Address-of and dereference operators

Copyright (c) 2026 - 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.
This document was produced by using LibreOffice.

& address-of operator

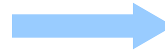
***** dereference operator

Address-of operator and dereferencing operator

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

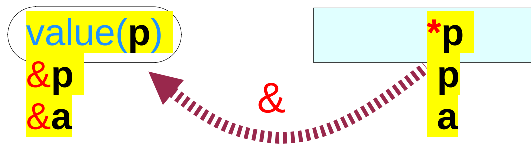
*a **variable** has a memory location whose value can be changed by an assignment*

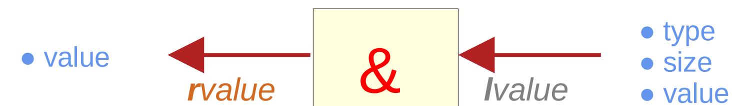
*a **variable** must be an **l**value*



***&variable** returns the address of a variable*

***&variable** returns an **r**value*



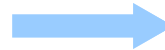


Address-of operator and dereferencing operator

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

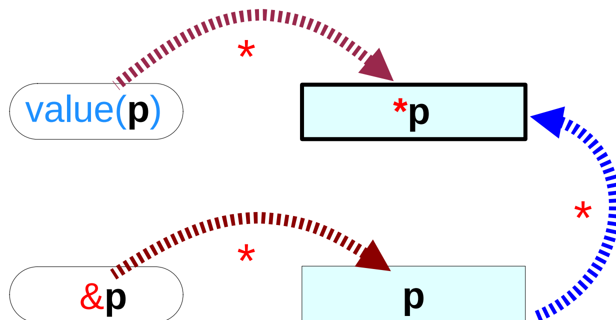
*an **address** can be an address value or a pointer variable that holds an address value*

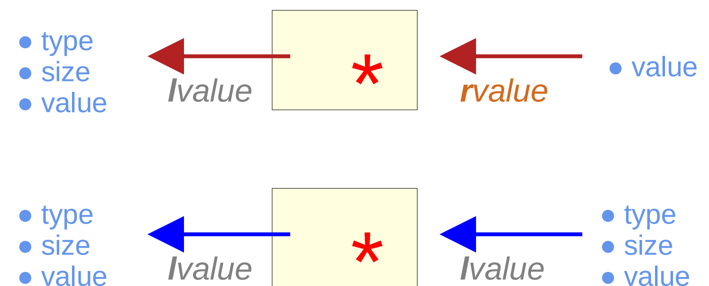
*an **address** must be an **r**value, or evaluated to be an **r**value*



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

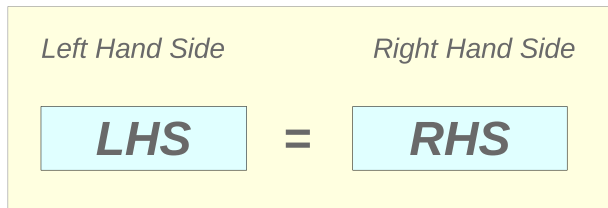
*** **address** returns an **l**value*





*p ≡ *value(p)

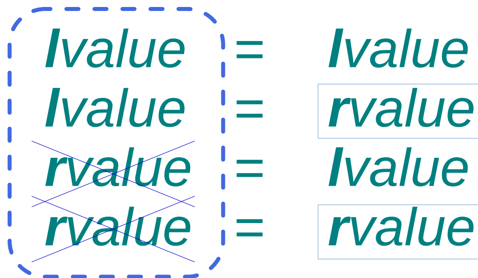
lvalue and rvalue in assignments

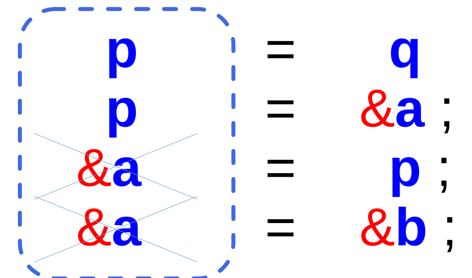


- *in the **LHS,** only **l**value can exist*
- ***r**value can exist only in the **RHS***

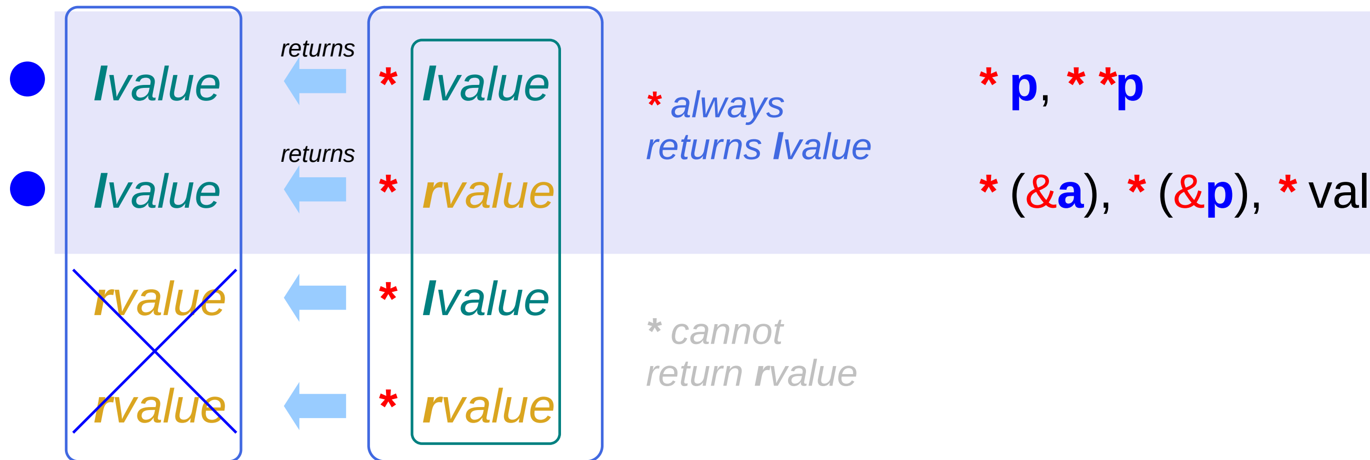
```
int   a,  b = 10 ;
int  *p, *q = &a ;
```

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





lvalue and rvalue with * and & operators

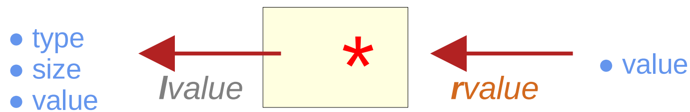


* always returns lvalue

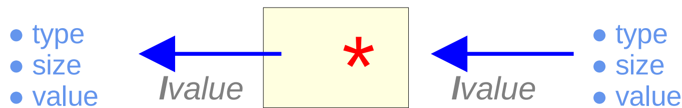
* p, * *p

* (&a), * (&p), * val

* cannot return rvalue



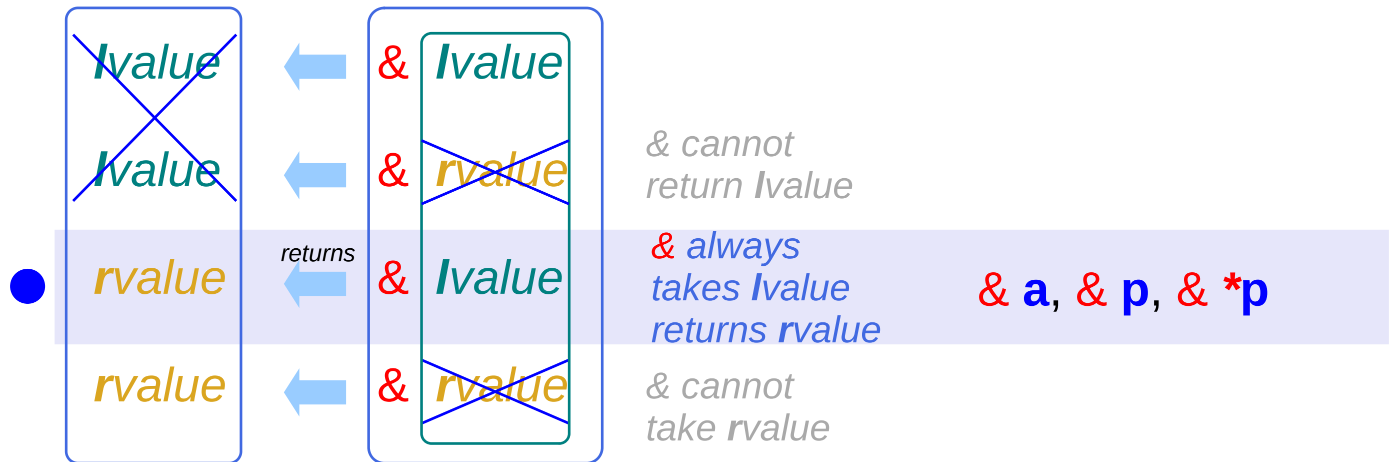
* can be applied to either an **lvalue** variable or a **rvalue** address



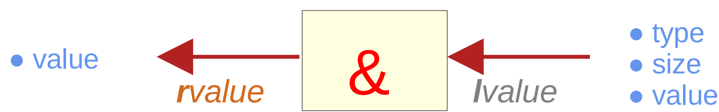
***p** ≡ *value(**p**)

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

lvalue and rvalue with * and & operators



& a, **& p**, **& *p**



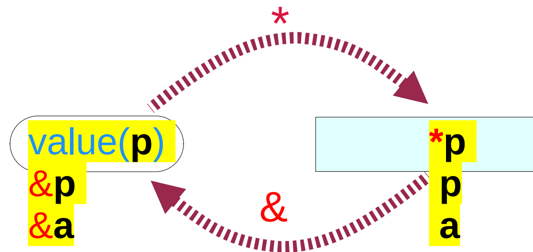
& can be applied
to only an **lvalue** variable and
returns only an **rvalue** address

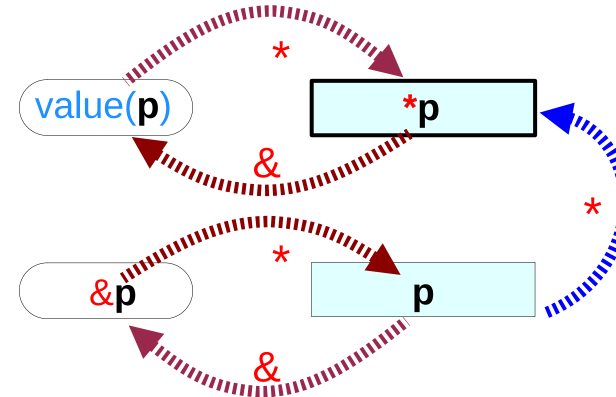
Variable types

a	non-pointer variables	***l**value*
p	pointer variables	***l**value*
*** p**	dereferenced variables	***l**value*
val	address values	***r**value*

C operators : *, &

C operators : *, &



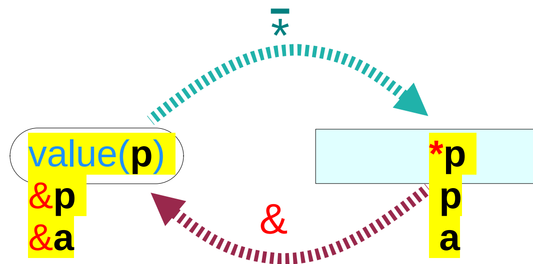


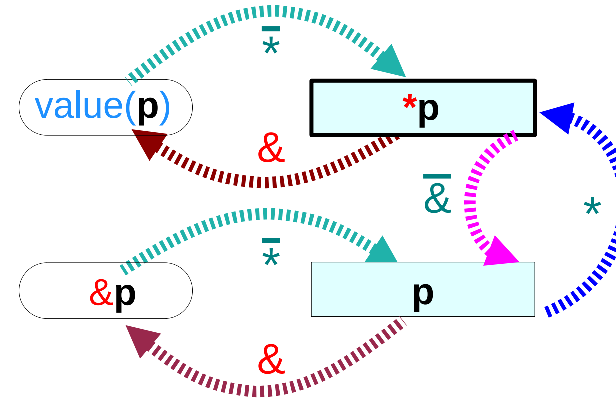
			*	~~a~~	lvalue
lvalue	p		*	p	lvalue
lvalue	* p		*	* p	lvalue
lvalue	* val		*	val	rvalue

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

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

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





				*	~~a~~	lvalue
	lvalue	p	⇐	*	p	lvalue
	lvalue	* p	⇐	*	* p	lvalue
★	lvalue	$\overline{*}$ val	⇐	$\overline{*}$	val	rvalue

	rvalue	& p	⇐	&	a	lvalue
	rvalue	& p	⇐	&	p	lvalue
	rvalue	& * p	⇐	&	* p	lvalue
★	lvalue	p	⇐	$\overline{\&}$	* p	lvalue

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

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

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

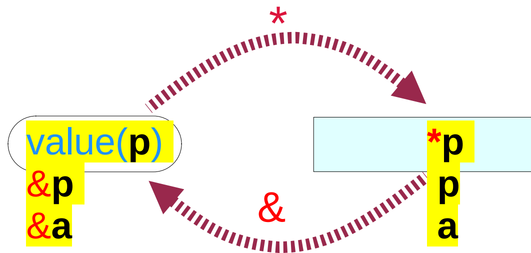
rvalue	& p ⇐	&	a	lvalue
rvalue	& p ⇐	&	p	lvalue
rvalue	& * p ⇐	&	* p	lvalue
		&	~~val~~	rvalue

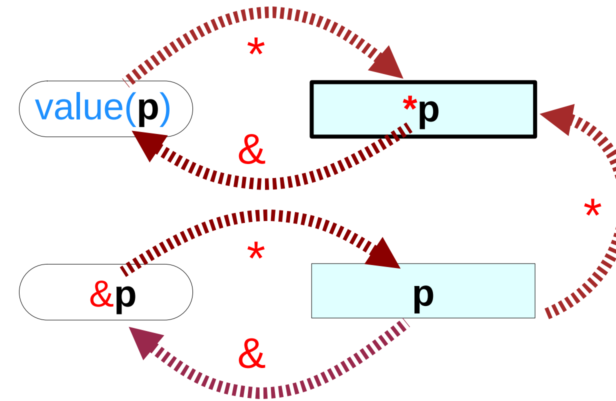
		$\bar{*}$	~~a~~	lvalue
		$\bar{*}$	~~p~~	lvalue
		$\bar{*}$	* ~~p~~	lvalue
★ lvalue	$\bar{*}$ val ⇐	$\bar{*}$	val	rvalue

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

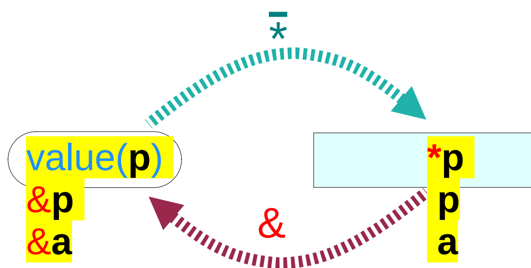
Comparisons (1)

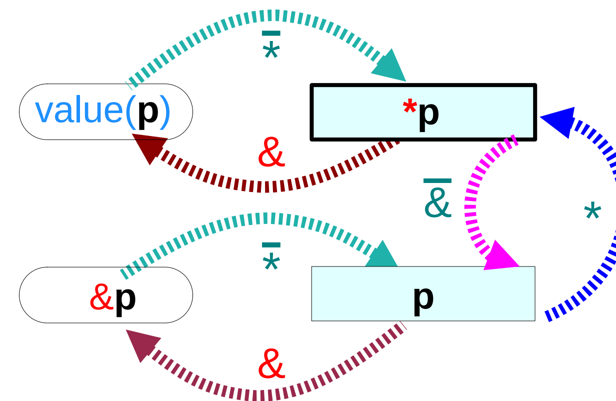
C operators : *, &





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





Comparisons (2)

C operators : *, &

			*	~~a~~	lvalue
lvalue	p	⇐	*	p	lvalue
lvalue	* p	⇐	*	* p	lvalue
lvalue	* val	⇐	*	val	rvalue

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

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

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

	rvalue	& a	⇐	&	a	lvalue
	rvalue	& p	⇐	&	p	lvalue
	rvalue	& * p	⇐	&	* p	lvalue
★	lvalue	p	⇐	$\bar{\&}$	* p	lvalue

Comparisons (3)

C operators : *, &

			*	~~a~~	*l*value
*l*value	p	⇐	*	p	*l*value
*l*value	* p	⇐	*	* p	*l*value
*l*value	* val	⇐	*	val	*r*value

*r*value	& p	⇐	&	a	*l*value
*r*value	& p	⇐	&	p	*l*value
*r*value	& * p	⇐	&	* p	*l*value
			&	~~val~~	*r*value

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

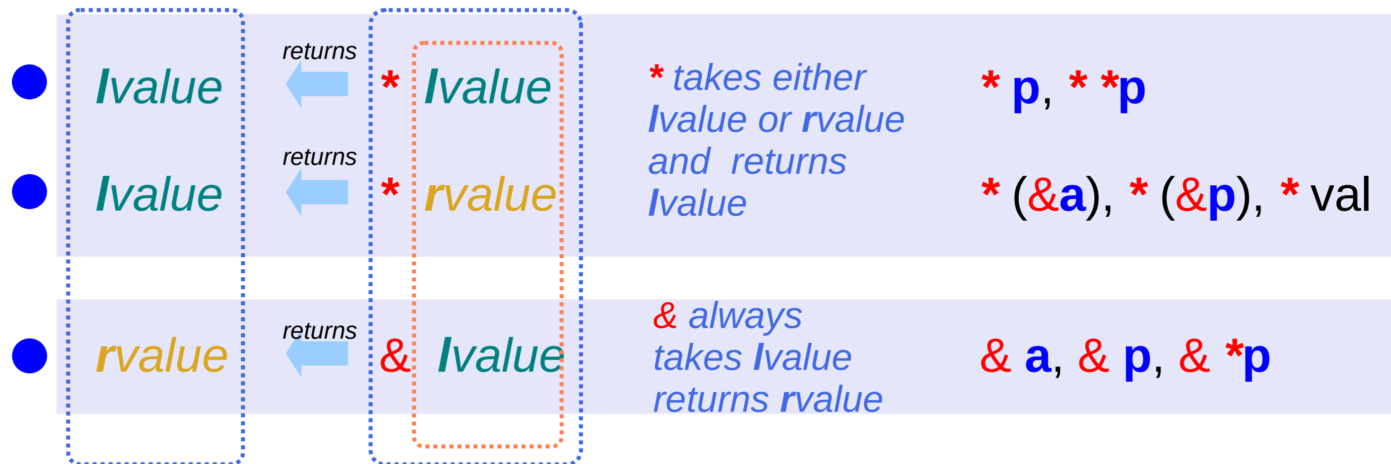
				*	~~a~~	*l*value
	*l*value	p	⇐	*	p	*l*value
	*l*value	* p	⇐	*	* p	*l*value
★				*	~~val~~	*r*value

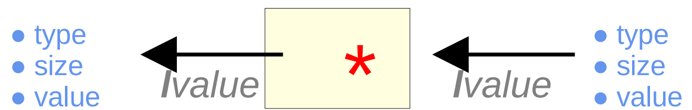
*r*value	& p	⇐	&	a	*l*value
*r*value	& p	⇐	&	p	*l*value
*r*value	& * p	⇐	&	* p	*l*value
			&	~~val~~	*r*value

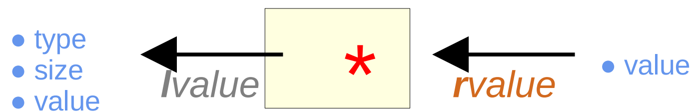
				$\bar{*}$	~~a~~	*l*value
				$\bar{*}$	~~p~~	*l*value
				$\bar{*}$	* ~~p~~	*l*value
★	*l*value	$\bar{*}$ val	⇐	$\bar{*}$	val	*r*value

				$\bar{\&}$	~~a~~	*l*value
				$\bar{\&}$	~~p~~	*l*value
★	*l*value	p	⇐	$\bar{\&}$	* p	*l*value
				$\bar{\&}$	~~val~~	*r*value

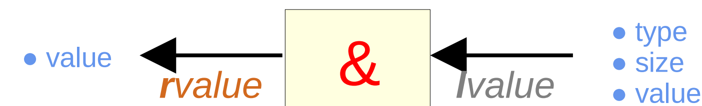
lvalue and rvalue with * and & operators





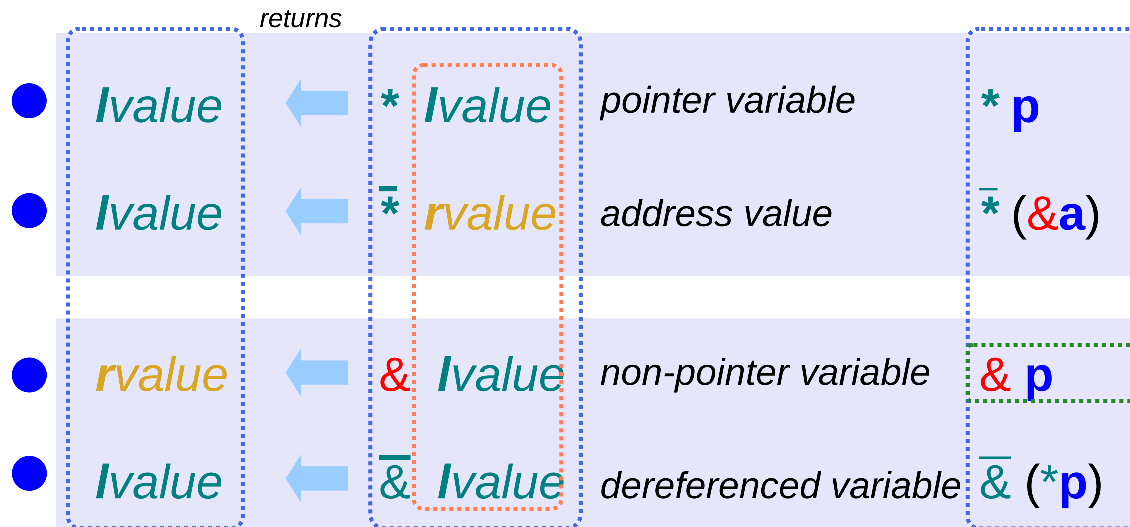


* (an **lvalue** variable)
* (an **rvalue** address)
* **operand** : an **lvalue** variable
* can be applied successively.



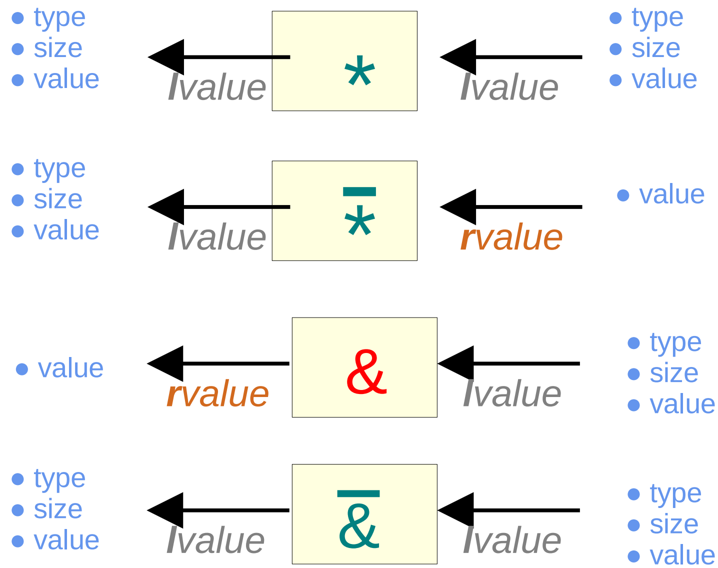
& (an **lvalue** variable)
& **operand** : an **rvalue** address,
& cannot be applied successively.

lvalue and rvalue with * and & operators



* can be applied successively.
$\overline{\&}$ can be applied successively.

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



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

$\overline{*}$ (an **rvalue** address)
$\overline{*}$ **operand**: an **lvalue** variable

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

$\overline{\&}$ (an dereferenced **lvalue** variable)
$\overline{\&}$ **operand**: an **lvalue** variable

Recursive application of the address-of operator

$* = \bar{*}v$

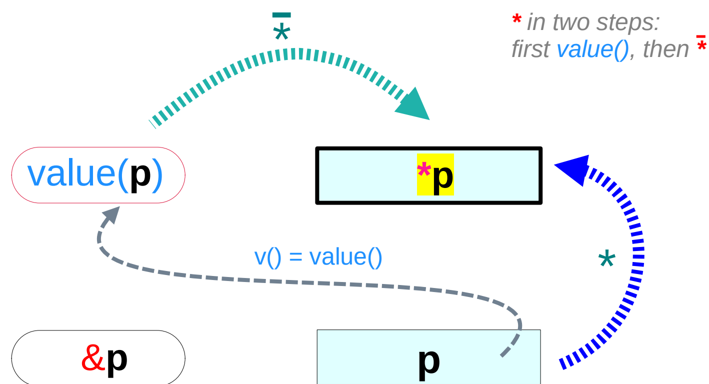
$*v^{-1} = \bar{*}vv^{-1}$

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

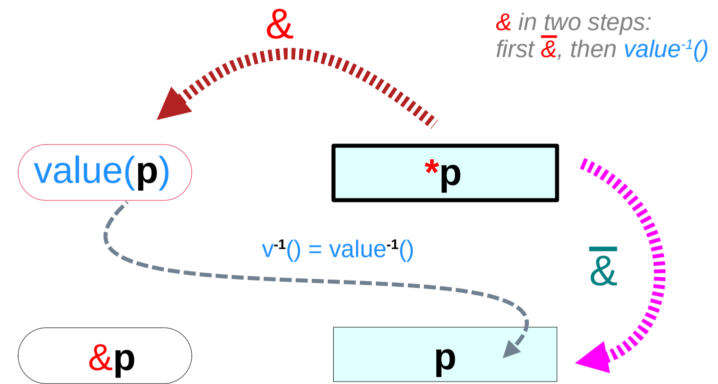
$\bar{\&} = v^{-1}\&$

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

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



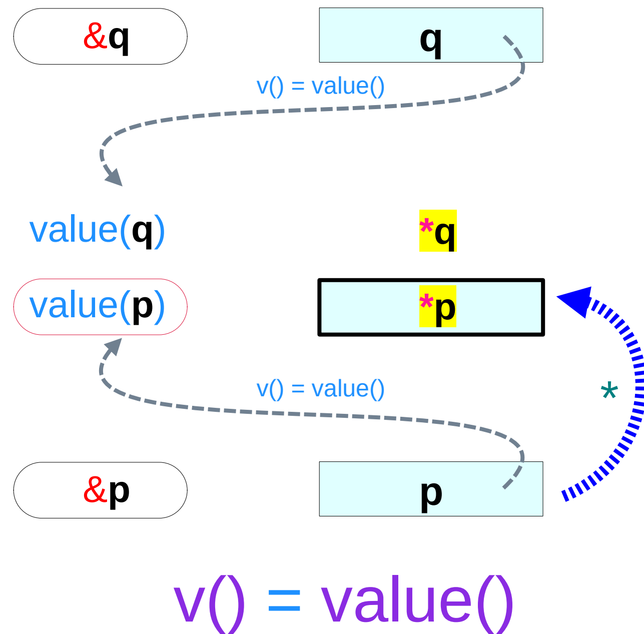
$v() = value()$



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

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



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

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

$* = \bar{*}v$

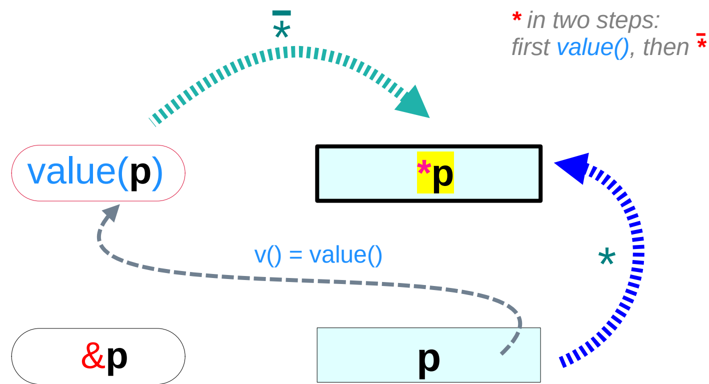
~~$*v^{-1} = \bar{*}vv^{-1}$~~

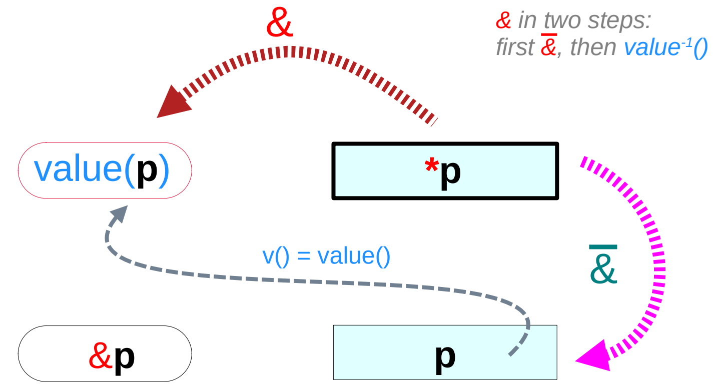
~~$*v^{-1} = \bar{*}$~~

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

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

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





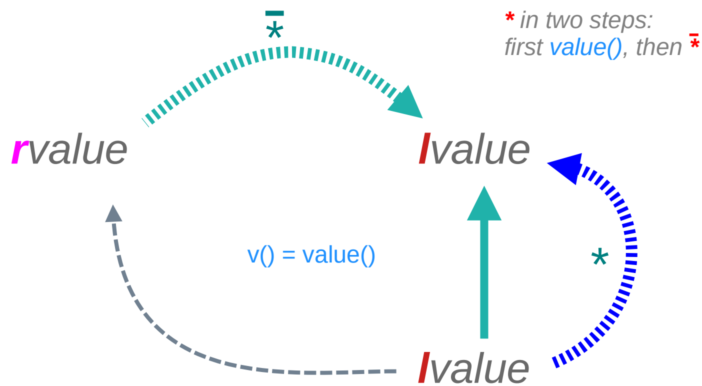
$v() = value()$

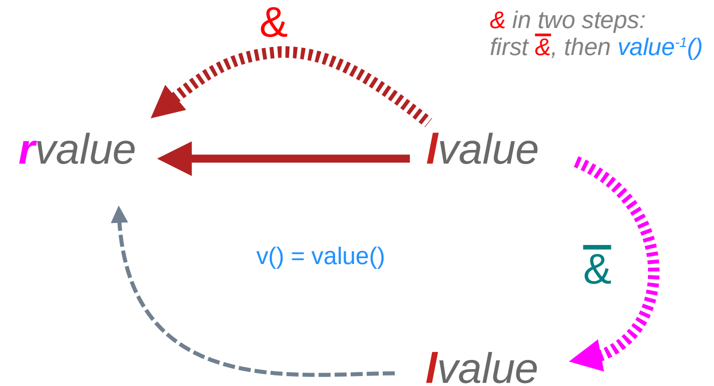
~~$v^{-1}() = value^{-1}()$~~

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

$* = \overline{*}v$

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

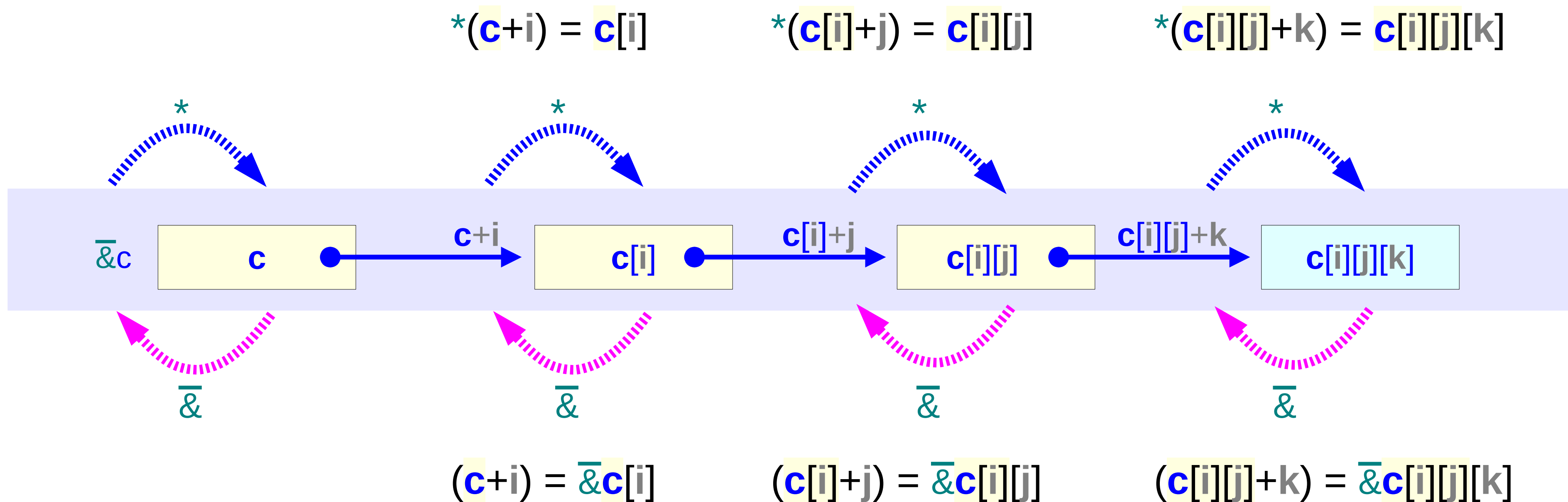




v() = value()

v() = value()

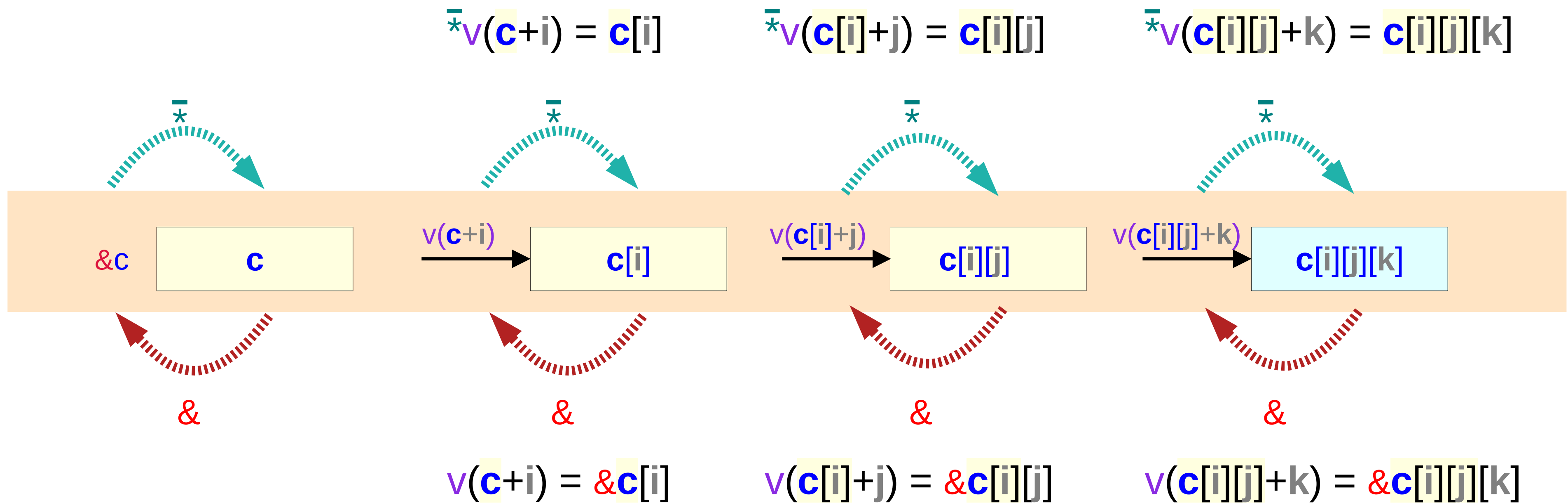
Equivalence Relations I : * and $\overline{\&}$ operators



equivalence relations I

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

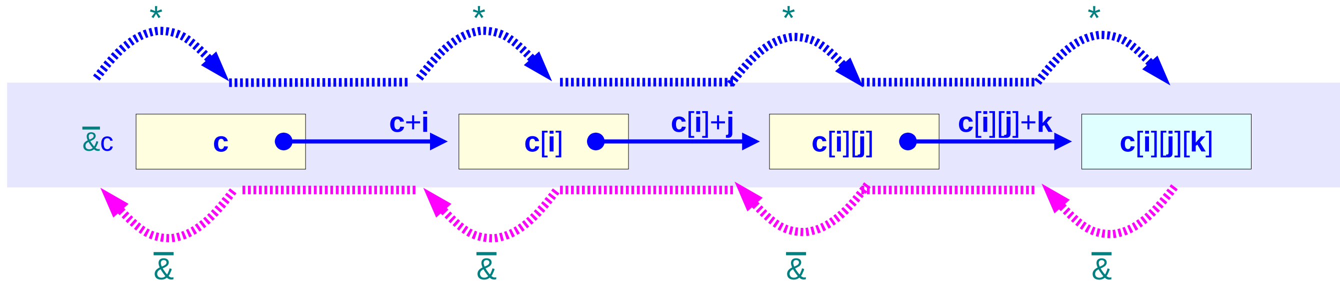
Equivalence Relations II : $\bar{*}$ and & operators



equivalence relations II

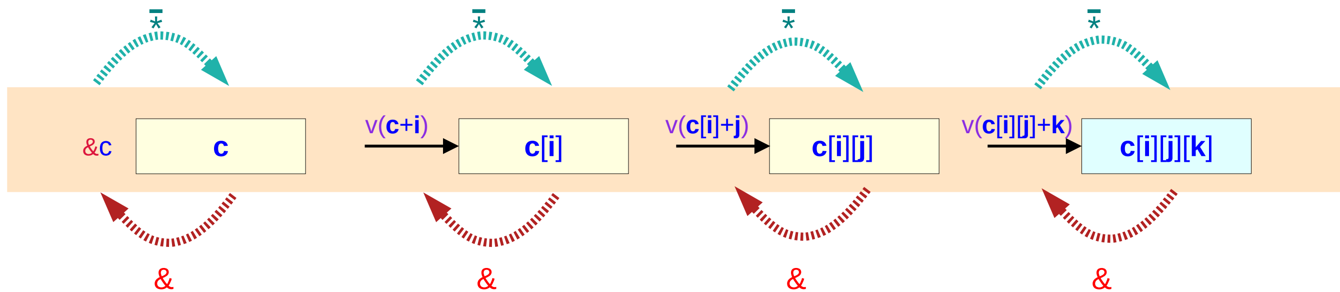
&c[i] ≡ v(c+i)	&c[i][j] ≡ v(c[i]+j)	&c[i][j][k] ≡ v(c[i][j]+j)
c[i] ≡ $\bar{*}$v(c+i)	c[i][j] ≡ $\bar{*}$v(c[i]+j)	c[i][j][k] ≡ $\bar{*}$v(c[i][j]+k)

Equivalence Relations (1)



equivalence relations I

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



equivalence relations II

&c[i]	≡ v(c+i)	&c[i][j]	≡ v(c[i]+j)	&c[i][j][k]	≡ v(c[i][j]+j)
c[i]	≡ $\overline{*}$v(c+i)	c[i][j]	≡ $\overline{*}$v(c[i]+j)	c[i][j][k]	≡ $\overline{*}$v(c[i][j]+k)

Equivalence Relations (2)

*l*value → * → *l*value *l*value ← $\overline{\&}$ ← *l*value	*(c+i) = c[i] (c+i) = $\overline{\&}$c[i]	*(c[i]+j) = c[i][j] (c[i]+j) = $\overline{\&}$c[i][j]	*(c[i][j]+k) = c[i][j][k] (c[i][j]+k) = $\overline{\&}$c[i][j][k]
*r*value → $\overline{*}$ → *l*value *r*value ← & ← *l*value	$\overline{*}$v(c+i) = c[i] v(c+i) = &c[i]	$\overline{*}$v(c[i]+j) = c[i][j] v(c[i]+j) = &c[i][j]	$\overline{*}$v(c[i][j]+k) = c[i][j][k] v(c[i][j]+k) = &c[i][j][k]

*l*value → * → *l*value *r*value → $\overline{*}$ → *l*value	*(c+i) = c[i] $\overline{*}$v(c+i) = c[i]	*(c[i]+j) = c[i][j] $\overline{*}$v(c[i]+j) = c[i][j]	*(c[i][j]+k) = c[i][j][k] $\overline{*}$v(c[i][j]+k) = c[i][j][k]
*l*value ← $\overline{\&}$ ← *l*value *r*value ← & ← *l*value	(c+i) = $\overline{\&}$c[i] v(c+i) = &c[i]	(c[i]+j) = $\overline{\&}$c[i][j] v(c[i]+j) = &c[i][j]	(c[i][j]+k) = $\overline{\&}$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
```

```c
#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]);


}
```

Dual Types and Address Replication

int (*)[3][4] → int (*)[4] → int (*) → int
int [2][3][4] → int [3][4] → int [4] → int

equivalence relations I

&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)

equivalence relations II

&c[i]	≡ v(c+i)	c[i]	≡ *v(c+i)
&c[i][j]	≡ v(c[i]+j)	c[i][j]	≡ *v(c[i]+j)
&c[i][j][k]	≡ v(c[i][j]+k)	c[i][j][k]	≡ *v(c[i][j]+k)

address replication

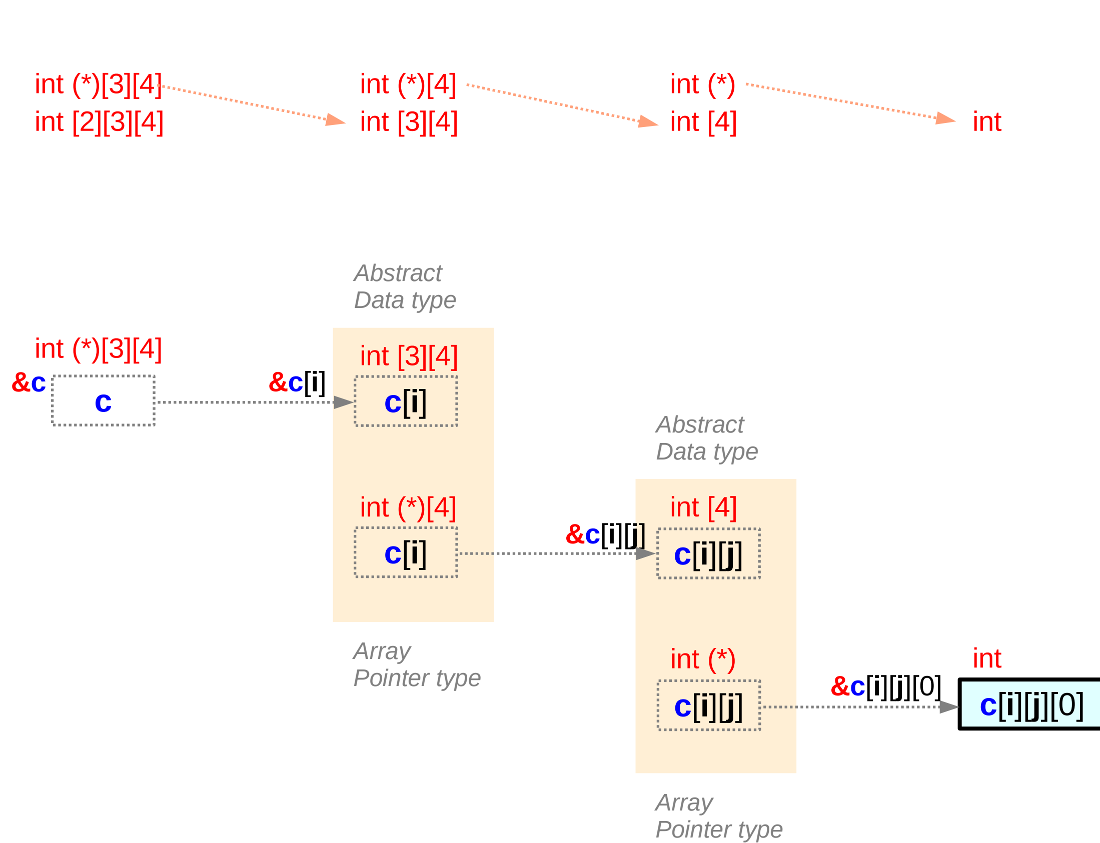
&c	= v(c)		
&c[i]	= v(c[i])	v(c+i)	= v(c[i])
&c[i][j]	= v(c[i][j])	v(c[i]+j)	= v(c[i][j])
&c[i][j][k]	≠ v(c[i][j][k])	v(c[i][j]+k)	≠ v(c[i][j][k])

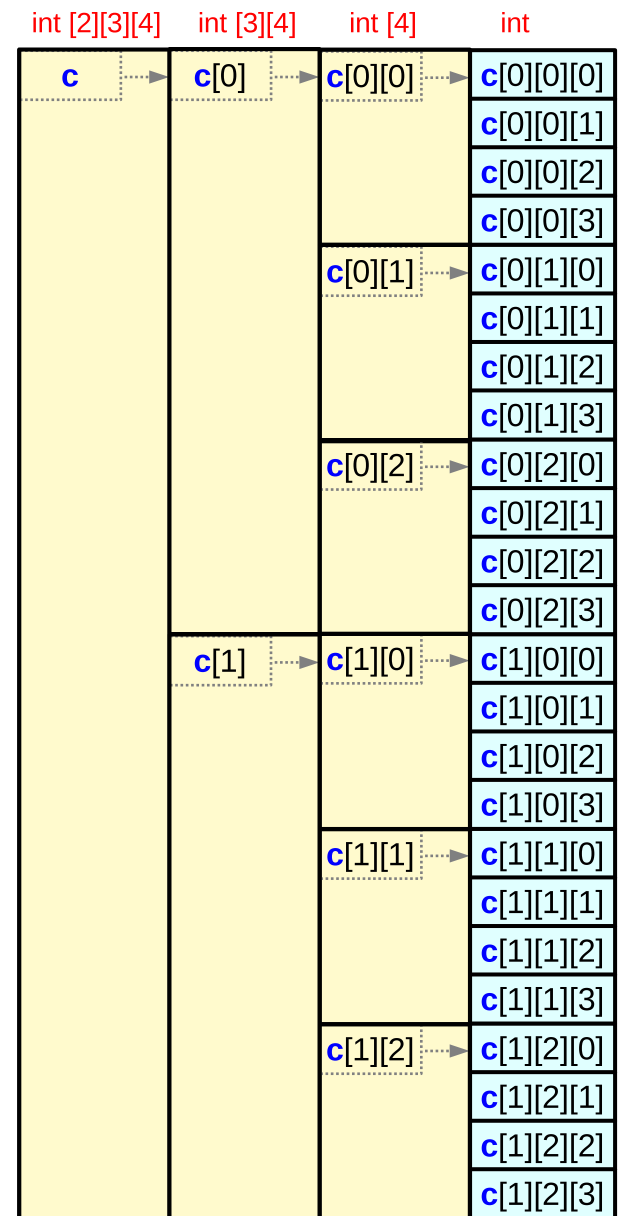
c[i], c[i][0] point to the same data c[i][0][0]
c, c[0], c[0][0] point to the same data c[0][0][0]

int [2][3][4] | int [3][4] | int [4] | int

c → c[0] → c[0][0] → c[0][0][0], c[0][0][1], c[0][0][2], c[0][0][3]
c[0][1] → c[0][1][0], c[0][1][1], c[0][1][2], c[0][1][3]
c[0][2] → c[0][2][0], c[0][2][1], c[0][2][2], c[0][2][3]
c[1] → c[1][0] → c[1][0][0], c[1][0][1], c[1][0][2], c[1][0][3]
c[1][1] → c[1][1][0], c[1][1][1], c[1][1][2], c[1][1][3]
c[1][2] → c[1][2][0], c[1][2][1], c[1][2][2], c[1][2][3]

Dual Types





Address Replications (1)

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

equivalence relations I

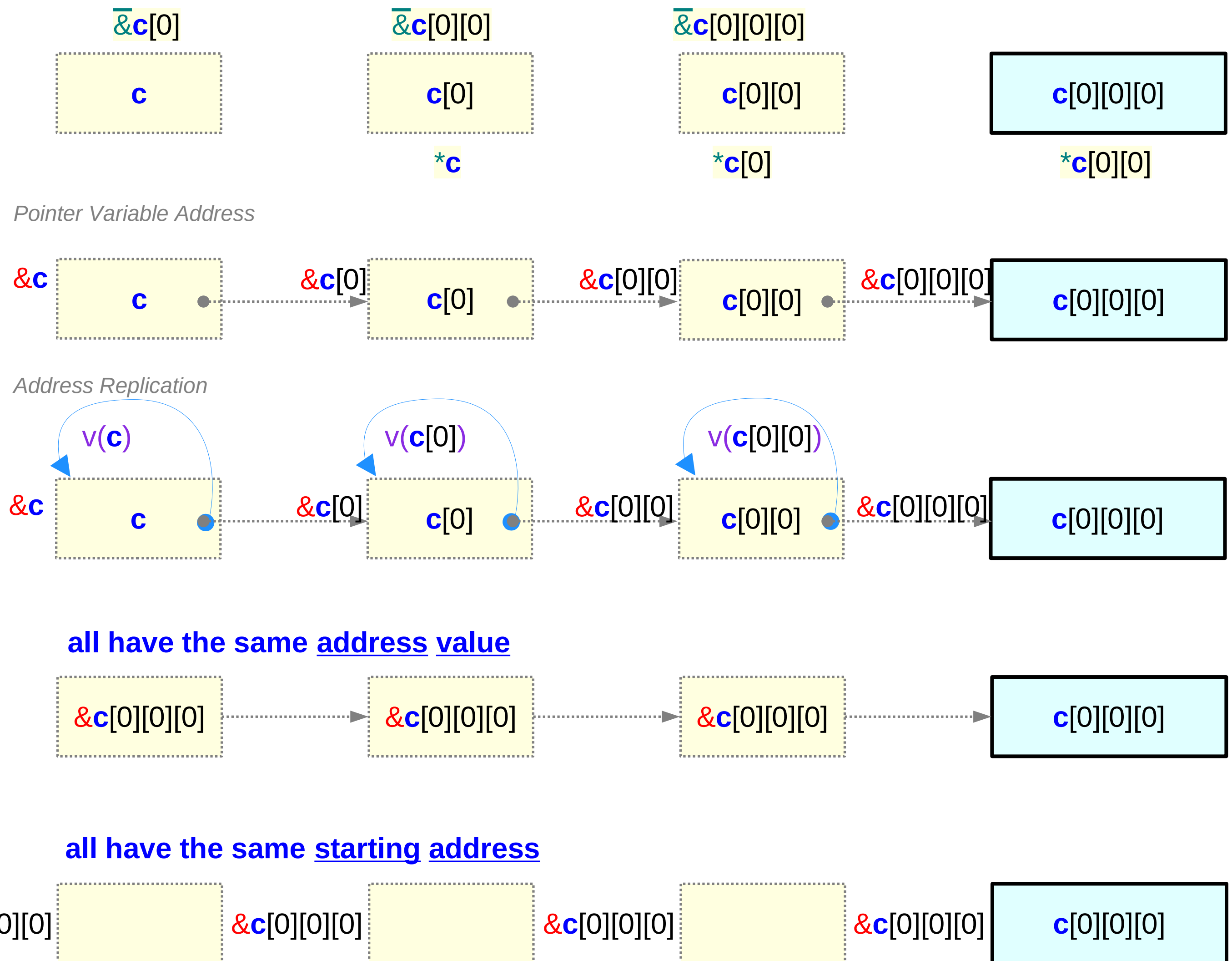
&c[i] ≡ (c+i)
&c[i][j] ≡ (c[i]+j)
&c[i][j][k] ≡ (c[i][j]+k)

equivalence relations II

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

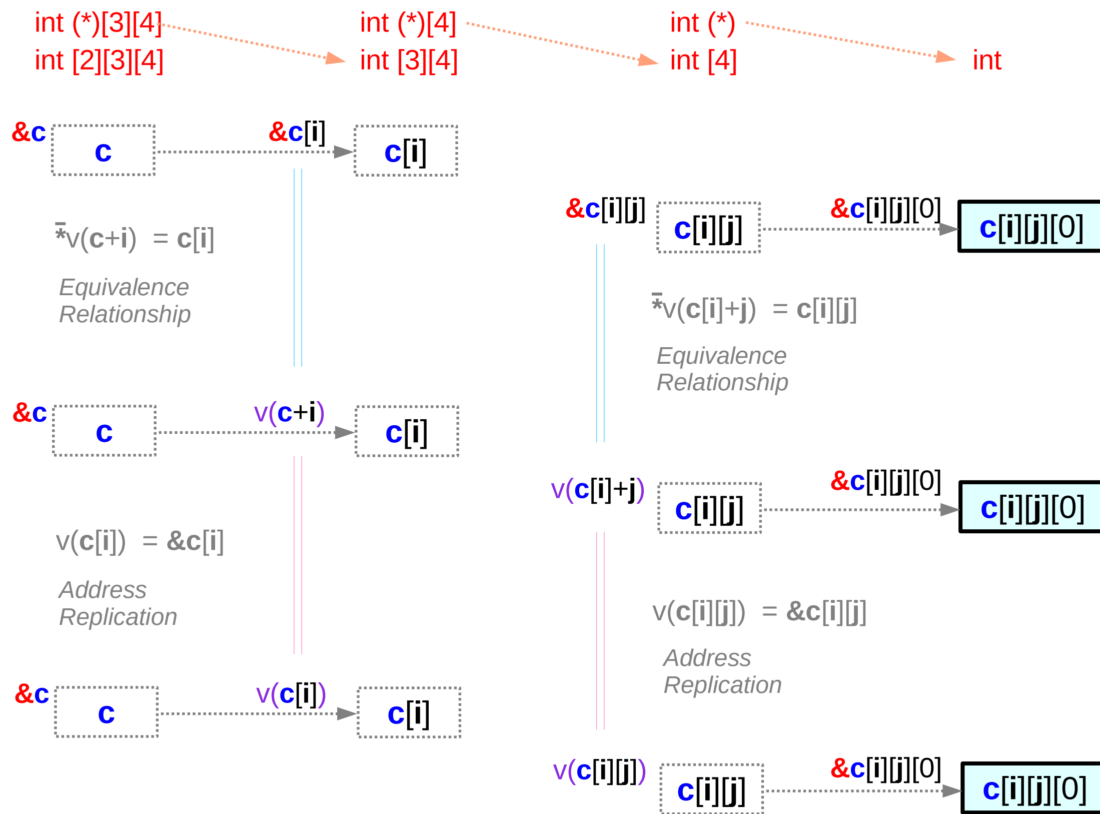
address replication

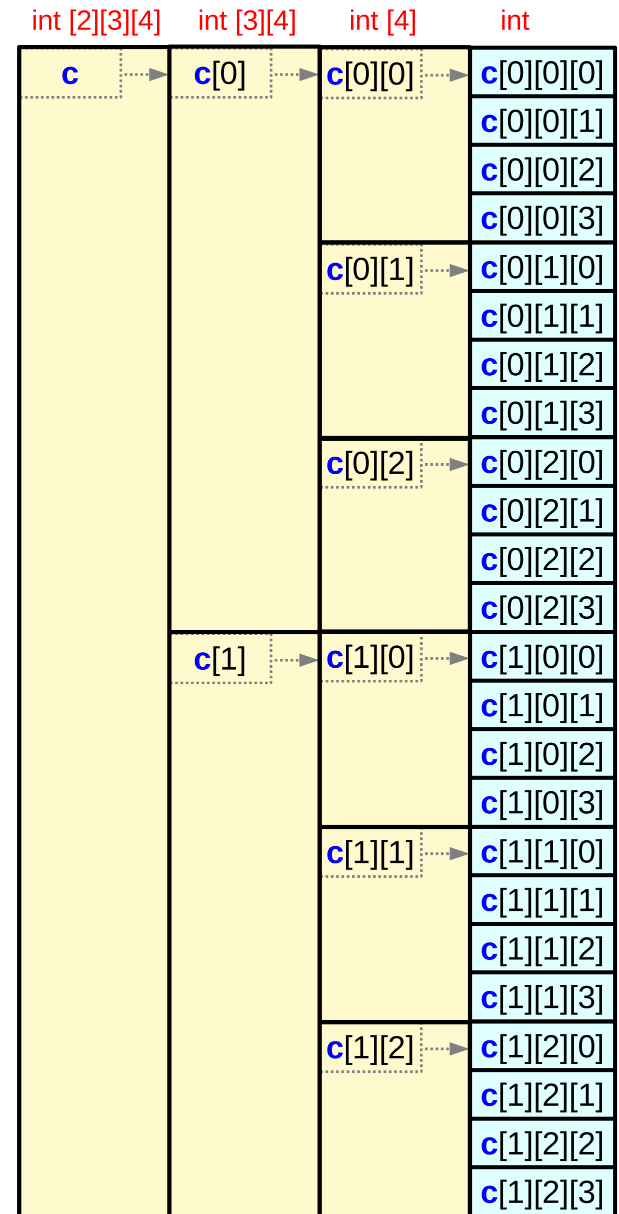
&c = v(c)
&c[i] = v(c[i])
&c[i][j] = v(c[i][j])
&c[i][j][k] ≠ v(c[i][j][k])



Young Won Lim
3/5/26

Address Replications (2)





address replication

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

Address Replications (3)

Pointer Variable Address

&c → c → &c[0] → c[0] → &c[0][0] → c[0][0] → &c[0][0][0] → c[0][0][0]

&c[i] → c[i] → &c[i][0] → c[i][0] → &c[i][0][0] → c[i][0][0]

&c[i][j] → c[i][j] → &c[i][j][0] → c[i][j][0]

Equivalence Relationship

c → v(c+0) → c[0] → v(c[0]+0) → c[0][0] → &c[0][0][0] → c[0][0][0]

v(c+i) → c[i] → v(c[i]+0) → c[i][0] → &c[i][0][0] → c[i][0][0]

v(c[i]+j) → c[i][j] → &c[i][j][0] → c[i][j][0]

Address Replication

v(c) → c → v(c[0]) → c[0] → v(c[0][0]) → c[0][0] → &c[0][0][0] → c[0][0][0]

v(c[i]) → c[i] → v(c[i][0]) → c[i][0] → &c[i][0][0] → c[i][0][0]

v(c[i][j]) → c[i][j] → &c[i][j][0] → c[i][j][0]

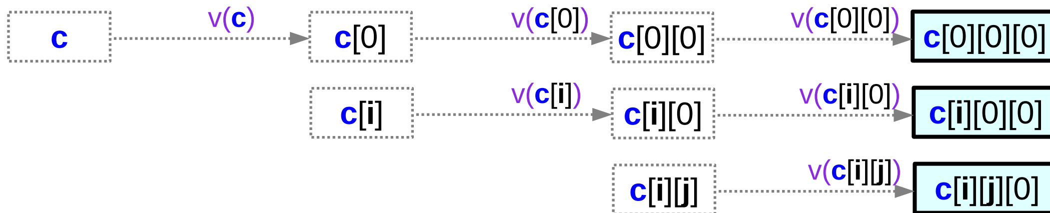
address replication

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

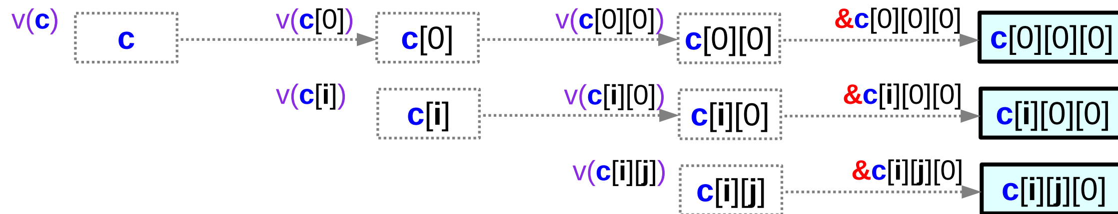
int [2][3][4]	int [3][4]	int [4]	int
c	c[0]	c[0][0]	c[0][0][0]
			c[0][0][1]
			c[0][0][2]
			c[0][0][3]
		c[0][1]	c[0][1][0]
			c[0][1][1]
			c[0][1][2]
			c[0][1][3]
		c[0][2]	c[0][2][0]
			c[0][2][1]
			c[0][2][2]
			c[0][2][3]
	c[1]	c[1][0]	c[1][0][0]
			c[1][0][1]
			c[1][0][2]
			c[1][0][3]
		c[1][1]	c[1][1][0]
			c[1][1][1]
			c[1][1][2]
			c[1][1][3]
		c[1][2]	c[1][2][0]
			c[1][2][1]
			c[1][2][2]
			c[1][2][3]

Address Replications (4)

by array pointer value



by address replication

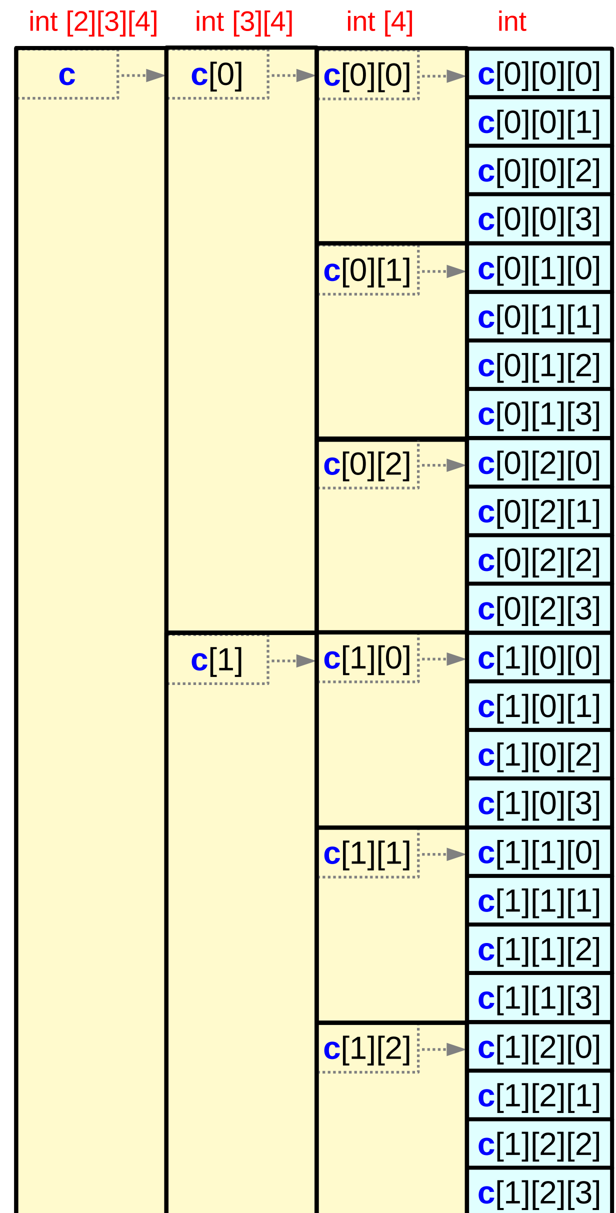


by equating these

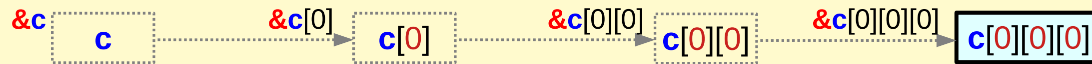
v(c) ◄ v(c[0]) ◄ v(c[0][0]) ◄ &c[0][0][0]

v(c[i]) ◄ v(c[i][0]) ◄ &c[i][0][0]

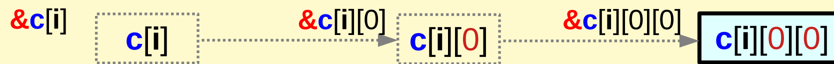
v(c[i][j]) ◄ &c[i][j][0]



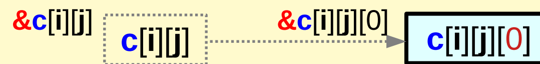
Address Replications (5)



c, c[0], c[0][0] have the same address of c[0][0][0]

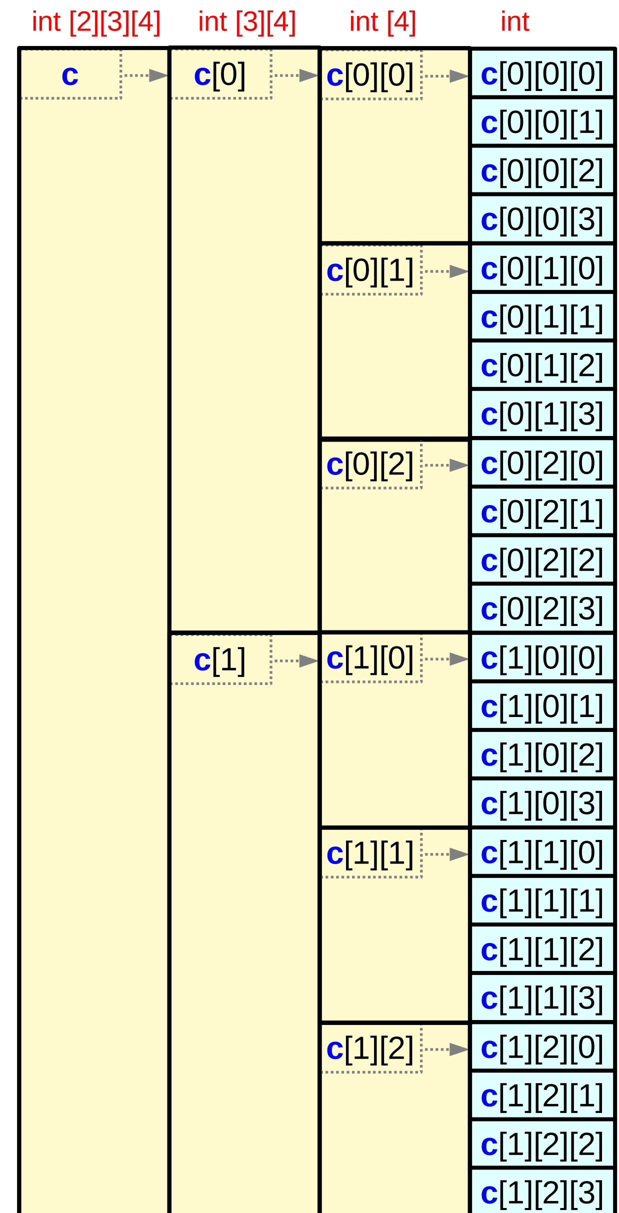


c[i], c[i][0] have the same address of c[i][0][0]



c[i][j] have the same address of c[i][j][0]

Only c[i][j][k] has a physical location
and its unique address &c[i][j][k]



Types, sizes, and values of sub-arrays (1)

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 (*)

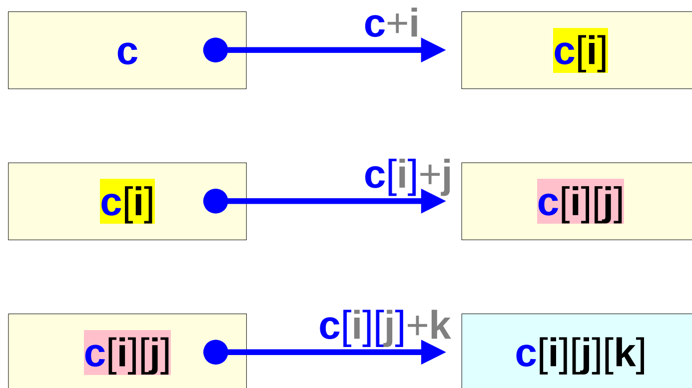
pointers to arrays

Types, sizes, and values of sub-arrays (2)

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

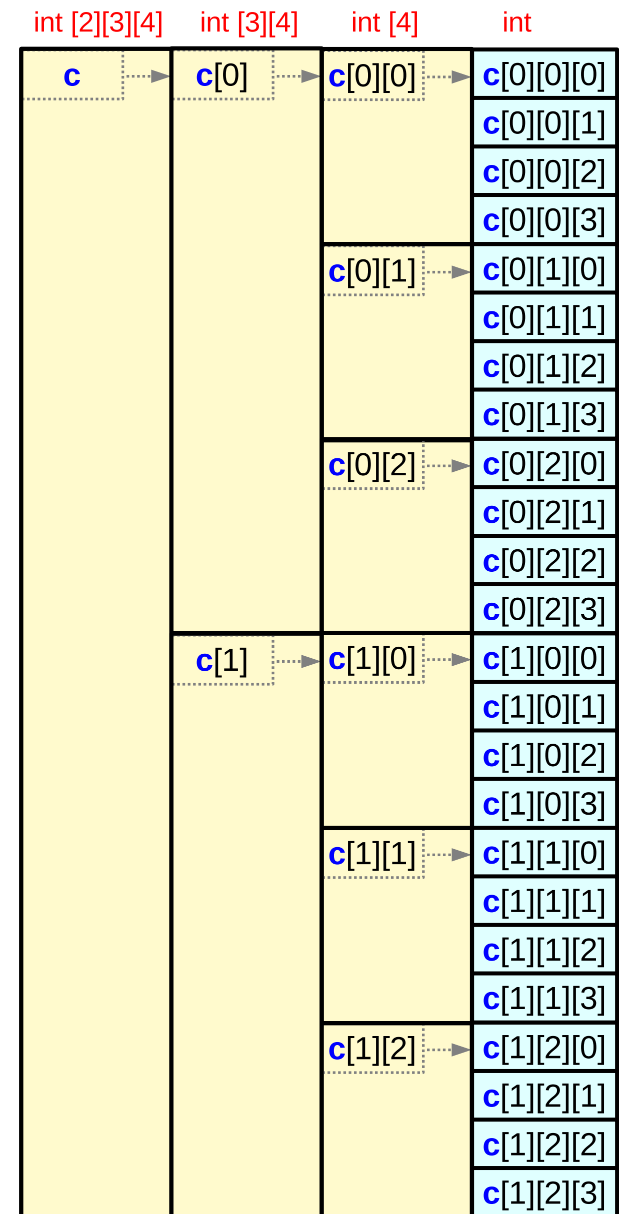
type(c) = int (*)[3][4]
type(c[i]) = int (*)[4]
type(c[i][j]) = int (*)

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



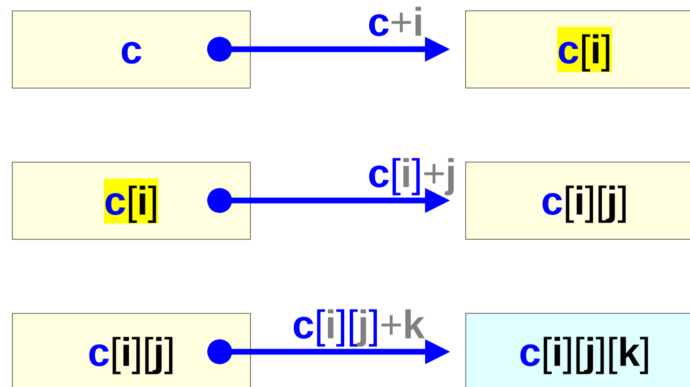
value(c) = &c[0][0][0]
value(c[i]) = &c[i][0][0]
value(c[i][j]) = &c[i][j][0]
value(c[i][j][k]) = int data

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



Types, sizes, and values of sub-arrays (3)

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



virtual array pointer

value(**c**)	= &**c**[0][0][0]	type(**c**)	= int (*)[3][4]
value(**c[i]**)	= &**c**[**i**][0][0]	type(**c[i]**)	= int (*)[4]
value(**c[i][j]**)	= &**c**[**i**][**j**][0]	type(**c[i][j]**)	= int (*)

abstract array data

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

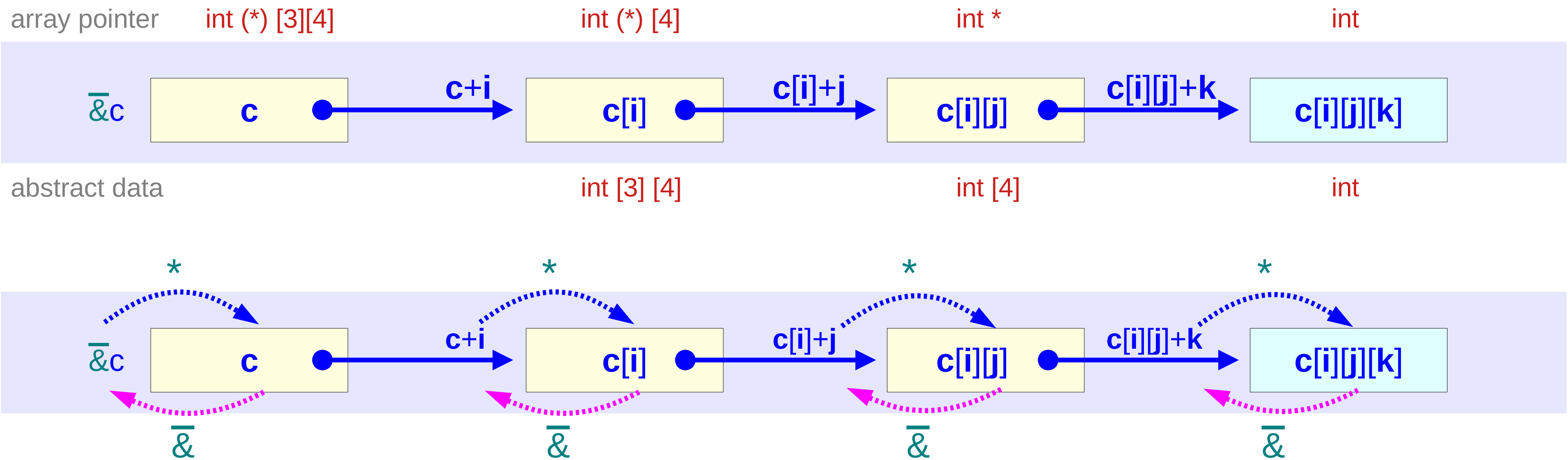
Types of sub-arrays (3)

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

c		:: int [2] [3][4]

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

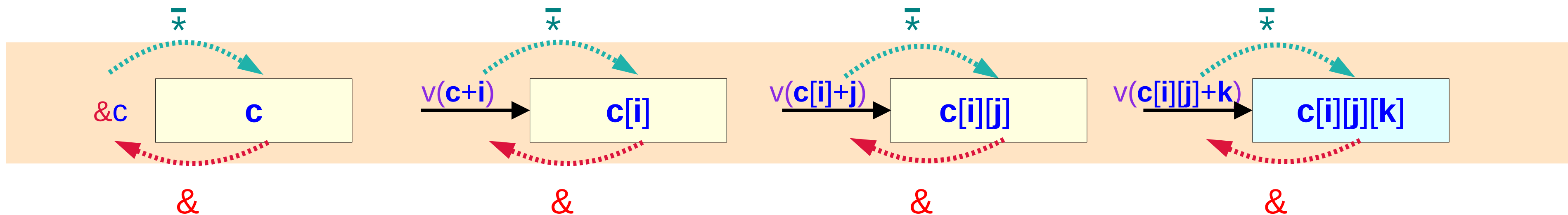
int c[2][3][4];

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

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

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

mathematical expressions obtained by mathematical operator $\bar{\&}$



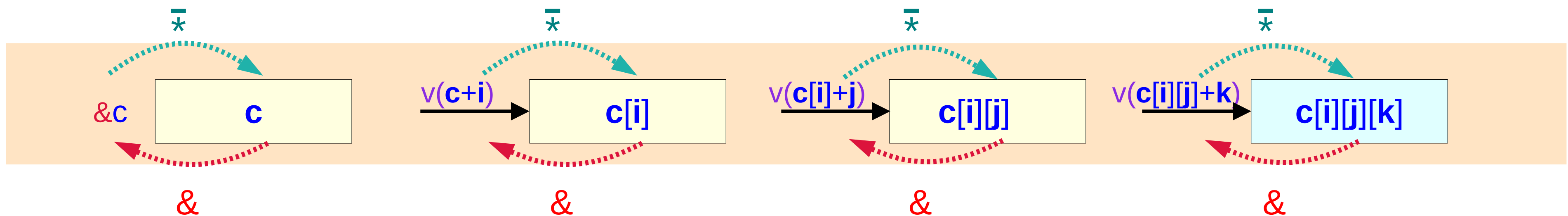
Dimensional Parentheses (2)

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

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

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

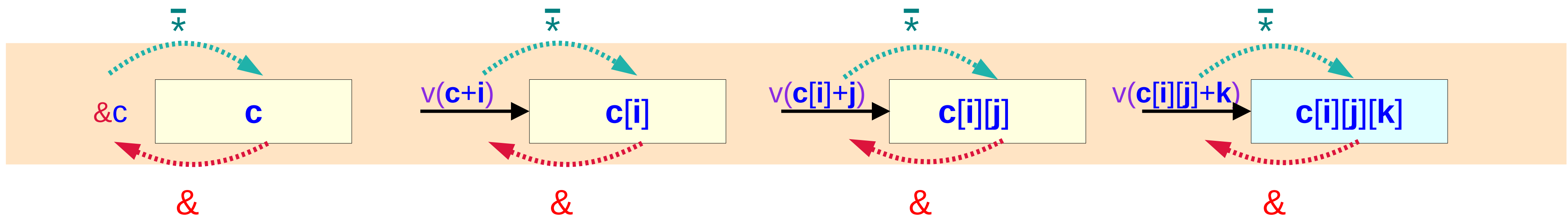
&c[i]	= v(c[i])
&c[i][j]	= v(c[i][j])
&c[i][j][k]	= v(c[i][j][k])

equivalence

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

address replication

&c[i][j][k] ≠ value(c[i][j][k]) — *real*
&c[i][j] = value(c[i][j]) — *virtual*
&c[i] = value(c[i]) — *virtual*



Dimensional Parentheses (4)

Array **Pointer** Approach

skip **i** * sizeof(int [3][4])
= **i** *3*4*4 bytes from **c**

$(c + i)_{3 \cdot 4 \cdot 4}$ ➡ c[i]

skip **j** * sizeof(int [4])
= **j** *4*4 bytes from **c[i]**

$(c[i] + j)_{4 \cdot 4}$ ➡ c[i][j]

skip **k** * sizeof(int)
= **k** *4 bytes from **c[i][j]**

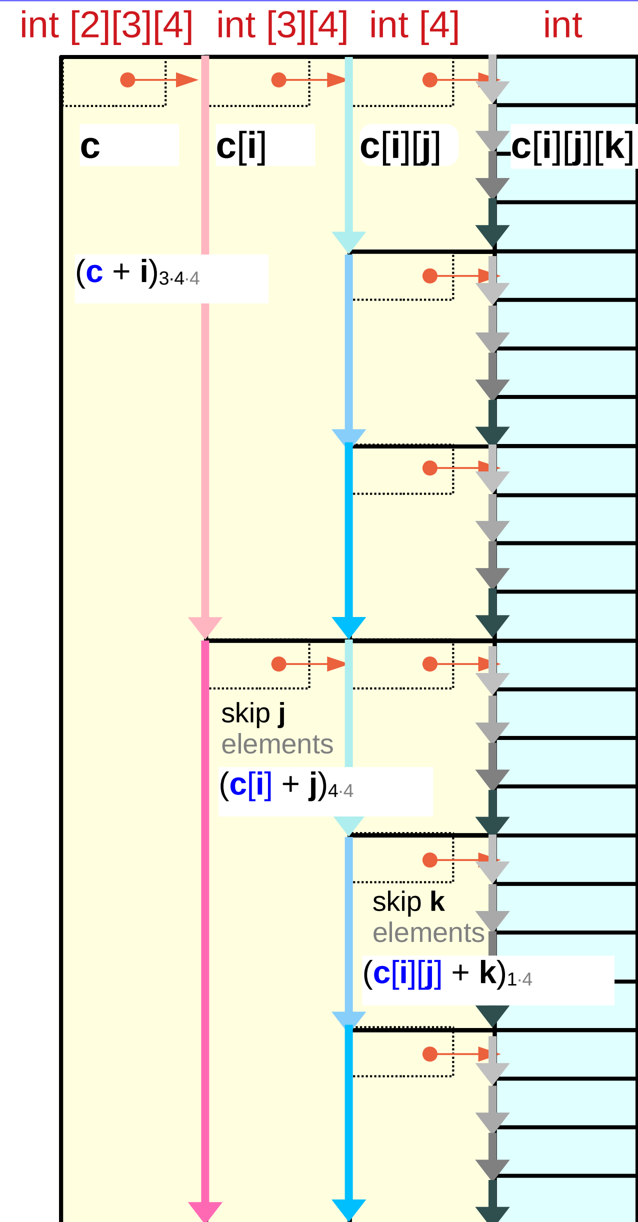
$(c[i][j] + k)_{1 \cdot 4}$ ➡ c[i][j][k]

- ***subarray partitioning***
- ***address replication***

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

size information

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



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

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

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

v(x + y) = v(v(x) + v(y))

Equivalence Relationship

Address Replication

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

v(x + y) = v(v(x) + v(y))

Equivalence Relationship

Address Replication

v(v(v(c+i)+j)+k) ⟸ v(c+i) ⟸ &c[i]

v(x + y) = v(v(x) + v(y))

*(c[i][j]+k) ≡ c[i][j][k]

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

&c[i][j] = v(c[i][j]) = &c[i][j][0]

v(c[i]+j)

*(c[i]+j) ≡ c[i][j]

*v(c[i]+j)

&c[i] = v(c[i]) = &c[i][0])

v(c+i)

$v(((c+i)+j)+k) = v(c) + i \cdot sizeof(3 \cdot 4) + j \cdot sizeof(4) + k$

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

static allocation

$$
\begin{aligned}
v(((c+i)+j)+k) &= v(v(v(c + i) + j) + k) \\
&= (v(v(c + i) + j) + k) \\
&= ((v(c + i) + j*sizeof(4)) + k) \\
&= (((v(c) + i*sizeof(3*4)) + j*sizeof(4)) + k) \\
&= v(c) + i*sizeof(3*4) + j*sizeof(4) + k
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
v(c+i) &= v(c) + i * sizeof(3*4) \\
v((c+i)+j) &= v(c) + i * sizeof(3*4) + j * sizeof(4) \\
v(((c+i)+j)+k) &= v(c) + i * sizeof(3*4) + j * sizeof(4) + k
\end{aligned}
$$

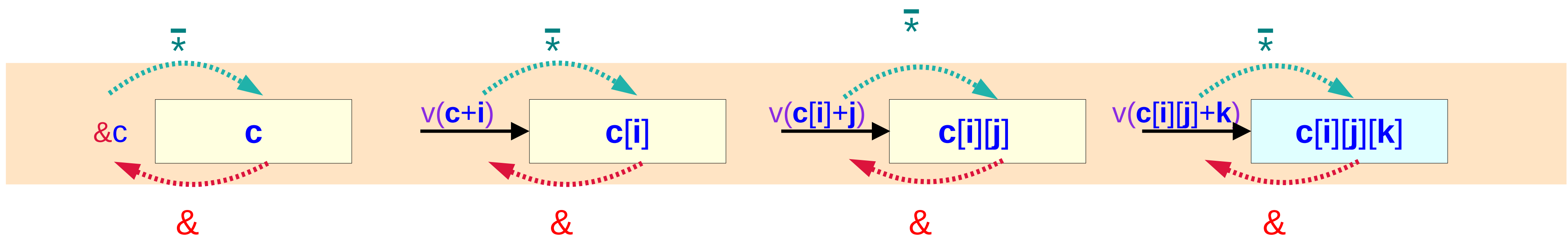
&c[i][j][k] = v(((c+i)+j)+k) = v(v(v(c+i)+j)+k)

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

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

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



Types, sizes, and values of sub-arrays

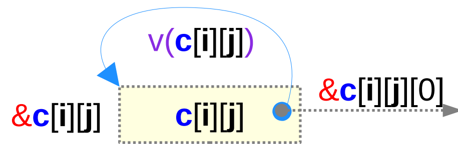
int c [2][3][4] ;   **static allocation**

reduced information :
value only
Address Replication

&c[i][j] = v(c[i][j]) = &c[i][j][0]

full information :
value and type

c[i][j] = *v(c[i][j]) ≠ c[i][j][0]



c[i][j] = *v(c[i][j]) = *v(c[i][j]+0) = c[i][j][0]   type mismatch

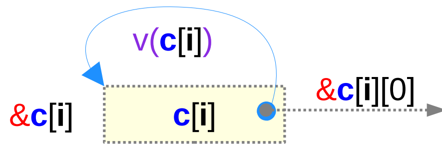
address replication must be performed as the final step

reduced information :
value only
Address Replication

&c[i] = v(c[i]) = &c[i][0]

full information :
value and type

c[i] = *v(c[i]) ≠ c[i][0)



c[i] = *v(c[i]) = *v(c[i]+0) = c[i][0]   type mismatch

address replication must be performed as the final step

Types, sizes, and values of sub-arrays

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

&c[i][j] = v(c[i][j]) = &c[i][j][0]

&c[i][j] = v(c[i][j])   c[i][j] = *v(c[i][j]) = *v(c[i][j]+0) = ~~c[i][j][0]~~   type mismatch   *address replication must be performed as the final step*

Address Replication   applying * (Equivalence)

&c[i] = v(c[i]) = &c[i][0]

&c[i] = v(c[i])   c[i] = *v(c[i]) = *v(c[i]+0) = ~~c[i][0]~~   type mismatch   *address replication must be performed as the final step*

Address Replication   applying * (Equivalence)

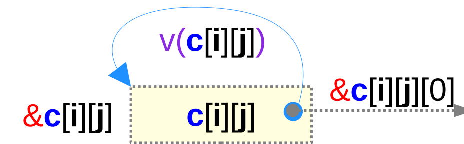
Types, sizes, and values of sub-arrays

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]

Address Replication

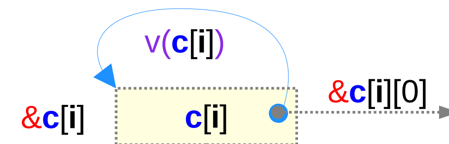
&**c**[**i**][**j**] = v(**c**[**i**][**j**]) = &**c**[**i**][**j**][0]



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

Address Replication

&**c**[**i**] = v(**c**[**i**]) = &**c**[**i**][0])



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]

Address Replications (4)

int (*)[3][4] → int (*)[4] → int (*) → int

int [2][3][4] → int [3][4] → int [4] → int

&c v(c[0]) c → &c[0] v(c[0]) c[0] → &c[0][0] v(c[0][0]) c[0][0] → &c[0][0][0] c[0][0][0]

&c[0][1] v(c[0][1]) c[0][1] → &c[0][1][0] c[0][0][0]

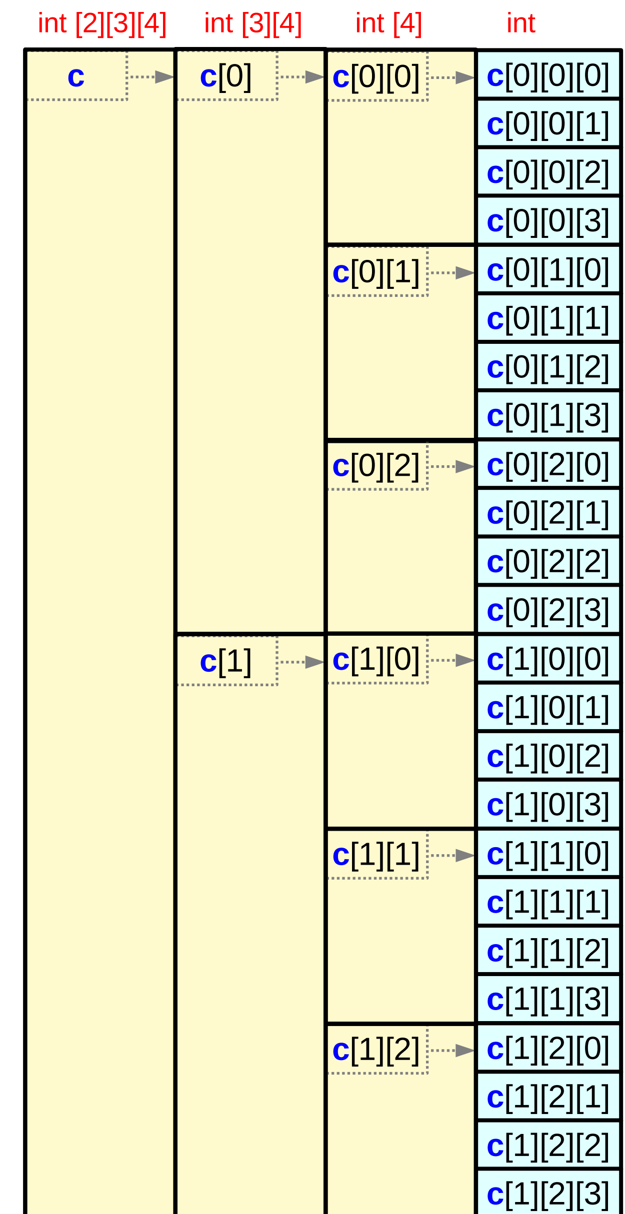
&c[0][2] v(c[0][2]) c[0][2] → &c[0][2][0] c[0][0][0]

&c[1] v(c[1]) c[1] → &c[1][0] v(c[1][0]) c[1][0] → &c[1][0][0] c[0][0][0]

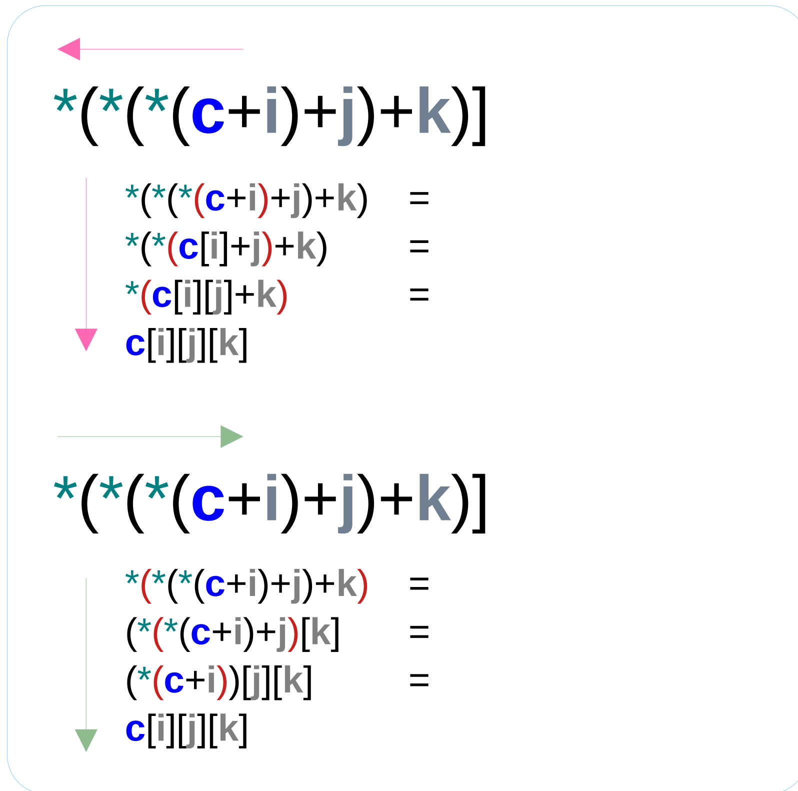
&c[1][1] v(c[1][1]) c[1][1] → &c[1][1][0] c[0][0][0]

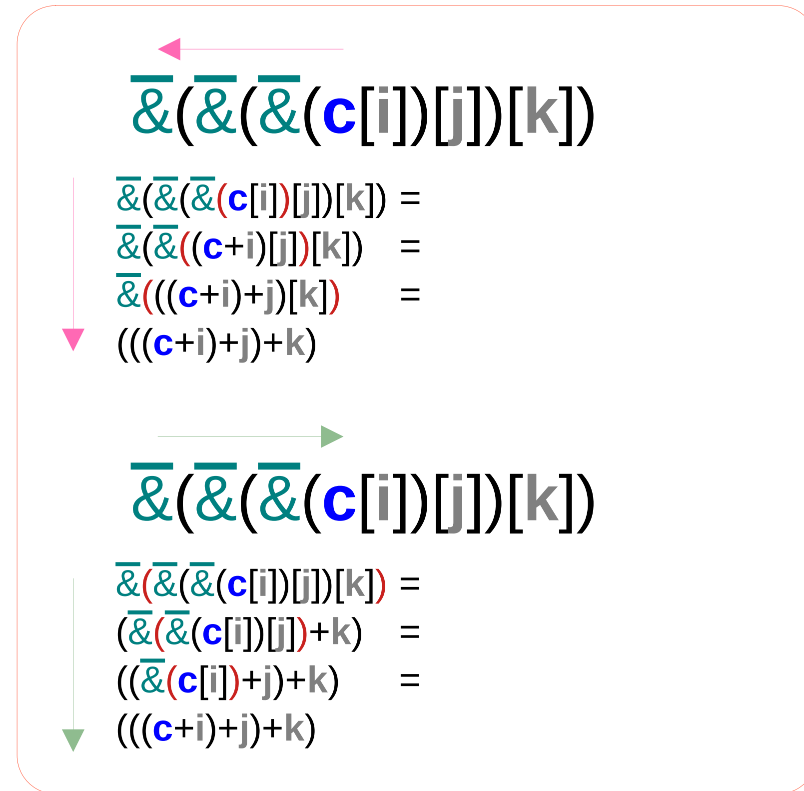
&c[1][2] v(c[1][2]) c[1][2] → &c[1][2][0] c[i][j][0]

Address Replication

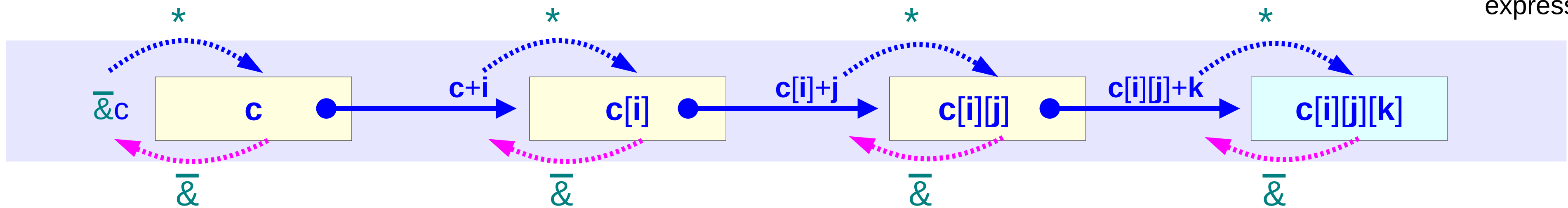


Outward and Inward Recursive applications of * and &





mathematical expressions



Recursive applications of $\bar{*}$ and & are <u>not</u> possible (2)

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

not allowed
even in mathematical expressions

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

not allowed
in C expressions

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

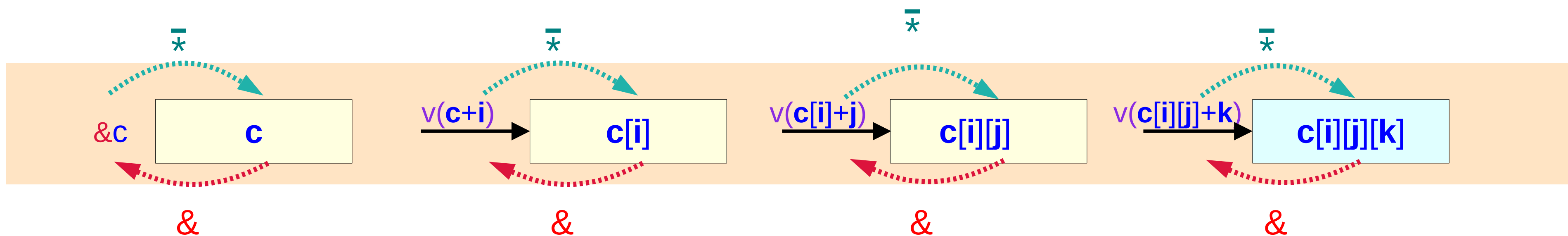
allowed
in mathematical expressions

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

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

allowed
in mathematical expressions

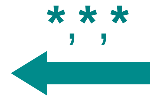
$\overline{\&}(\overline{\&}(\overline{\&}(c[i])[j])[k])$



Recursive applications of * and $\overline{\&}$

legal C exp

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



legal math. exp

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

||

legal C exp

c[i][j][k]

||

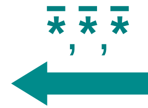
legal math. exp

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

Recursive applications of $\bar{*}$ and &

legal math exp

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



legal math. exp

v(v(v(c+i)+j)+k)

||

legal C exp

c[i][j][k]

||

legal math. exp

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

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

equivalence

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

address replication

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

v(c+i)	= &c[i]	= v(c[i])
v(c[i]+j)	= &c[i][j]	= v(c[i][j])
v(c[i][j]+k)	= &c[i][j][k]	= v(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 $\bar{*}$ and & are not possible (3)

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

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

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

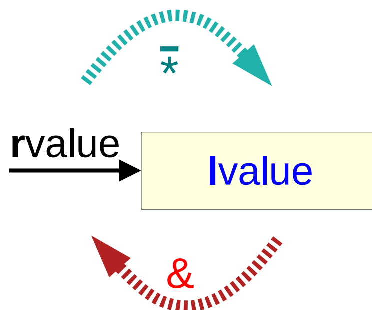
successive application of **&** is not possible,
unless intermediate variables and proper
type casting

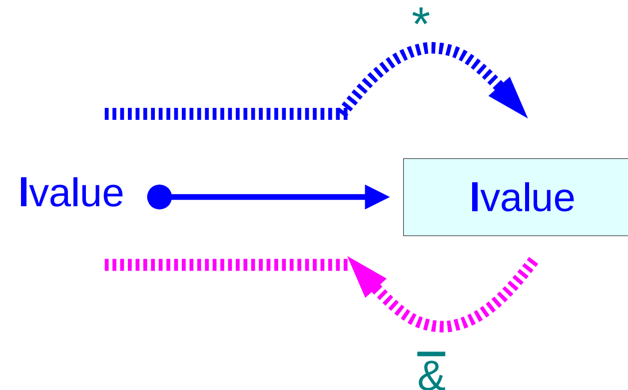
$\bar{*}$ must be applied to an **rvalue** address
value, $\bar{*}$ operator cannot be
applied successively.

$\bar{\&}$ 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.





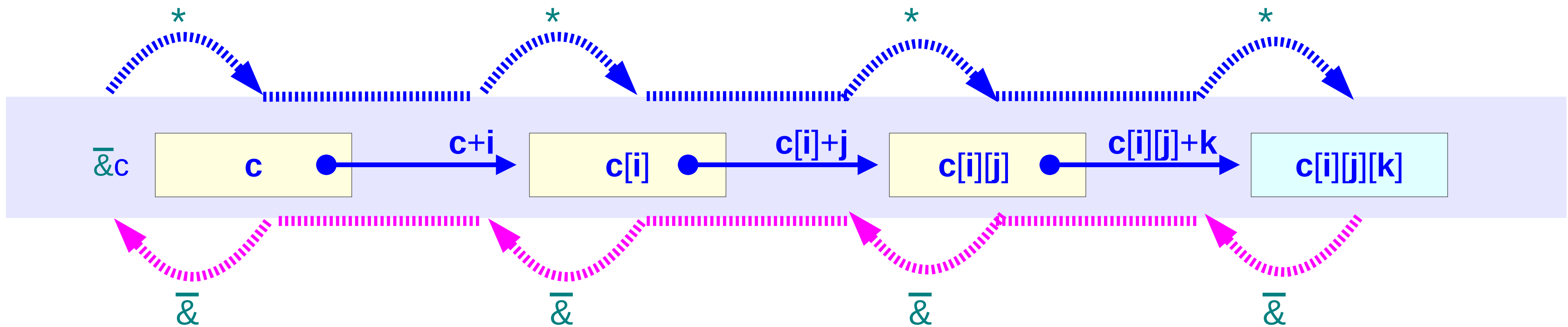
Inverse Relations between * and $\overline{\&}$

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)

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

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



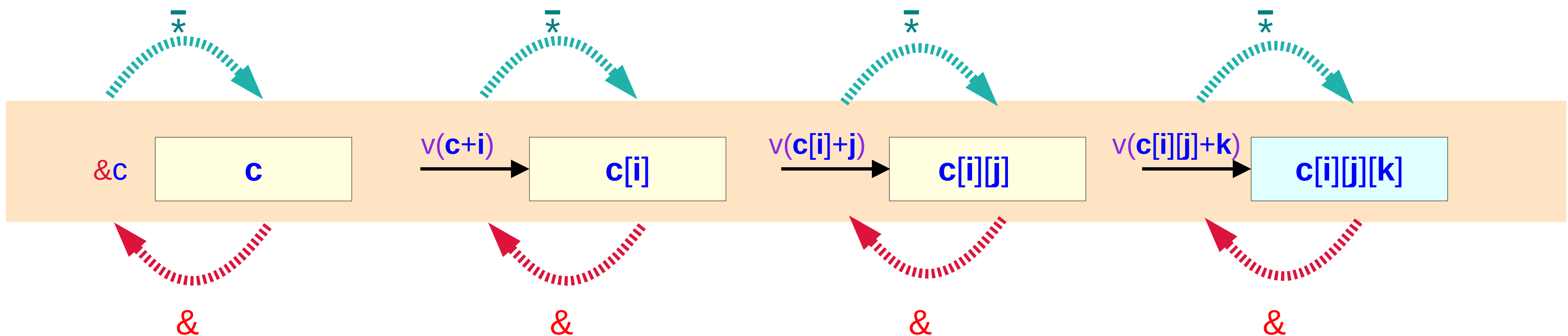
Inverse Relations between $\bar{*}$ and &

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 $\overline{*}$ and &

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

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

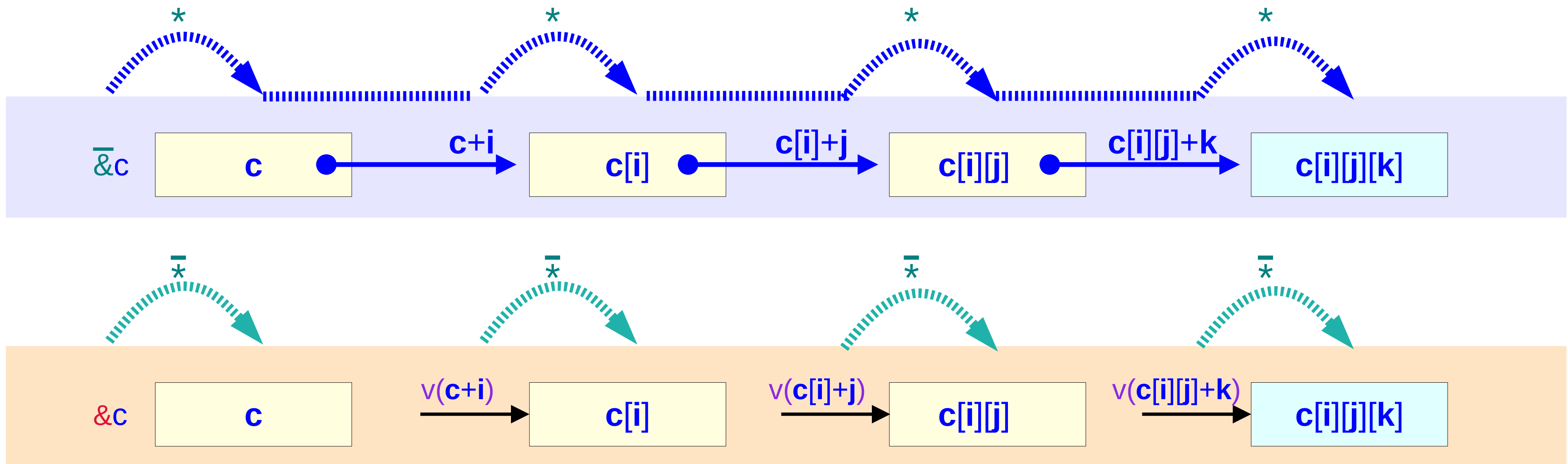
&(c[i][j][k]) = v(c[i][j] + k)
&(c[i][j]) = v(c[i] + j)
&(c[i]) = v(c + i)

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]

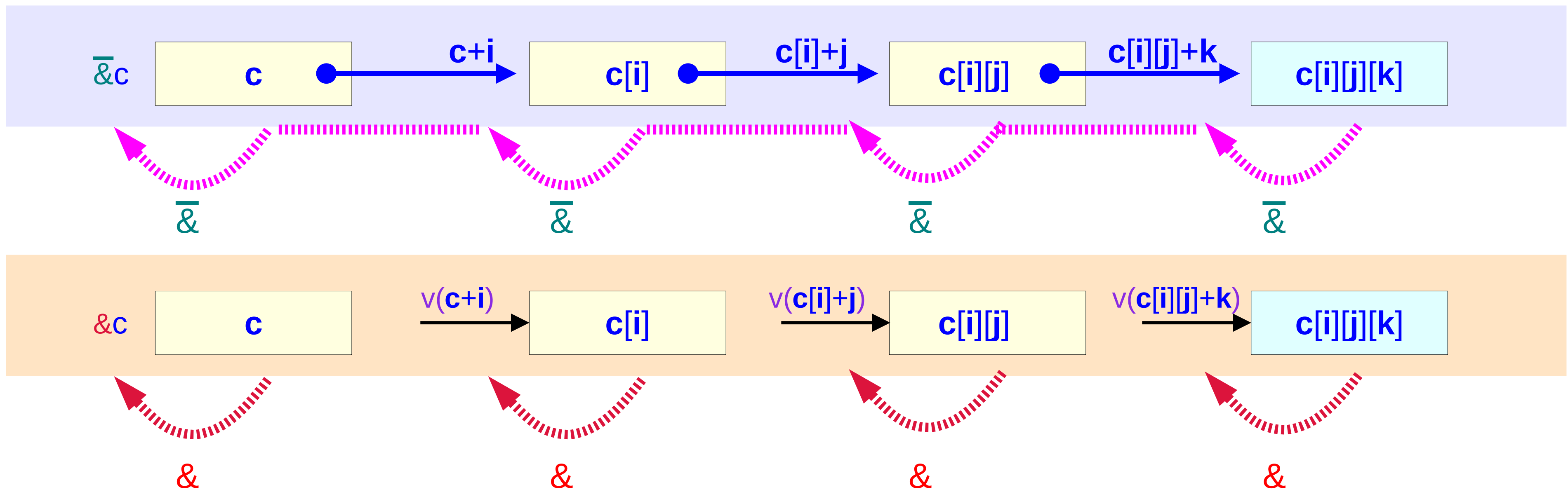
c[i][j][k] = *(c[i][j] + k)
c[i][j] = *(c[i] + j)
c[i] = *(c + i)

c[i][j][k] = $\bar{*}$v(c[i][j] + k) = $\bar{*}$(value(c[i][j]) + k * sizeof(c[0][0][0]))
c[i][j] = $\bar{*}$v(c[i] + j) = $\bar{*}$(value(c[i]) + j * sizeof(c[0][0]))
c[i] = $\bar{*}$v(c + i) = $\bar{*}$(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

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

*$\overline{\&}$(c[i][j][k]) = *$\overline{\&}$(*(c[i][j]+k)) = *(c[i][j] +k)
*$\overline{\&}$(c[i][j]) = *$\overline{\&}$(*(c[i]+j)) = *(c[i] +j)
*$\overline{\&}$(c[i]) = *$\overline{\&}$(*(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] = $\overline{*}$&(*(c[i][j]+k)) = $\overline{*}$(c[i][j] + k * sizeof(c[0][0][0])
c[i][j] = $\overline{*}$&(*(c[i]+j)) = $\overline{*}$(c[i] + j * sizeof(c[0][0]))
c[i] = $\overline{*}$&(*(c+i)) = $\overline{*}$(c + i * sizeof(c[0]))

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

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

c[i][j][k] = *(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] = $\overline{*}$(c[i][j] + k * sizeof(c[0][0][0])
c[i][j] = $\overline{*}$(c[i] + j * sizeof(c[0][0]))
c[i] = $\overline{*}$(c + i * sizeof(c[0]))

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

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

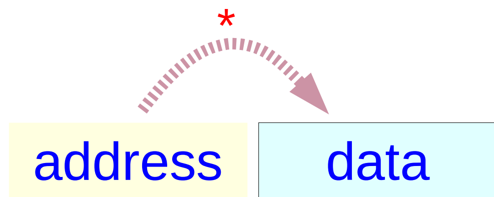
c[i][j][k] = *(c[i][j] +k)
c[i][j] = *(c[i] +j)
c[i] = *(c +i)

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

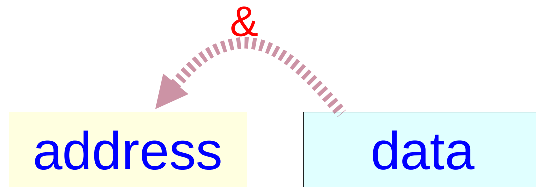
c[i][j][k] = $\overline{*}$ value(c[i][j] + k)
c[i][j] = $\overline{*}$ value(c[i] + j)
c[i] = $\overline{*}$ value(c + i)

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

Primitive Data Type

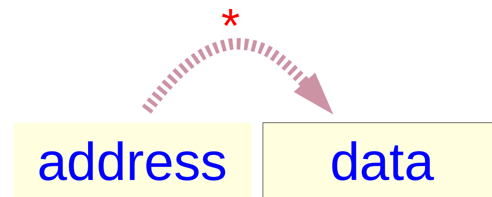


rvalue &a ⇨(*) *lvalue* a
constant *variable*
lvalue p ⇨(*) *lvalue* *p
pointer *variable*

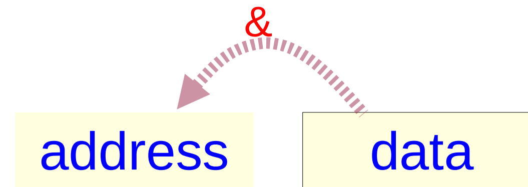


rvalue &a ⇦(&) *lvalue* a
constant *variable*
~~*lvalue*~~ p ⇦(&) *lvalue* *p
pointer *variable*

Pointer Data Type

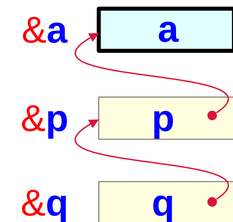


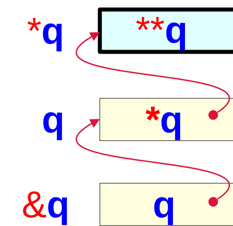
rvalue &q ⇨(*) *lvalue* q
constant *double pointer*
lvalue q ⇨(*) *lvalue* *q
double pointer *pointer*



rvalue &q ⇦(&) *lvalue* q
constant *double pointer*
~~*lvalue*~~ q ⇦(&) *lvalue* *q
double pointer *pointer*

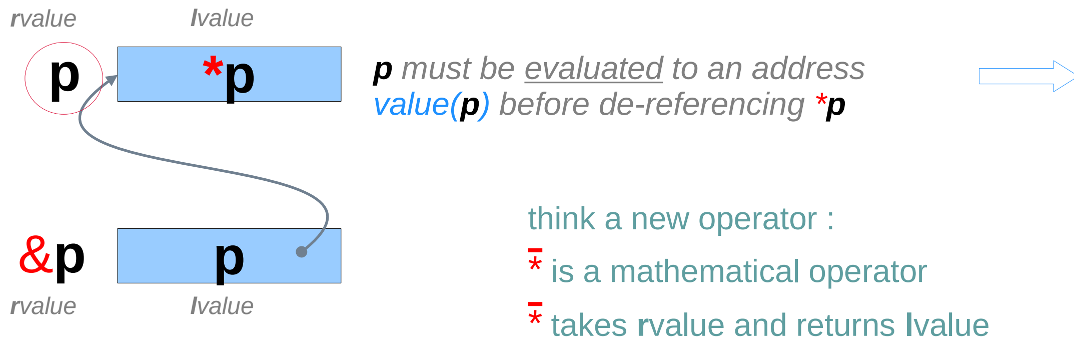
```
int    a ;
int *  p ;
int ** q ;
```

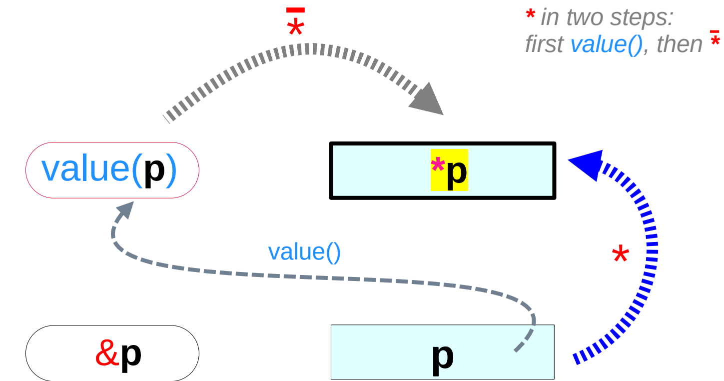




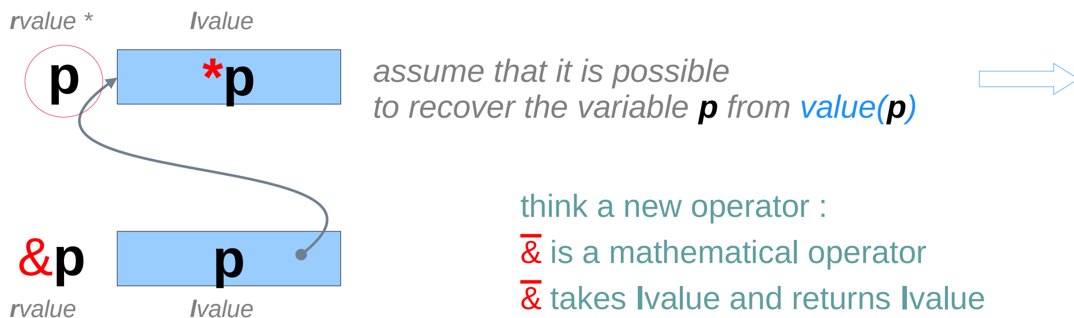
& and * operators in pointer de-referecing

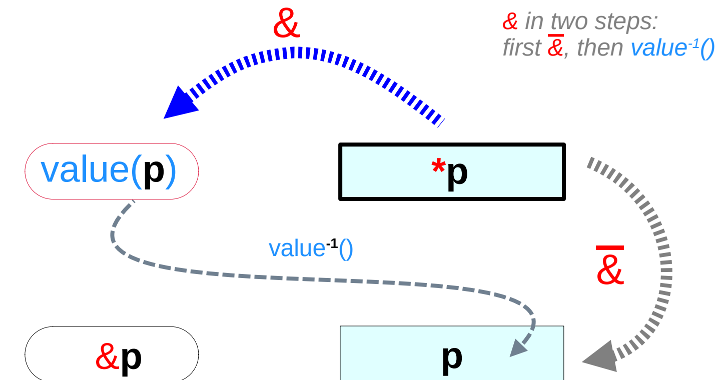
Two step De-reference * operation



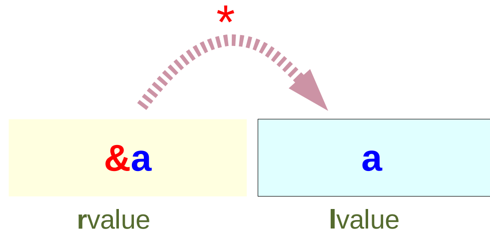


Two step Address-of & operation



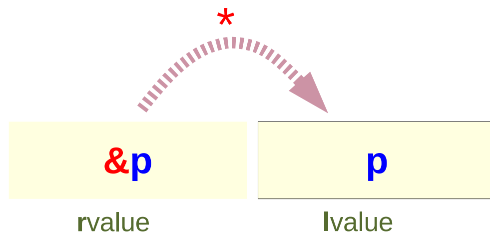


Type, Size, and Value attributes of an **lvalue**



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

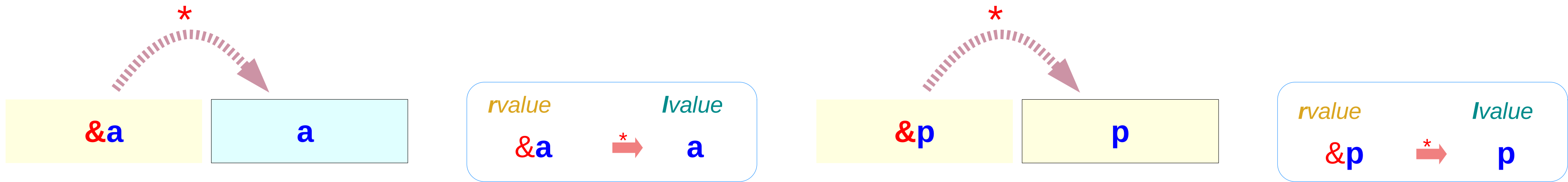
- Value

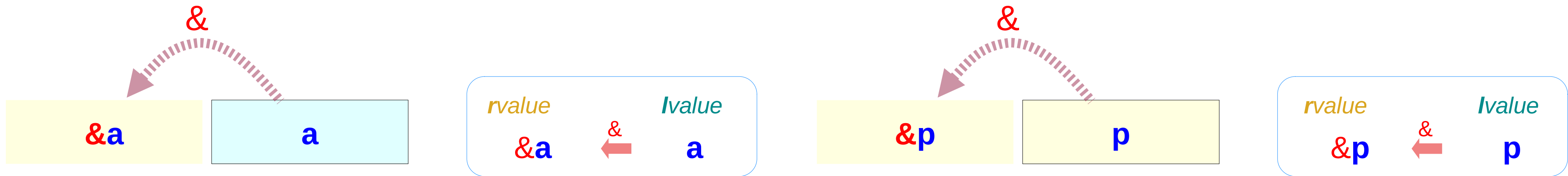
assume the function value()

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

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

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

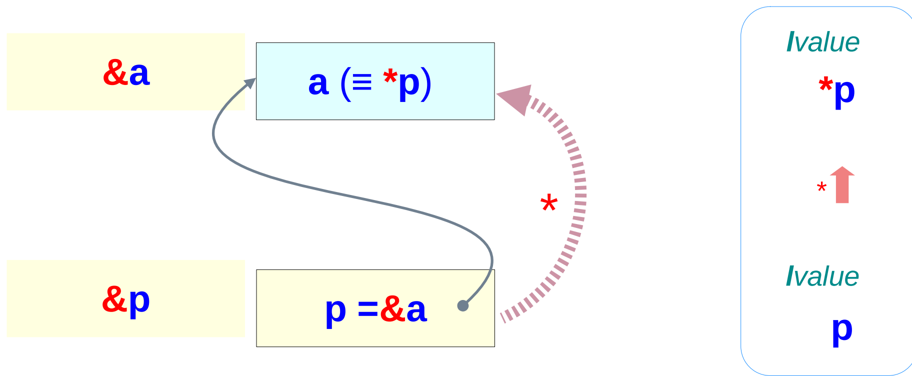


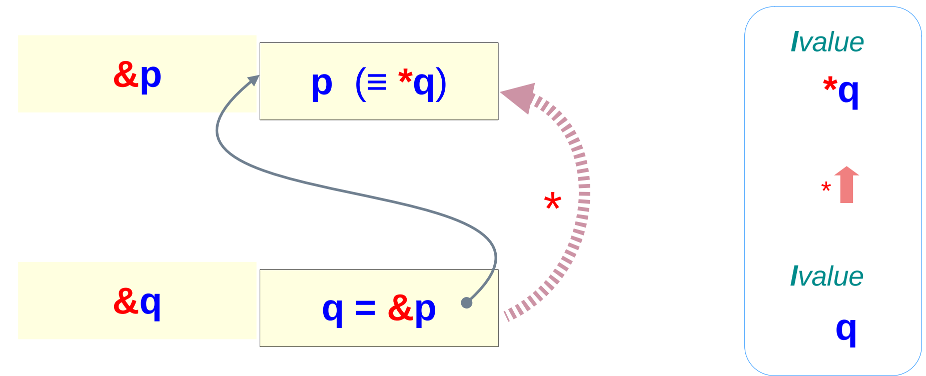


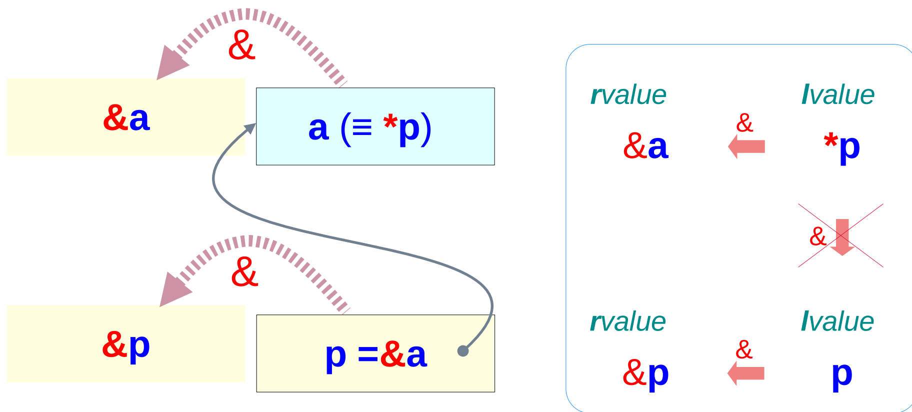
*&a = a

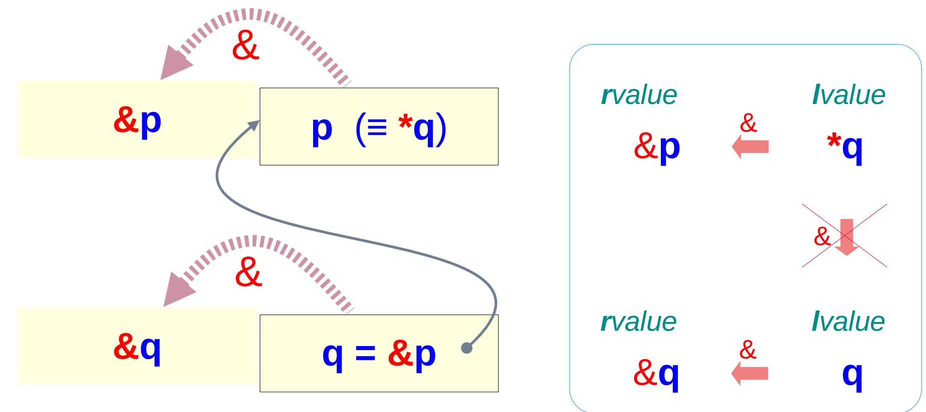
*&p = p

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









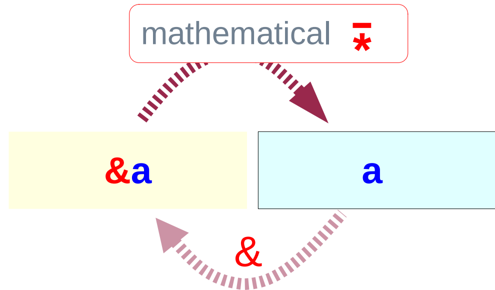
*&p = p

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

*&q = q

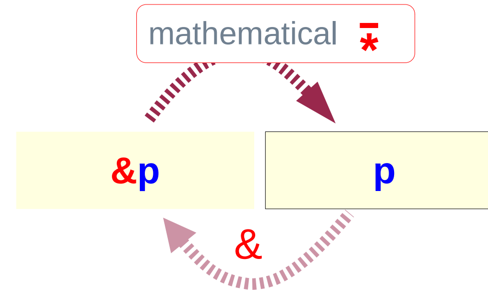
~~&*q = q~~ value(q)

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



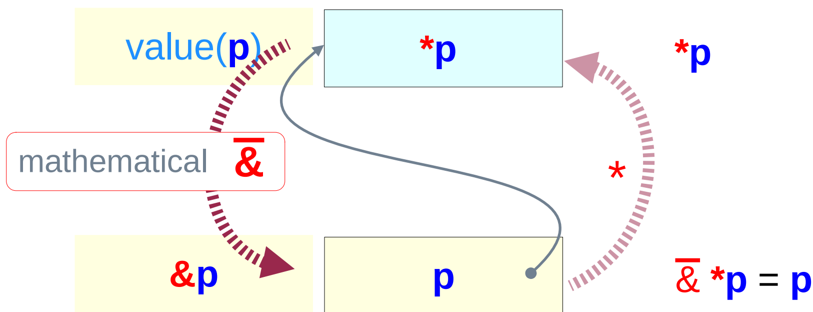
$\overline{*}$ &a = a

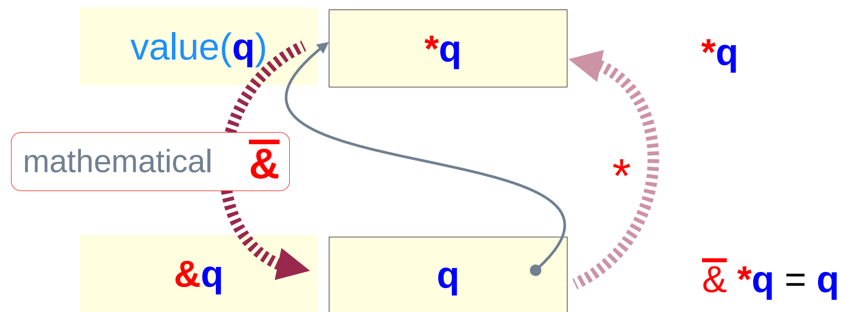
&a



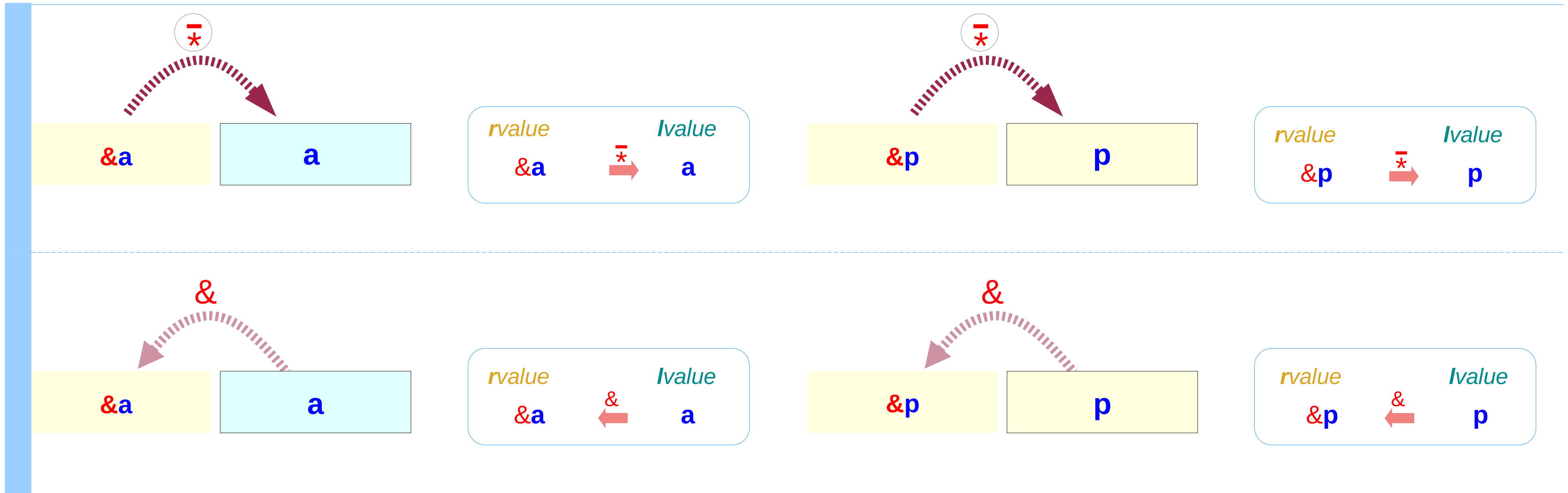
$\overline{*}$ &p = p

&p





Inverse operators * and &



*&a = a

&*&a = &a

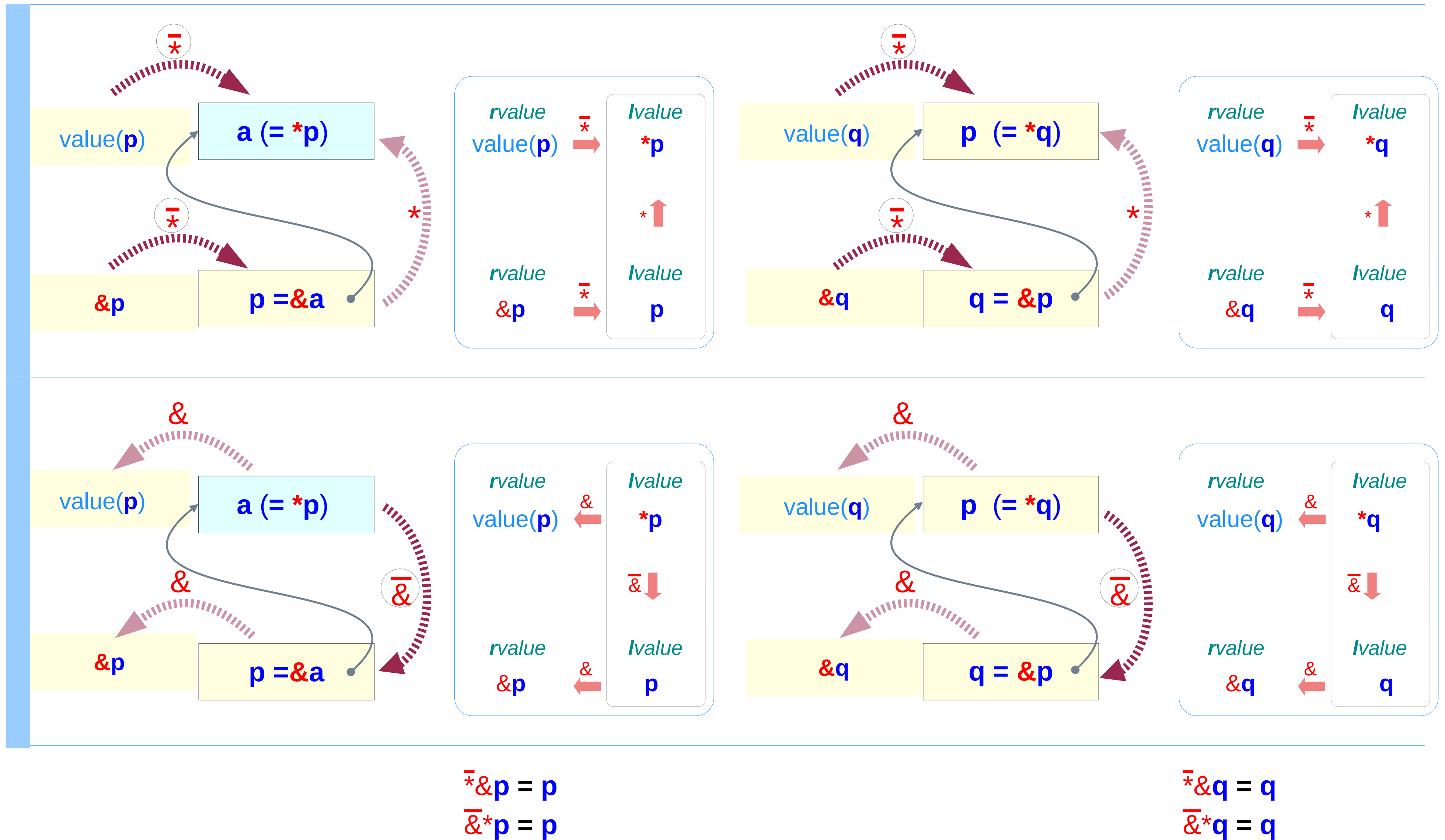
* and & are
inverse operators
to each other

*&p = p

&*&p = &p

* and & are
inverse operators
to each other

Inverse operators &̄ and *

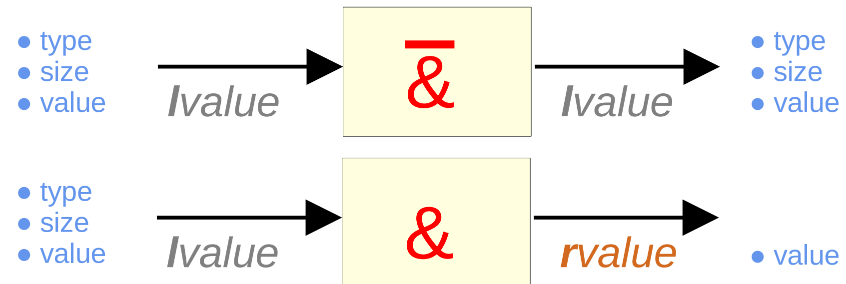


*̄&p = p
&̄*p = p

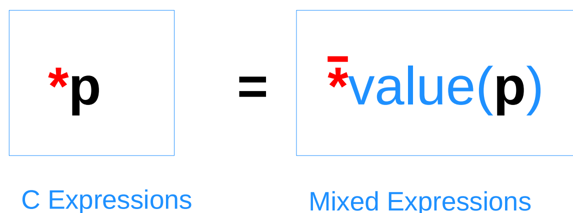
*̄&q = q
&̄*q = q

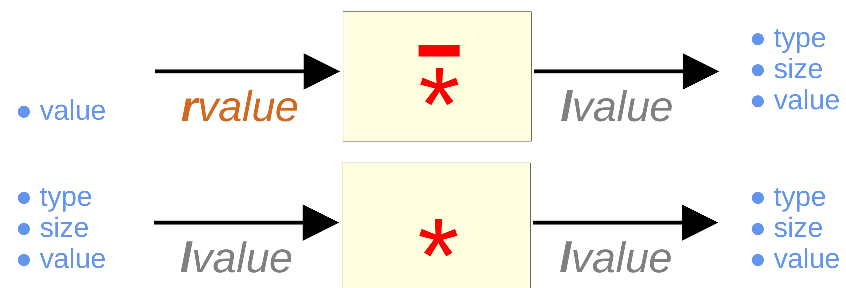
C operators and mathematical operators

Address-of operation

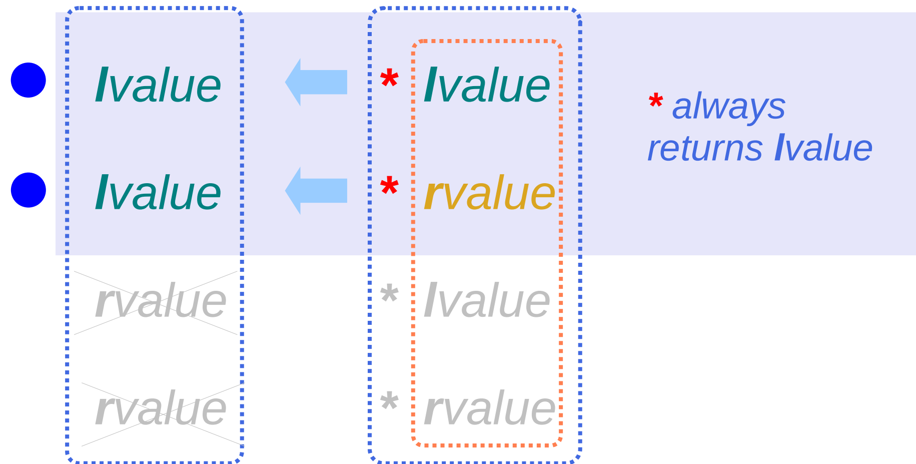


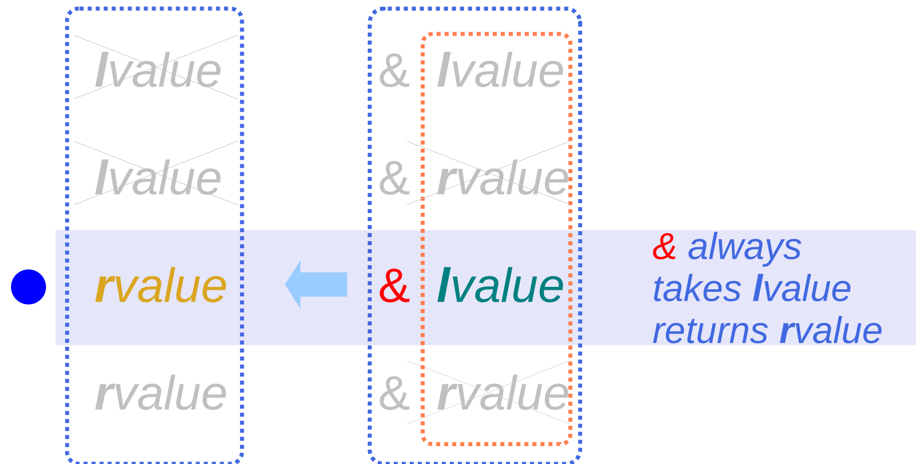
De-reference operation

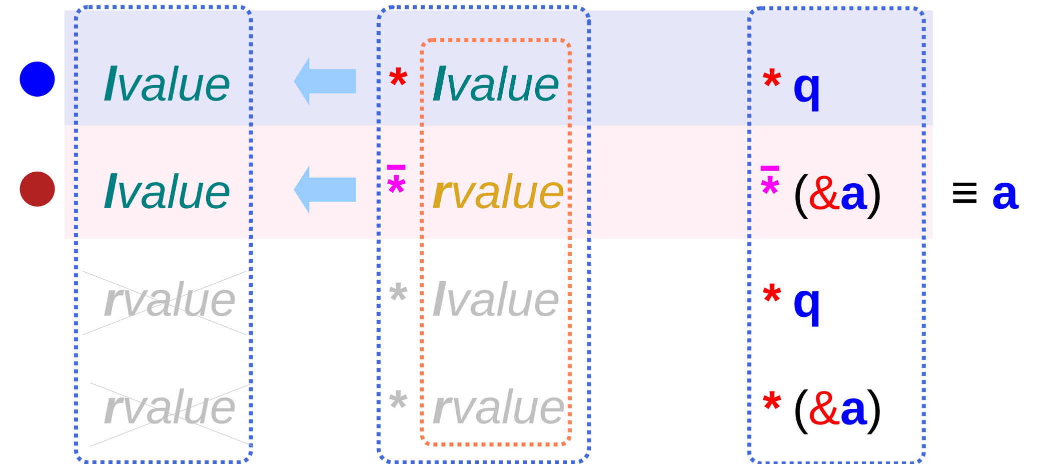


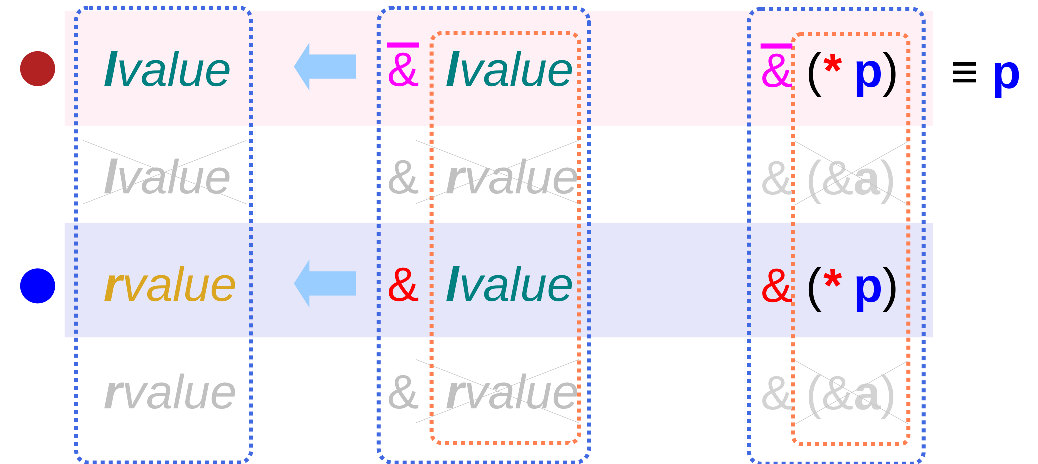


lvalue and rvalue with * and & operators







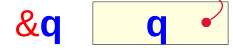


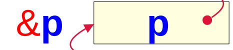
a, p, q	: lvalues	… variables	… RW
*p, *q, **q	: lvalues	… variables	… RW
&a, &p, &q	: rvalues	… constants	… RO

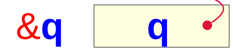
Examples of inverse operators

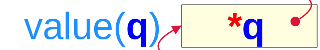
Extended Operators	C Operators
$\bar{*}$&a ➡ a	*&a ➡ a
$\bar{*}$&p ➡ p	*&p ➡ p
$\bar{*}$&q ➡ q	*&q ➡ q
$\bar{*}$value(p) ➡ *p ➡ a	*p ➡ a
$\bar{*}$value(q) ➡ *q ➡ p	*q ➡ p
$\bar{*}$value(*q) ➡ **q ➡ *p ➡ a	**q ➡ *p ➡ a
$\bar{\&}$*p ➡ p	&*p ➡ &a
$\bar{\&}$*q ➡ q	&*q ➡ &p
$\bar{\&}$**q ➡ *q	&**q ➡ &a

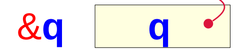
```
int    a ;
int *  p = & a ;
int ** q = & p ;
```











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

$\overline{*}$ address value

$\overline{*}$ & variable

$\overline{\&}$ * pointer

dereferenced pointer

$\overline{\&}$ ~~pointer~~

must be a dereferenced pointer

not considering
simultaneous pointing

& $\overline{*}$ value(**P**) ➡ value(**P**)

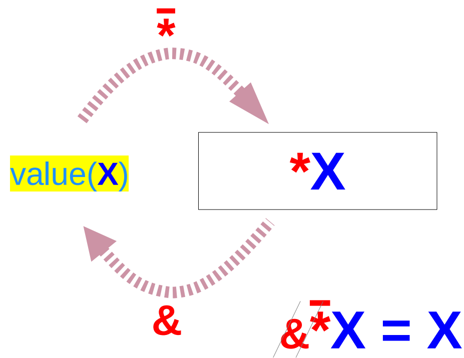
$\overline{*}$ &**X** ➡ **X**

$\overline{\&}$ ***P** ➡ **P**

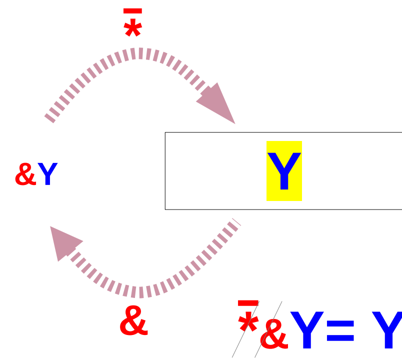
* $\overline{\&}$**Q** ➡ **Q**

Q must be a dereferenced pointer
i.e, **Q = *p** ➡ $\overline{\&}$**Q = p**

& and mathematical *̄

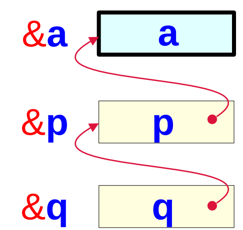


&*̄X = X



*̄&Y= Y

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



& *̄value(a) = value(a)

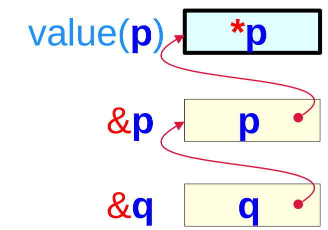
& *̄value(p) = value(p)

& *̄value(q) = value(q)

*̄ &a = a

*̄ &p = p

*̄ &q = q



& *a = value(a)

& *p = value(p)

& *q = value(q)

*̄ &a = a

*̄ &p = p

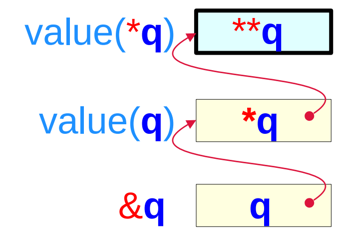
*̄ &q = q

* &a = a

* &p = p

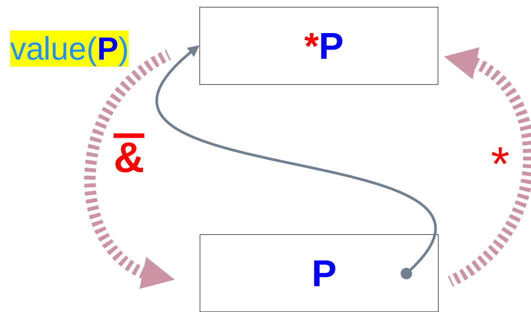
* &q = q

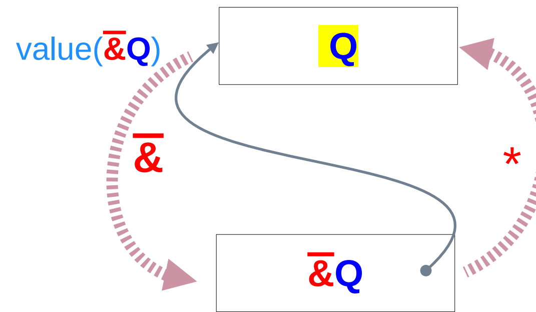
C expressions

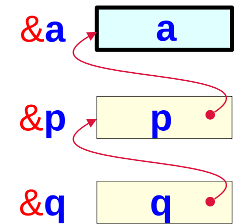


* and mathematical $\overline{\&}$

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



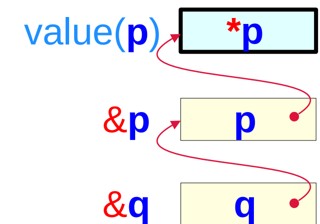




$\overline{\&}$*P = P

if not considering simultaneous pointing

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



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

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

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

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

& *p = value(p)

& *q = value(q)

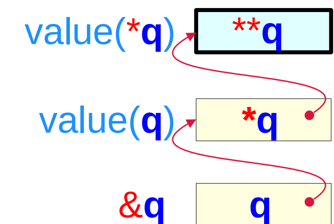
$\overline{*}$ &*p = *p

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

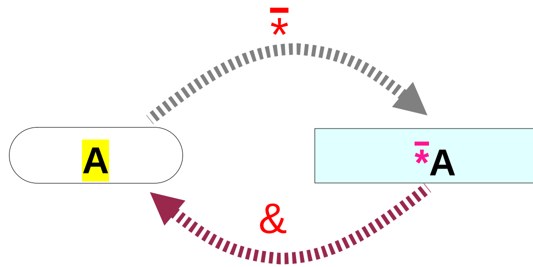
* &*p = *p

* &*q = *q

C operators

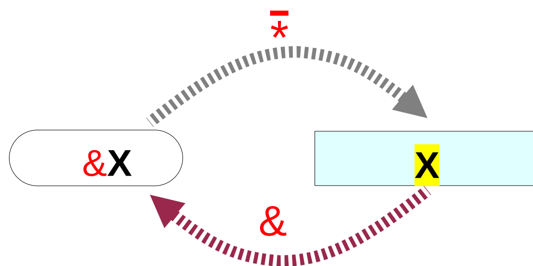


Requirements of address and data



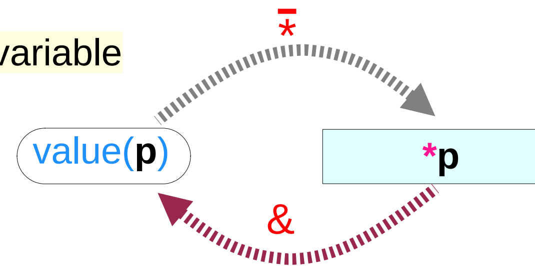
A : an address value - a number

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



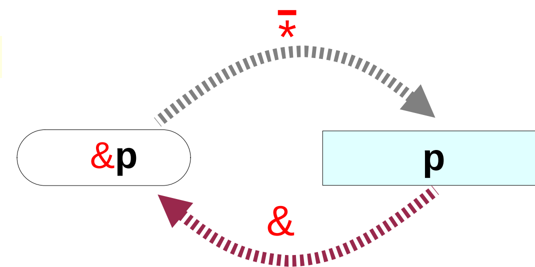
X : a variable *p, p, a

***p :** a dereferenced variable



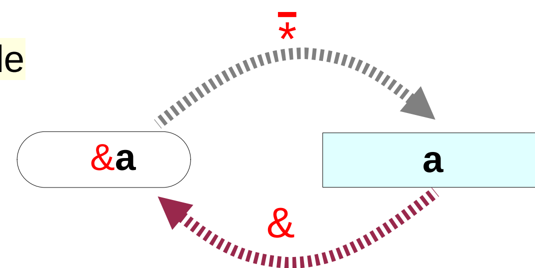
Pointer Dereferencing

p : a pointer variable



Pointer Data

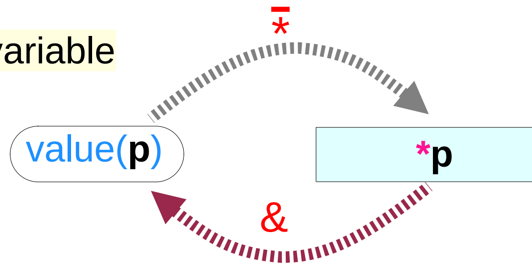
a : a primitive variable



Non-pointer Data

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

***p :** a dereferenced variable

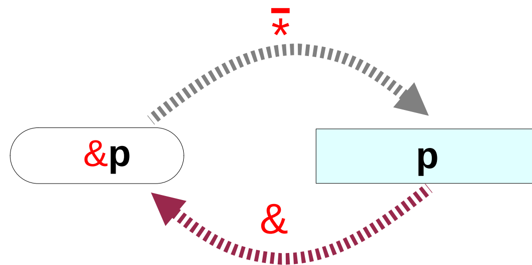


$\bar{*}$ value(p) = *p

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

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

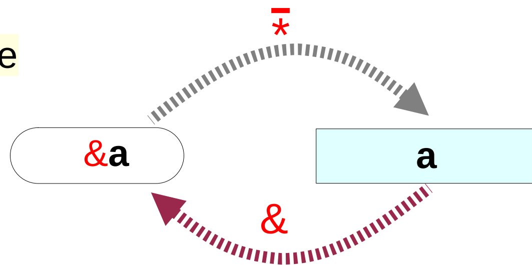
p : a pointer variable



$\bar{*}$ &p = p

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

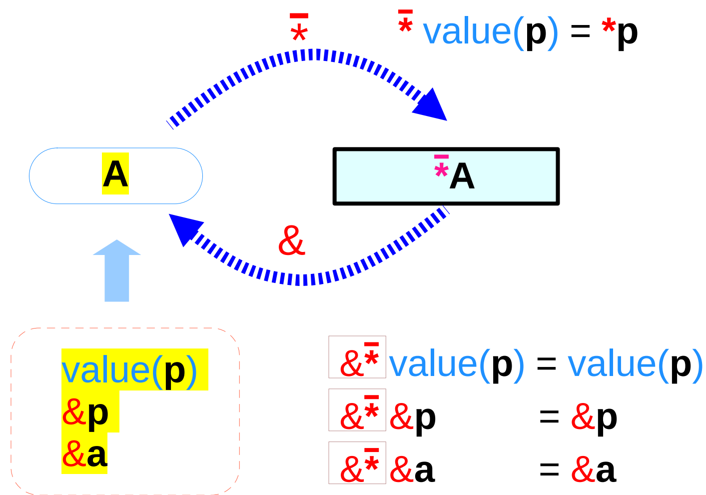
a : a primitive variable

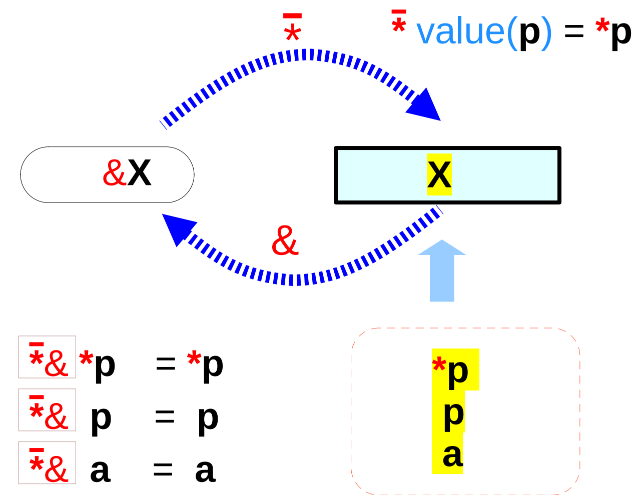


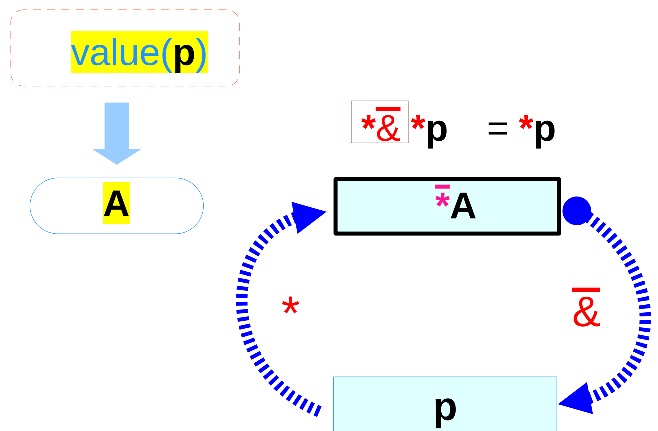
$\bar{*}$ &a = a

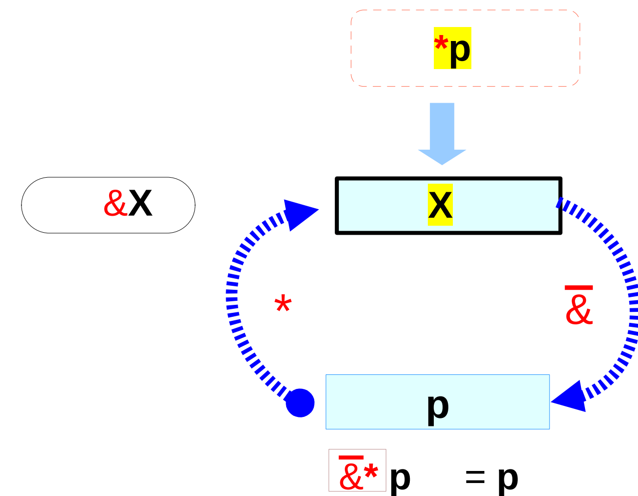
&$\bar{*}$ &a = &a

Requirements of pointed address and data



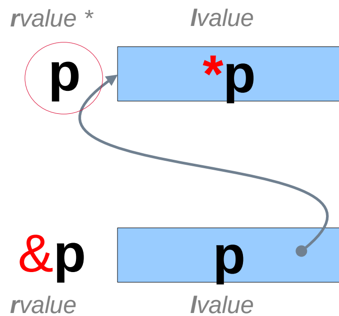




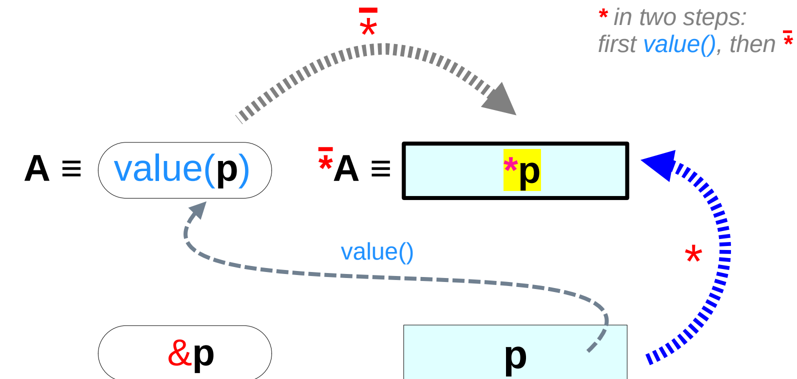


& and * operators in pointer de-referecing (1)

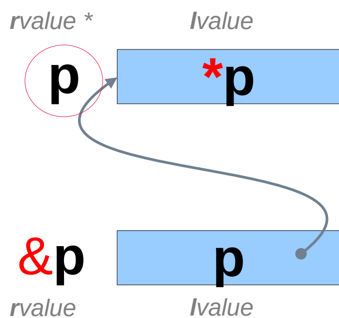
Two step De-reference * operation



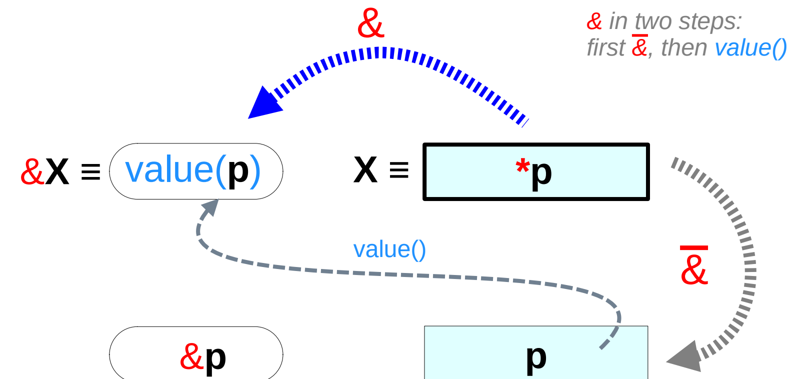
C Expressions		Mixed Expressions
*p	=	$\overline{*}$value(p)
$\overline{*}$A	=	*value^{-1}(A)



Two step Address-of & operation

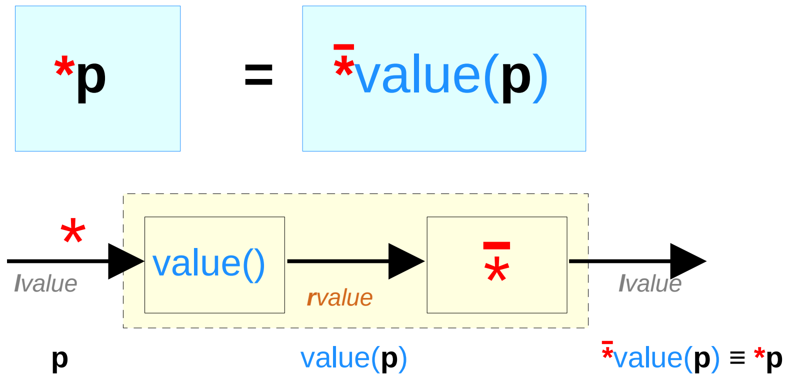


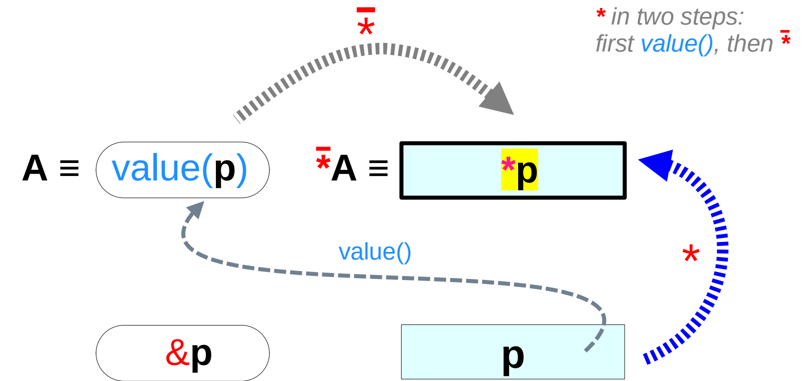
C Expressions		Mixed Expressions
&X	=	value($\overline{\&}$X)
$\overline{\&}$X	=	value^{-1}(&X)



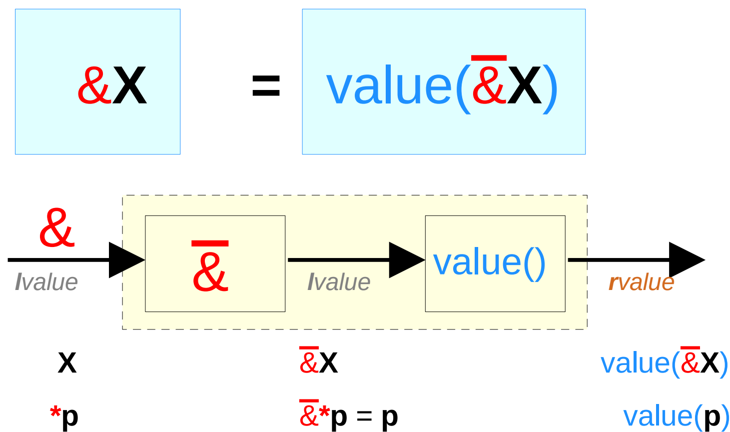
& and * operators in pointer de-referecing (2)

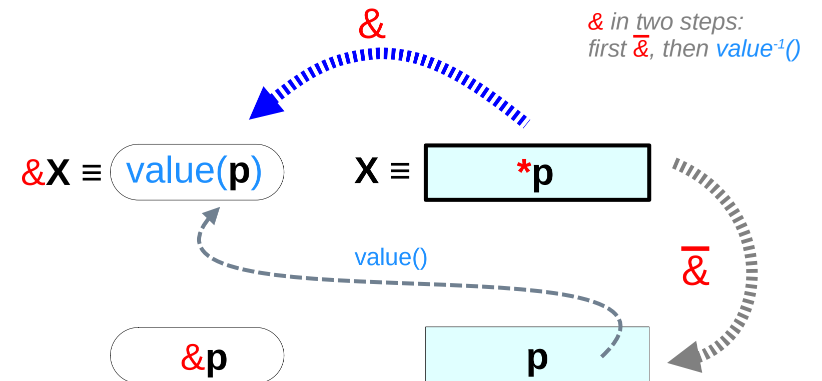
Two step De-reference * operation



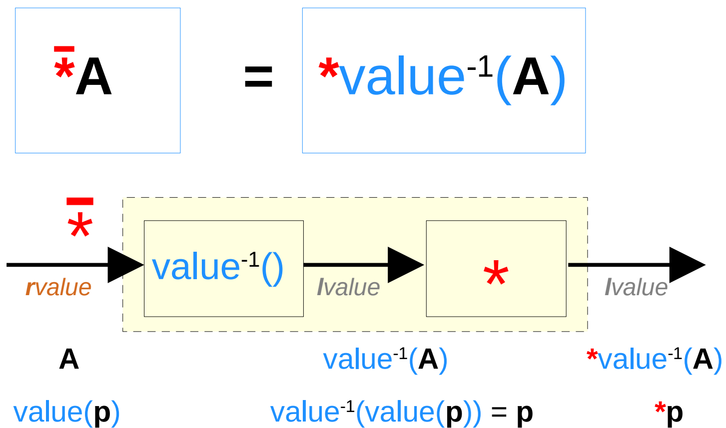


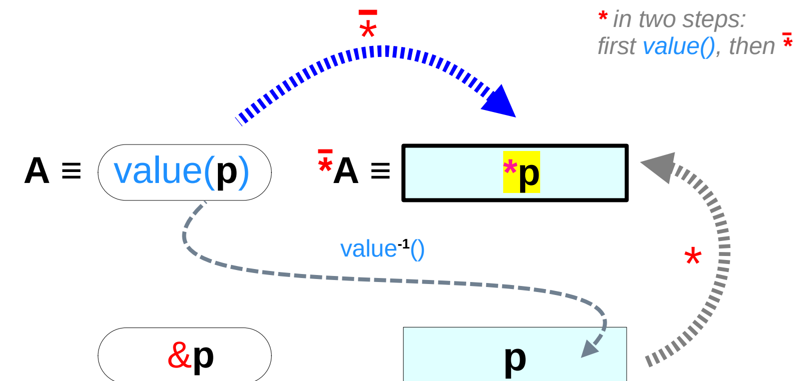
Two step Address-of & operation



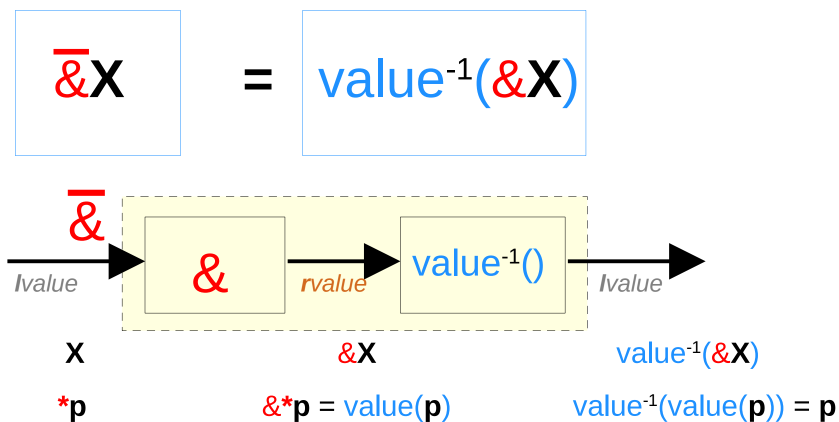


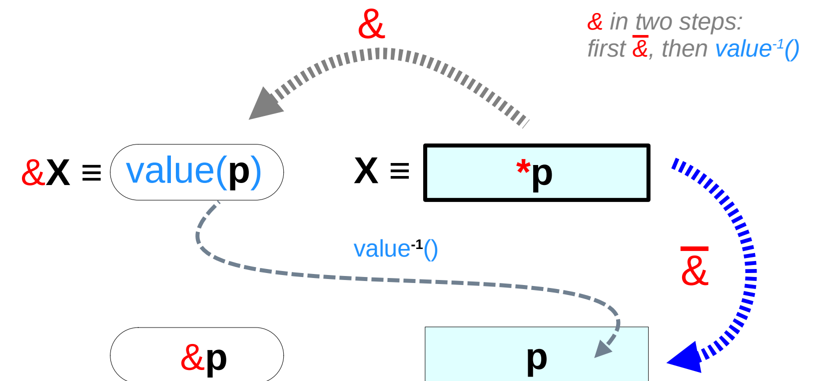
Two step De-reference * operation





Two step Address-of & operation





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

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

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

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

C Expressions

& X

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

& p

& a

&

&X

X

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

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

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

$\overline{\&}$ X = p

$\overline{\&}$ *p ≡ p

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

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

&

$\overline{\&}$ in two steps:
first &, then value^{-1}()

&X

X

value^{-1}()

$\overline{\&}$

&p

p

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

for $\bar{*}$**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

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

$\bar{*}$ **A**

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

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

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

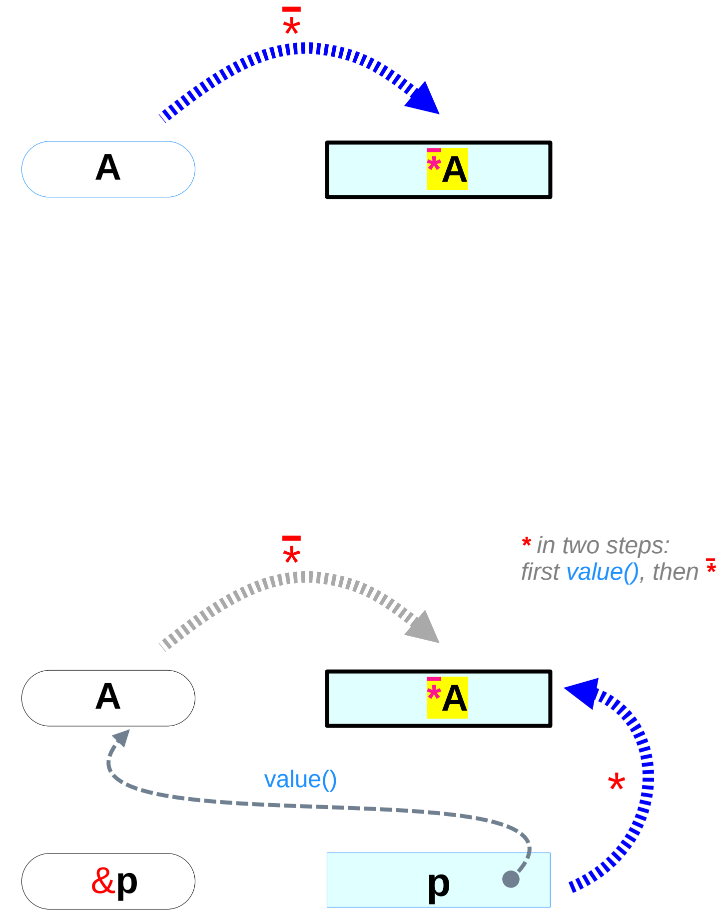
***p**	=	$\bar{*}$value(**p**)

C Expressions

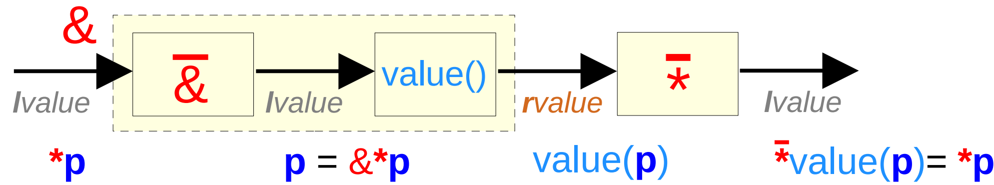
* **p**

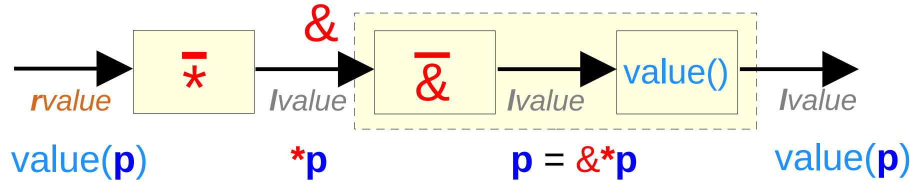
* **p** ≡ $\bar{*}$value(**p**)

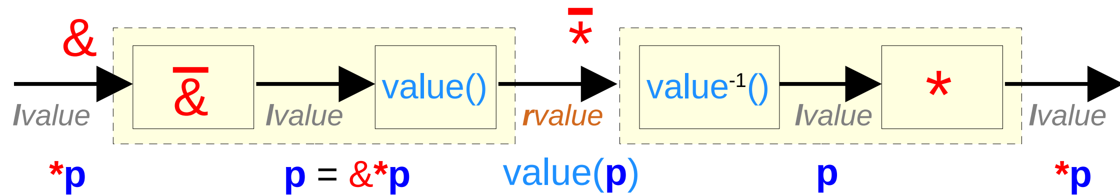
~~$\bar{*}$ &**p** = **p**~~
~~$\bar{*}$ &**a** = **a**~~

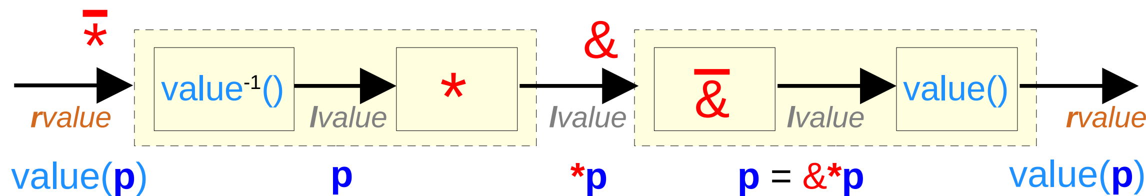


Inverse operators : & and $\overline{*}$ in pointer de-referencing





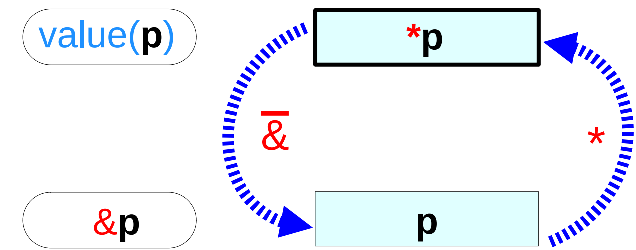


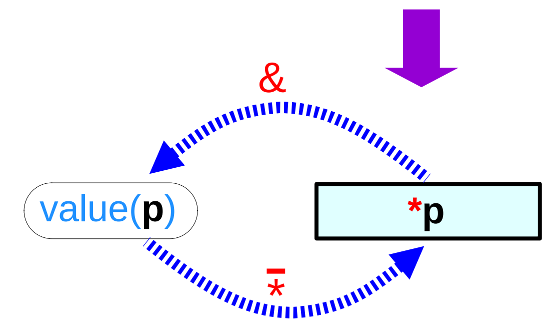


$\overline{*}$ value(p) = *p

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

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





Inverse operators of & and *

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

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

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

lvalue → * → [value() → rvalue → $\bar{*}$] → lvalue → $\bar{\&}$ → lvalue

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

lvalue → $\bar{\&}$ → lvalue → * → [value() → rvalue → $\bar{*}$] → lvalue

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

lvalue → * → [value() → rvalue → $\bar{*}$] → lvalue → $\bar{\&}$ → [& → rvalue → $value^{-1}$()] → lvalue

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

lvalue → $\bar{\&}$ → [& → rvalue → $value^{-1}$()] → lvalue → * → [value() → rvalue → $\bar{*}$] → lvalue

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

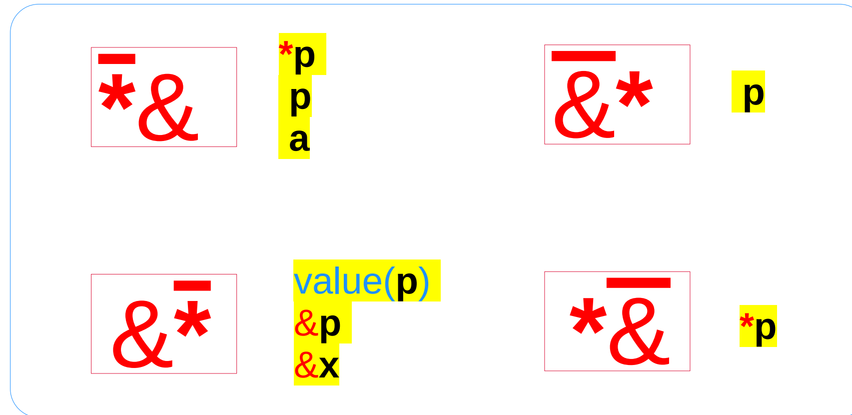
value(p) *p $\bar{\&}$ * &p p

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

Inverse operators in pointer referencing (1)

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

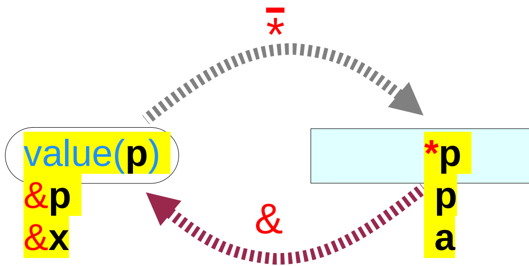
$\&\overline{*}$ value(p) = value(p)
$\&\overline{*}$ &p = &p
$\&\overline{*}$ &a = &a

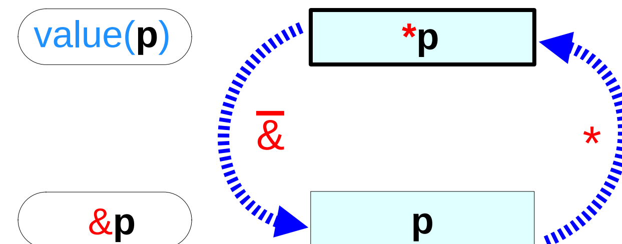


Math expressions

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

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

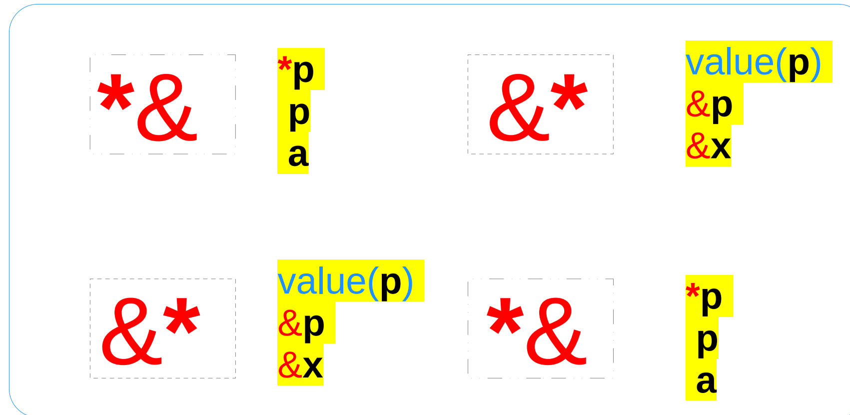




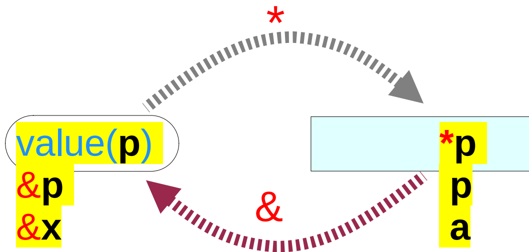
Inverse operators in pointer referencing (2)

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

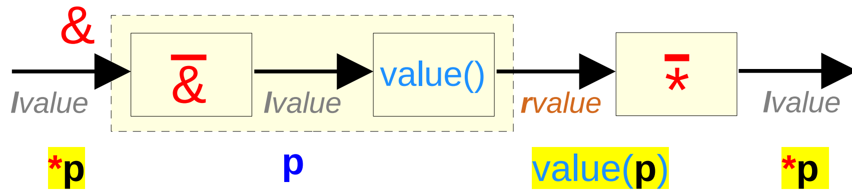
&* value(p) = value(p)
&* &p = &p
&* &a = &a

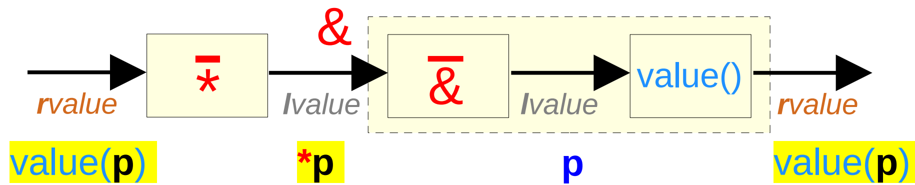


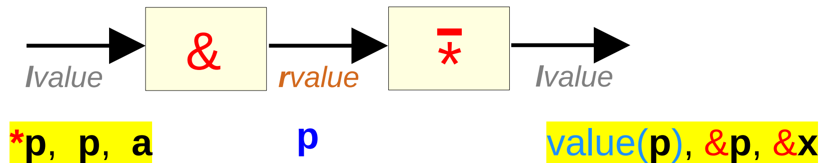
C expressions

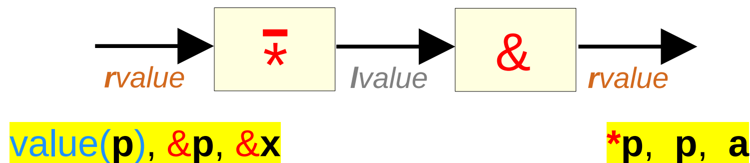


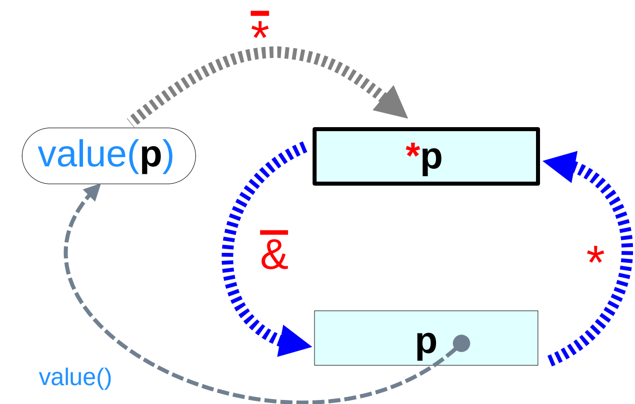
Inverse operators in pointer referencing (3)

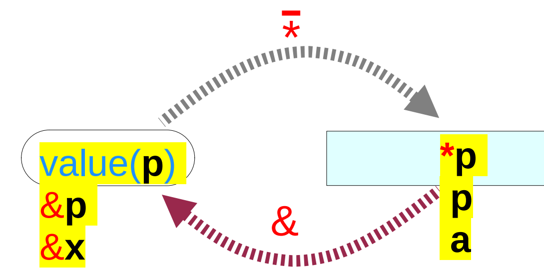




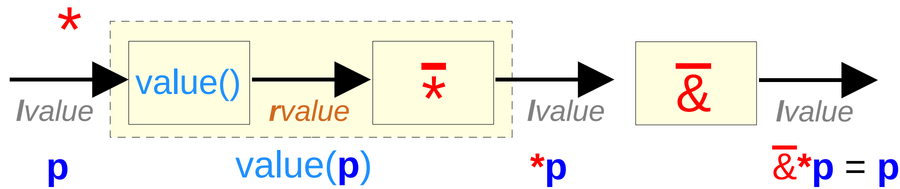


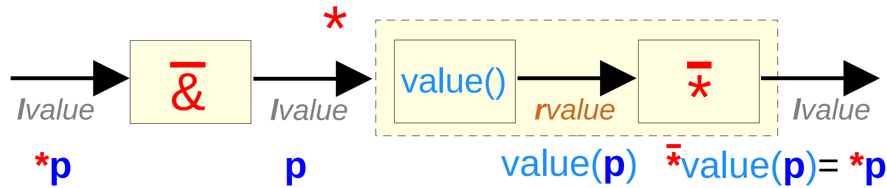


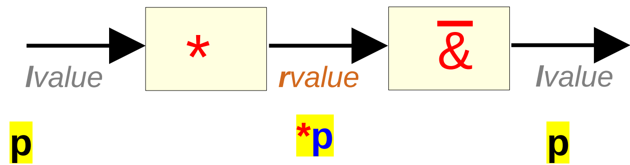


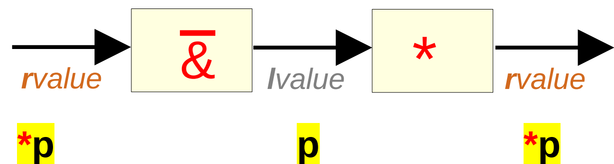


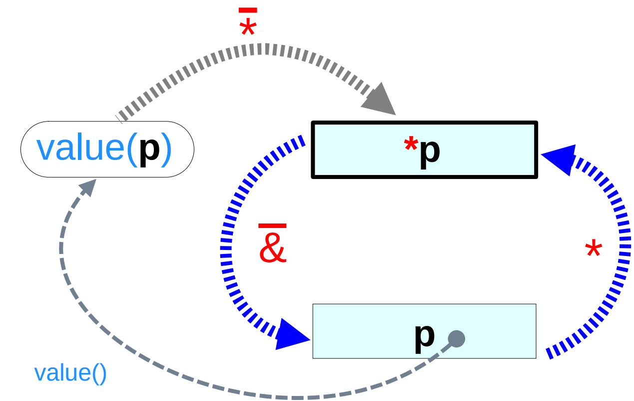
Inverse operators in pointer referencing (3)

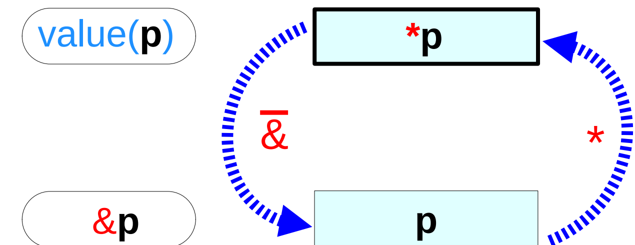












References

[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