

int c [2][3][4] ;

equivalences

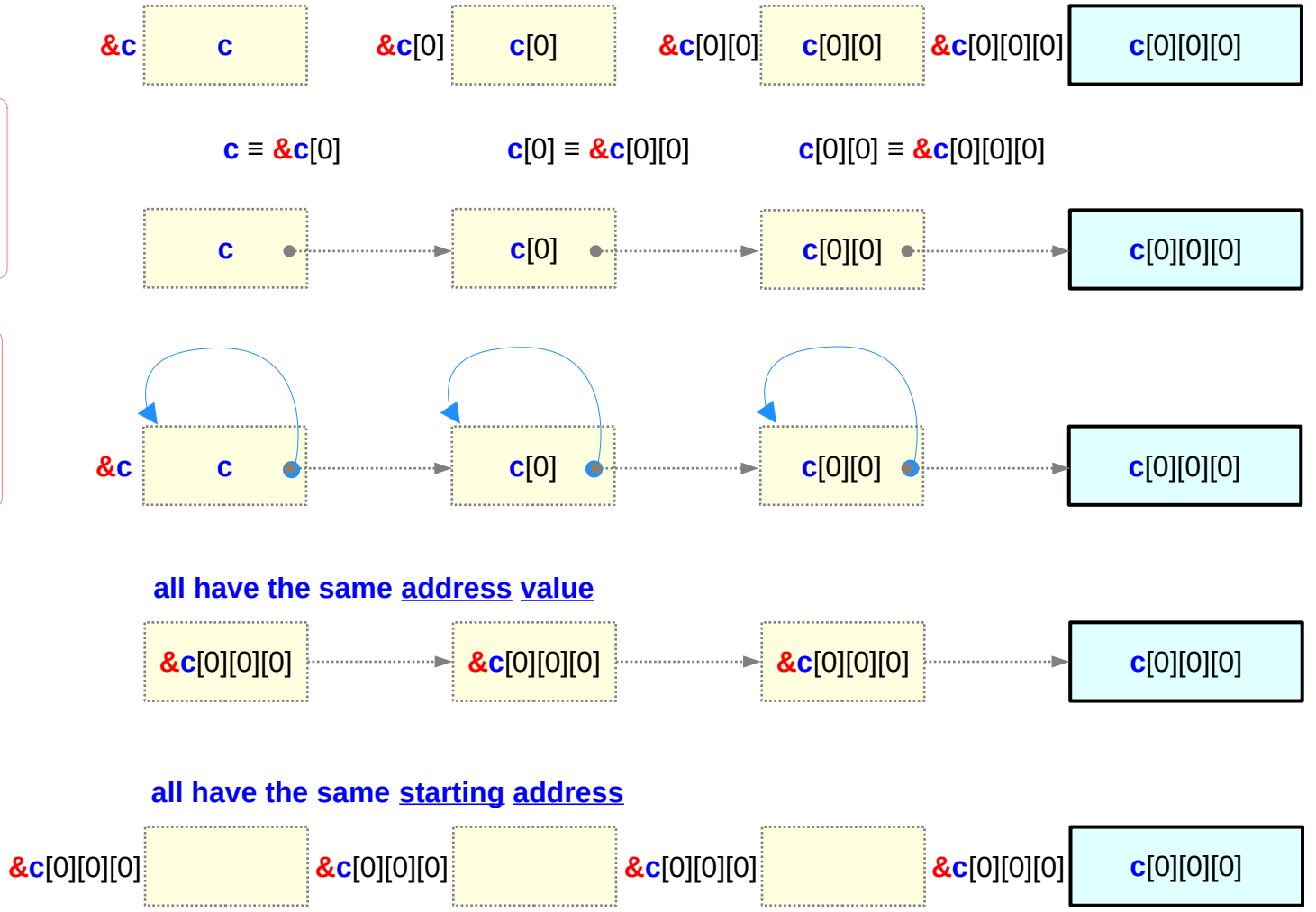
c[i][j] ≡ &c[i][j][0]
c[i] ≡ &c[i][0]
c ≡ &c[0]

address replication

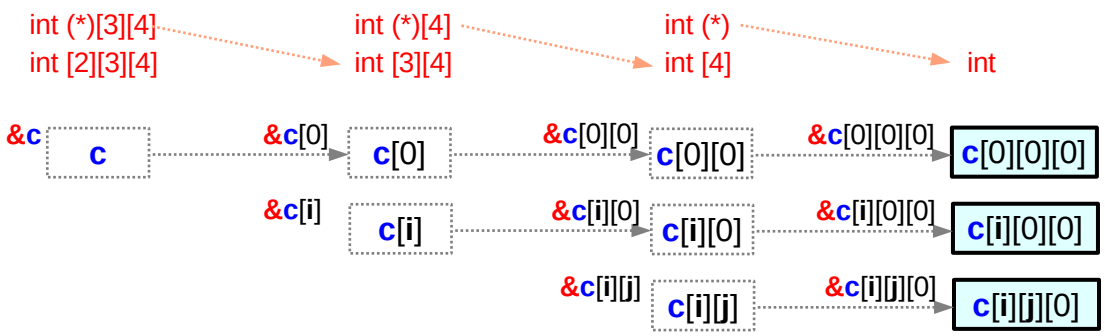
value(c[i][j]) = value(&c[i][j])
value(c[i]) = value(&c[i])
value(c) = value(&c)

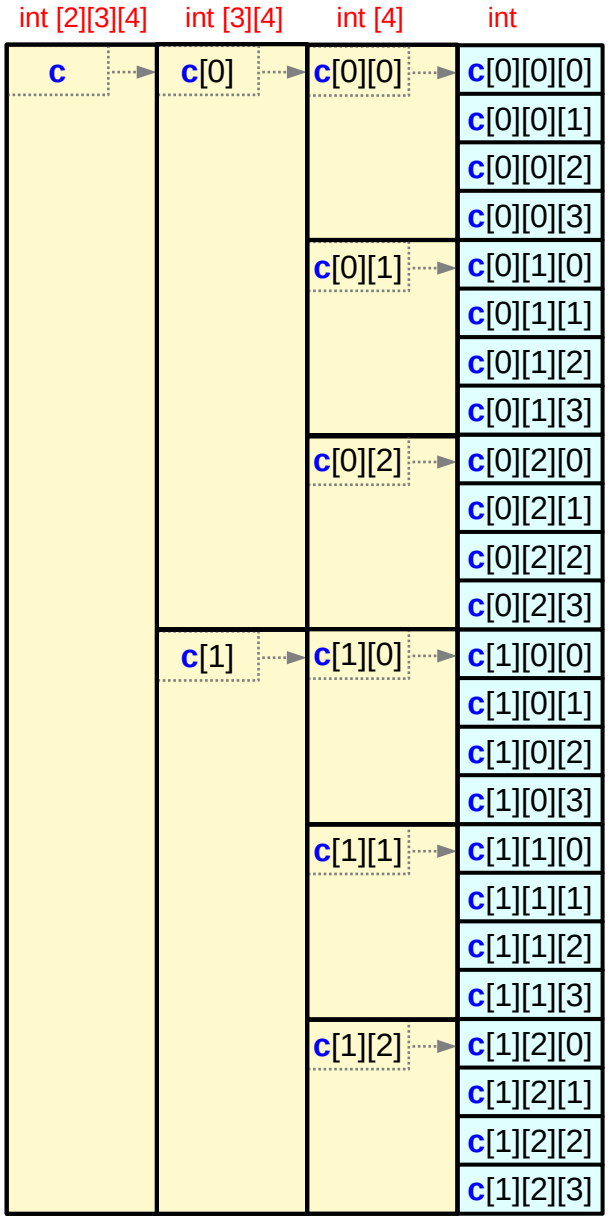
c, c[0], c[0][0] :
these virtual pointers have the same address value

a physical location has a unique address



Referencing sub-arrays





equivalence relations

c[i][j]	≡ *(c[i]+j)	&c[i][j]	≡ (c[i]+j)	&c[i][0]	≡ c[i]
c[i]	≡ *(c+i)	&c[i]	≡ (c+i)	&c[0]	≡ c

address replication

value(c[i][j]) = value(&c[i][j]) = value(c[i]+j) = *value(c[i]+j)
value(c[i]) = value(&c[i]) = value(c+i) = *value(c+i)

c[i], c[i][0] point to the same data c[i][0][0]
c, c[0], c[0][0] point to the same data c[0][0][0]

Types, sizes, and values of sub-arrays

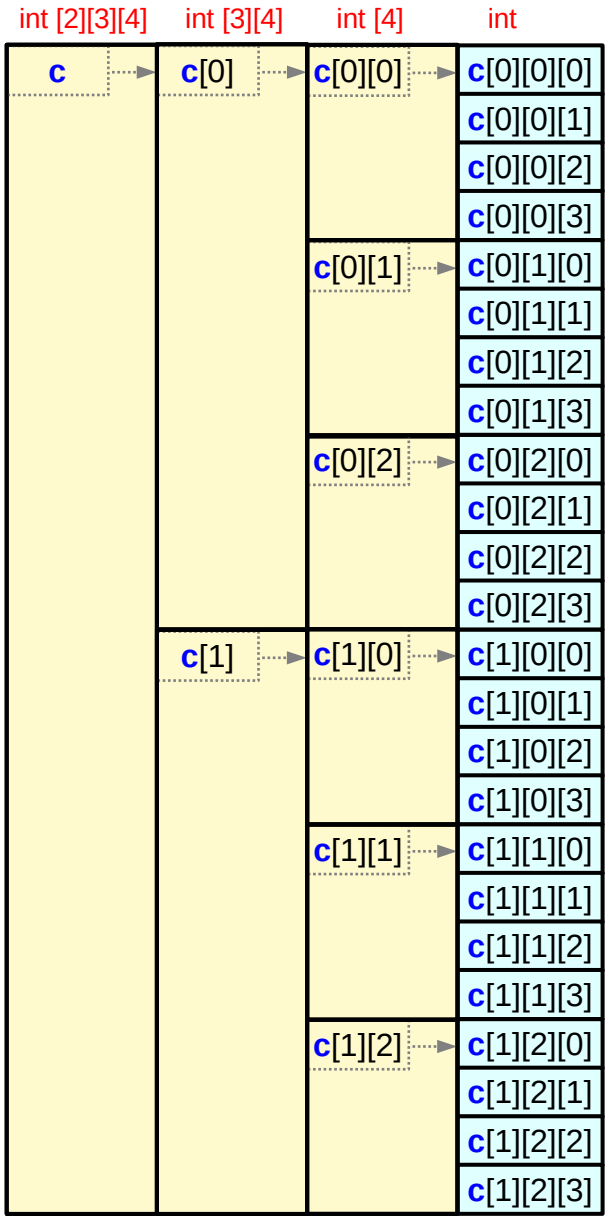
int **c** [2][3][4] ; **static allocation**

value(**c**) = value(**c**[0]) = value(**c**[0][0]) = &**c**[0][0][0]
value(**c**[0][1]) = &**c**[0][1][0]
value(**c**[0][2]) = &**c**[0][2][0]
value(**c**[1]) = value(**c**[1][0]) = &**c**[1][0][0]
value(**c**[1][1]) = &**c**[1][1][0]
value(**c**[1][2]) = &**c**[1][2][0]

sizeof(**c**) = 2*3*4 * sizeof(int)
sizeof(**c**[**i**]) = 3*4 * sizeof(int)
sizeof(**c**[**i**][**j**]) = 4 * sizeof(int)

type(**c**) = int [2][3][4]
int (*)[3][4]
type(**c**[**i**]) = int [3][4]
int (*)[4]
type(**c**[**i**][**j**]) = int [4]
int (*)

pointers to arrays



Dimensional Parentheses

int c[2][3][4];

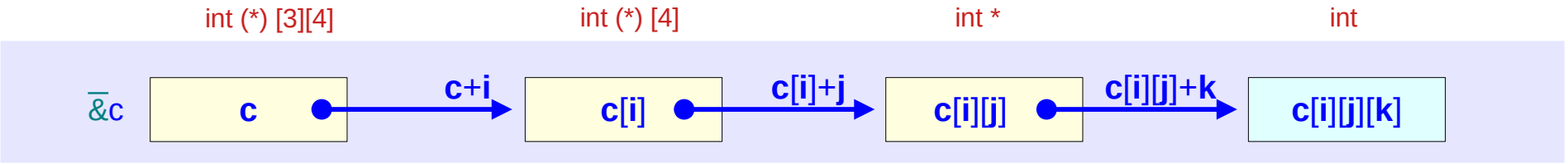
(c+i)	:: int (*) [3][4]
((c+i)+j)	:: int (*) [4]
(((c+i)+j)+k)	:: int (*)

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

(c+i)	$= (c+i)_{[3][4]}$	≠	c+i
((c+i)+j)	$= ((c+i)_{[3][4]}+j)_{[4]}$	≠	c+i+j
(((c+i)+j)+k)	$= (((c+i)_{[3][4]}+j)_{[4]}+k)$	≠	c+i+j+k

dimensional parentheses
not a simple parenthesis, but associated with the dimensions of an array defined

mathematical expressions obtained by mathematical operator &



Types of sub-arrays (2)

int c[2][3][4];

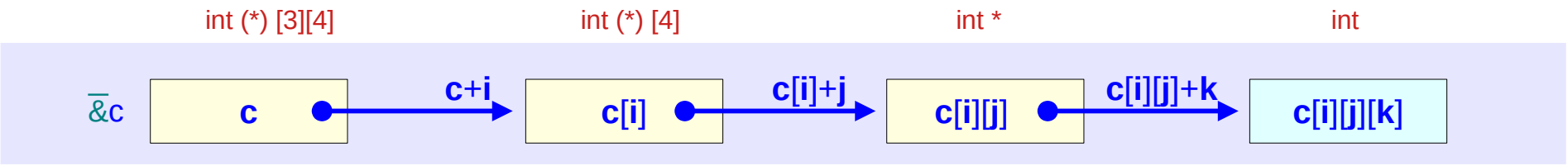
(c+i)	:: int (*) [3][4]
((c+i)+j)	:: int (*) [4]
(((c+i)+j)+k)	:: int (*)

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

(c+i)	$= (c+i)_{[3][4]}$	≠	c+i
((c+i)+j)	$= ((c+i)_{[3][4]}+j)_{[4]}$	≠	c+i+j
(((c+i)+j)+k)	$= (((c+i)_{[3][4]}+j)_{[4]}+k)$	≠	c+i+j+k

not a simple parenthesis, but associated with the dimensions of an array defined

mathematical expressions obtained by mathematical operator &



Types of sub-arrays (2)

```
int c[2][3][4];
```

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

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

```
int c[2][3][4];
```

&c[i]	:: int (*) [3][4]
((c+i)+j)	:: int (*) [4]
(((c+i)+j)+k)	:: int (*)

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

Types of sub-arrays (2)

```
int c[2][3][4];
```

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

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

```
int c[2][3][4];
```

&c[i]	:: int (*) [3][4]
((c+i)+j)	:: int (*) [4]
(((c+i)+j)+k)	:: int (*)

v(c+i)	= v(c) + i * sizeof(3*4)
v((c+i)+j)	= v(c+i) + j * sizeof(4)
v(((c+i)+j)+k)	= v((c+i)+j) + k

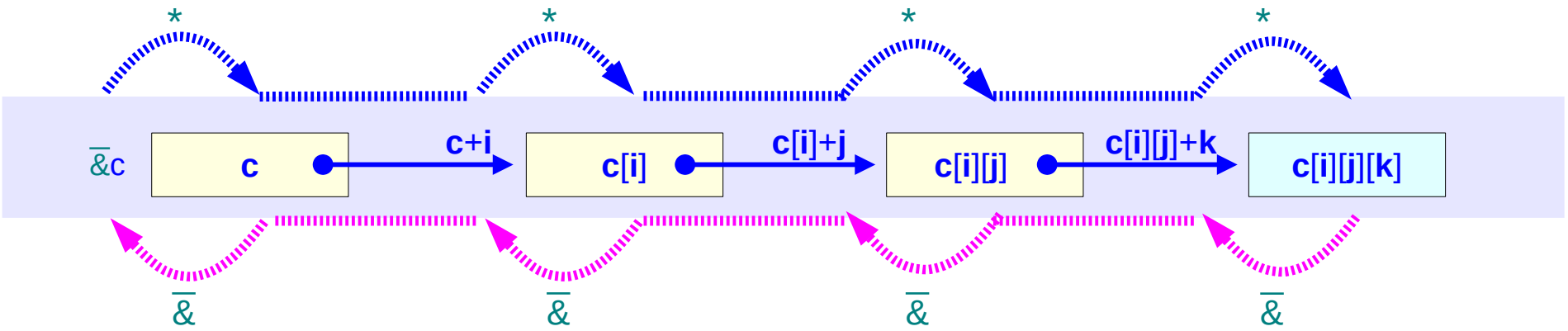
Recursive applications of * and & are possible

((*(c+i)+j)+k)]

((*(c+i)+j)+k) =
((c[i]+j)+k) =
*(c[i][j]+k) =
c[i][j][k]

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

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



Outward and Inward Recursive applications of * and &

((*(c+i)+j)+k)]

((*(c+i)+j)+k) =
((c[i]+j)+k) =
*(c[i][j]+k) =
c[i][j][k]

((*(c+i)+j)+k)]

((*(c+i)+j)+k) =
(*(*(c+i)+j)[k] =
(*(c+i))[j][k] =
c[i][j][k]

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

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

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

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

mathematical expressions

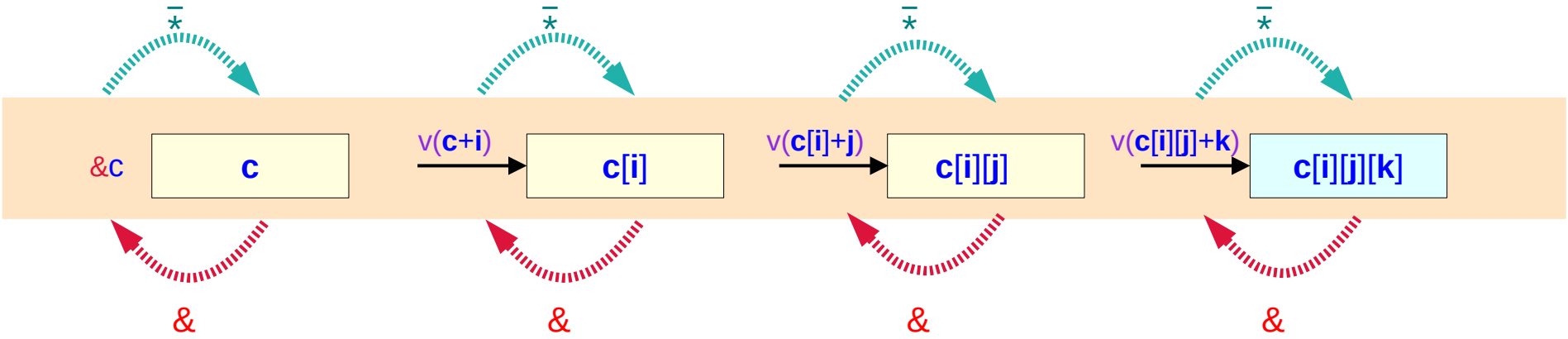
Recursive applications of * and & are not possible (2)

((*(c+i)+j)+k)

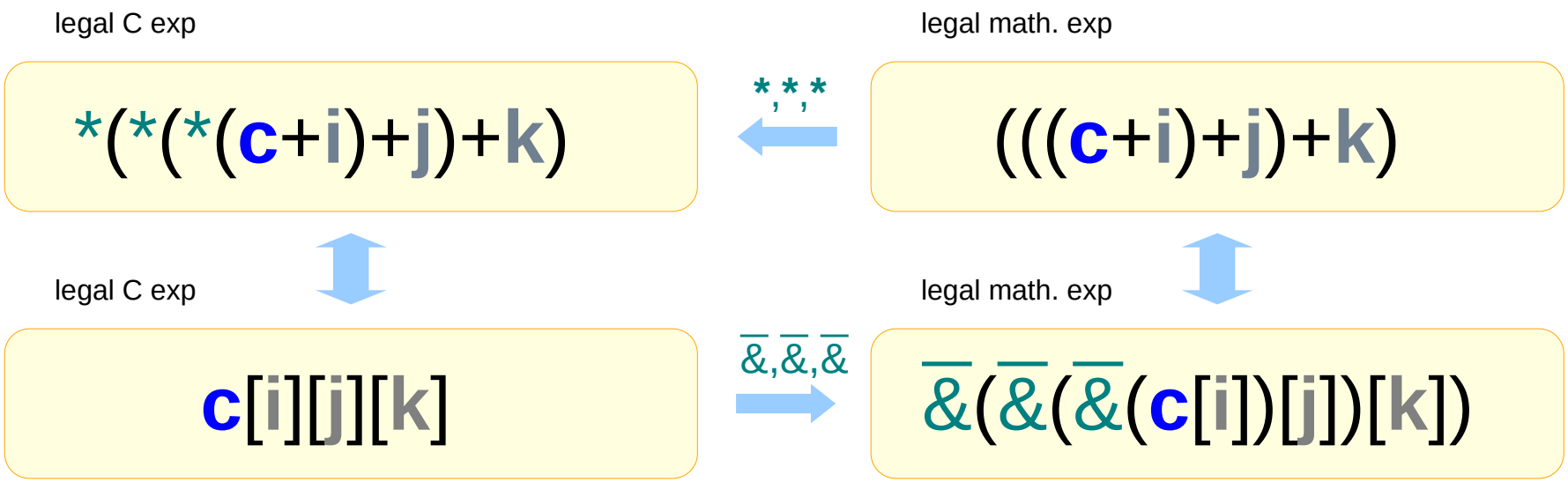
not allowed
even in mathematical expressions

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

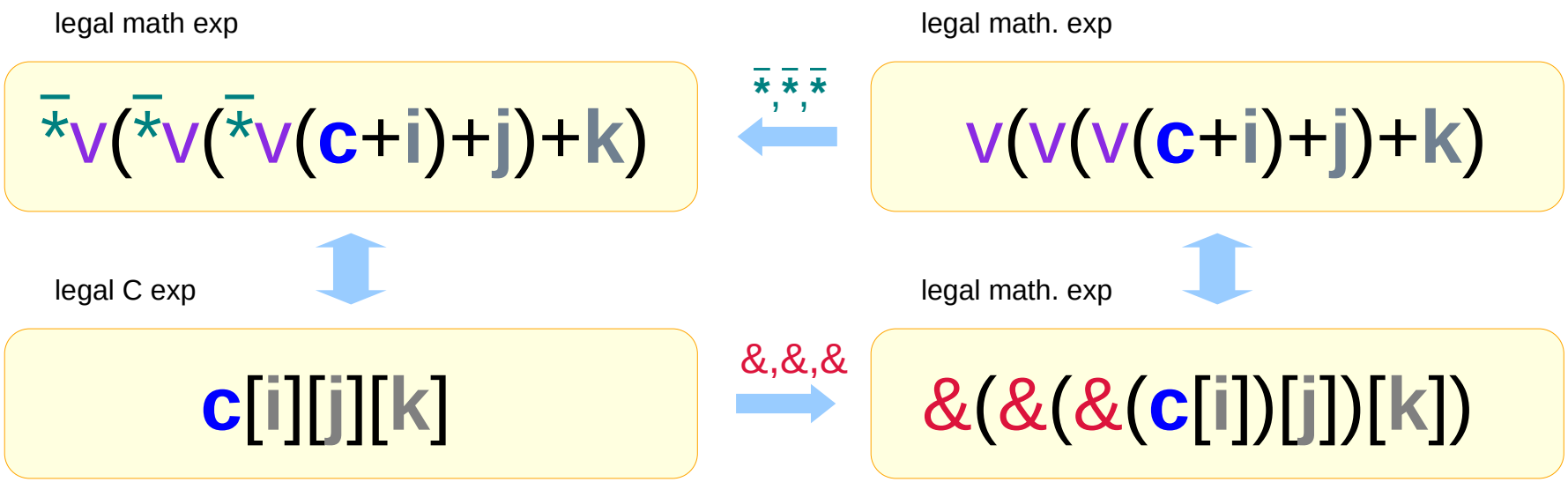
not allowed
in C expressions



Recursive applications of * and &



Recursive applications of *̄ and &



Recursive applications of $\bar{*}$ and & are not possible (3)

legal C expression

((*(c+i)+j)+k)]

legal math. expression

$\bar{*}v(\bar{*}v(\bar{*}v(c+i)+j)+k)$

illegal math. expression

$\bar{*}(\bar{*}(\bar{*}(c+i)+j)+k)$

legal math. expression

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

illegal math. expression

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

illegal C expression

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

Recursive applications of $\bar{*}$ and & are not possible (3)

legal C expression

((*(c+i)+j)+k)]

legal math. expression

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

legal math. expression

$\bar{*}v(\bar{*}v(\bar{*}v(c+i)+j)+k)$

legal math. expression

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

Recursive applications of *̄ and & are not possible (2)

c[i][j][k] = *̄v(c[i][j] + k)
= *̄v(*̄v(c[i] + j) + k)
= *̄v(*̄v(*̄v(c + i) + j) + k)

v(v(v(c + i) + j) + k) = (&(c[i]) + j) + k
= (&(&(c[i])[j])) + k
= &(&(&(c[i])[j])[k])

a[i] = &(c[i])
b[i][j] = (&(&(c[i])[j]))

legal math. expression

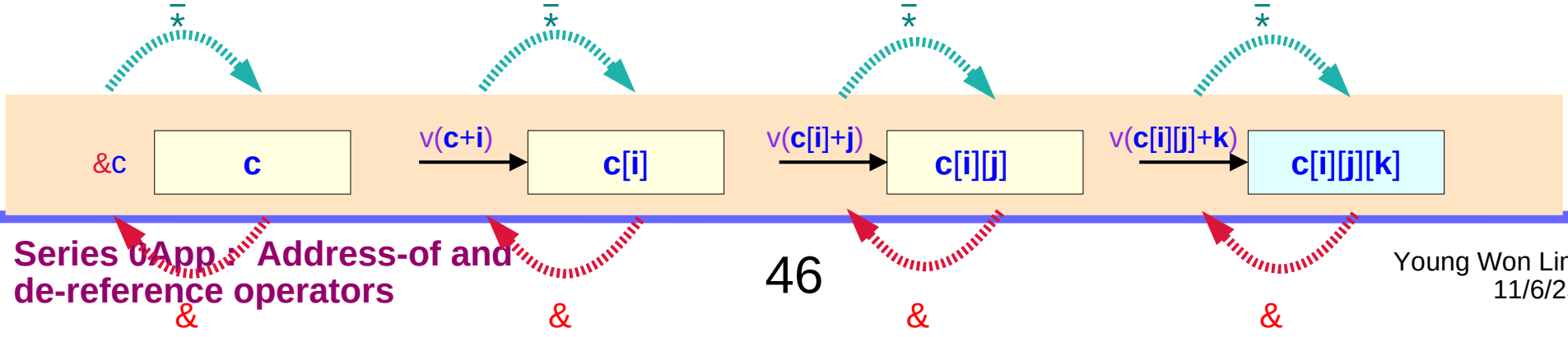
&̄(&̄(&̄(c[i])[j])[k])

legal math. expression

̄v&̄(̄v&̄(*̄v&̄(c[i])[j])[k])

illegal C expression

̄&(̄&(*̄&(c[i])[j])[k])



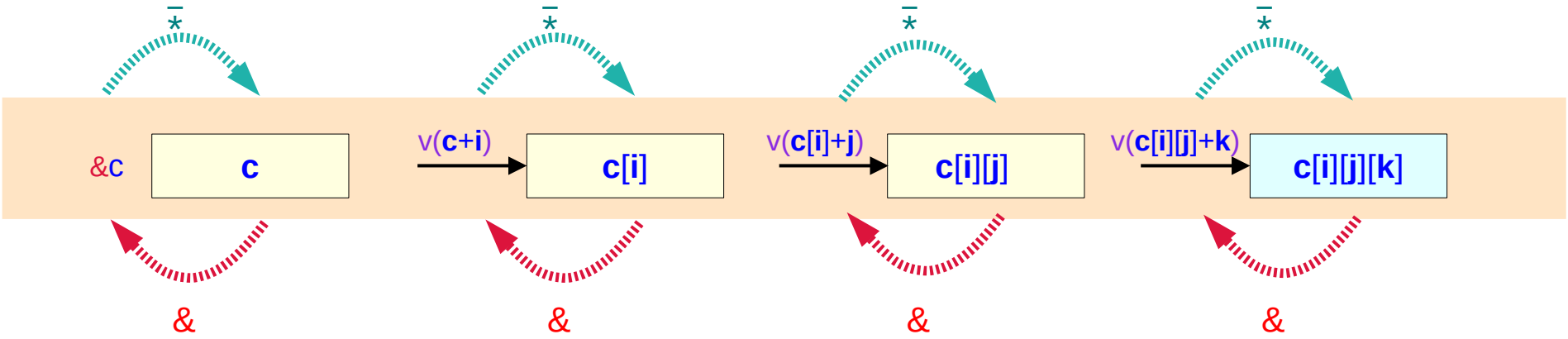
Recursive applications of *̄ and & are not possible (1)

̄v(̄v(*̄v(c+i)+j)+k)

v&̄(v&̄(v&̄(c[i])[j])[k])

̄v(̄v(*̄v(c+i)+j)+k) =
̄v(̄v(c[i]+j)+k) =
*̄v(c[i][j]+k) =
c[i][j][k]

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



Recursive applications of *̄ and & are not possible (2)

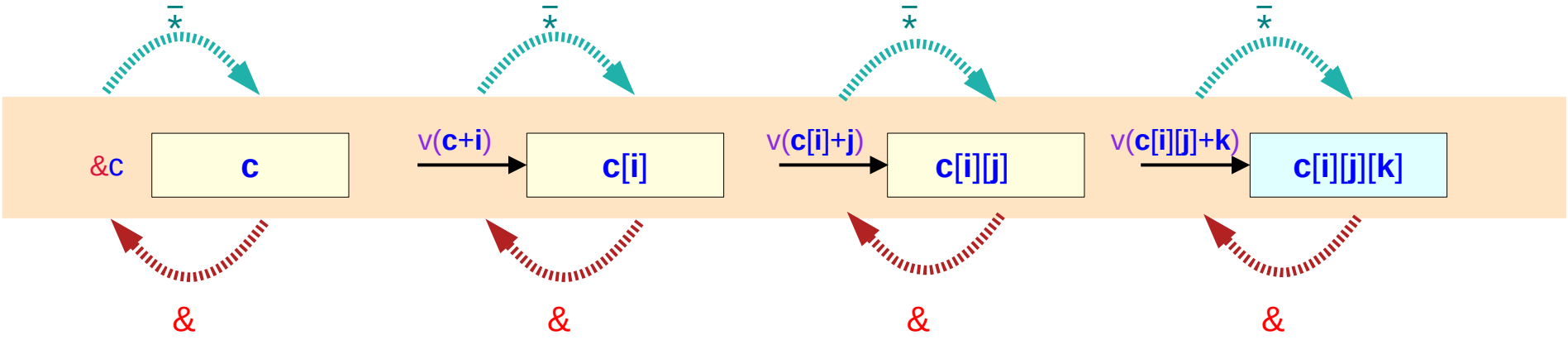
~~&(&(&(c[i])[j])[k])~~

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

&(&(&(c[i])[j])[k]) =
v&(&(&(c[i])[j])[k])=
v(&(&(c[i])[j])+k) =
v((&(c[i])+j)+k) =
v(((c+i)+j)+k)

&(&(c[i])[j])[k] =
v&(&(c[i])[j])[k] =
v((&(c[i])+j)[k] =
v((c+i)+j)[k]

&(c[i])[j][k] =
v&(c[i])[j][k] =
v((c+i)[j][k]



Recursive applications of $\bar{*}$ and & are not possible (3)

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

& takes only **lvalue** variable
returns **rvalue** address value

successive application of **&** is not possible, unless intermediate variables and proper type casting

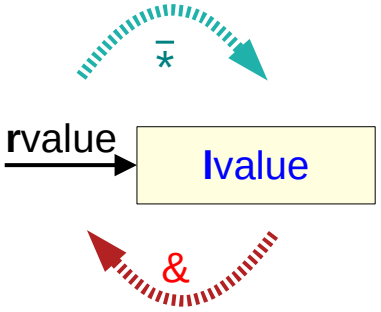
$\bar{*}$ must be applied to an **rvalue** address value, $\bar{*}$ operator cannot be applied successively.

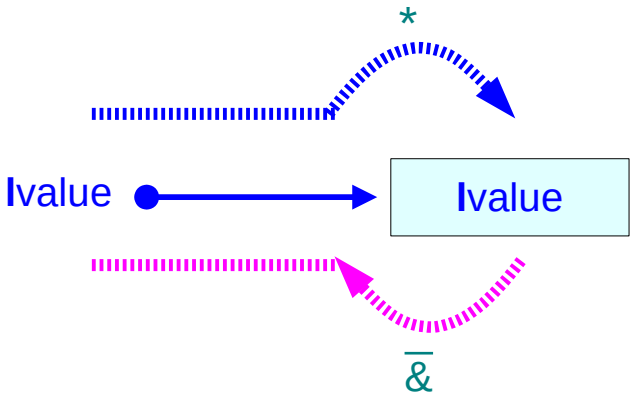
$\bar{\&}(\bar{\&}(\bar{\&}(c[i])[j])[k])$

$\bar{\&}$ takes a dereferenced **lvalue** variable
returns **lvalue** variable

successive application of $\bar{\&}$ is possible

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





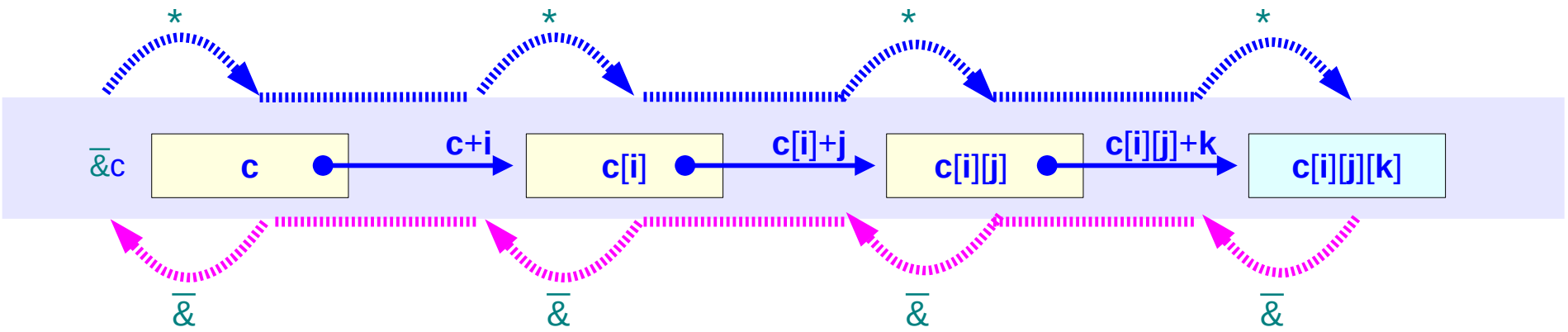
Inverse Relations between * and &

```
c[i][j][k]      = *(c[i][j] + k)
c[i][j]         = *(c[i] + j)
c[i]            = *(c + i)
```

```
c[i][j][k]      = *(c[i][j] + k)
                = *(*(c[i] + j) + k)
                = *(*(*(c + i) + j) + k)
```

```
&(c[i][j][k])   = c[i][j] + k
&(c[i][j])      = c[i] + j
&(c[i])         = c + i
```

```
((c + i) + j) + k = (&(c[i]) + j) + k
                  = (&(&(c[i])[j])) + k
                  = &(&(&(c[i])[j])[k])
```



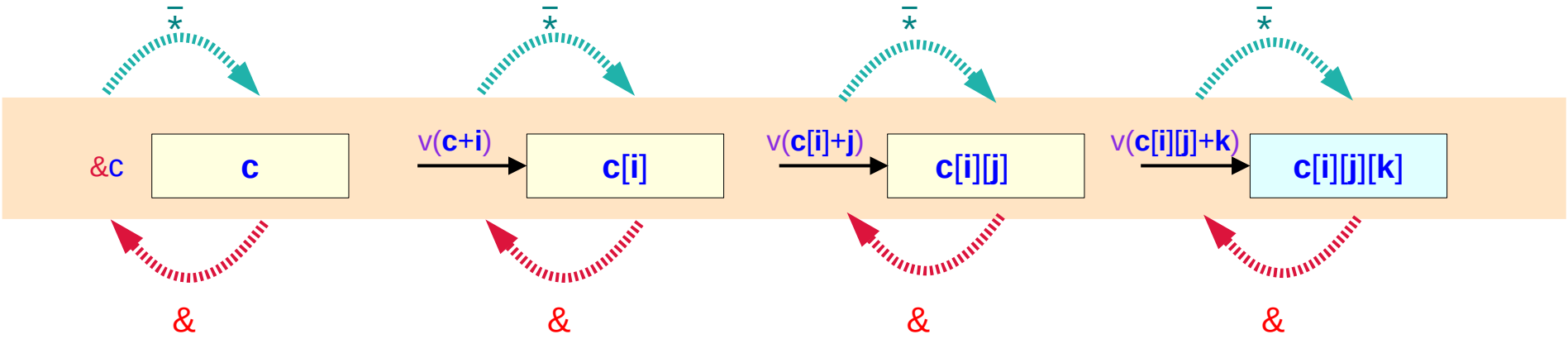
Inverse Relations between $\bar{*}$ and &

c[i][j][k]	= $\bar{*}$v(c[i][j] + k)
c[i][j]	= $\bar{*}$v(c[i] + j)
c[i]	= $\bar{*}$v(c + i)

c[i][j][k]	= $\bar{*}$v(c[i][j] + k)
	= $\bar{*}$v($\bar{*}$v(c[i] + j) + k)
	= $\bar{*}$v($\bar{*}$v($\bar{*}$v(c + i) + j) + k)

&(c[i][j][k])	= v(c[i][j] + k)
&(c[i][j])	= v(c[i] + j)
&(c[i])	= v(c + i)

v(v(v(c + i) + j) + k) = (&(c[i]) + j) + k
= (&(&(c[i])[j])) + k
= &(&(&(c[i])[j])[k])



Inverse Relations between *̄ and &

c[i][j][k] = *̄v(c[i][j] + k)
c[i][j] = *̄v(c[i] + j)
c[i] = *̄v(c + i)

c[i][j][k] = *̄v(c[i][j] + k)
= *̄v(*̄v(c[i] + j) + k)
= *̄v(*̄v(*̄v(c + i) + j) + k)

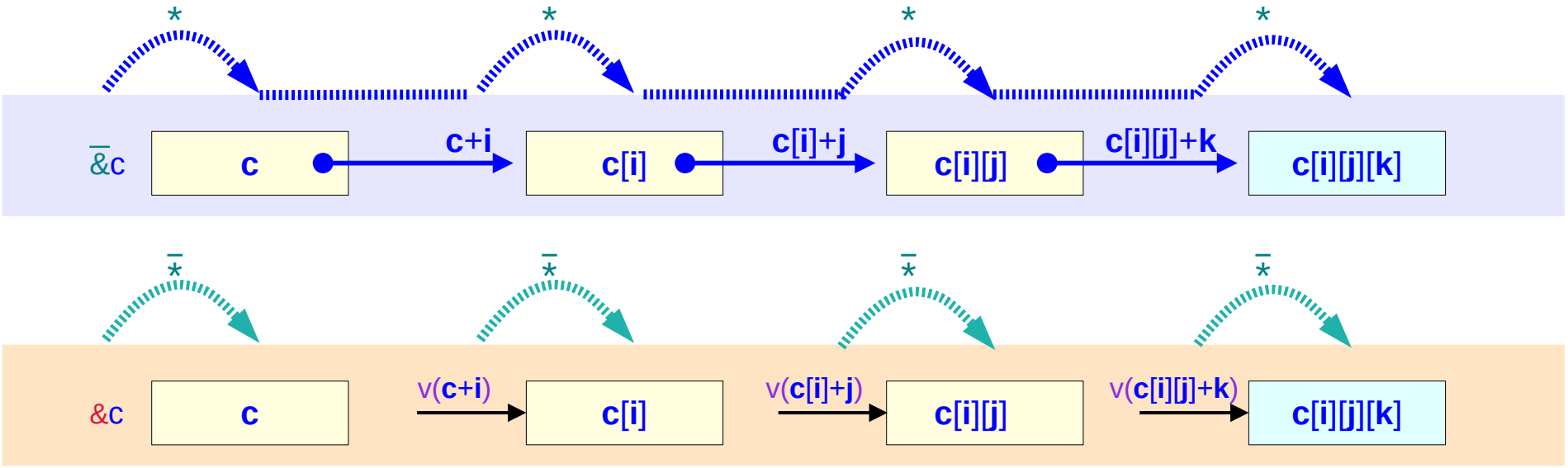
&(c[i][j][k]) = v(c[i][j] + k)
&(c[i][j]) = v(c[i] + j)
&(c[i]) = v(c + i)

v(v(v(c + i) + j) + k) = (&(c[i]) + j) + k
= (&(&(c[i])[j])) + k
= &(&(&(c[i])[j])[k])

v(v(v(c + i) + j) + k) = &(v(v(c + i) + j)[k]
= &(&(v(c + i)[j])[k]
= &(&(&(c[i])[j])[k]

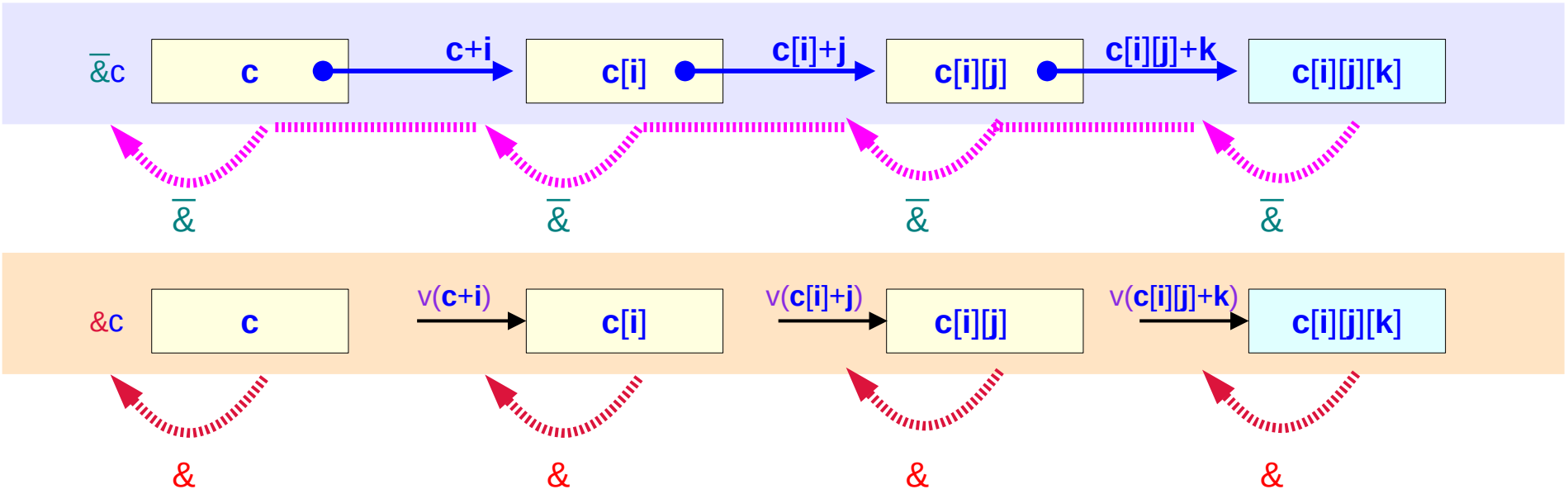
c[i][j][k] = *(c[i][j] + k)
c[i][j] = *(c[i] + j)
c[i] = *(c + i)

c[i][j][k] = *v(c[i][j] + k) = *(value(c[i][j]) + k * sizeof(c[0][0][0]))
c[i][j] = *v(c[i] + j) = *(value(c[i]) + j * sizeof(c[0][0]))
c[i] = *v(c + i) = *(value(c) + i * sizeof(c[0]))



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

&(c[i][j][k]) = v(c[i][j] + k) = (value(c[i][j]) + k * sizeof(c[0][0][0]))
&(c[i][j]) = v(c[i] + j) = (value(c[i]) + j * sizeof(c[0][0]))
&(c[i]) = v(c + I) = (value(c) + i * sizeof(c[0]))



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

```
&(c[i][j][k])      = &(*(c[i][j]+k))      = c[i][j]     + k
&(c[i][j])         = &(*(c[i]+j))         = c[i]        + j
&(c[i])            = &(*(c+i))            = c           + i

*&(c[i][j][k])     = *&(*(c[i][j]+k))     = *(c[i][j]   +k)
*&(c[i][j])        = *&(*(c[i]+j))        = *(c[i]      +j)
*&(c[i])           = *&(*(c+i))           = *(c         +i)


&(c[i][j][k])      = &(*(c[i][j]+k))      = c[i][j]     + k * sizeof(c[0][0][0])
&(c[i][j])         = &(*(c[i]+j))         = c[i]        + j  * sizeof(c[0][0])
&(c[i])            = &(*(c+i))            = c           + i  * sizeof(c[0])

c[i][j][k]         = *&(*(c[i][j]+k))     = *(c[i][j]   + k * sizeof(c[0][0][0])
c[i][j]            = *&(*(c[i]+j))        = *(c[i]      + j  * sizeof(c[0][0]))
c[i]               = *&(*(c+i))           = *(c         + i  * sizeof(c[0]))
```

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

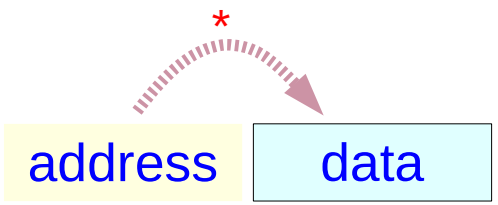
```
&(c[i][j][k])     = c[i][j]    + k
&(c[i][j])        = c[i]       + j
&(c[i])           = c          + i

c[i][j][k]        = *(c[i][j]  +k)
c[i][j]           = *(c[i]     +j)
c[i]              = *(c        +i)


&(c[i][j][k])     = c[i][j]    + k * sizeof(c[0][0][0])
&(c[i][j])        = c[i]       + j  * sizeof(c[0][0])
&(c[i])           = c          + i  * sizeof(c[0])

c[i][j][k]        = *(c[i][j]  + k * sizeof(c[0][0][0])
c[i][j]           = *(c[i]     + j  * sizeof(c[0][0]))
c[i]              = *(c        + i  * sizeof(c[0]))
```

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

&(c[i][j][k])	= c[i][j]	+ k	&(c[i][j][k])	= value(c[i][j]	+ k)
&(c[i][j])	= c[i]	+ j	&(c[i][j])	= value(c[i]	+ j)
&(c[i])	= c	+ i	&(c[i])	= value(c	+ i)
c[i][j][k]	= *(c[i][j]	+k)	c[i][j][k]	= * value(c[i][j]	+ k)
c[i][j]	= *(c[i]	+j)	c[i][j]	= * value(c[i]	+ j)
c[i]	= *(c	+i)	c[i]	= * value(c	+ i)

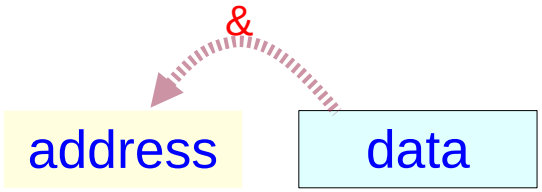
Address-of & and dereference * C operators (1)

Primitive Data Type



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

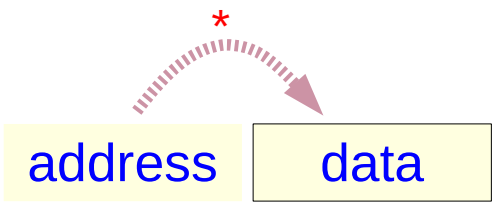
lvalue p → *lvalue* *p
pointer *variable*



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

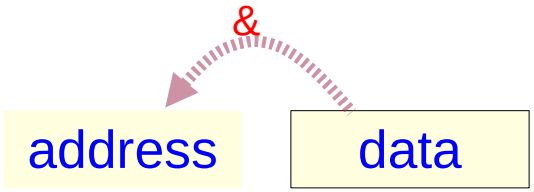
~~*lvalue*~~ p ← *lvalue* *p
pointer *variable*

Pointer Data Type



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

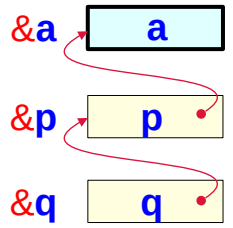
lvalue q → *lvalue* *q
double pointer *pointer*

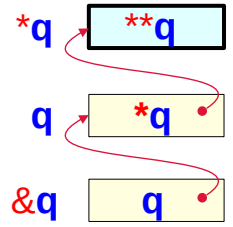


rvalue &q ← *lvalue* q
constant *double pointer*

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

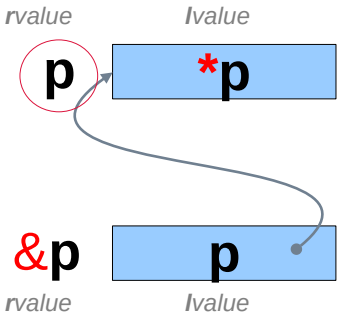
```
int   a ;
int *  p ;
int ** q ;
```





& and * operators in pointer de-referecing

Two step De-reference * operation

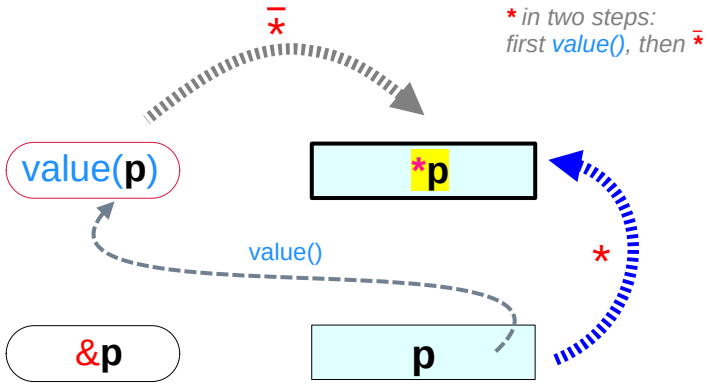


*p must be evaluated to an address value(p) before de-referencing *p*

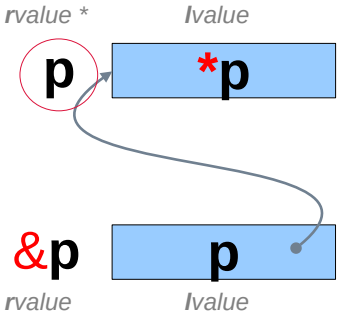
think a new operator :

$\overline{*}$ is a mathematical operator

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



Two step Address-of & operation

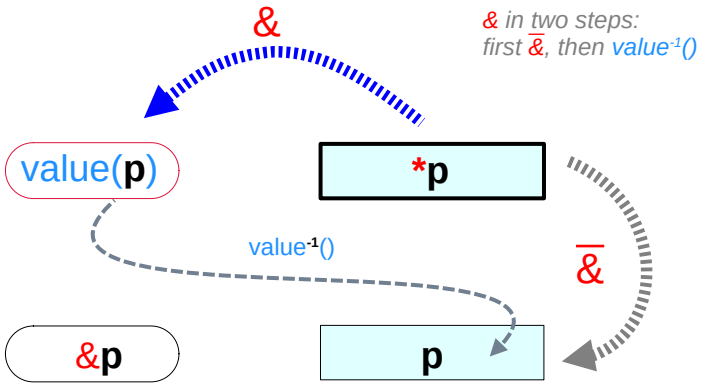


assume that it is possible to recover the variable p from value(p)

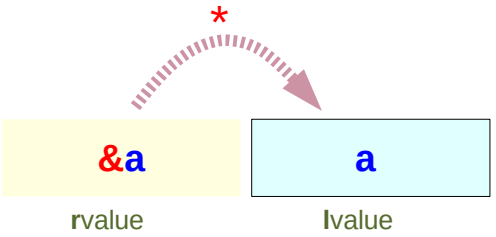
think a new operator :

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

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

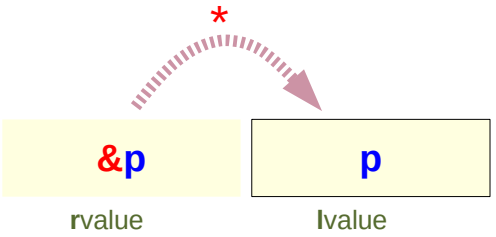


Type, Size, and Value attributes of an **lvalue**



lvalue ⟵ ***rvalue***

a ****&a***



lvalue ⟵ ***rvalue***

p ****&p***

lvalue is associated with a memory location

lvalue has the following attributes
- Type
- Size
- Value

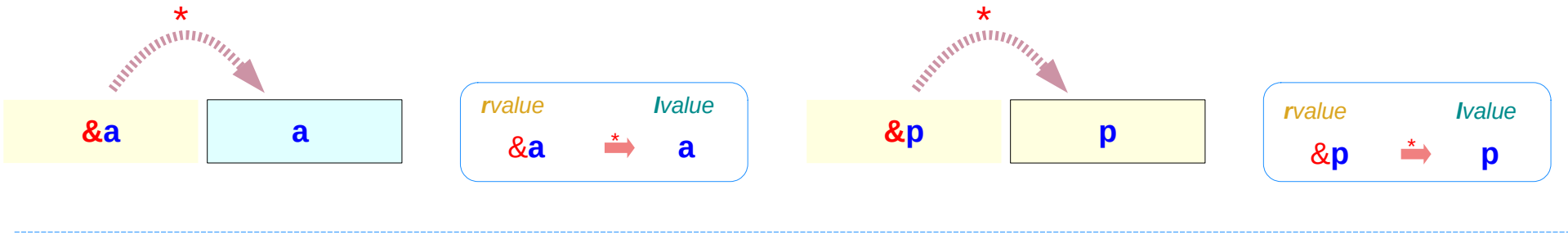
rvalue has the only attribute
- Value

assume the function value()

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

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

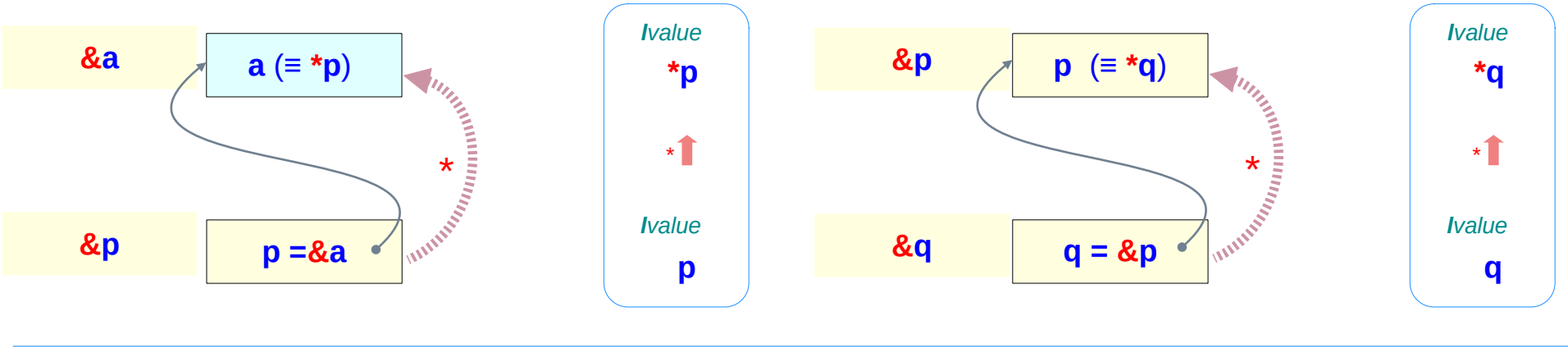
Address-of & and dereference * C operators (1)

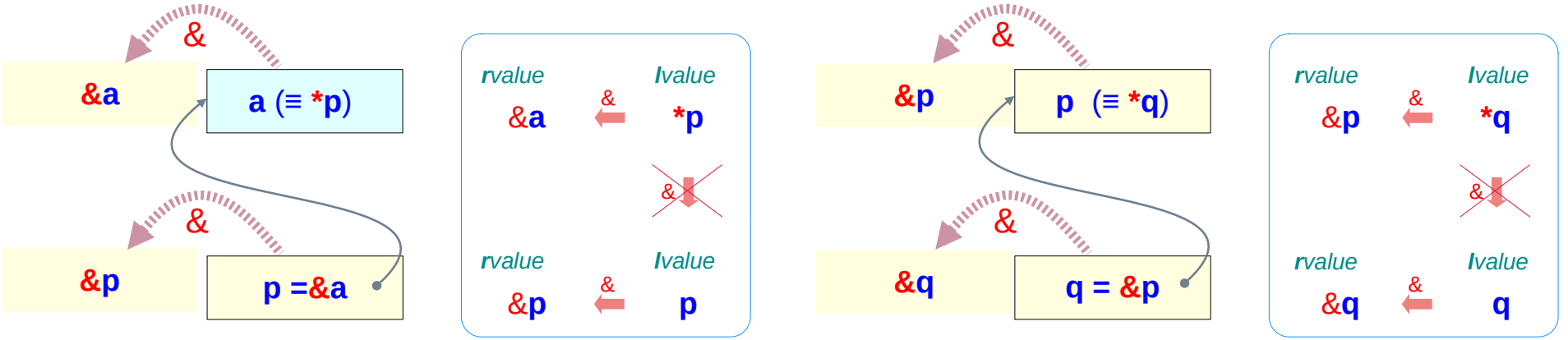


*&a = a

*&p = p

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





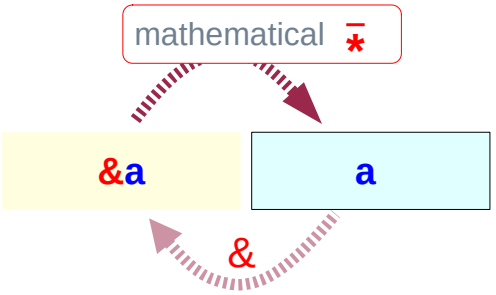
*&p = p

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

*&q = q

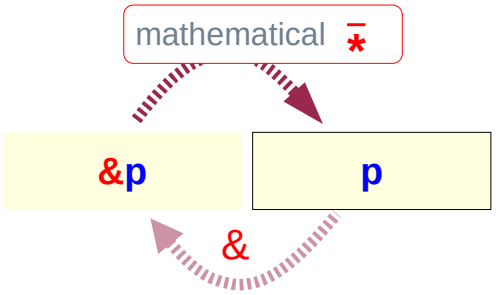
~~&*q = q~~ value(q)

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



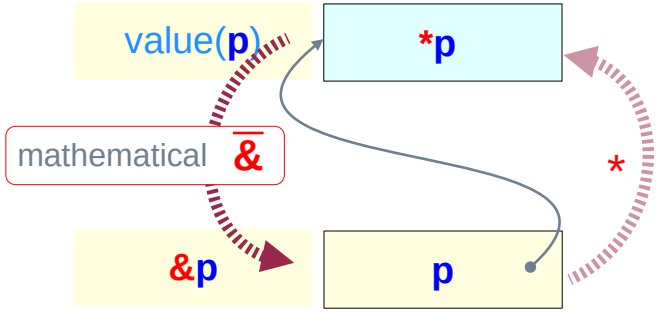
$\overline{*}$ &a = a

&a



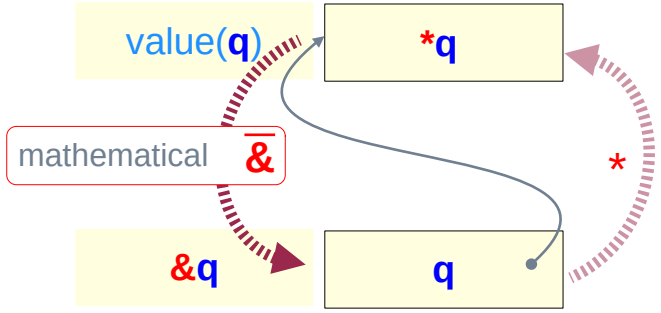
$\overline{*}$ &p = p

&p



*p

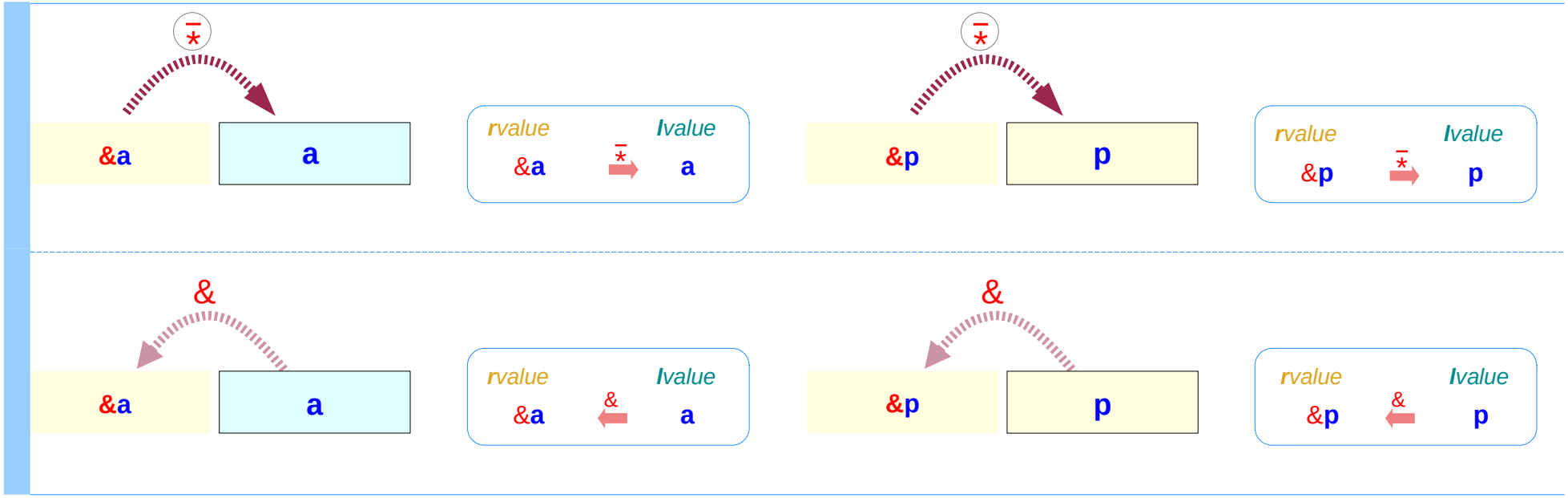
$\overline{\&}$ *p = p



*q

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

Inverse operators * and &



*&a = a

&*&a = &a

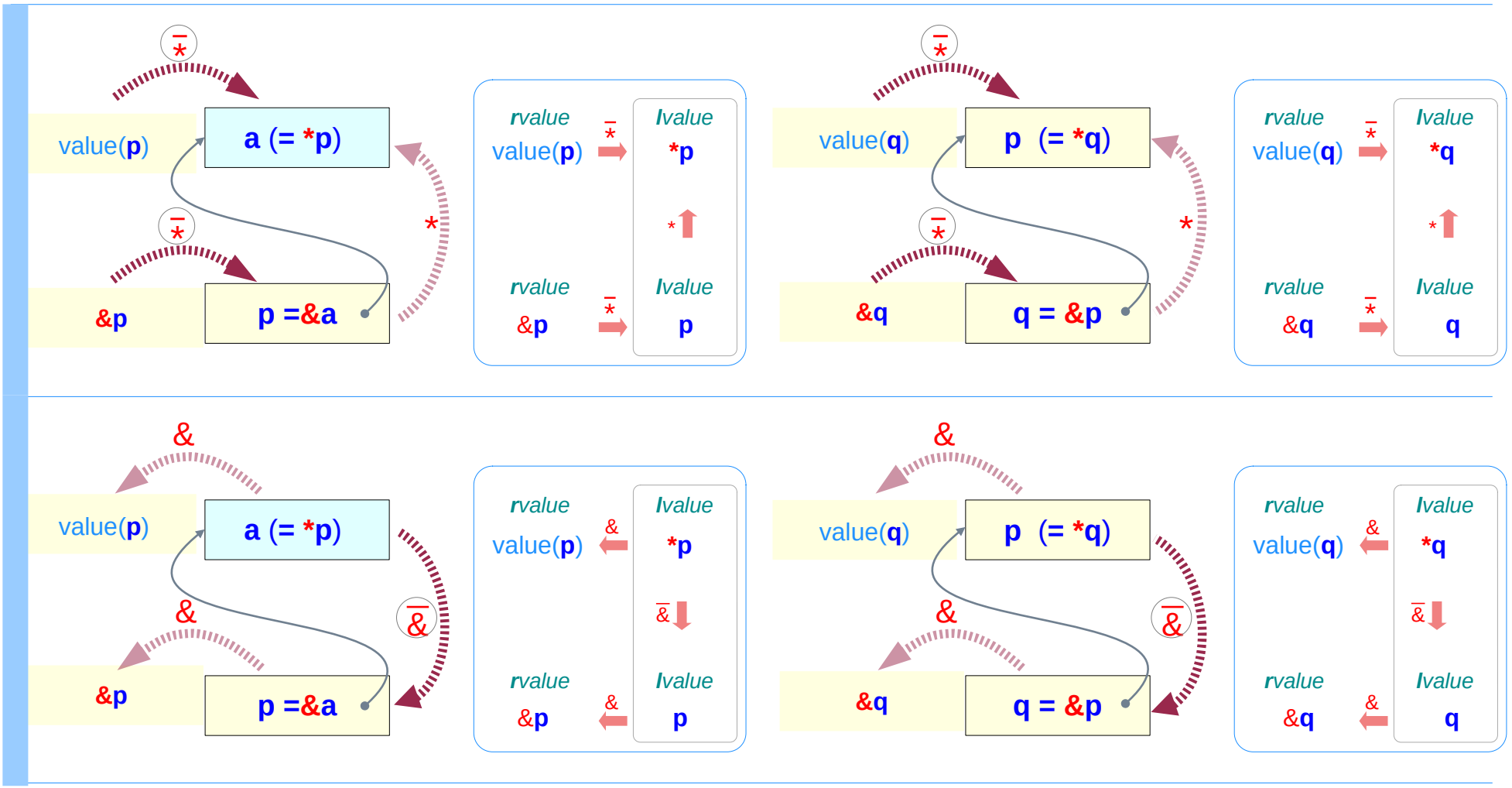
* and & are
inverse operators
to each other

*&p = p

&*&p = &p

* and & are
inverse operators
to each other

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



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

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

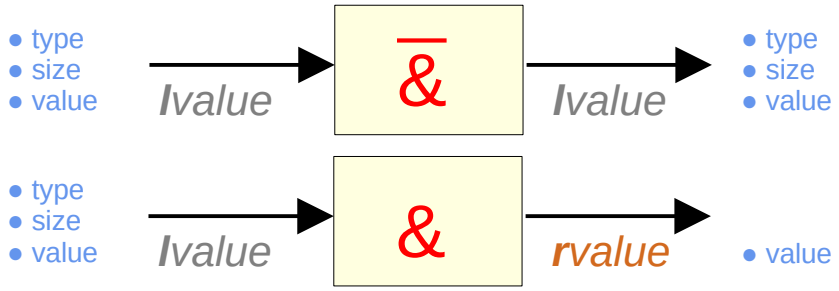
$\overline{*}\&q = q$

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

C operators and mathematical operators

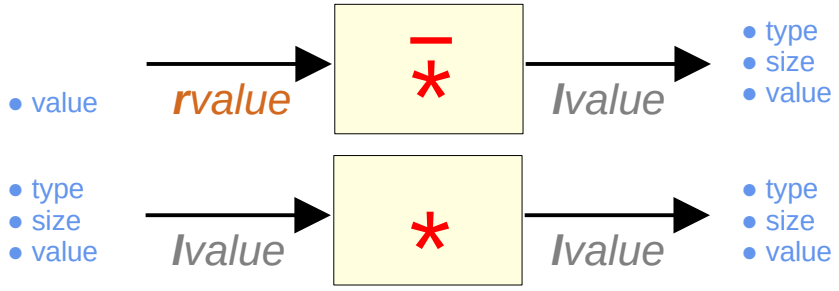
Address-of operation

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

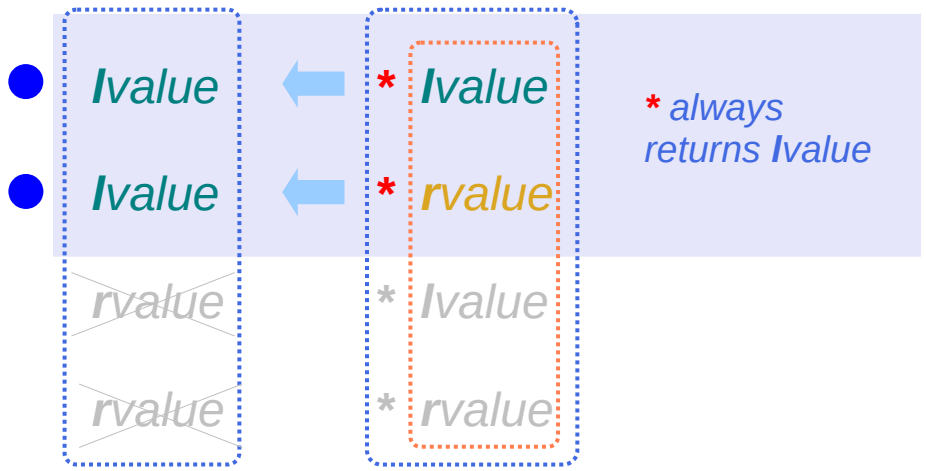


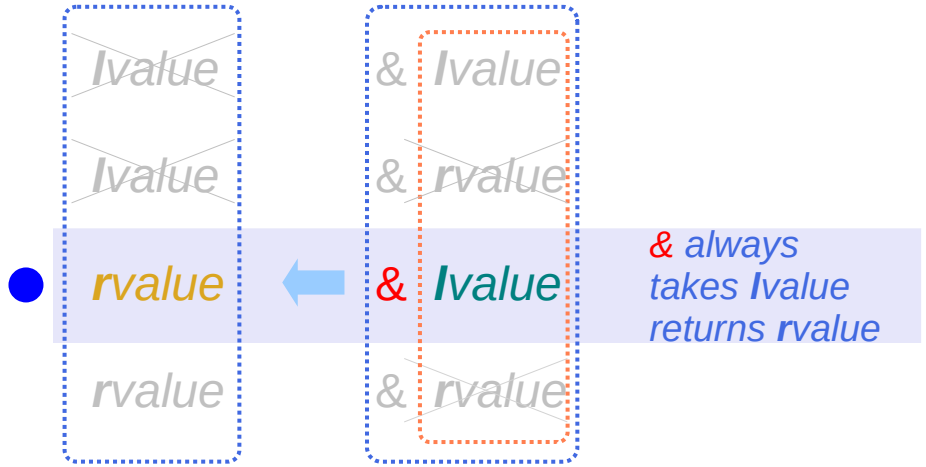
De-reference operation

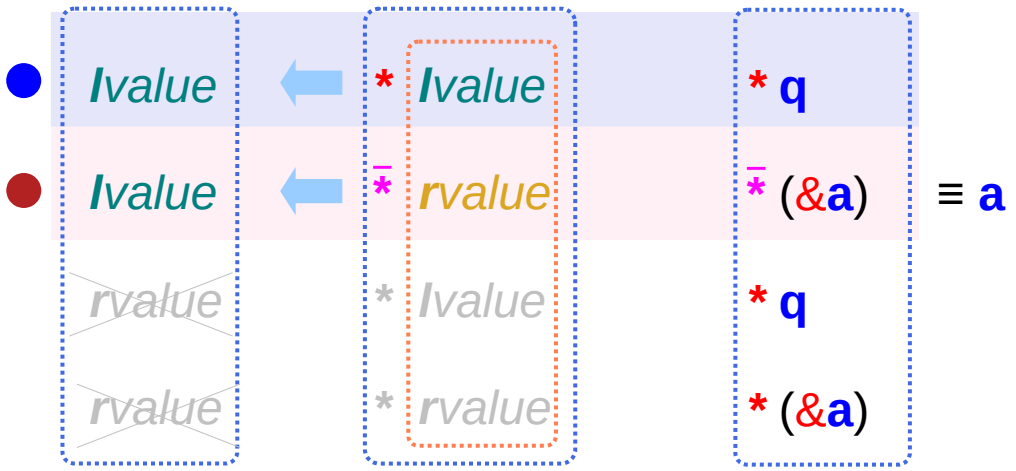
*p	=	*value(p)
C Expressions		Mixed Expressions

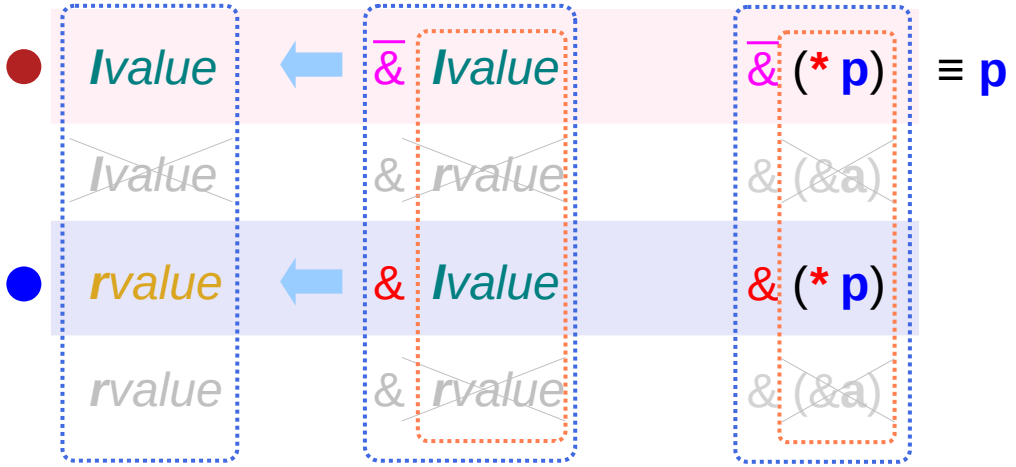


lvalue and rvalue with * and & operators









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

$*\&a \Rightarrow a$

$*\&p \Rightarrow p$

$*\&q \Rightarrow q$

$*p \Rightarrow a$

$*q \Rightarrow p$

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

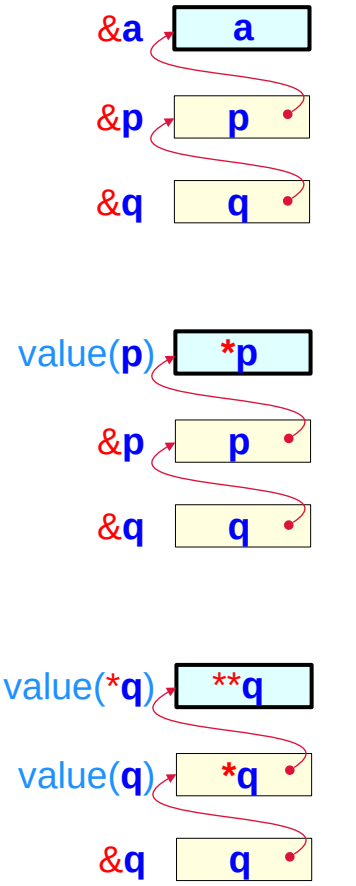
$\&*p \Rightarrow \&a$

$\&*q \Rightarrow \&p$

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

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

& ***P** ➡ **P**

dereferenced pointer

& ~~pointer~~

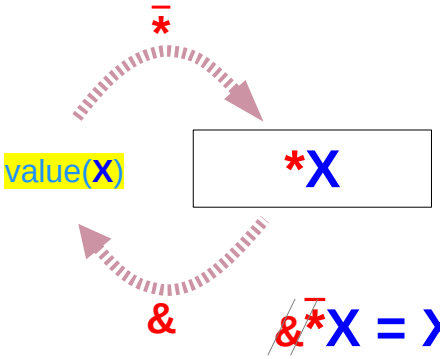
* &**Q** ➡ **Q**

must be a dereferenced pointer

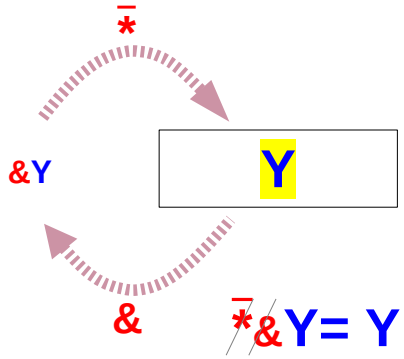
not considering
simultaneous pointing

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

& and mathematical *



& *X = X



*&Y= Y

& *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

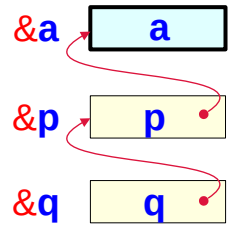
* &a = a

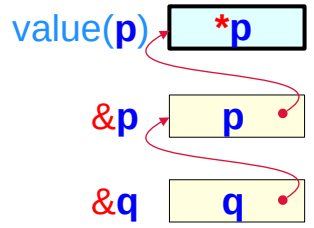
* &p = p

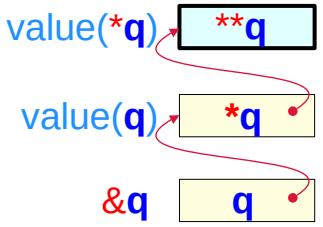
* &q = q

C expressions

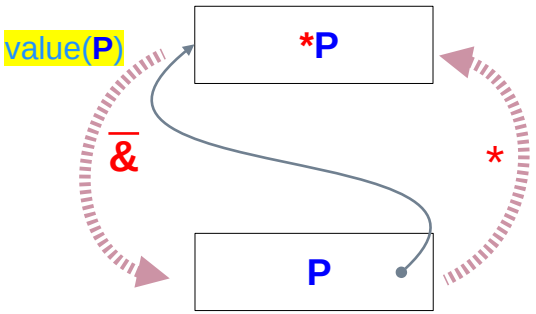
```
int    a ;
int *  p = & a ;
int ** q = & p ;
```

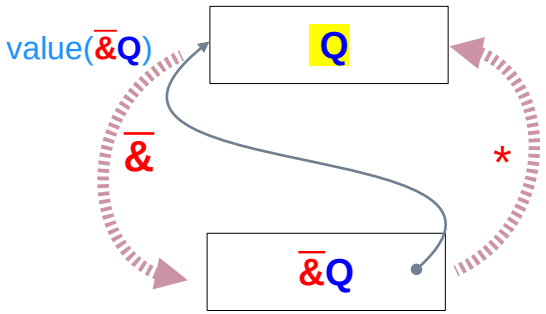






* and mathematical &





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

if not considering simultaneous pointing

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

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

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

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

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

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

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

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

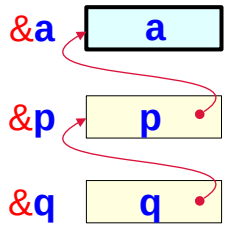
$\overline{{}^{*}}\ \&{}^{*}q = {}^{*}q$

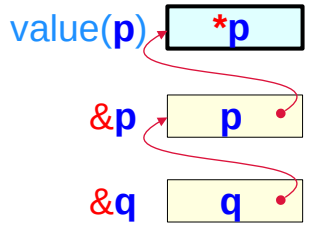
$* \ \&{}^{*}p = {}^{*}p$

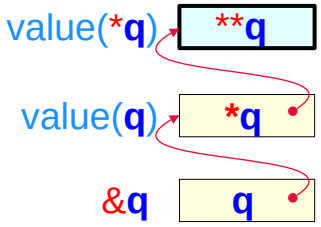
$* \ \&{}^{*}q = {}^{*}q$

C operators

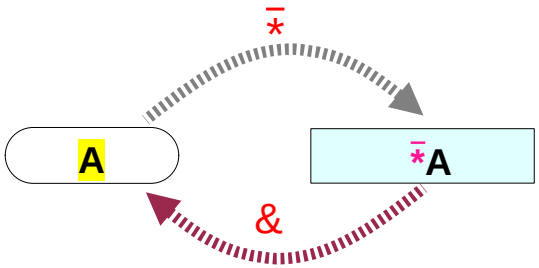
```
int    a ;
int *  p = & a ;
int ** q = & p ;
```





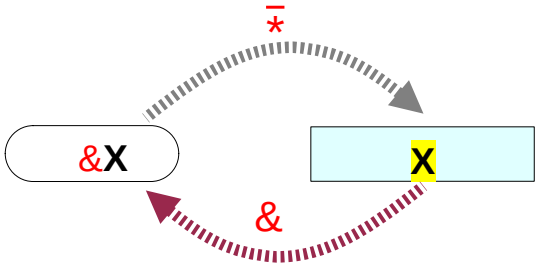


Requirements of address and data



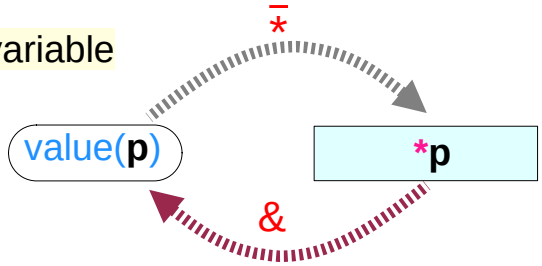
A : an address value - a number

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



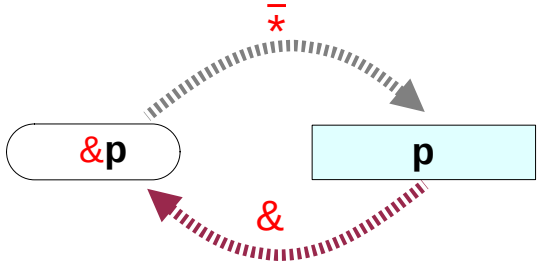
X : a variable ***p**, **p**, **a**

***p :** a dereferenced variable



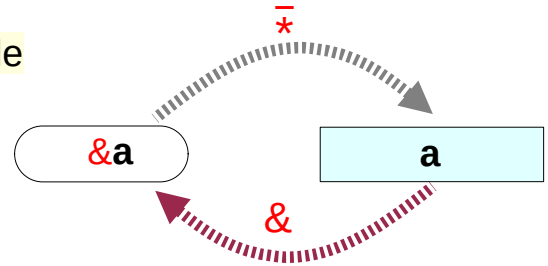
Pointer Dereferencing

p : a pointer variable



Pointer Data

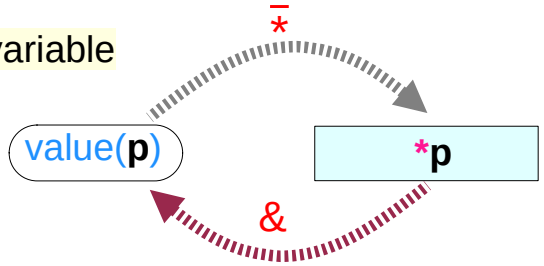
a : a primitive variable



Non-pointer Data

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

***p :** a dereferenced variable

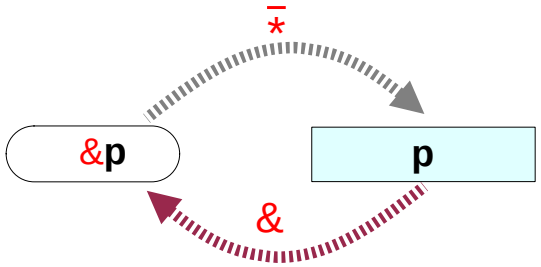


$\bar{*}$ value(**p**) = ***p**

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

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

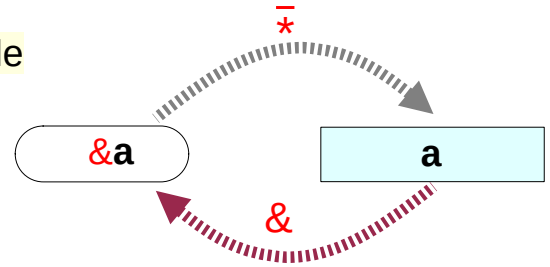
p : a pointer variable



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

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

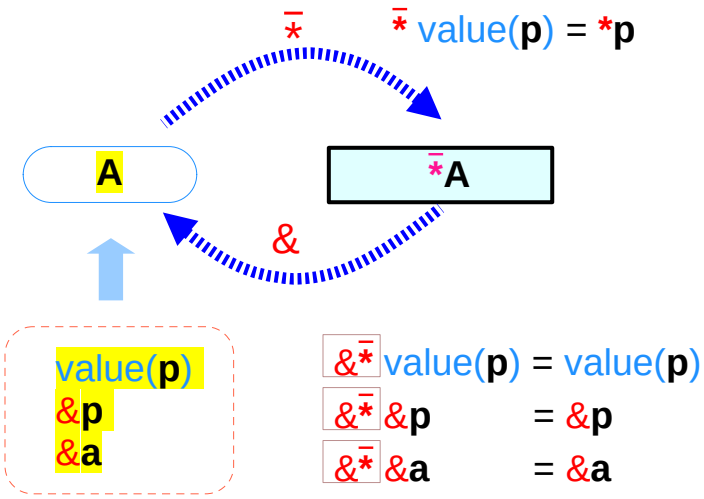
a : a primitive variable

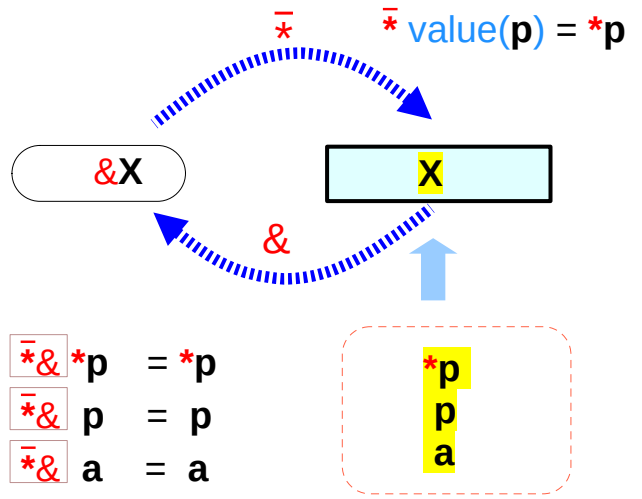


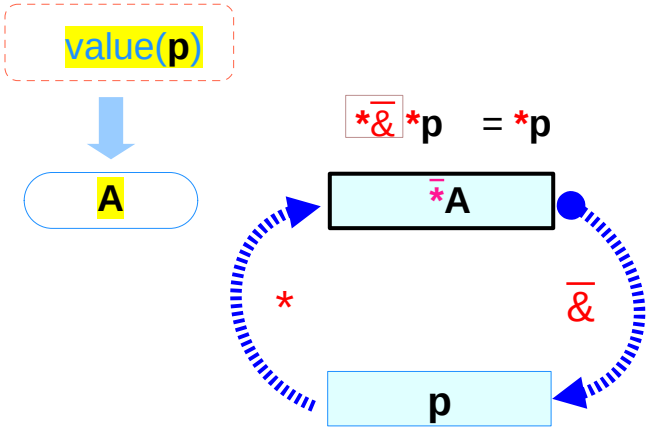
$\bar{*}$ &**a** = **a**

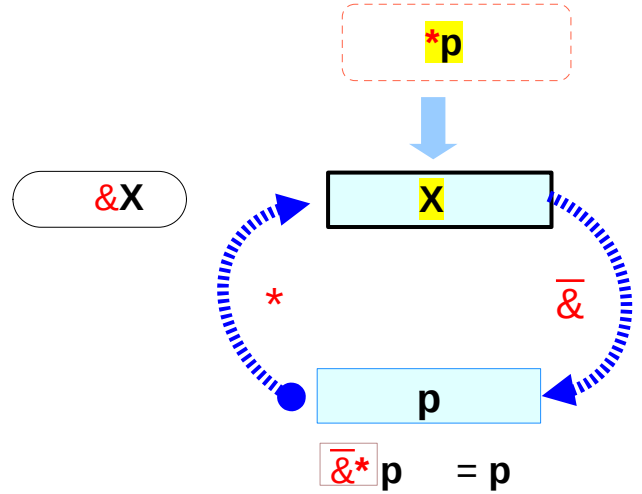
$\&\bar{*}$ &**a** = &**a**

Requirements of pointed address and data



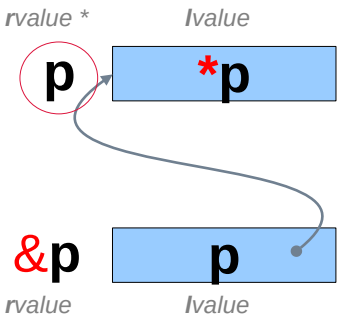




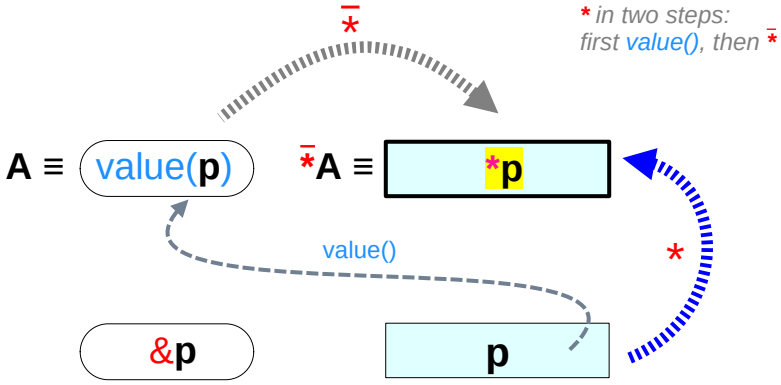


& and * operators in pointer de-referecing (1)

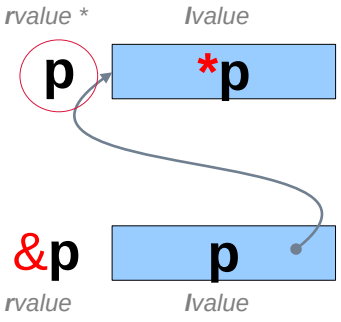
Two step De-reference * operation



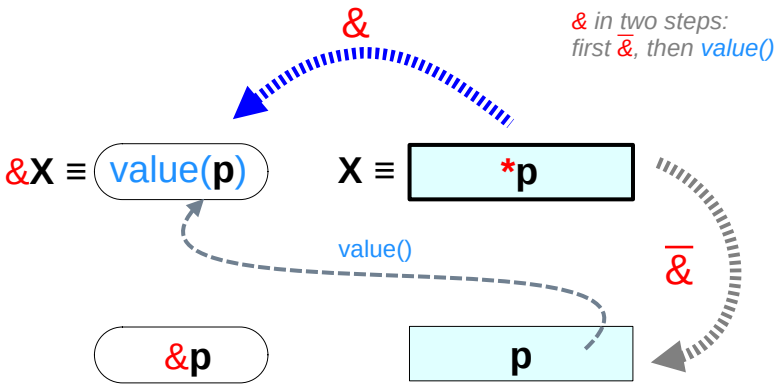
*p	=	$\bar{*}$value(p)
C Expressions		Mixed Expressions
$\bar{*}$A	=	*value^{-1}(A)



Two step Address-of & operation

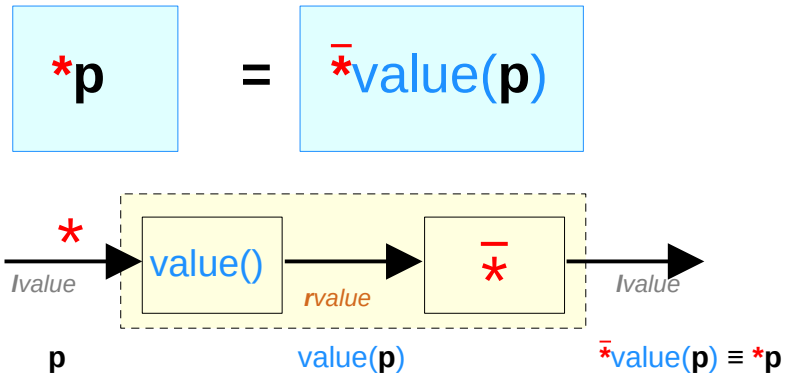


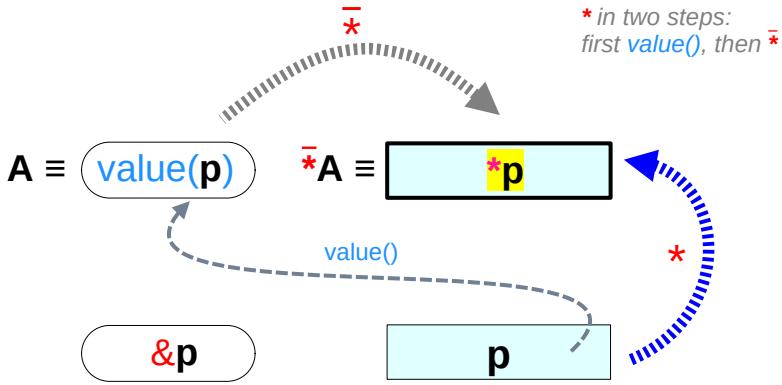
&X	=	value($\bar{\&}$X)
C Expressions		Mixed Expressions
$\bar{\&}$X	=	value^{-1}(&X)



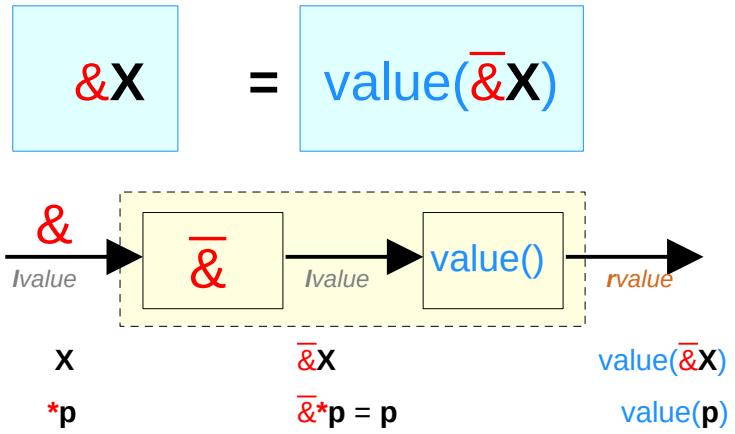
& and * operators in pointer de-referecing (2)

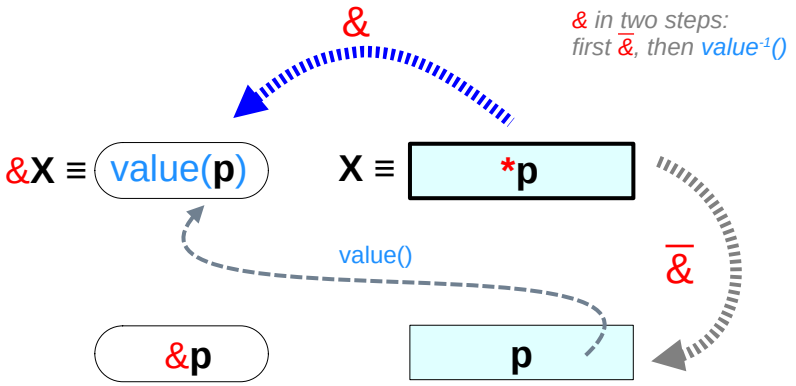
Two step De-reference * operation





Two step Address-of & operation

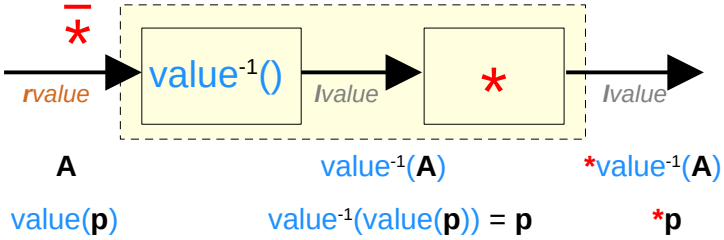


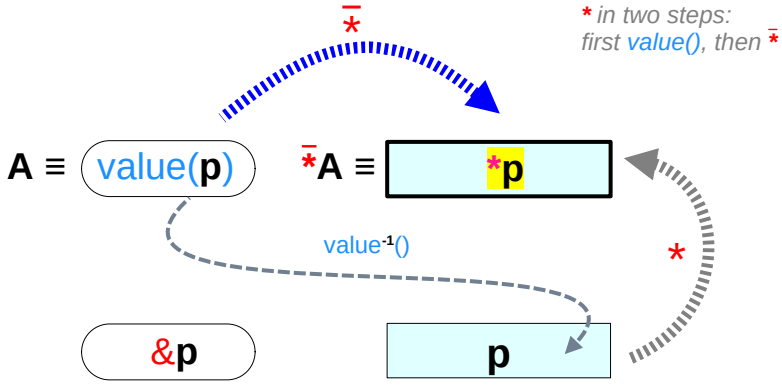


$\overline{\&}$ and $\overline{*}$ operators in pointer de-referecing (3)

Two step De-reference $\overline{*}$ operation

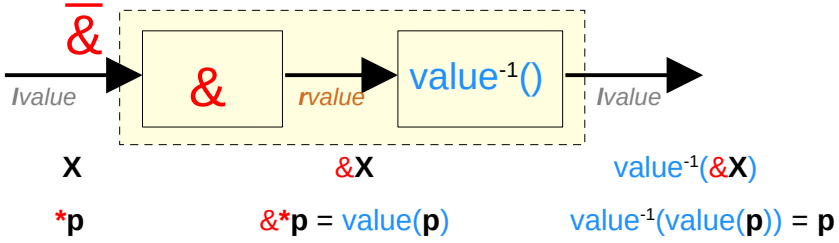
$\overline{*}A = *value^{-1}(A)$

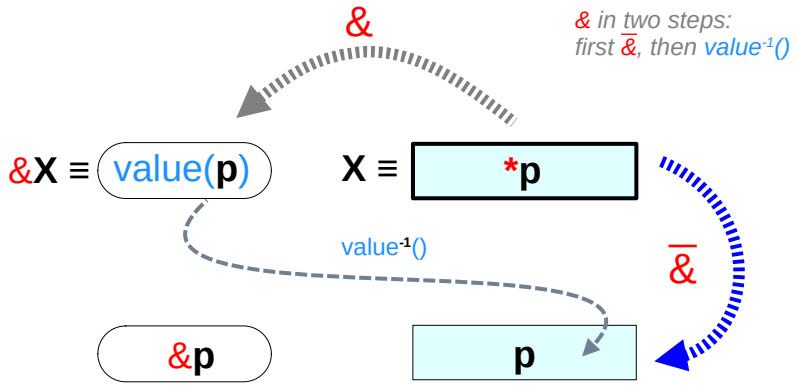




Two step Address-of $\overline{\&}$ operation

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





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

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

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

C Expressions

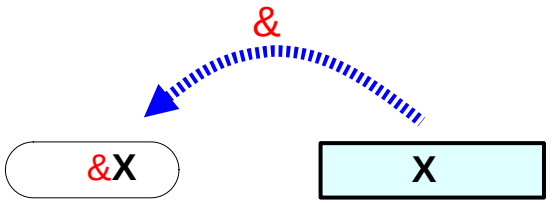
*p a dereferenced variable
p a pointer variable
a a primitive variable

& X

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

& p

& a



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

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

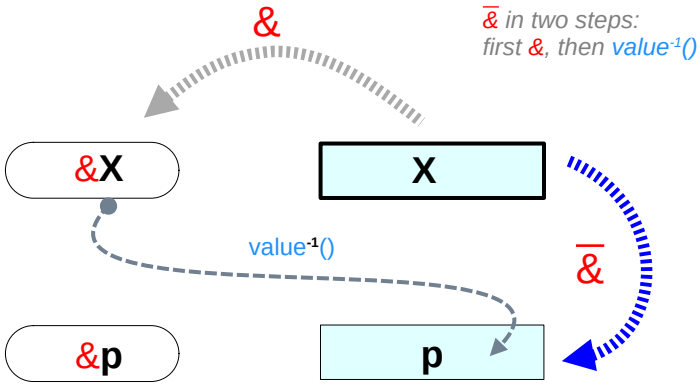
*p a dereferenced variable
~~p a pointer variable~~
~~a a primitive variable~~

$\overline{\&}$ 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*

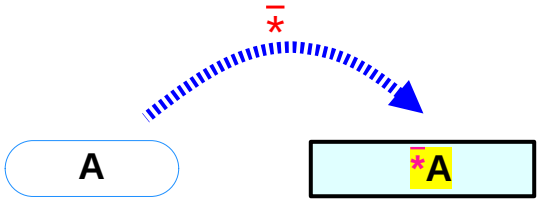
$$\bar{*}\mathbf{A} = \bar{*}\text{value}^{-1}(\mathbf{A})$$

*̄ **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*~~

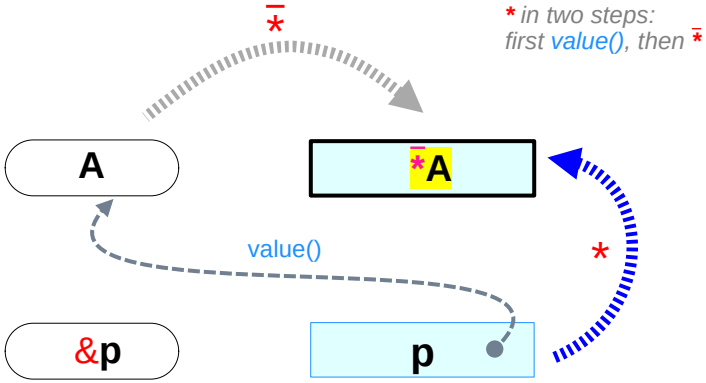
$$\mathbf{*p} = \bar{*}\text{value}(\mathbf{p})$$

***̄ **p**

***̄ **p** ≡ *̄value(**p**)

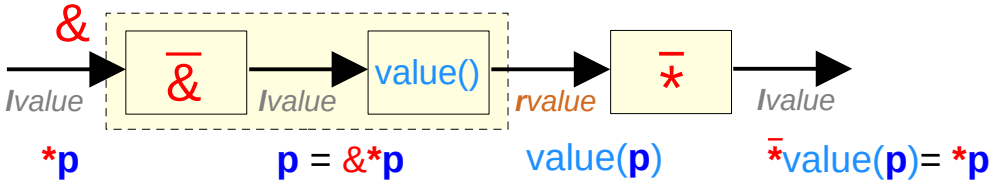
~~*̄ &**p** = **p**~~

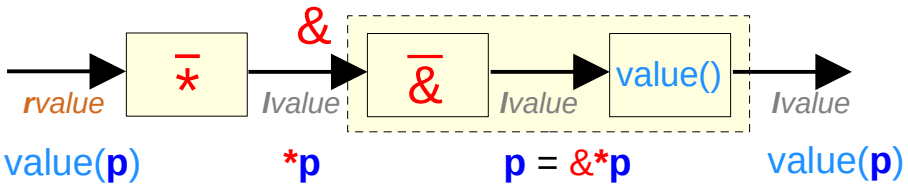
~~*̄ &**a** = **a**~~

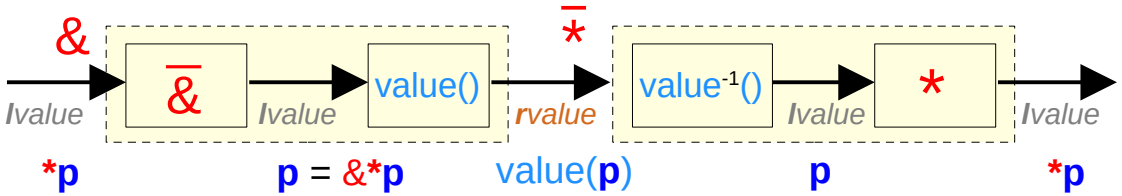


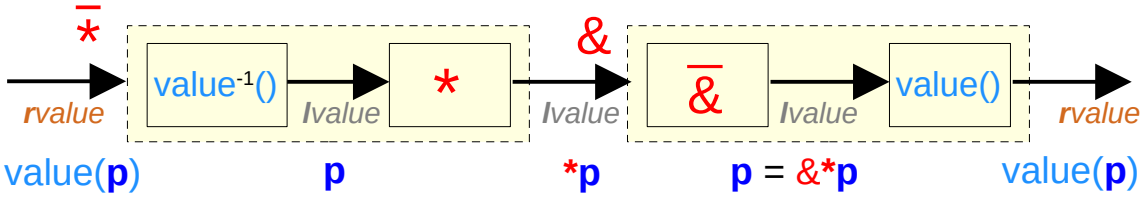
C Expressions

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





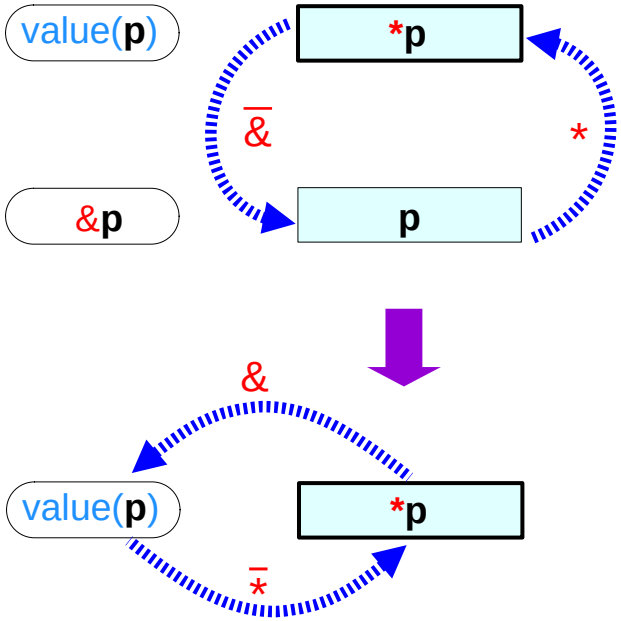




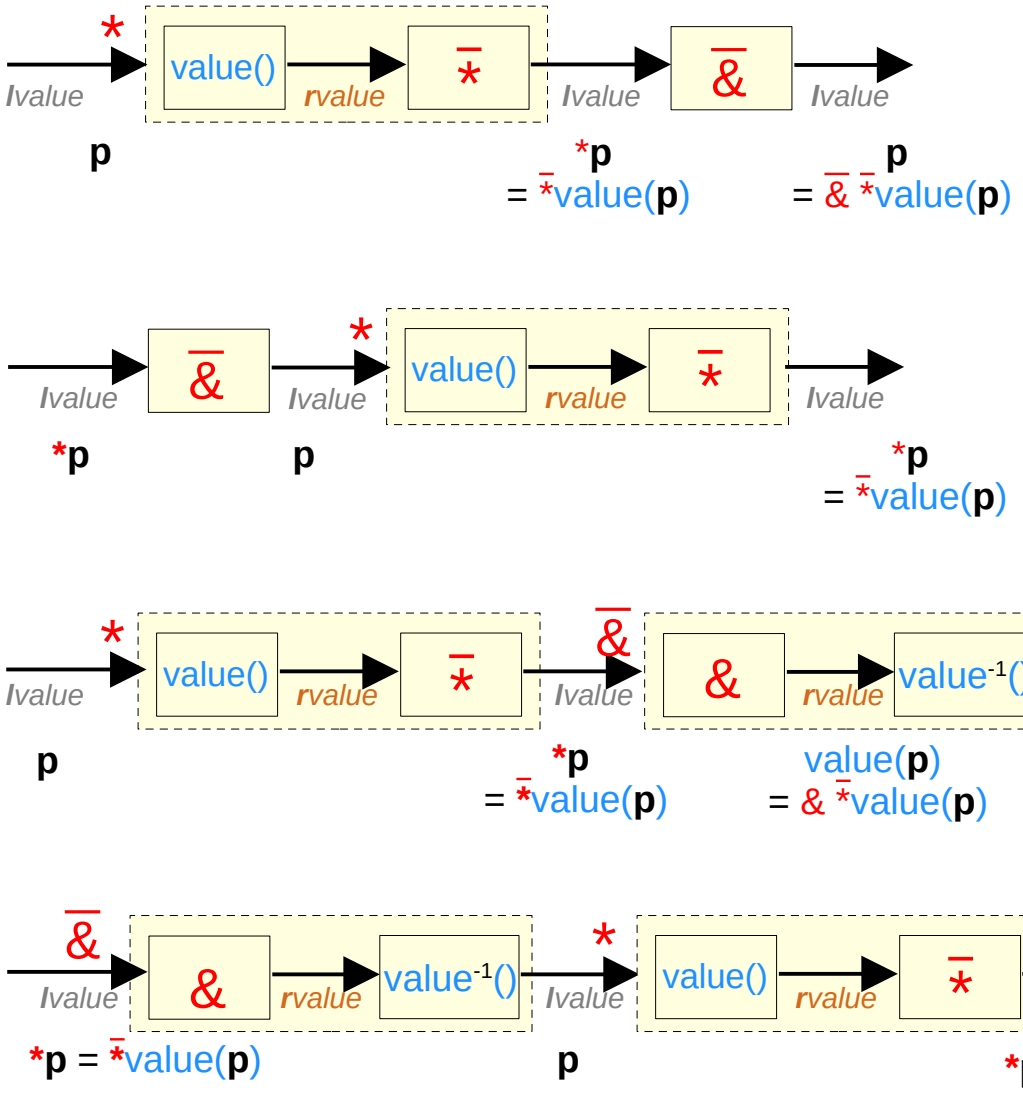
$\bar{*}$ value(**p**) = ***p**

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

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



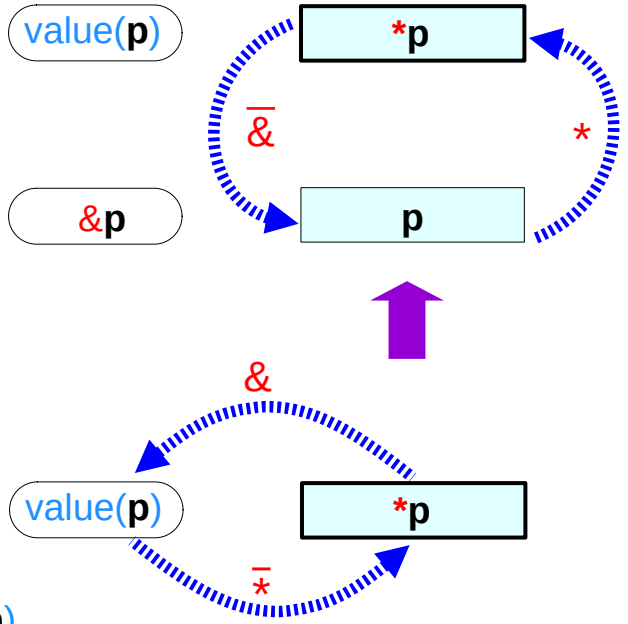
Inverse operators of & and *



*⁻¹ value(**p**) = ***p**

& * **p** = **p** **p** = & *⁻¹value(**p**)

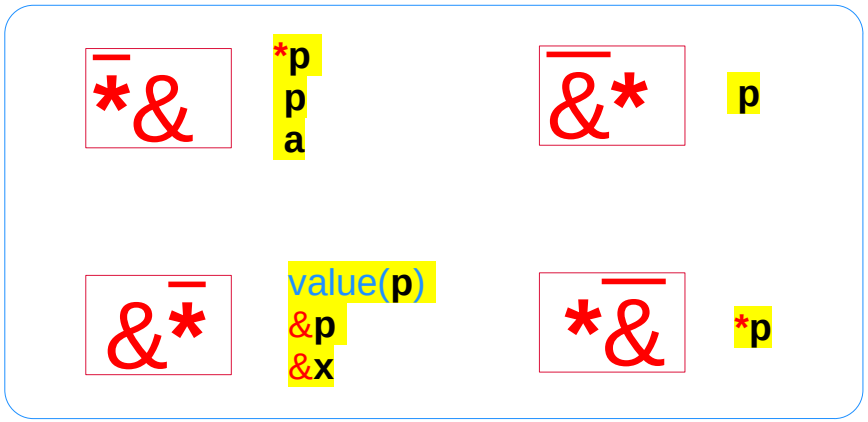
* & ***p** = ***p** value(**p**) = & *⁻¹value(**p**)



Inverse operators in pointer referencing (1)

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

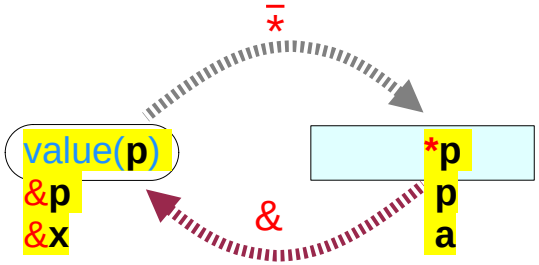
&* value(p) = value(p)
&* &p = &p
&* &a = &a

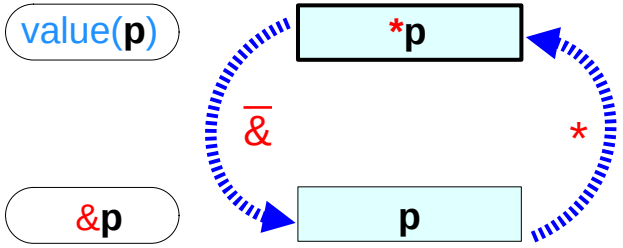


&* p = p

*& *p = *p

Math expressions

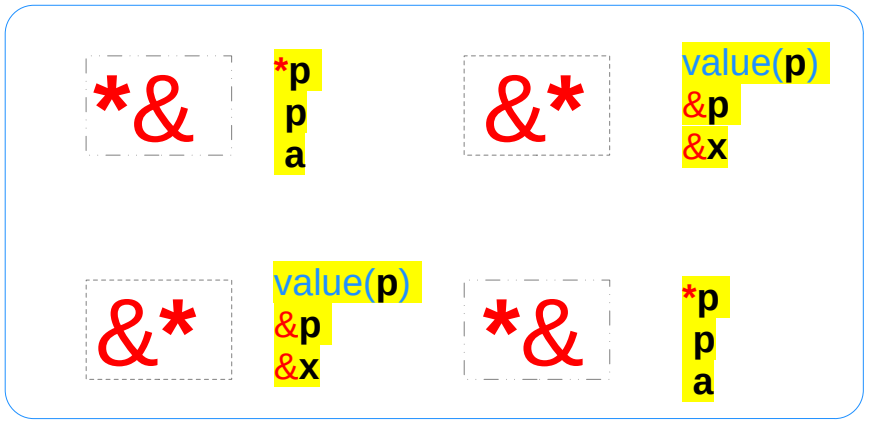




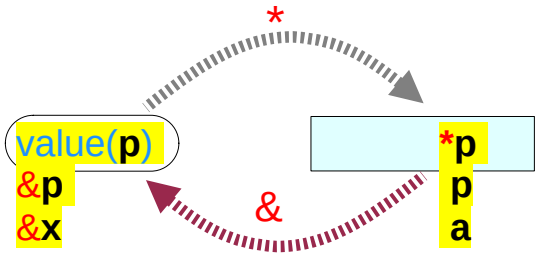
Inverse operators in pointer referencing (2)

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

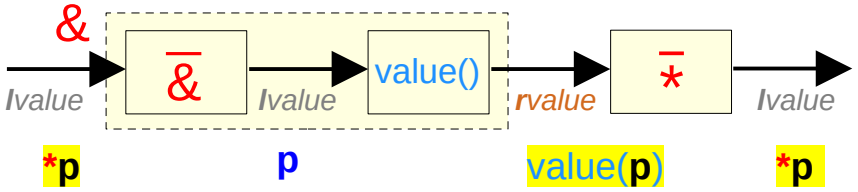
&* value(p) = value(p)
&* &p = &p
&* &a = &a

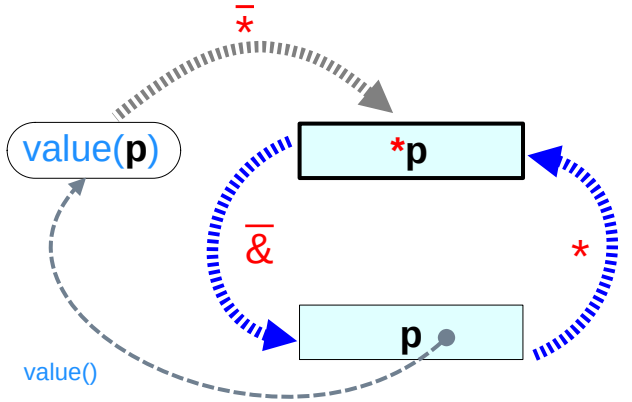


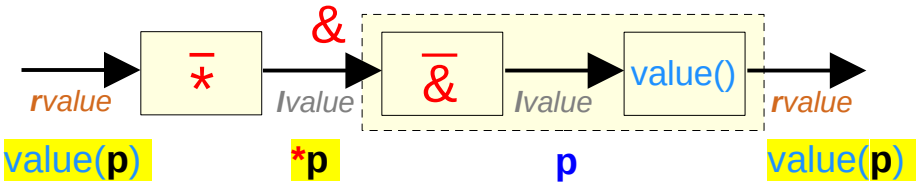
C expressions

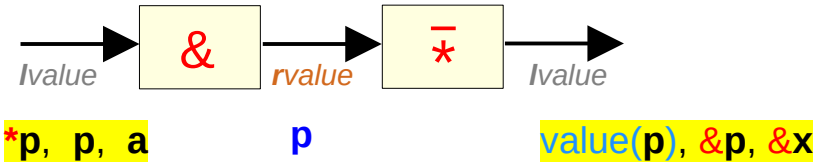


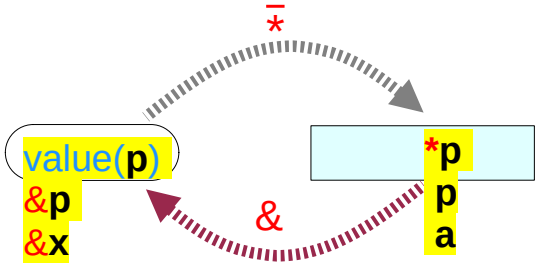
Inverse operators in pointer referencing (3)

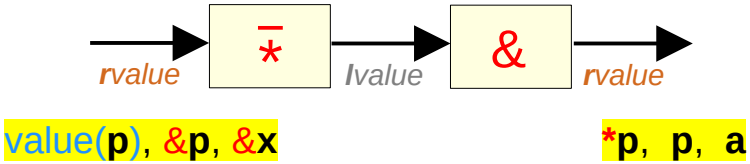




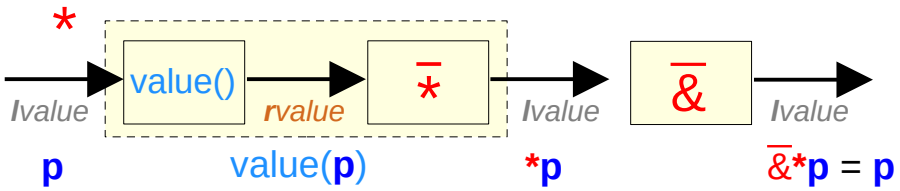


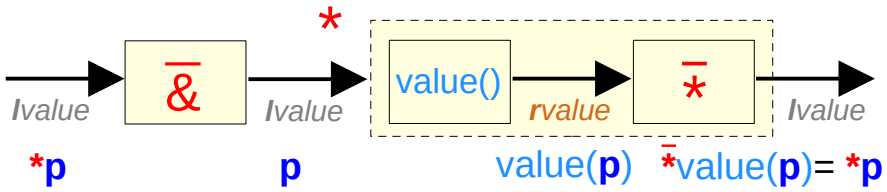


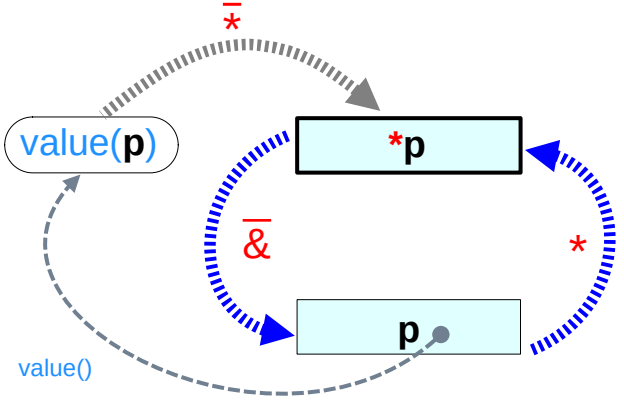


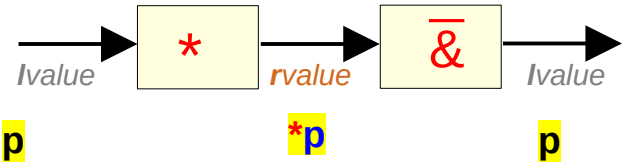


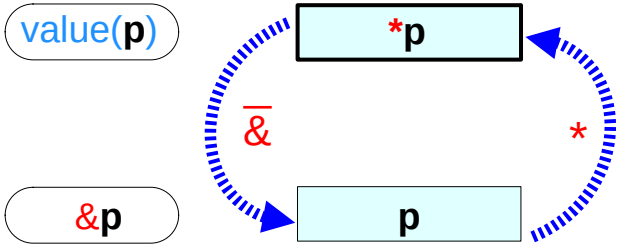
Inverse operators in pointer referencing (3)

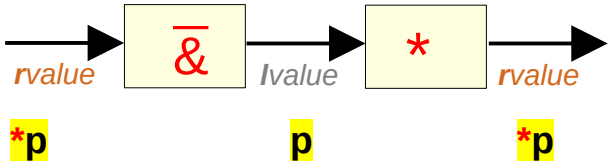












References

[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