OOPSLA’18 Boston

Reconciling High-Level Optimizations and Low-Level Code in LLVM

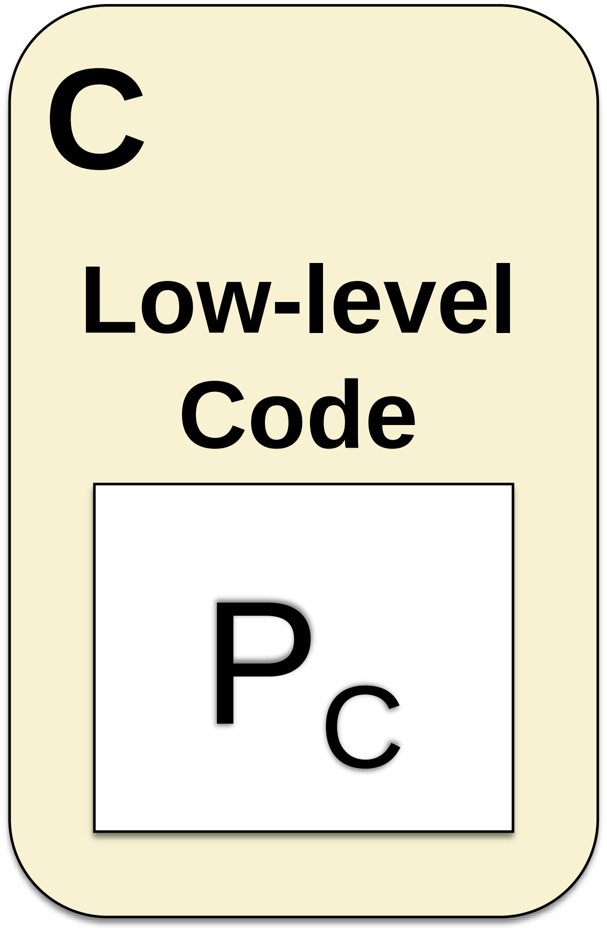
Seoul National Univ.
Juneyoung Lee
Chung-Kil Hur

MPI-SWS
Ralf Jung

University of Utah
Zhengyang Liu
John Regehr

Microsoft Research
Nuno P. Lopes

Overview

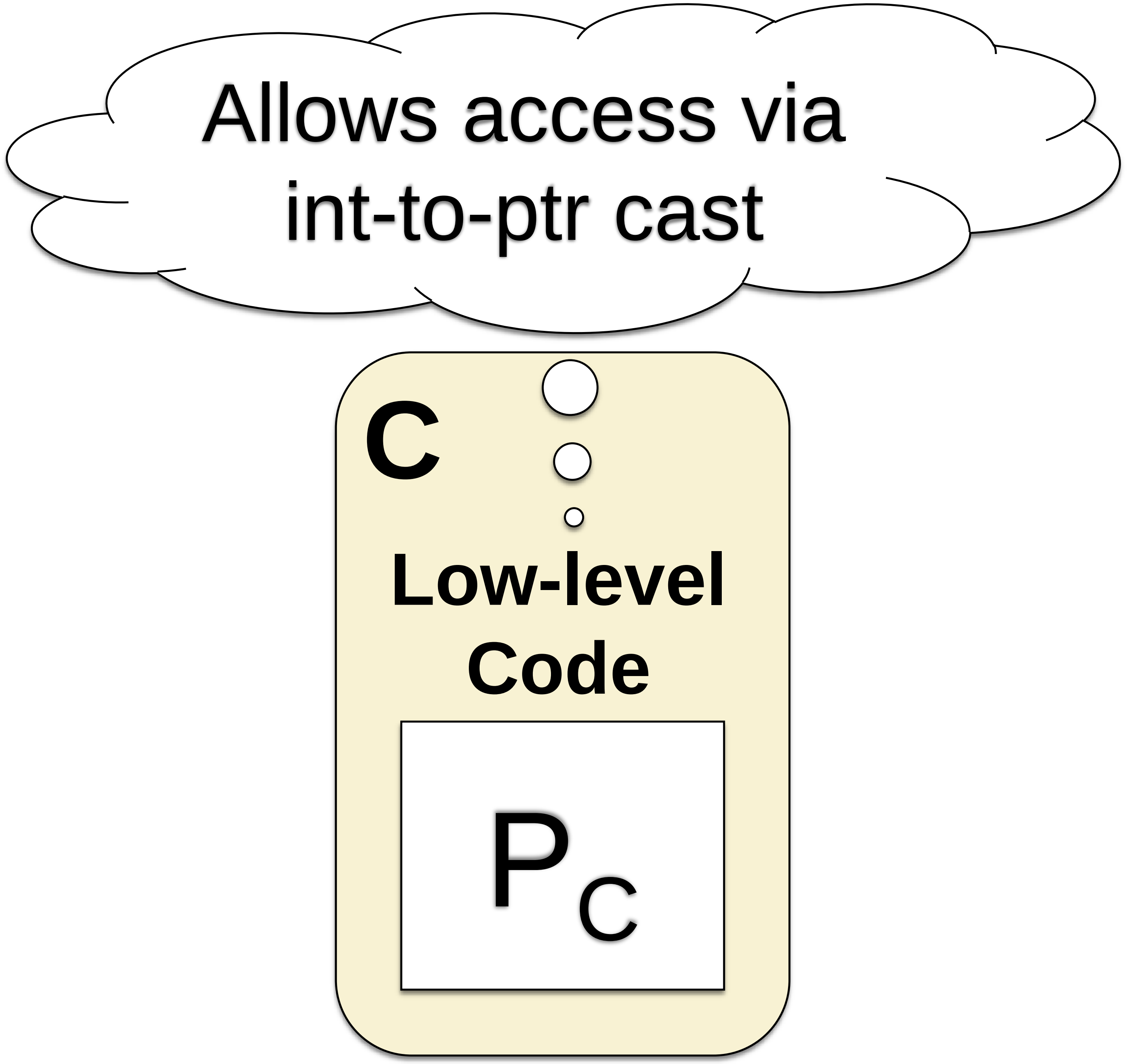


C

Low-level Code

P_C

Overview



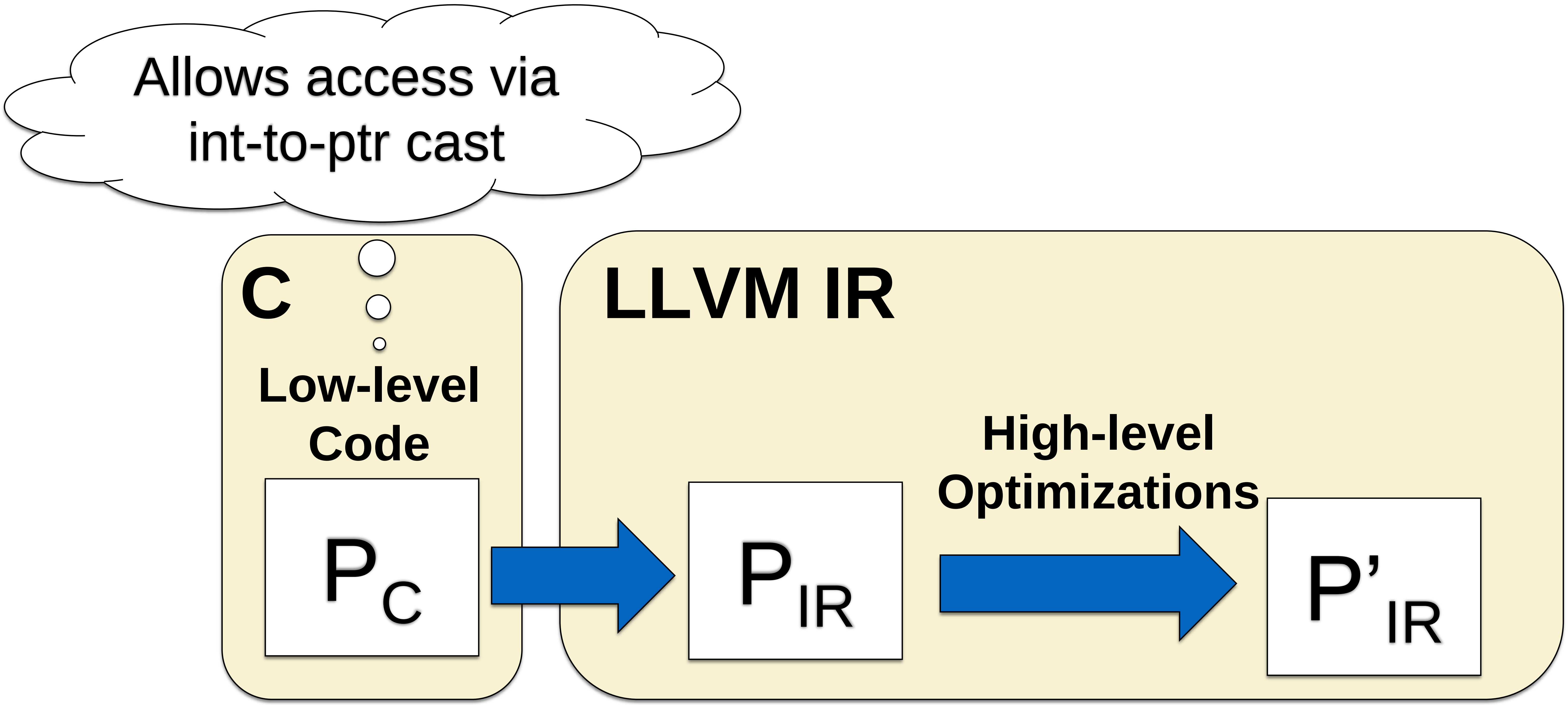
Allows access via int-to-ptr cast

C

Low-level Code

P_C

Overview



Allows access via int-to-ptr cast

C

Low-level Code

P_C

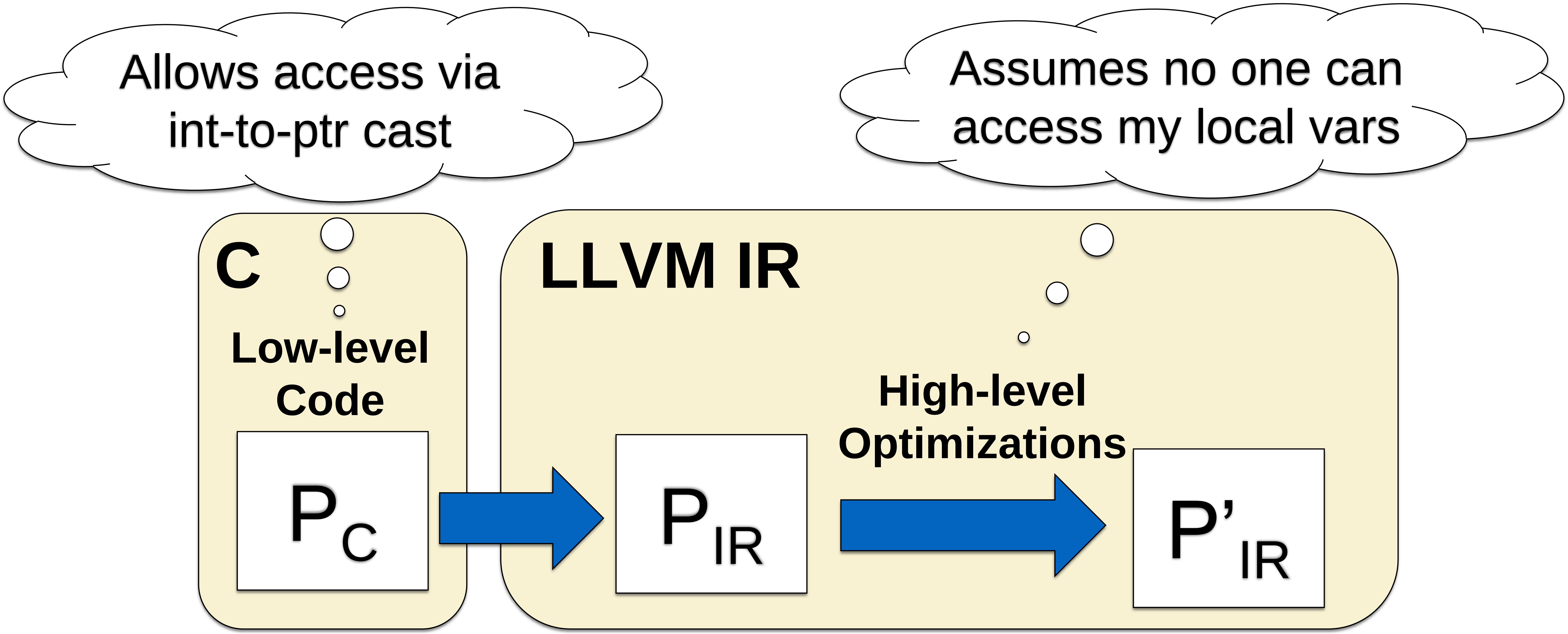
LLVM IR

P_{IR}

High-level Optimizations

P'_{IR}

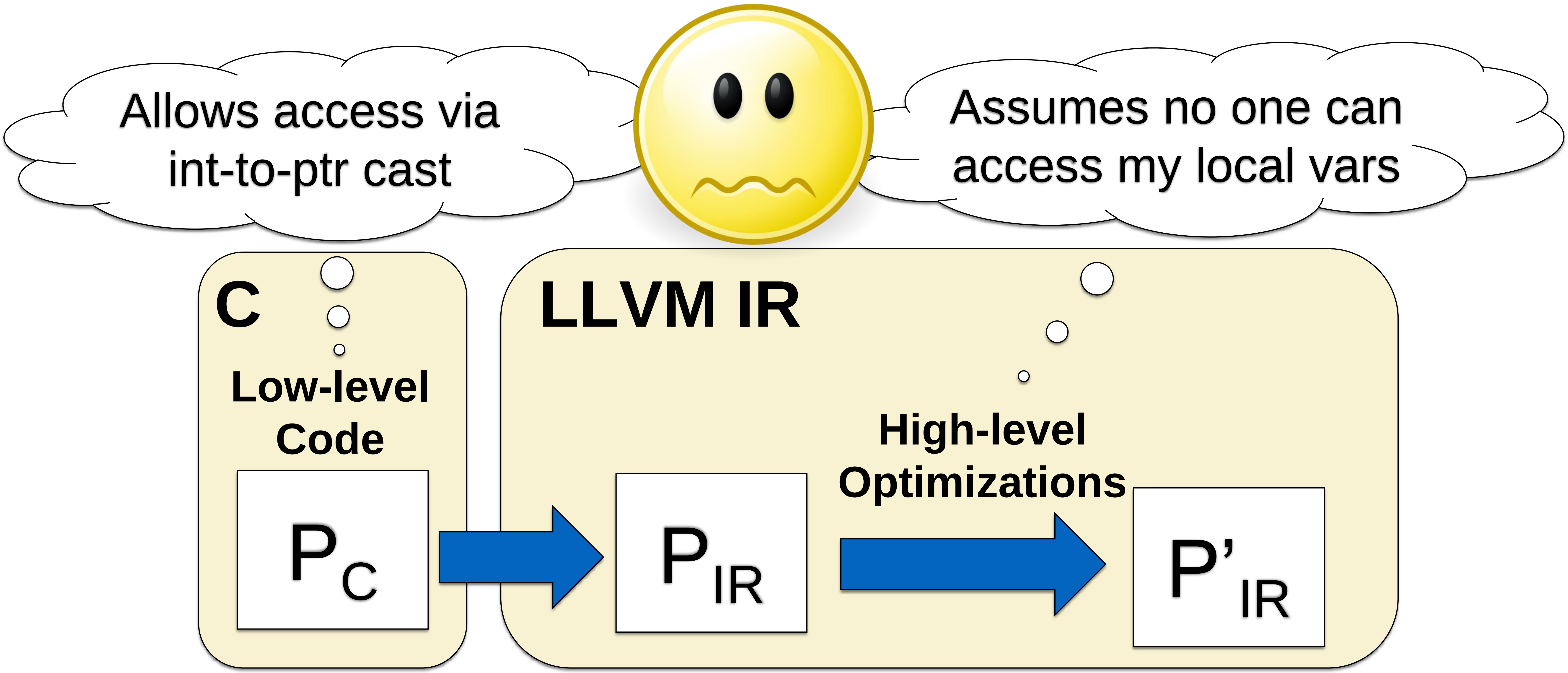
Overview



Allows access via int-to-ptr cast

Assumes no one can access my local vars

Overview



Allows access via int-to-ptr cast

Assumes no one can access my local vars

C

Low-level Code

P_C

LLVM IR

P_{IR}

High-level Optimizations

P'_{IR}

Finding a Good Memory Model

- A memory model specifies the behavior of memory operations
- As a result, it determines

1. Which low-level programs are valid
2. Which high-level assumptions are valid

- A good memory model should make valid both

1. Common low-level programs
2. Common high-level assumptions

Memory ≠ Byte Array

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

Memory ≠ Byte Array

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

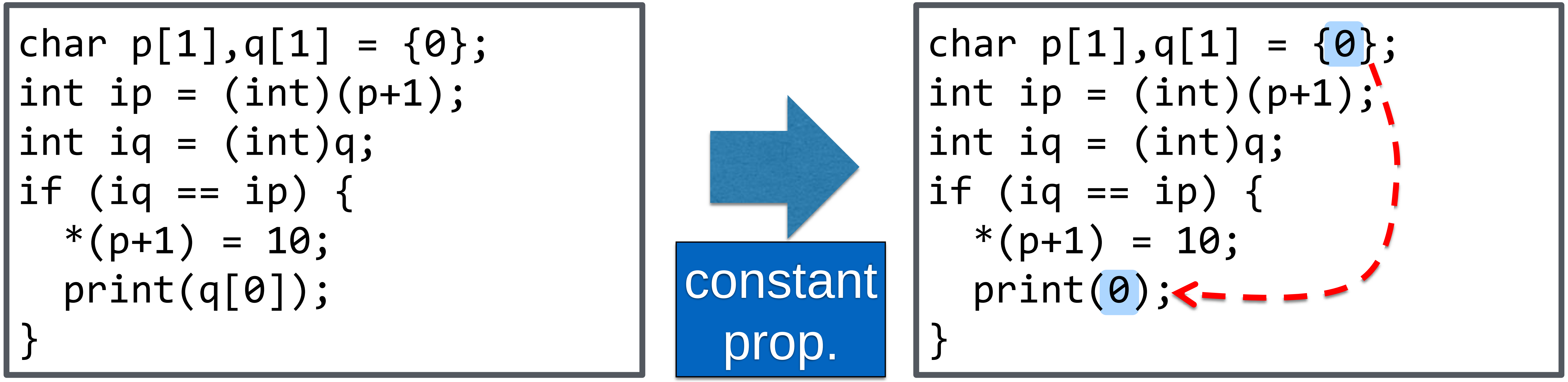
Memory ≠ Byte Array

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

Memory ≠ Byte Array

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

Memory ≠ Byte Array



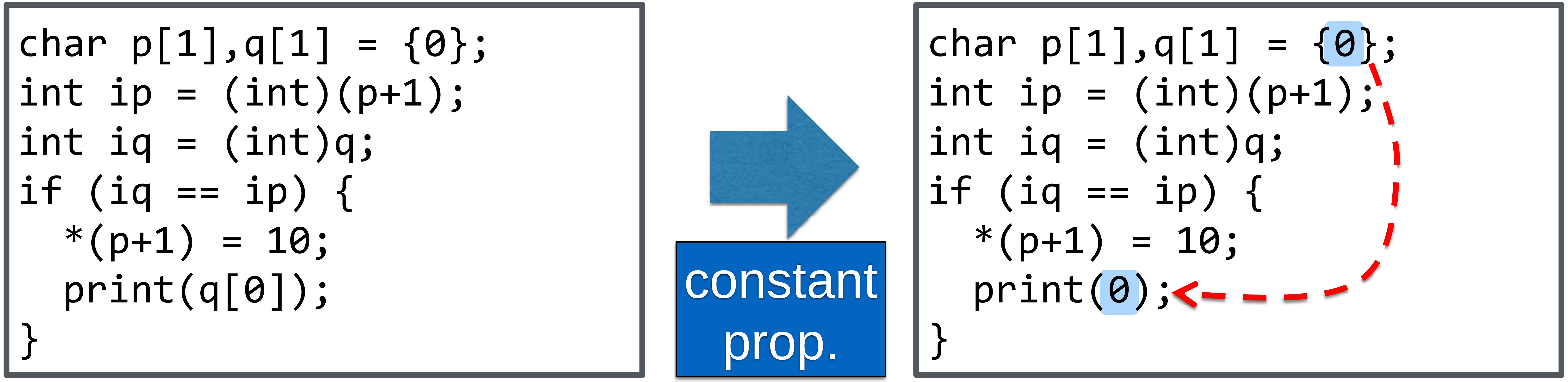
```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array

We use C syntax for LLVM IR code for readability

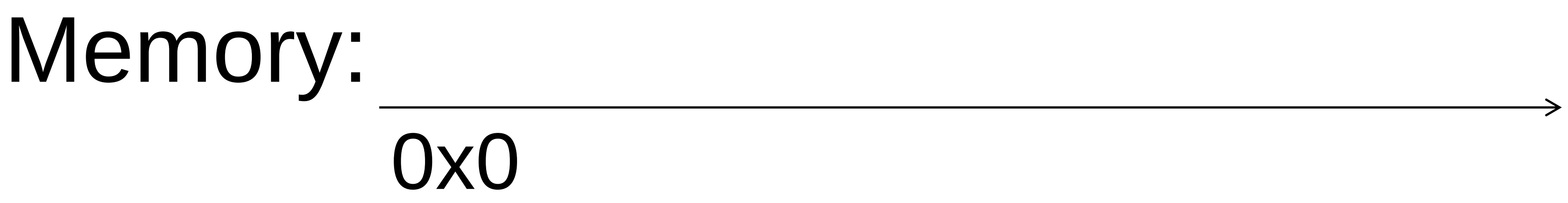


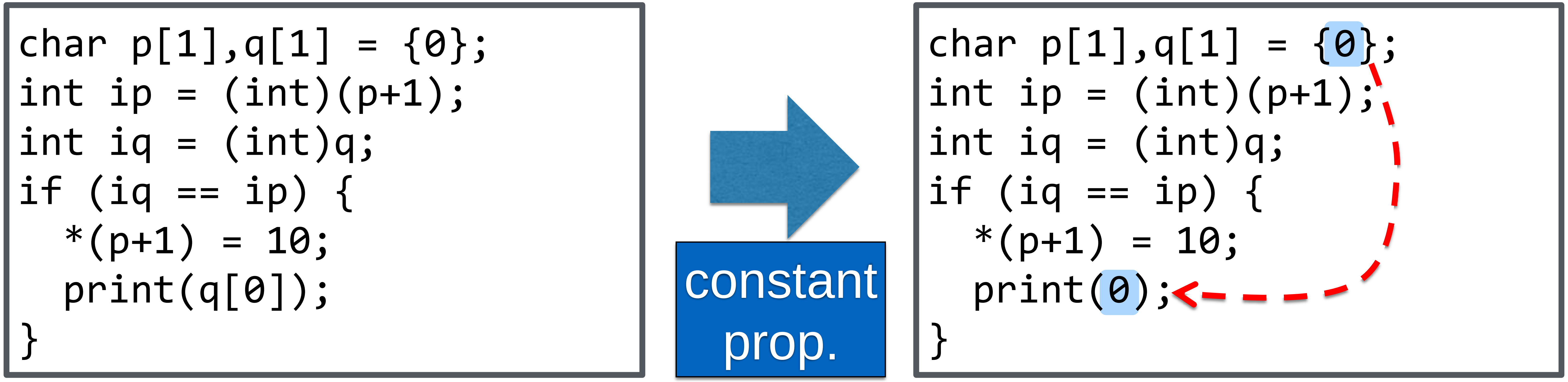
```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array



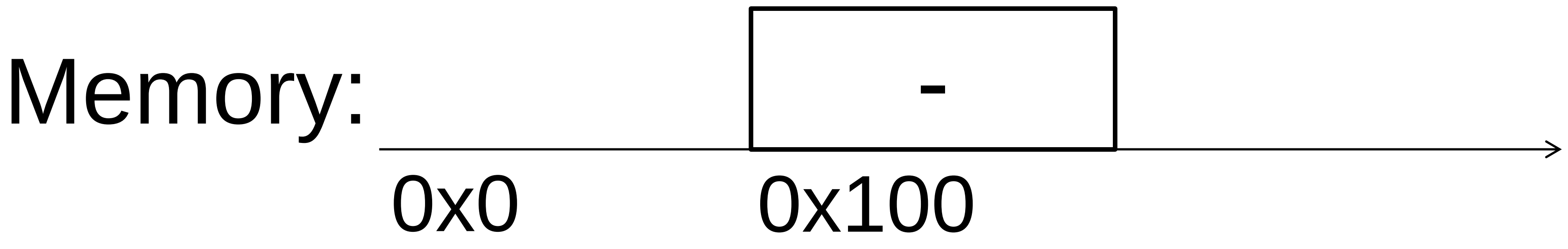


```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array



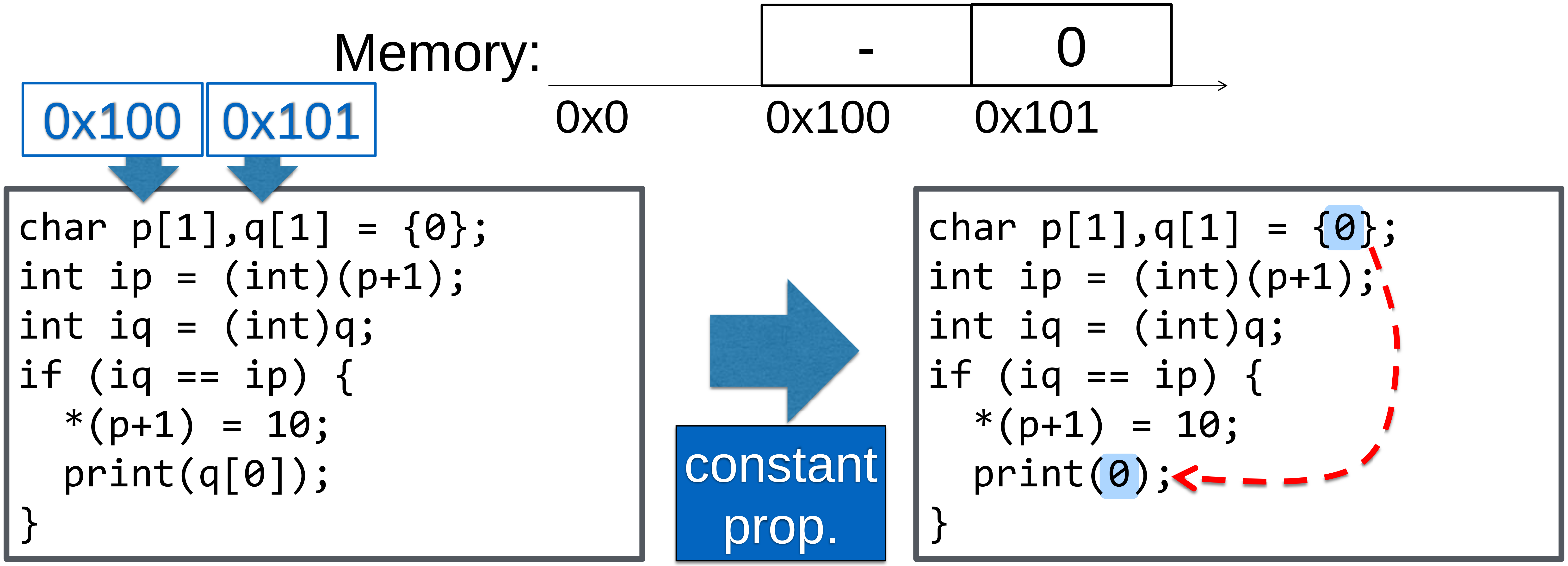
0x100

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array



Memory:

0x100	0x101
-	0

0x0 0x100 0x101

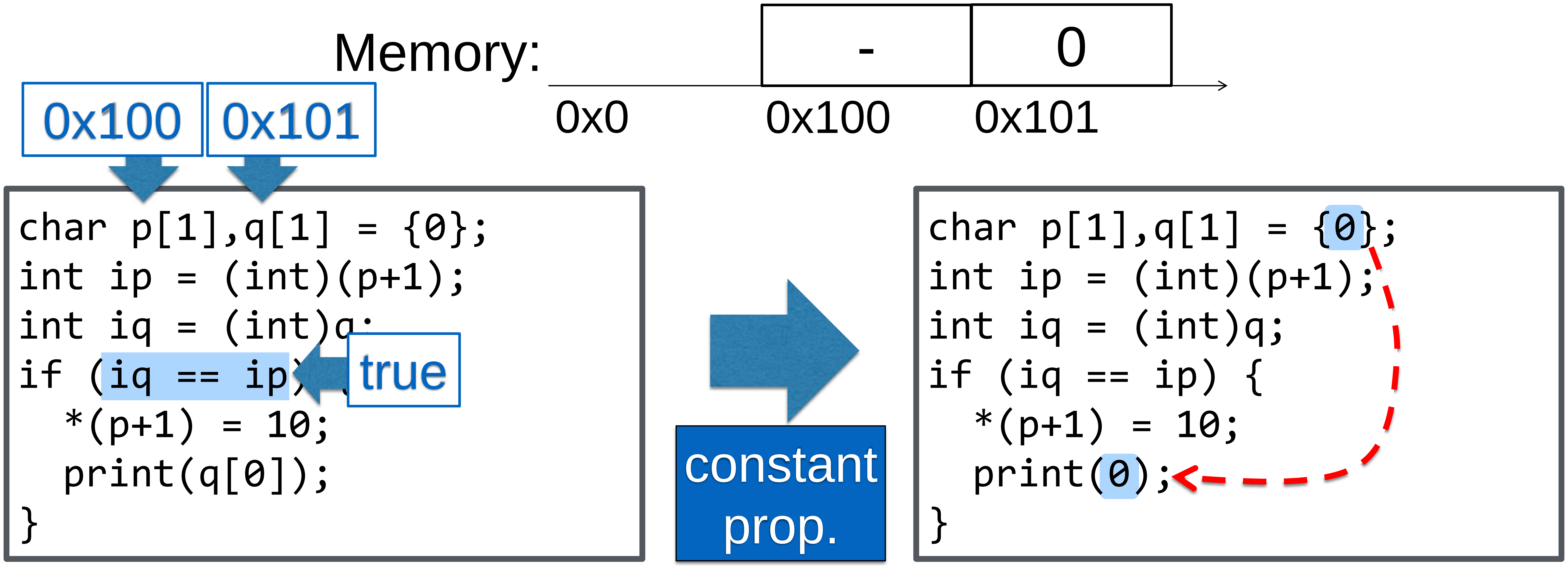
0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

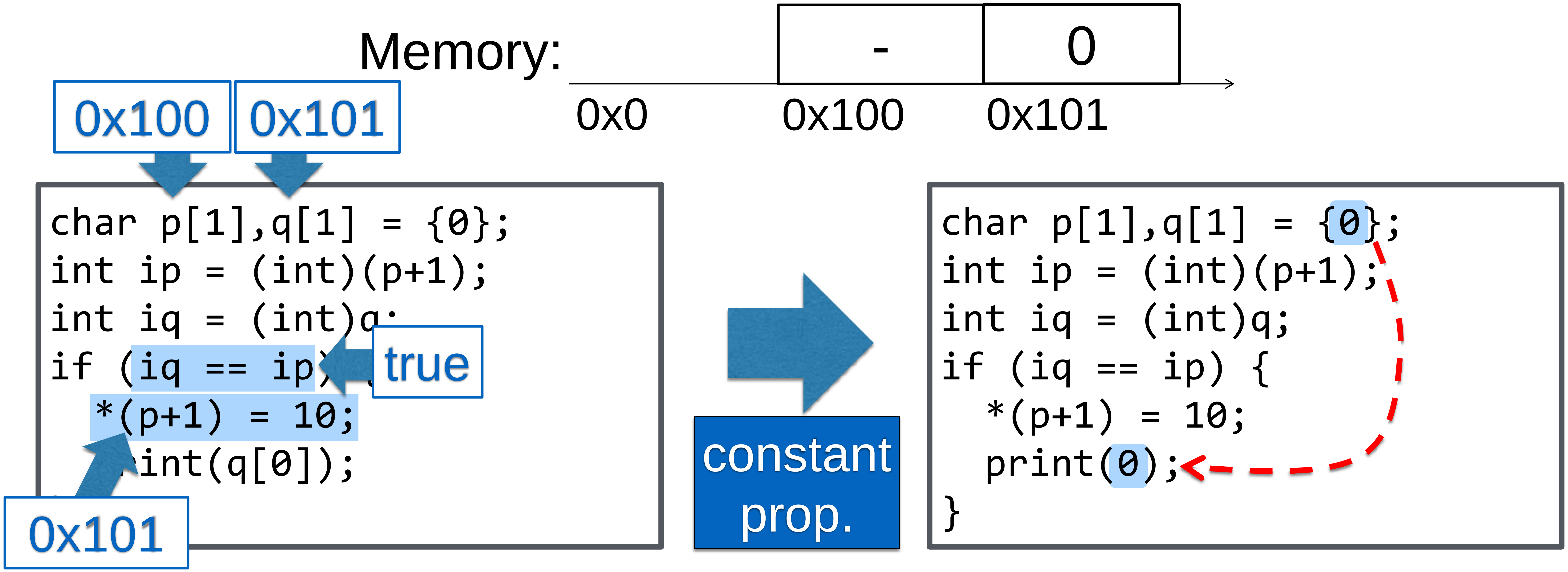
Memory ≠ Byte Array



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

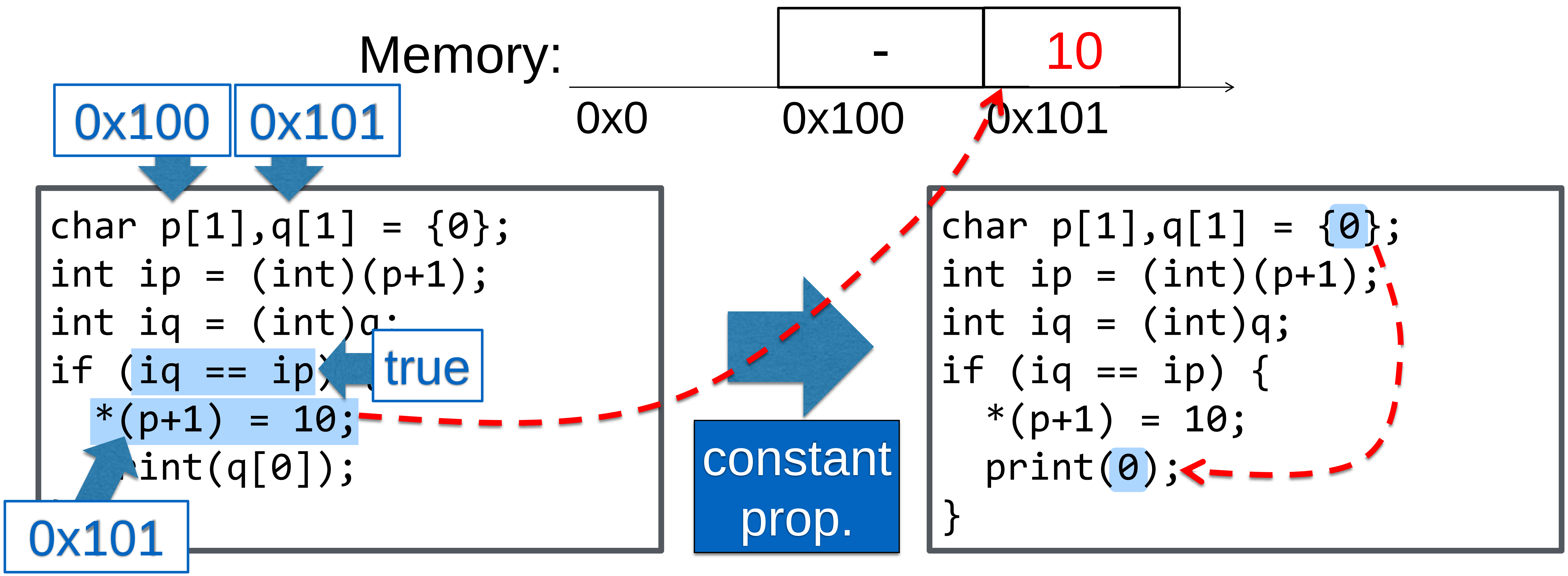
Memory ≠ Byte Array



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

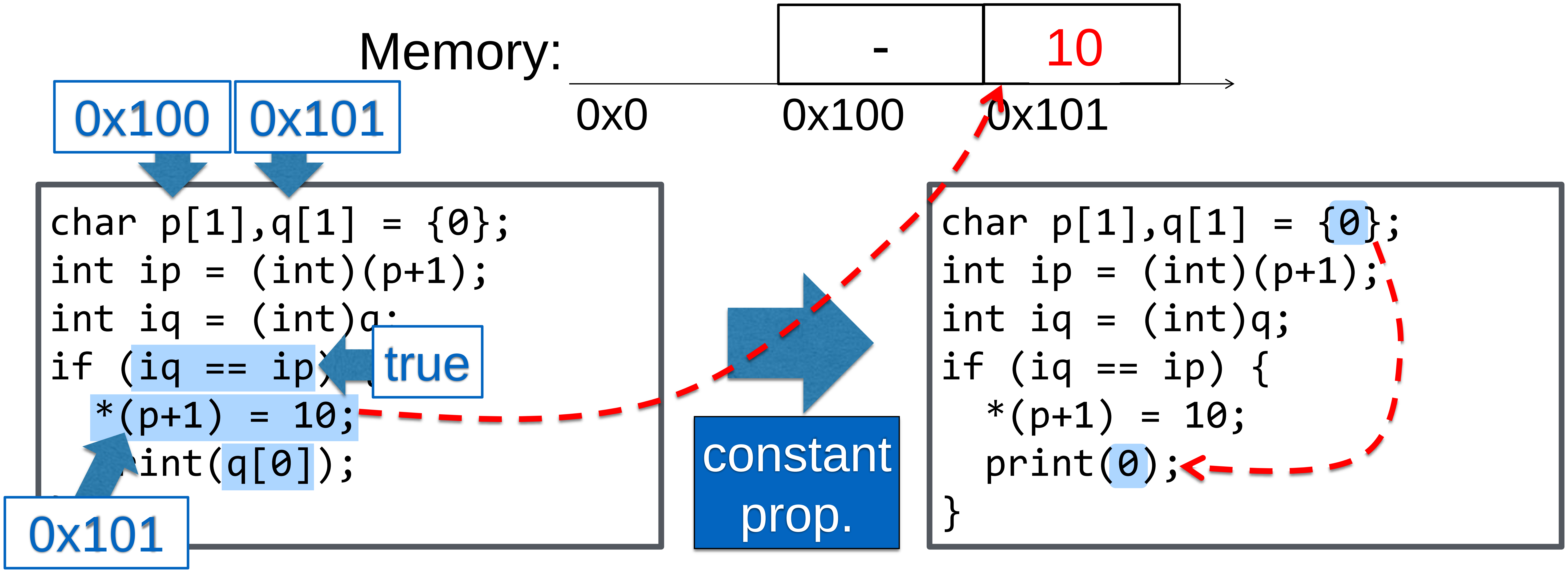
Memory ≠ Byte Array



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

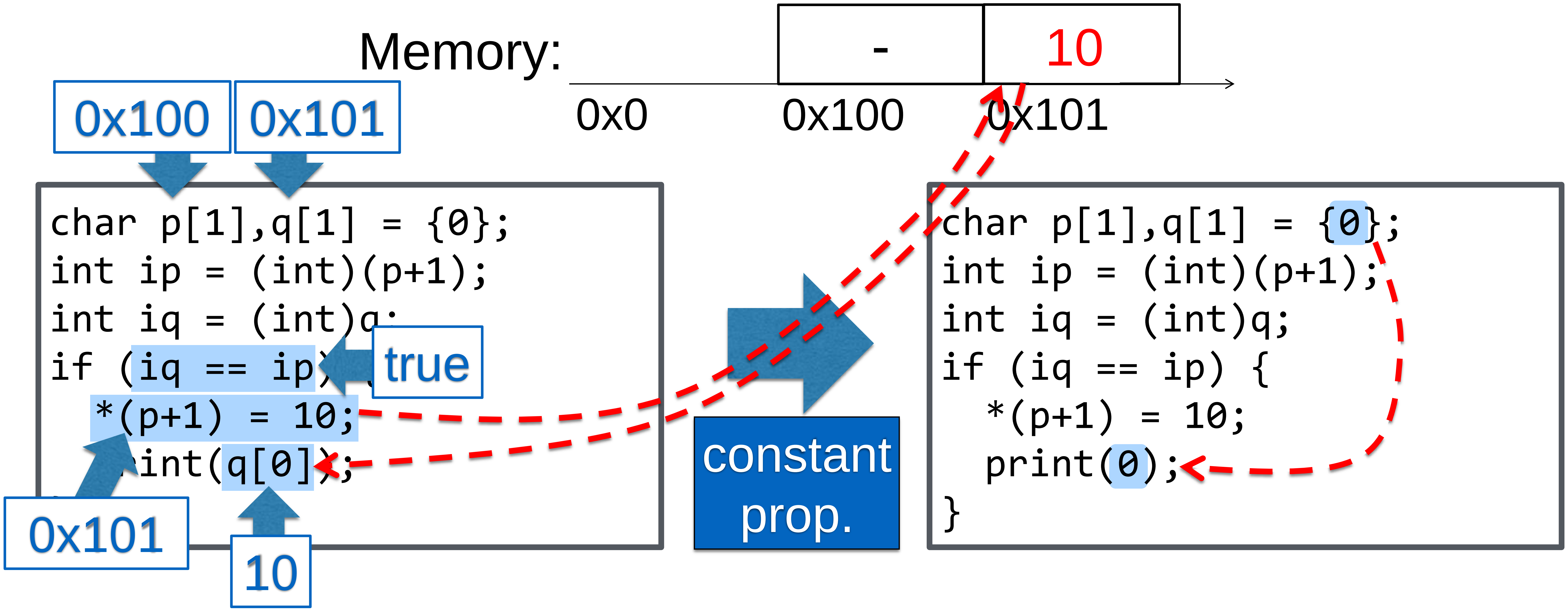
Memory ≠ Byte Array



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array



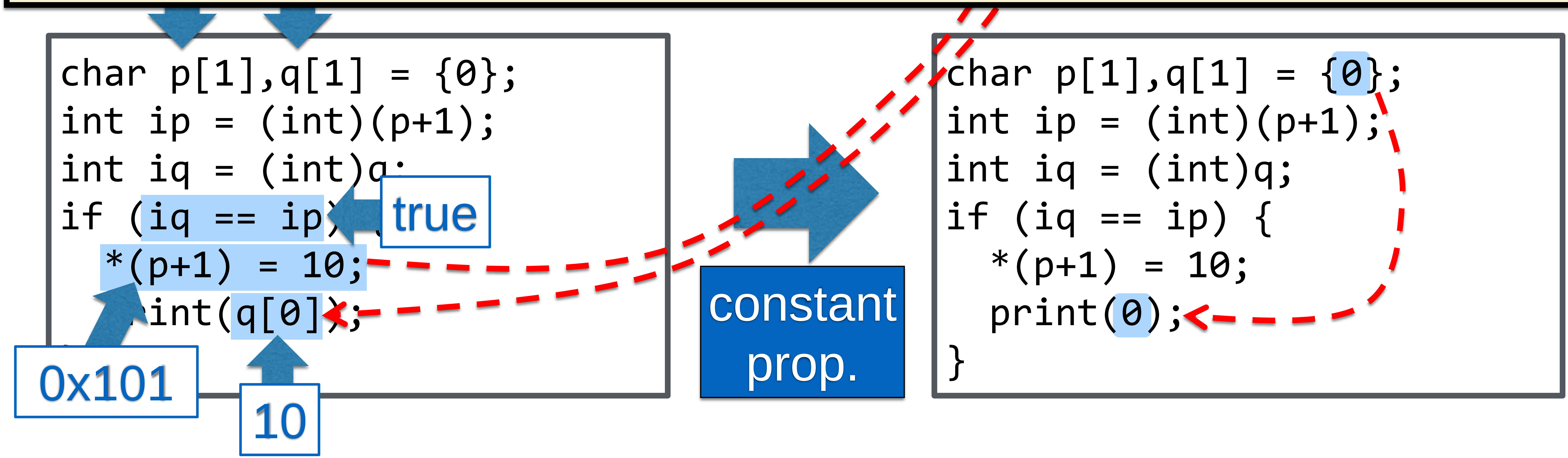
```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Memory ≠ Byte Array

Problem

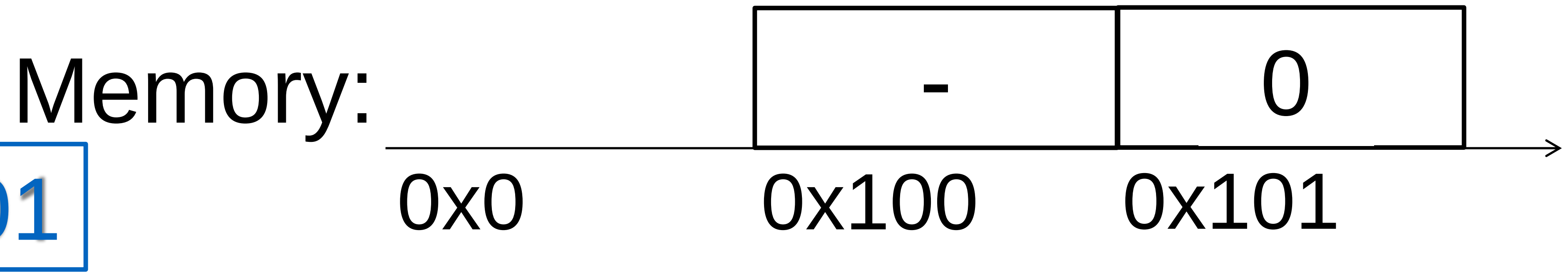
q can be accessed from p by pointer arithmetic



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Abstract Memory Explains Optimizations



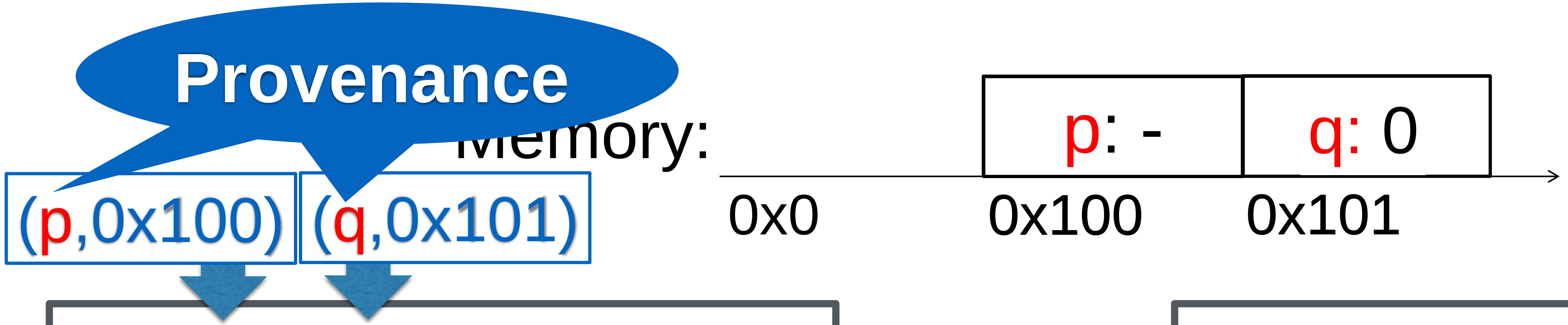
0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Abstract Memory Explains Optimizations



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

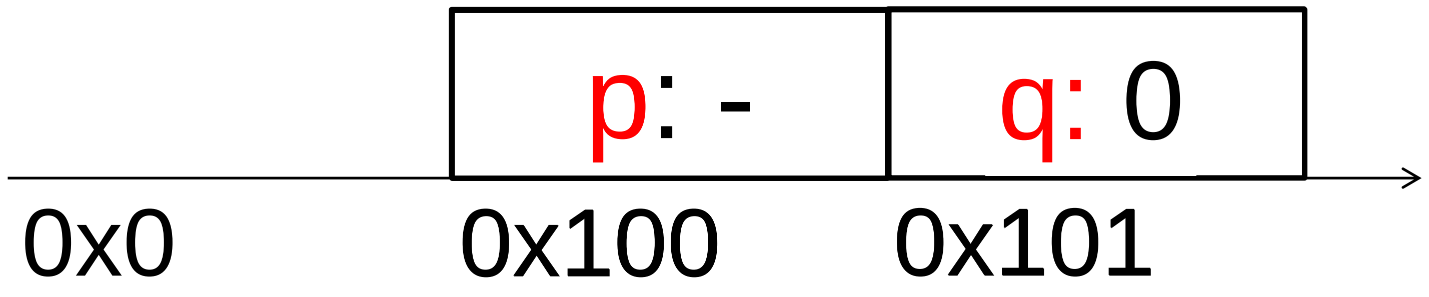
constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Abstract Memory Explains Optimizations

Provenance

(p,0x100) (q,0x101)



Memory:

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

true

constant prop.

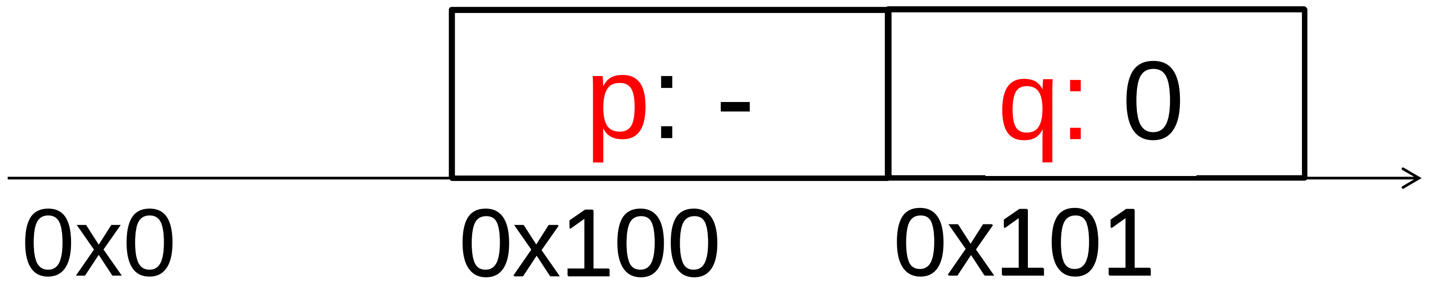
```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Abstract Memory Explains Optimizations

Provenance

(p,0x100) (q,0x101)

Memory:



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

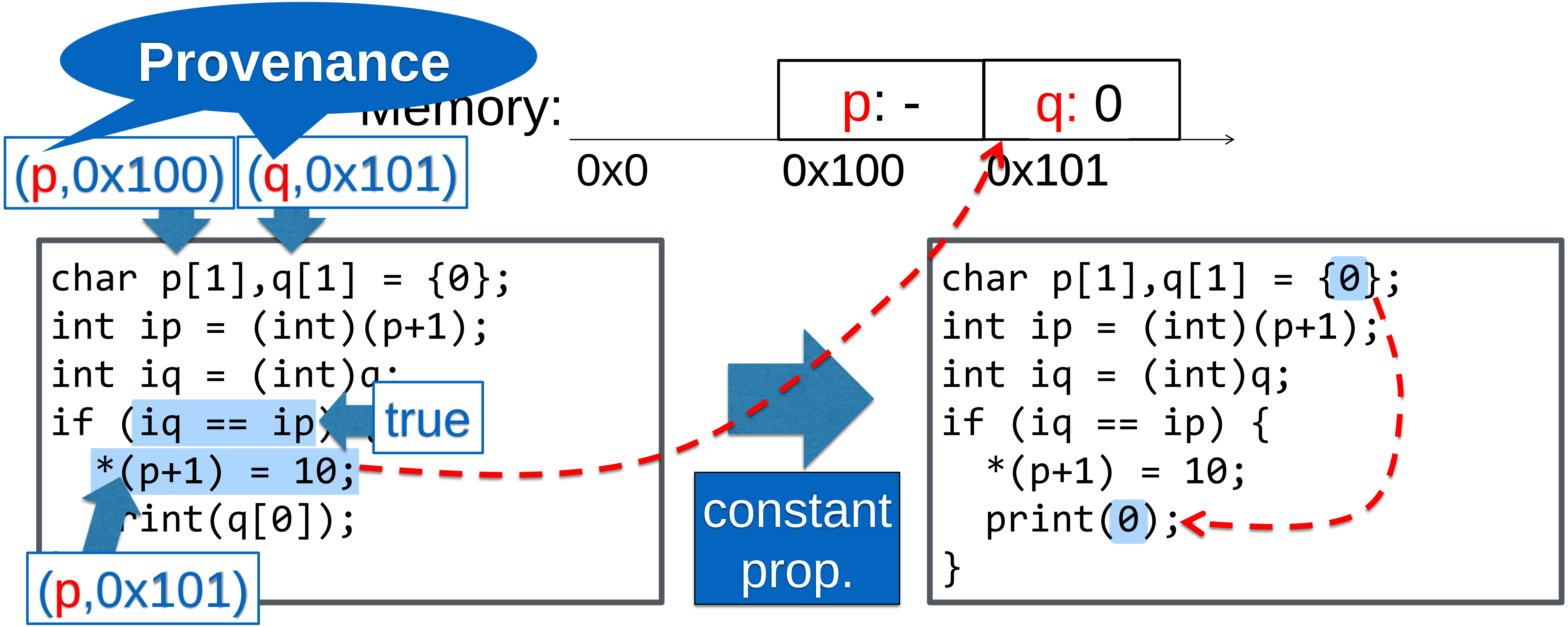
true

(p,0x101)

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

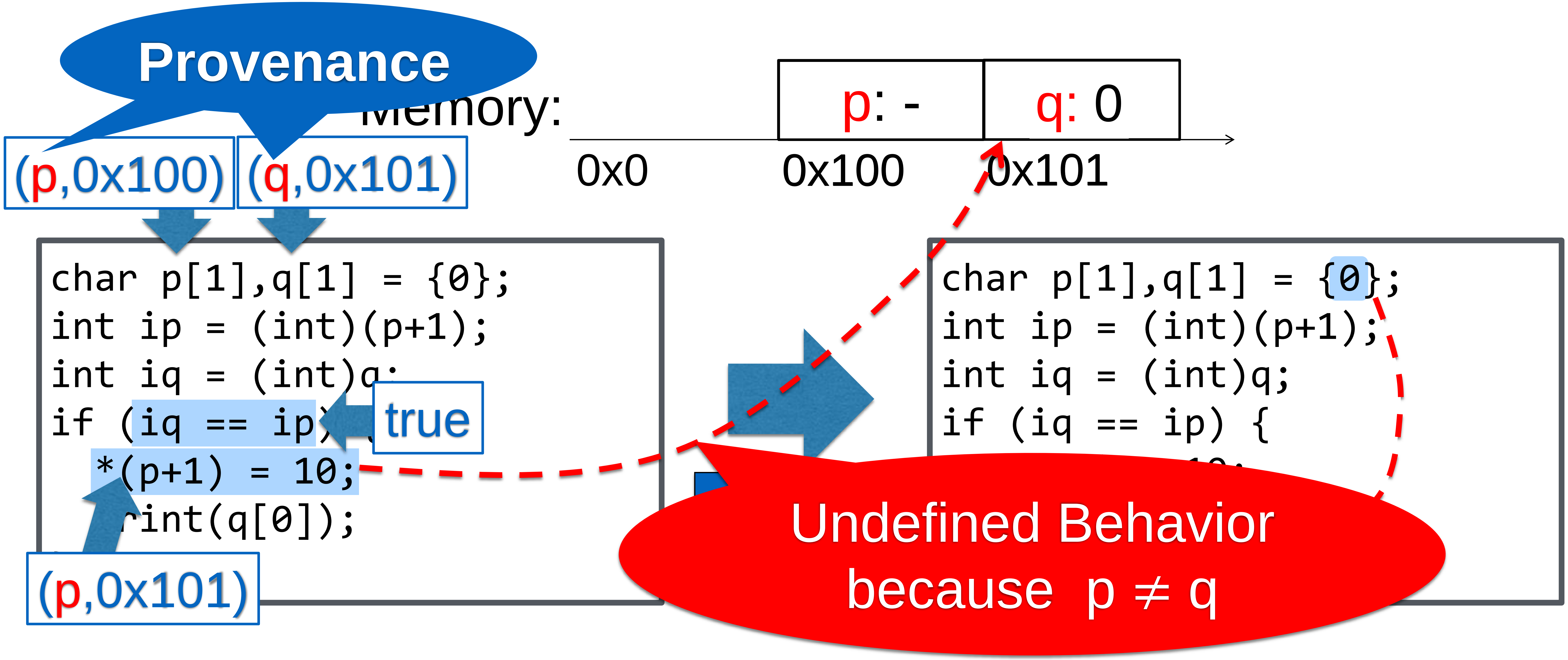
Abstract Memory Explains Optimizations



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Abstract Memory Explains Optimizations



```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
```

Abstract Memory Explains Optimizations

Provenance

Memory:

p: -

q: 0

(p,0x1

Principles of UB

1. Compilers assume input programs never raise UB

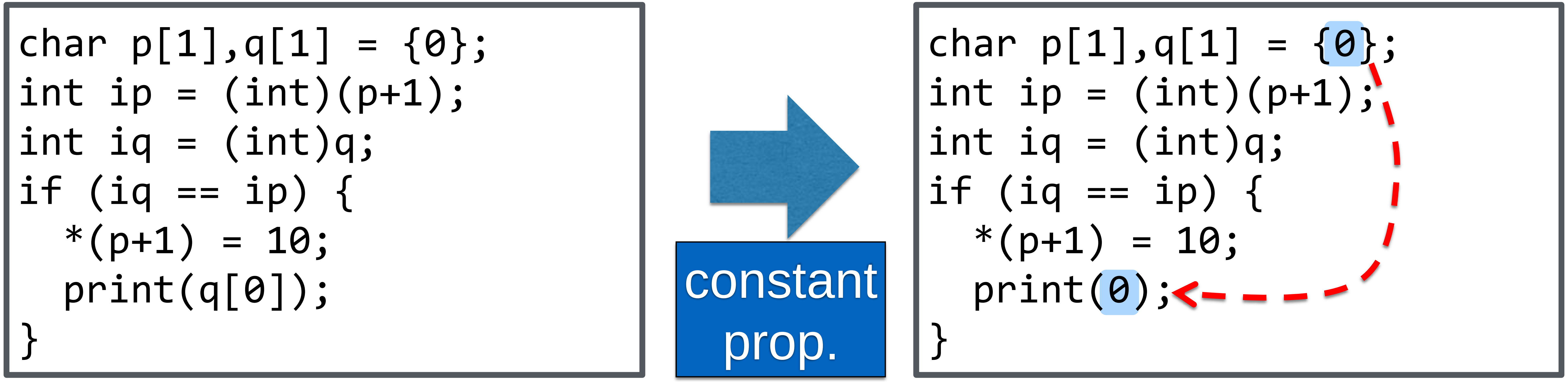
2. Programmers should not write programs raising UB

```
char
int
int
if (
  *(p+1) = 10;
  rint(q[0]);
```

(p,0x101)

Undefined Behavior because $p \neq q$

Miscompilation with Int-Ptr Casting

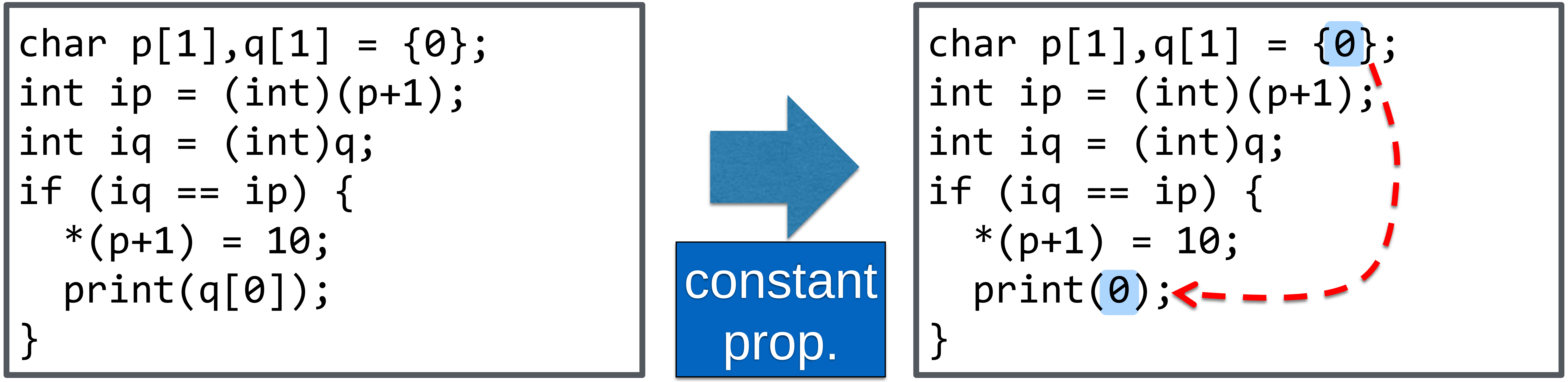


```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting

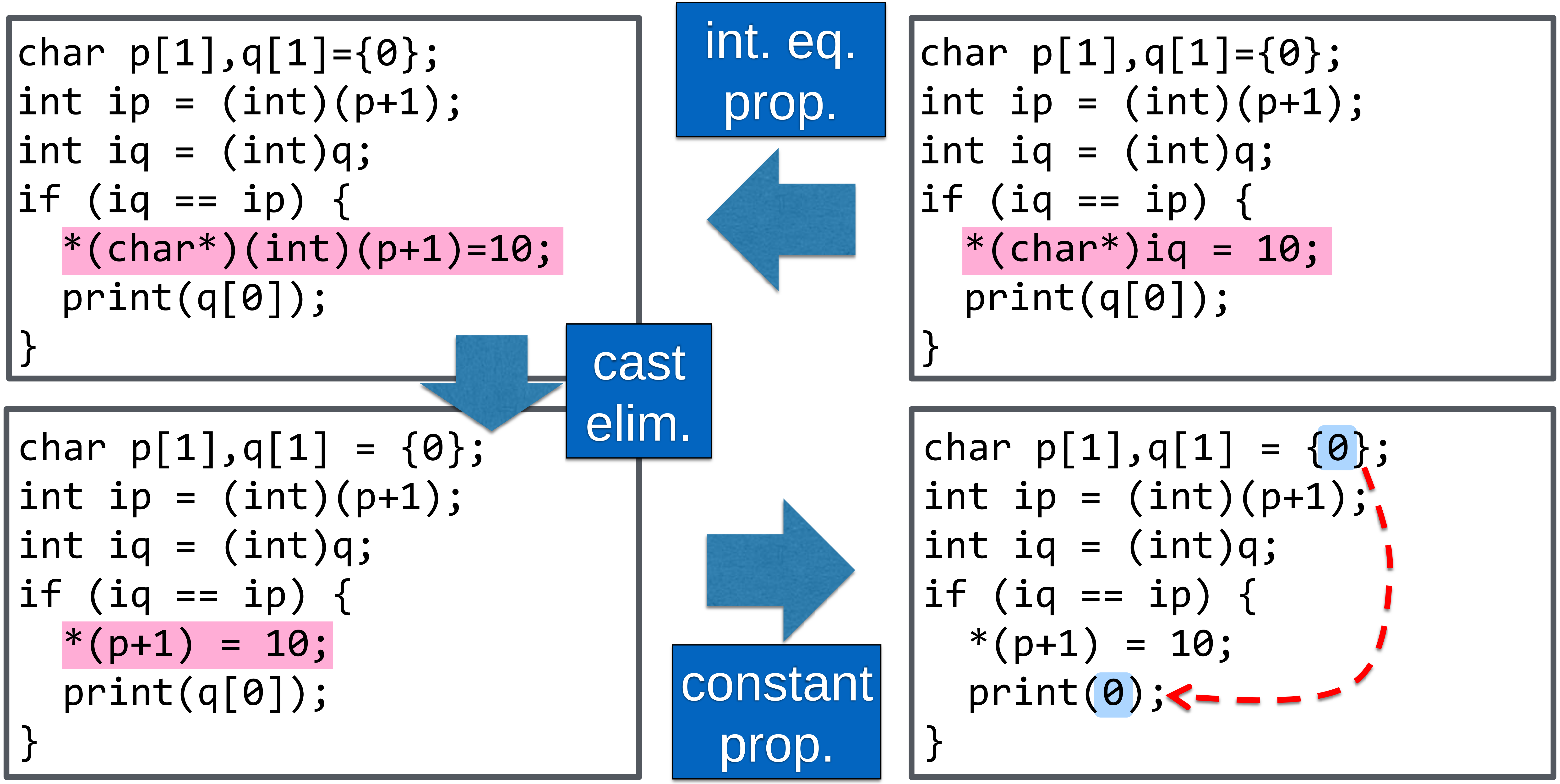


```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

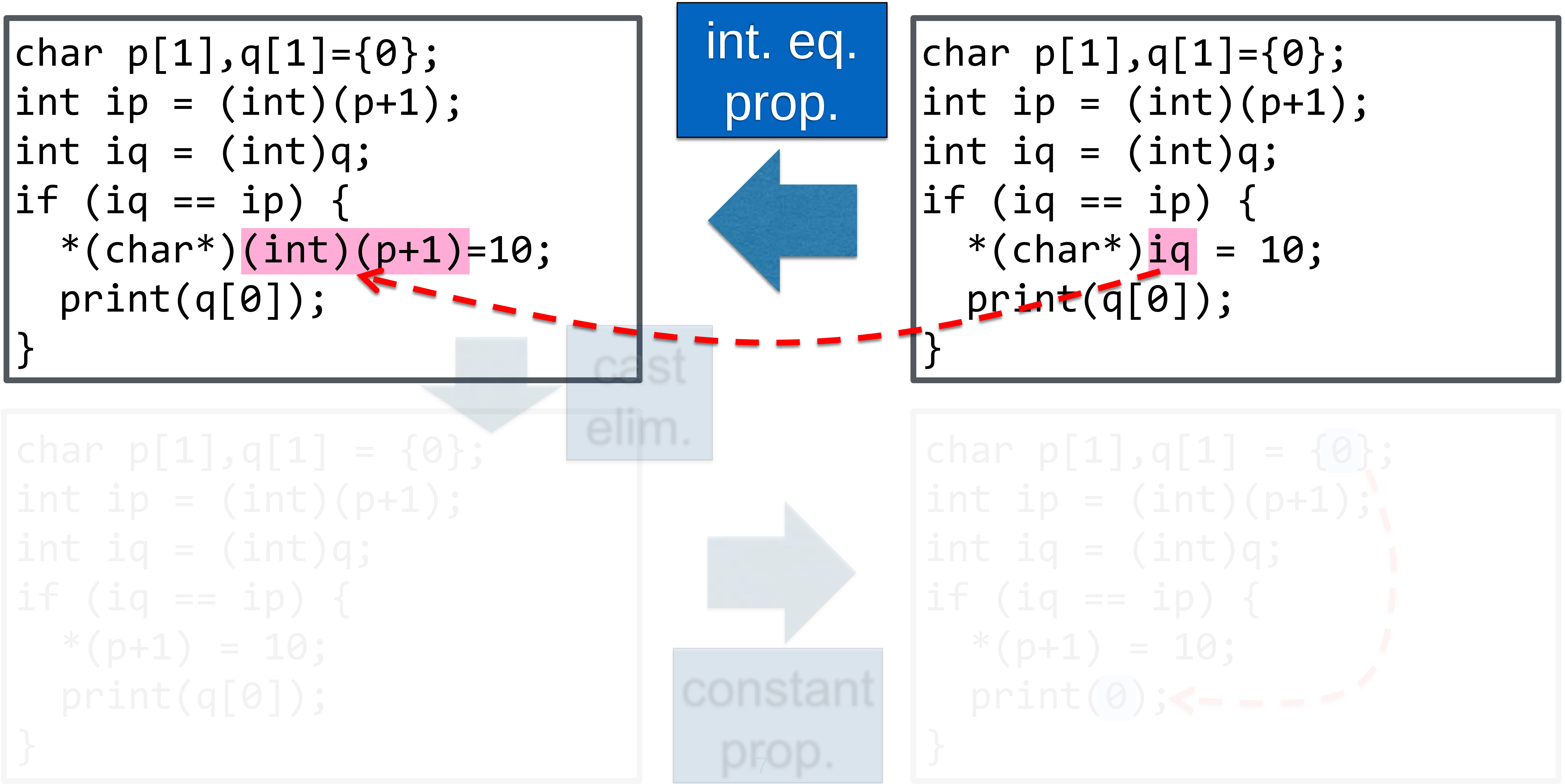
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

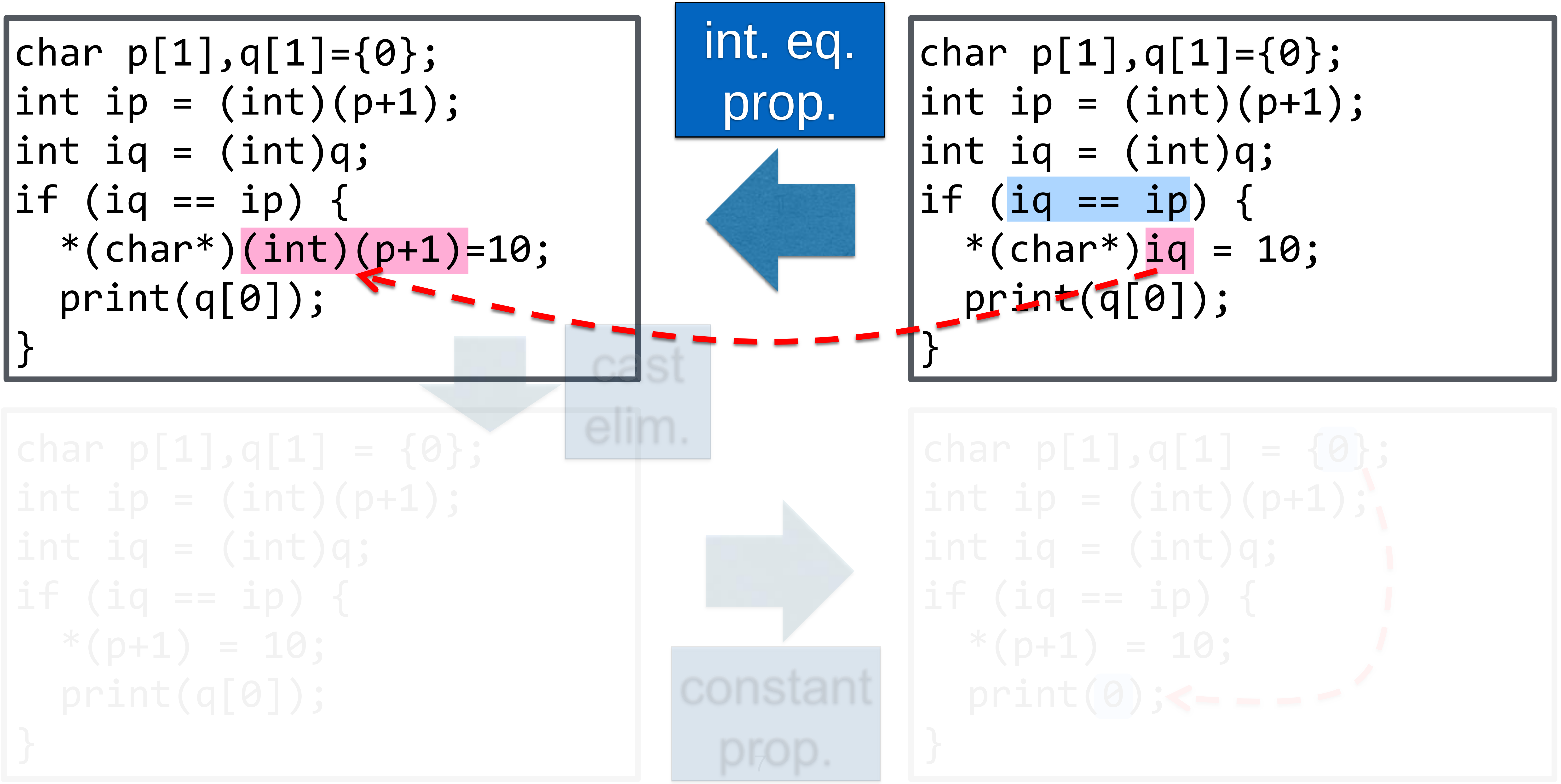
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

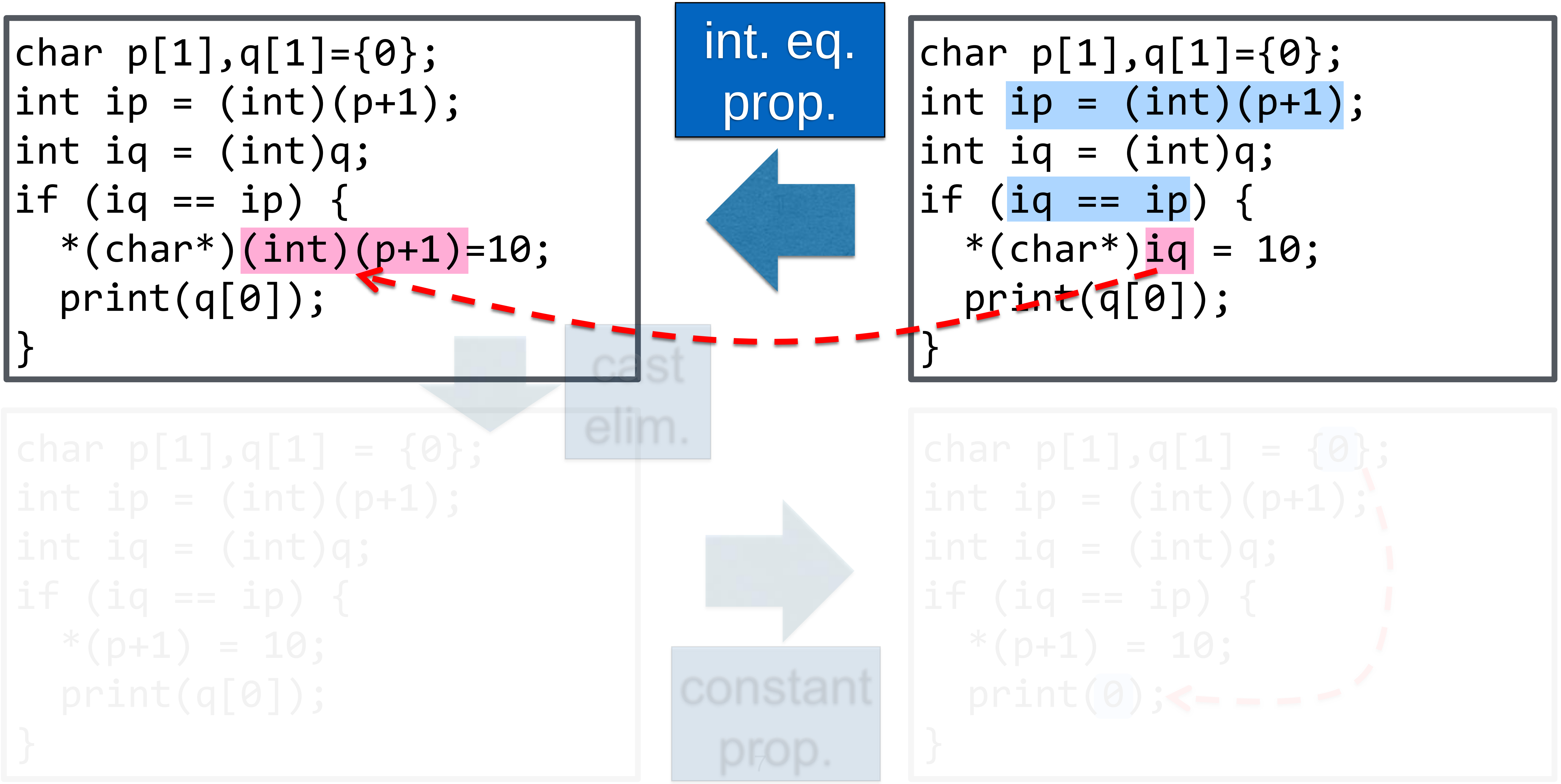
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

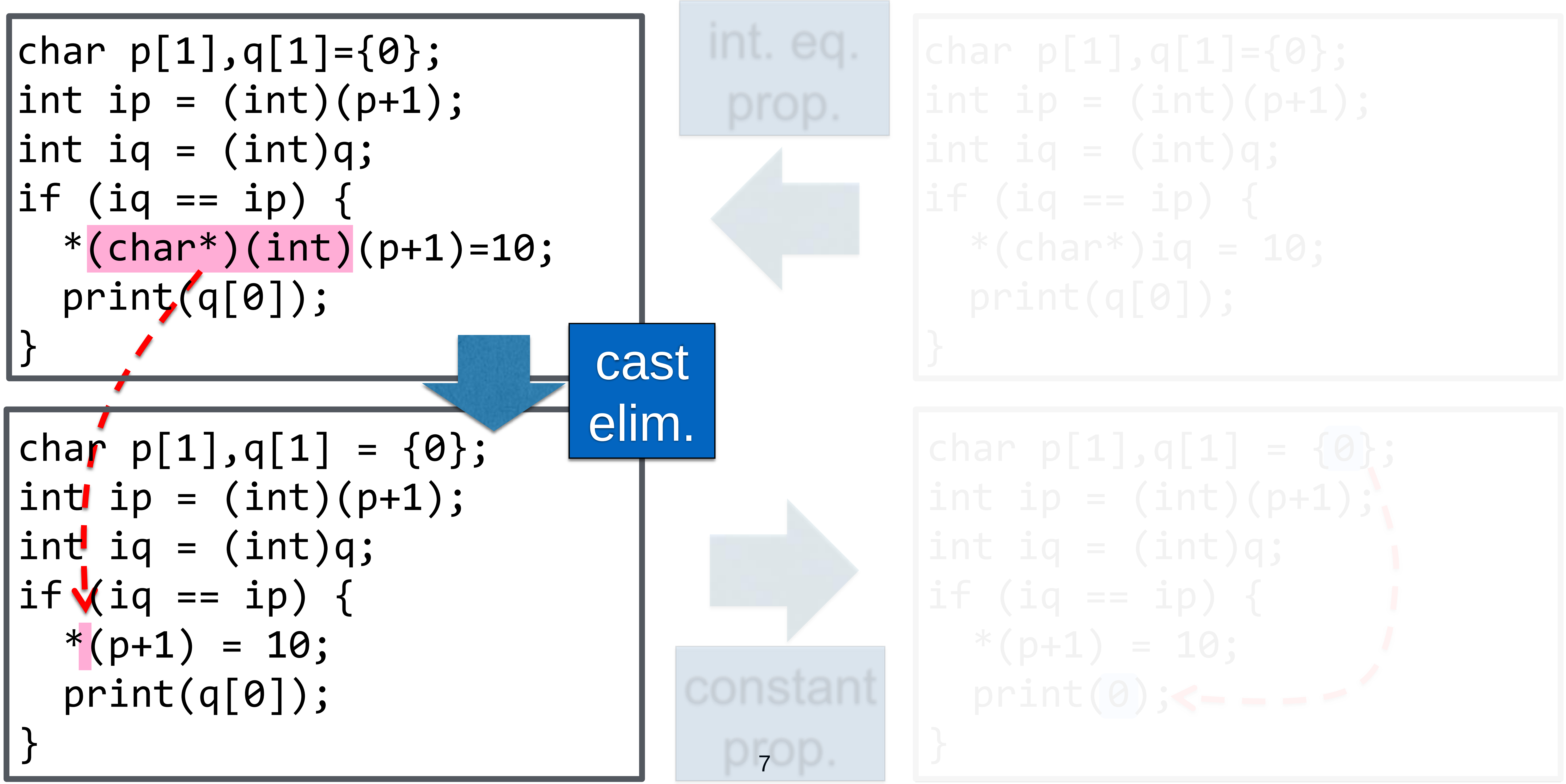
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

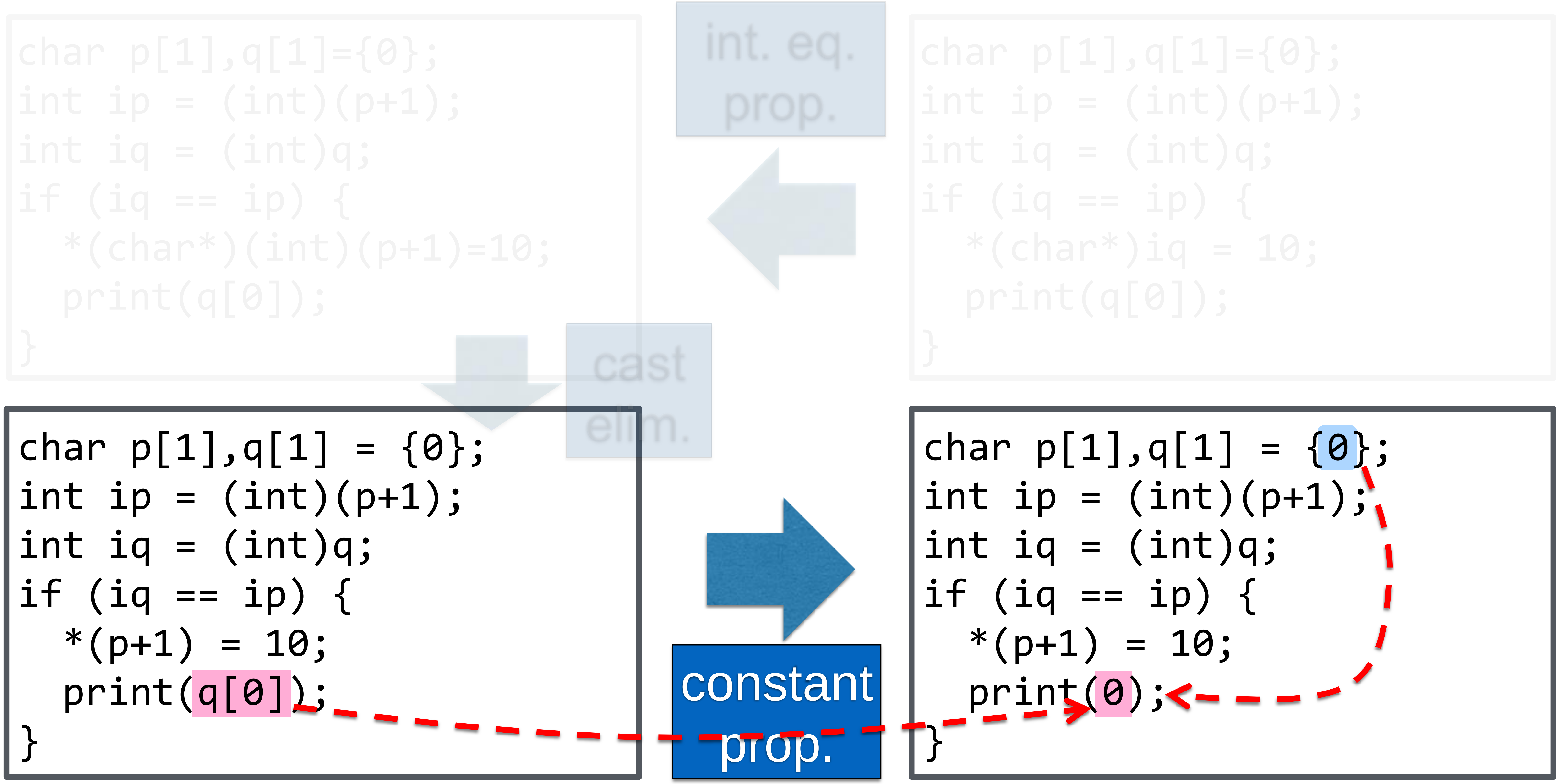
int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



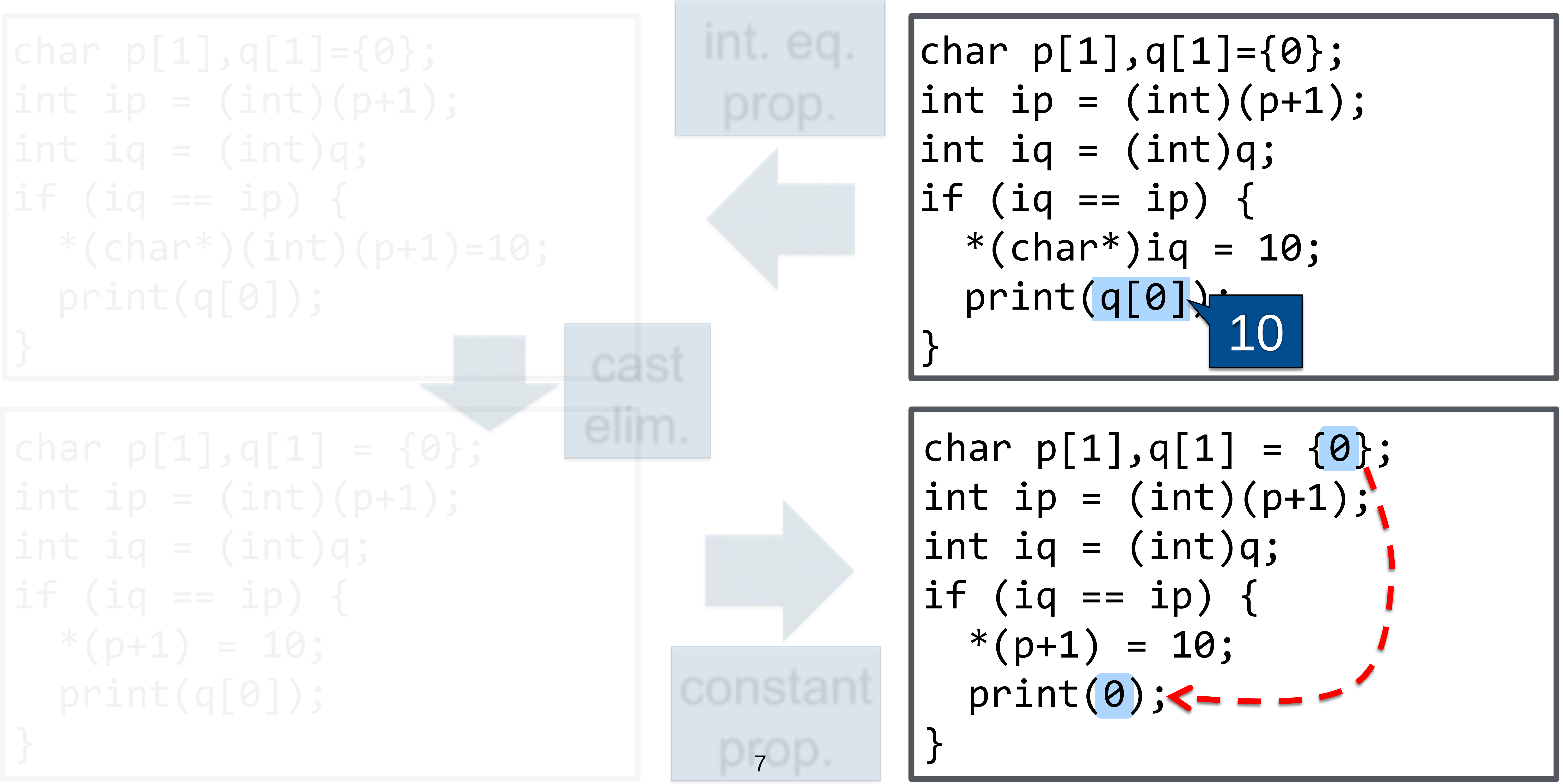
```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

10

Miscompilation with Int-Ptr Casting

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

10

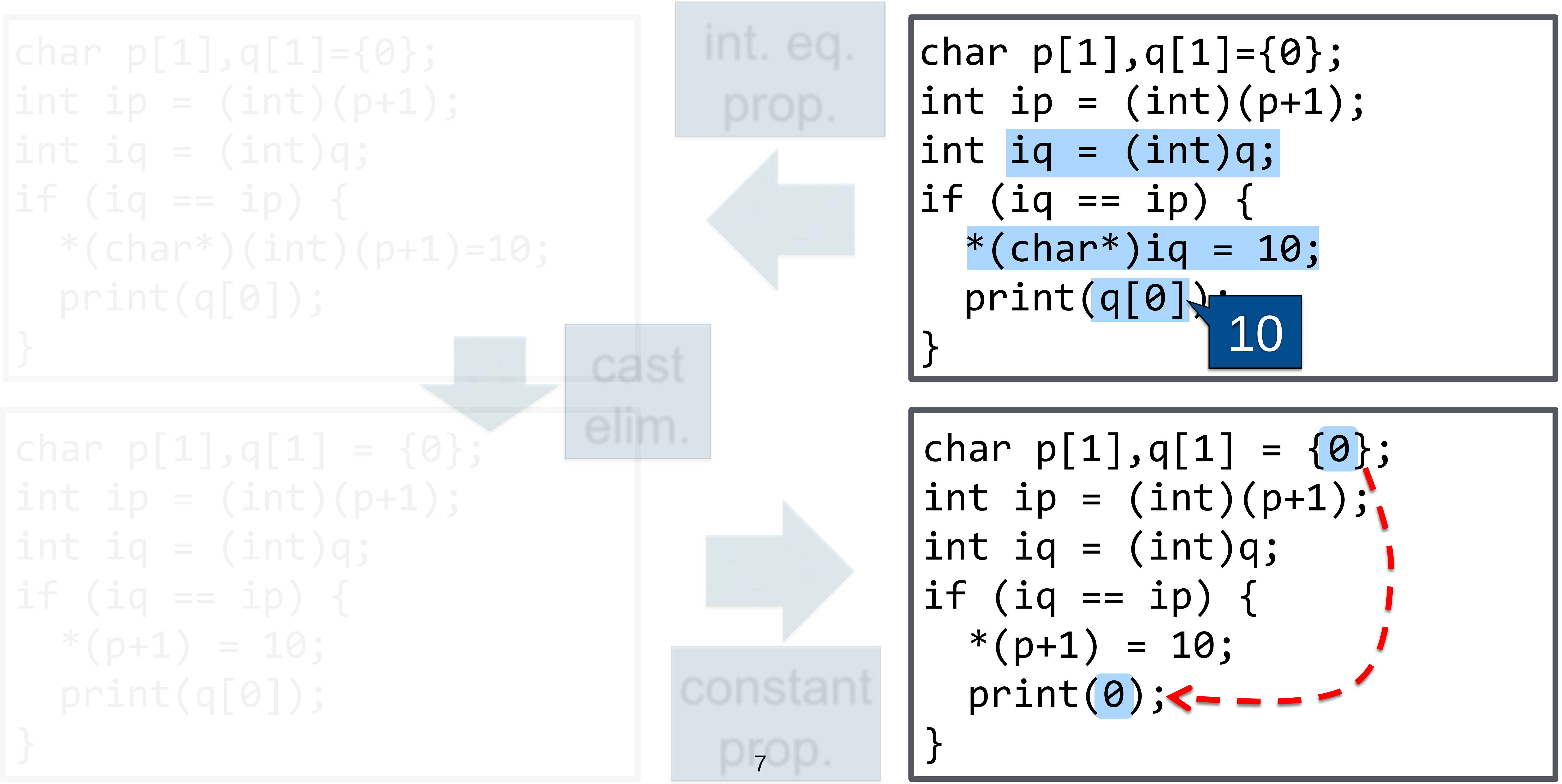
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting

```
cha
int
int
if
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

We found this miscompilation bug in both LLVM & GCC

```
  *(char*)iq = 10;
  print(q[0]);
}
```

10

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with Int-Ptr Casting

We found this miscompilation bug in both LLVM & GCC

Goal of this paper

Finding a good memory model for pointer ↔ integer casting

```
cha
int
int
if
  *(char*)(int)(p+1)=10;
  p
}
```

```
  *(char*)iq = 10;
```

```
cha
int
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

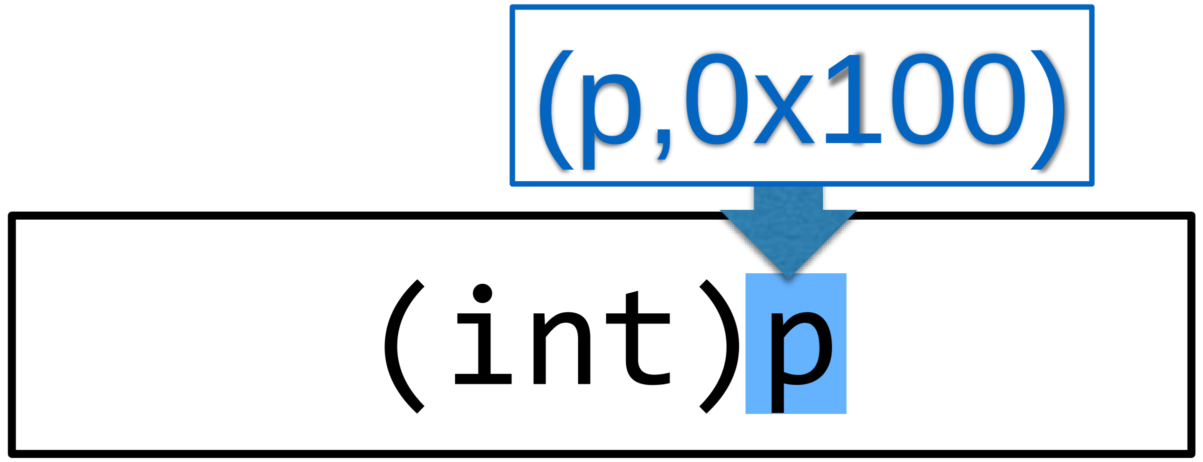
Problems &
Our Solutions

Problem 1

Pointer → Integer Casting?

```
(int)p
```

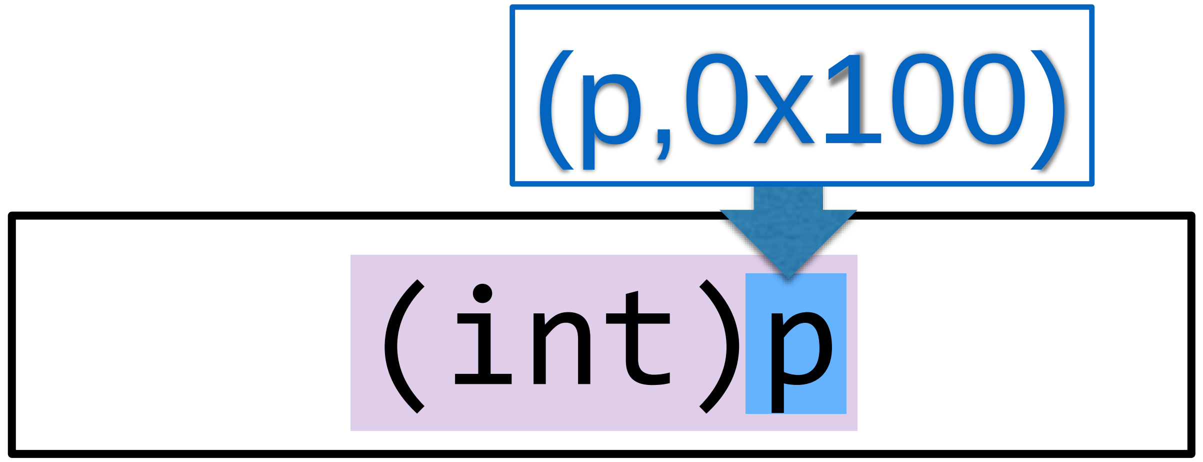
Pointer → Integer Casting?



(p,0x100)

```
(int)p
```

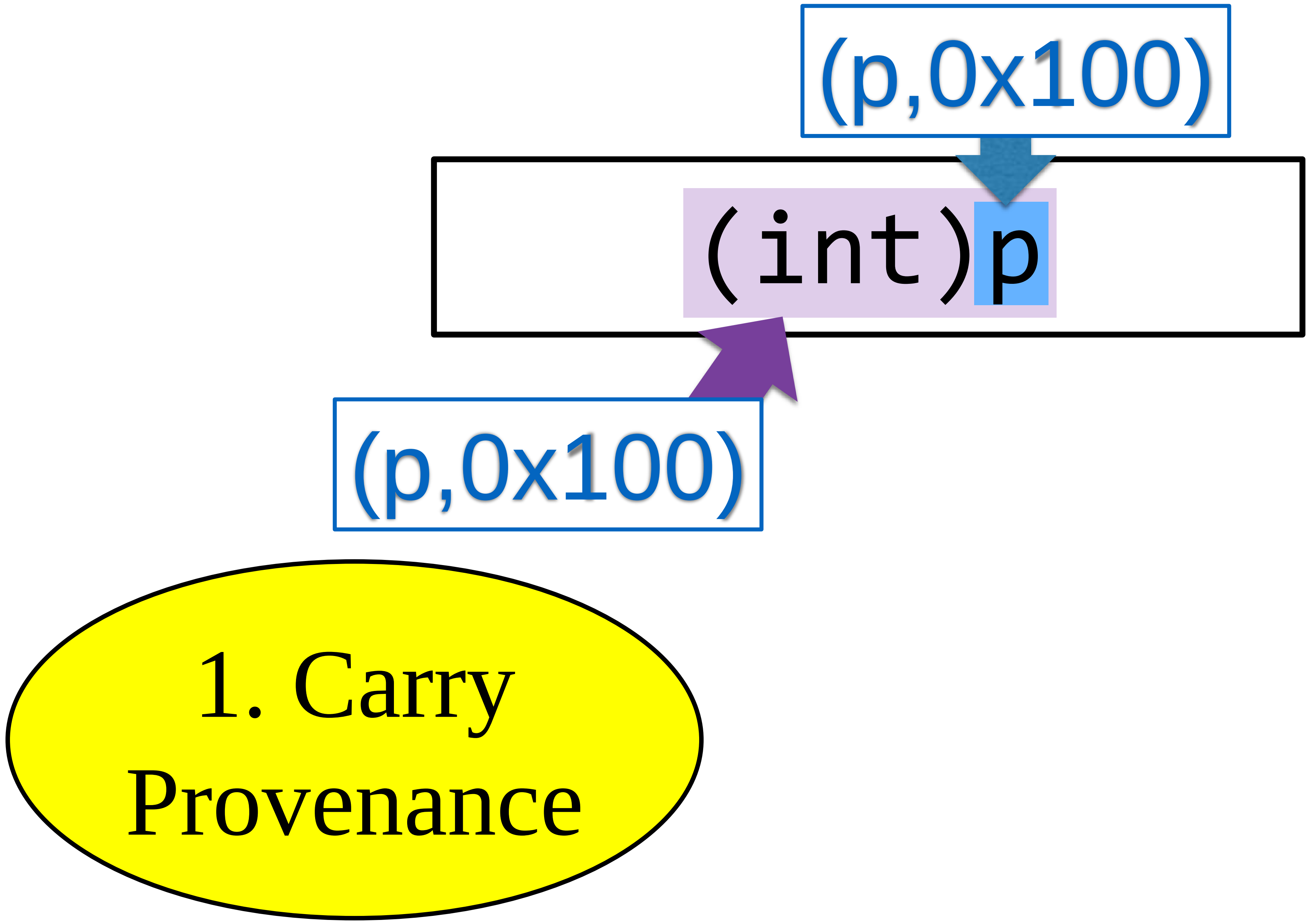
Pointer → Integer Casting?



(p,0x100)

```
(int)p
```

Pointer → Integer Casting?



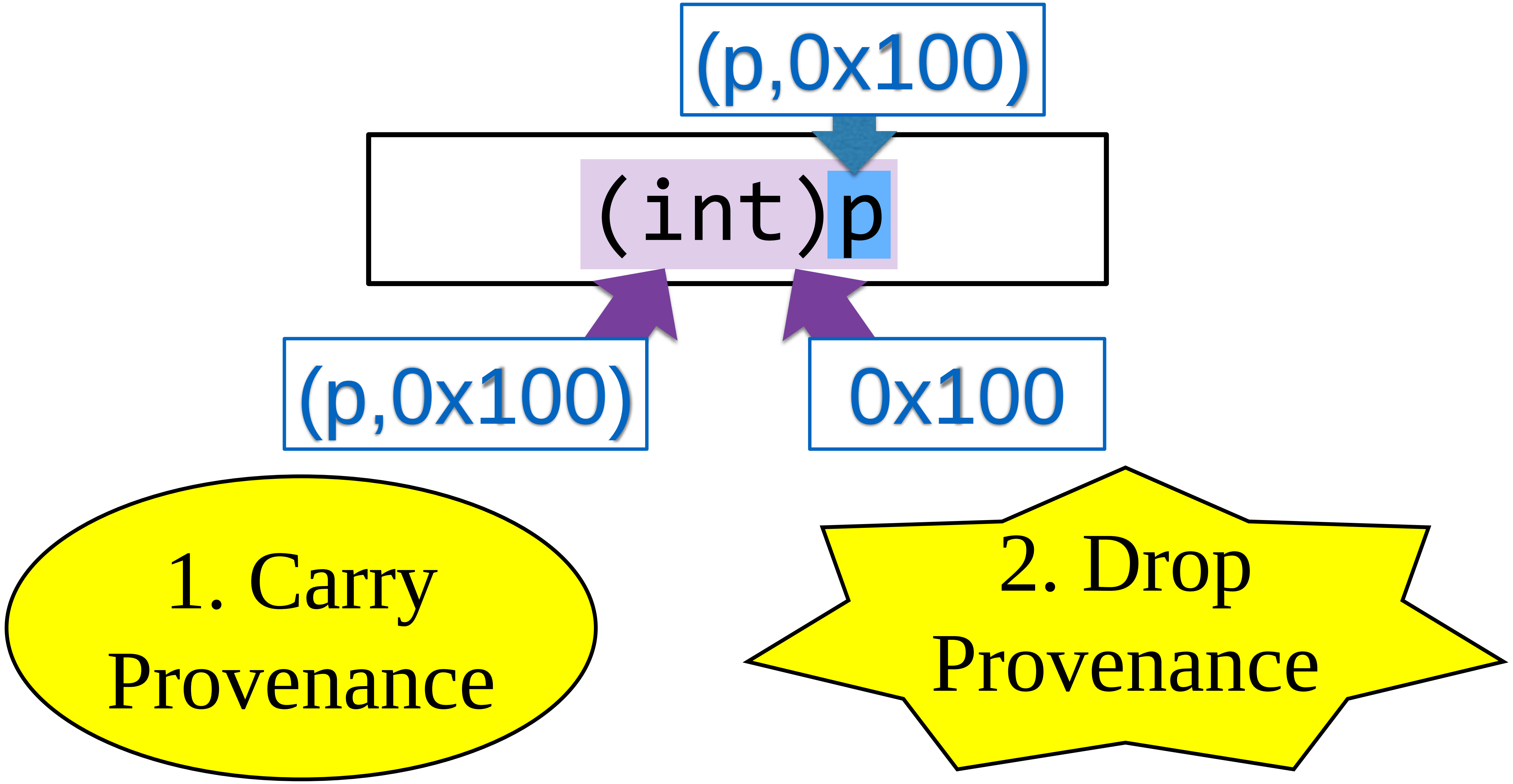
(p,0x100)

```
(int)p
```

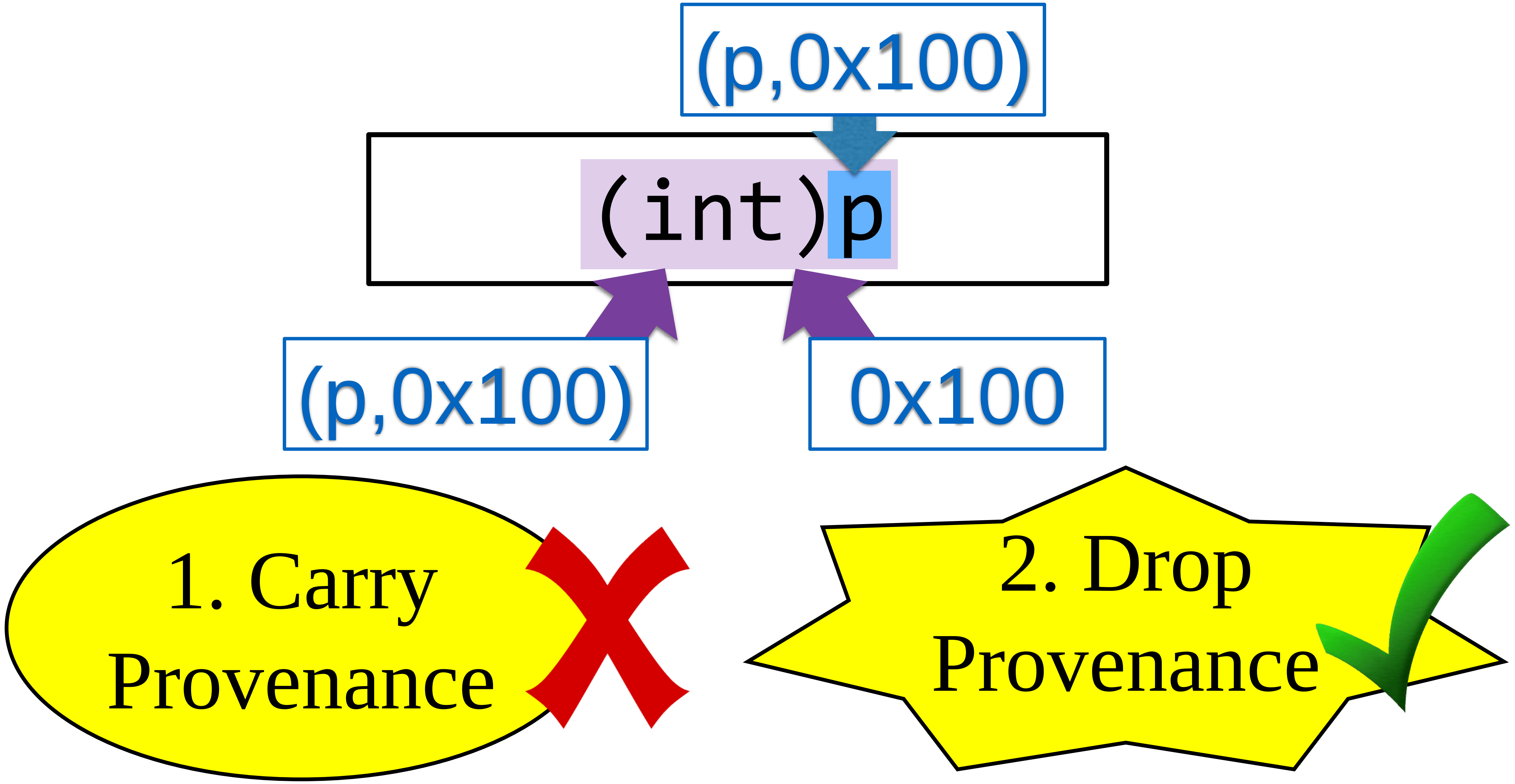
(p,0x100)

1. Carry Provenance

Pointer → Integer Casting?



Pointer → Integer Casting?



(p,0x100)

```
(int)p
```

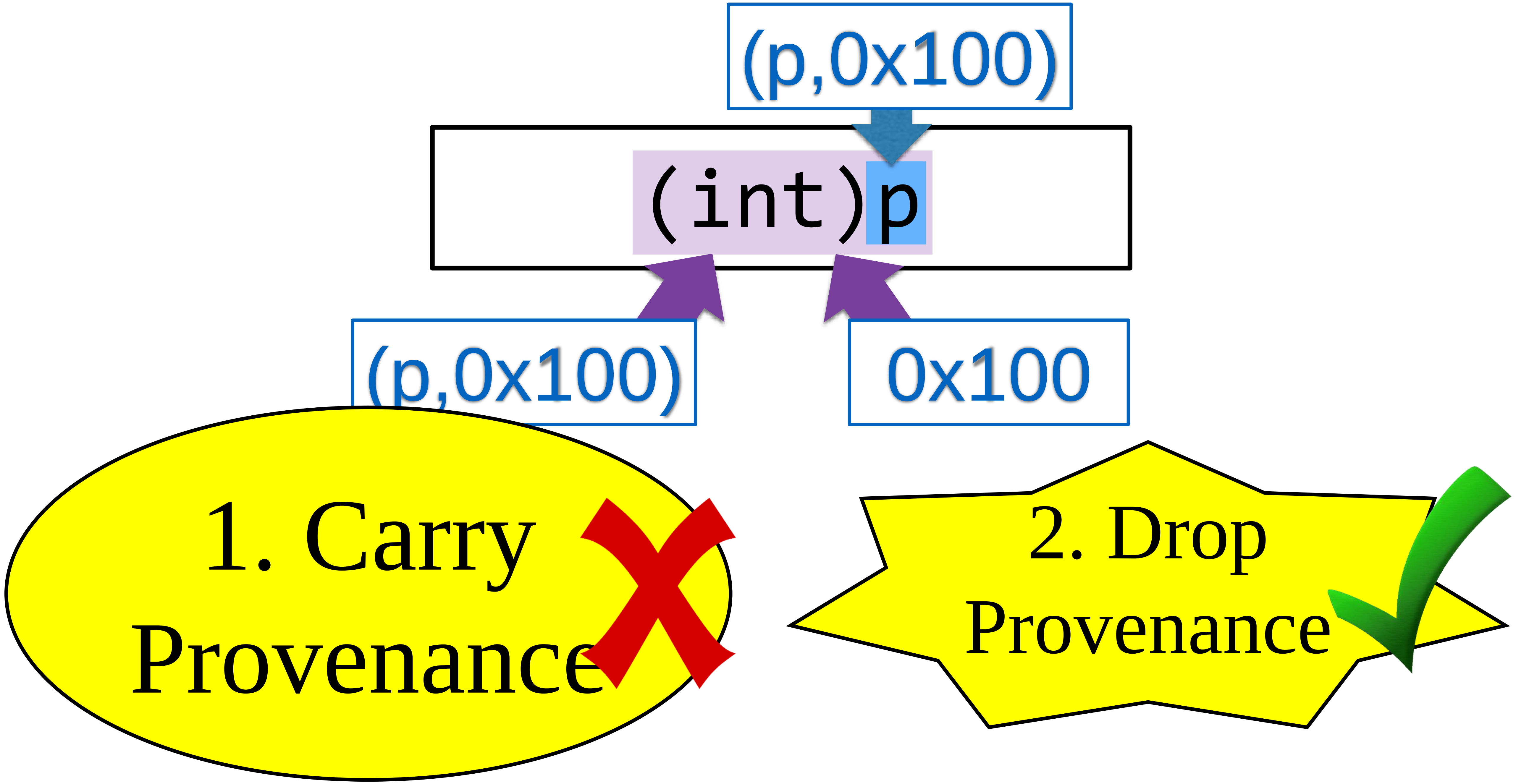
(p,0x100)

0x100

1. Carry Provenance ✗

2. Drop Provenance ✓

Pointer → Integer Casting?



(p,0x100)

`(int)p`

(p,0x100)

0x100

1. Carry Provenance ✗

2. Drop Provenance ✓

Carry Provenance: Integer Optimization Problem

1. Carry Provenance ✗

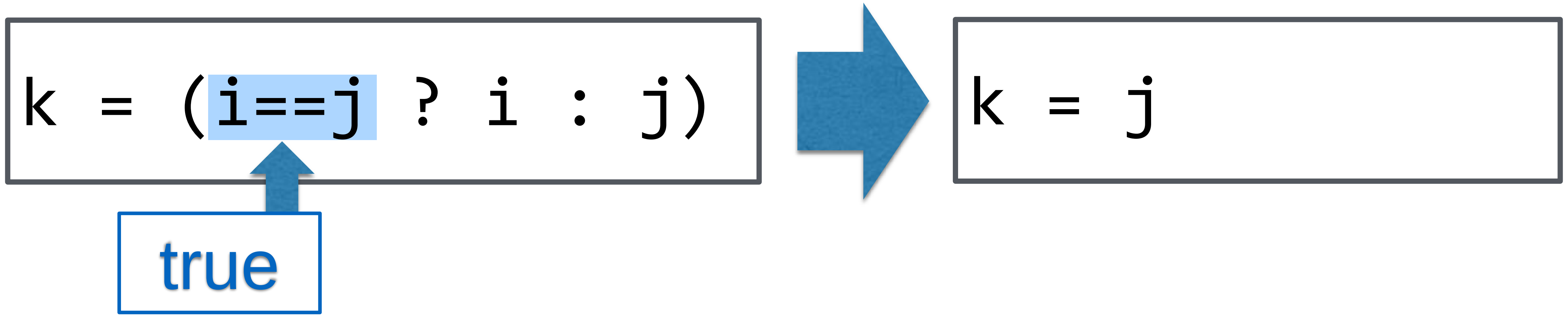
```
k = (i==j ? i : j)
```

→

```
k = j
```

Carry Provenance:
Integer Optimization Problem

1. Carry Provenance



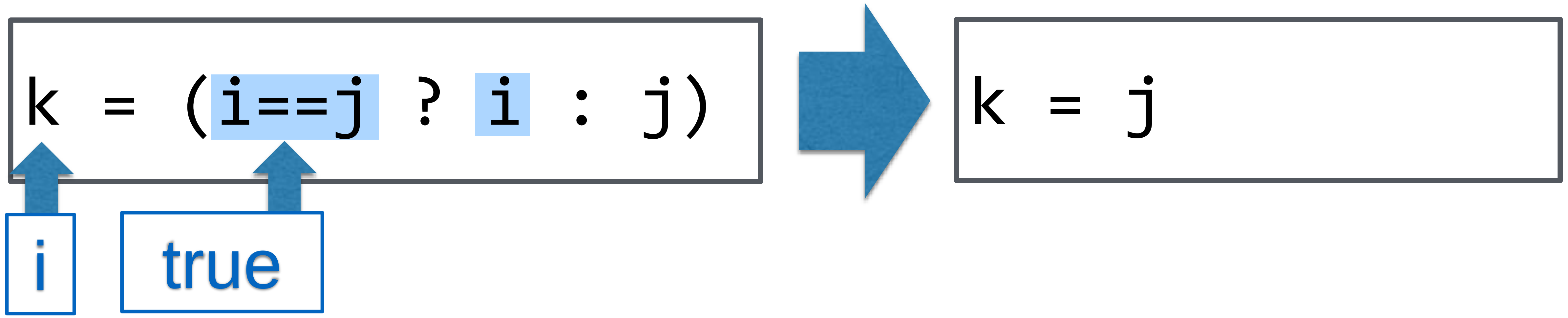
```
k = (i==j ? i : j)
```

true

```
k = j
```

Carry Provenance: Integer Optimization Problem

1. Carry Provenance

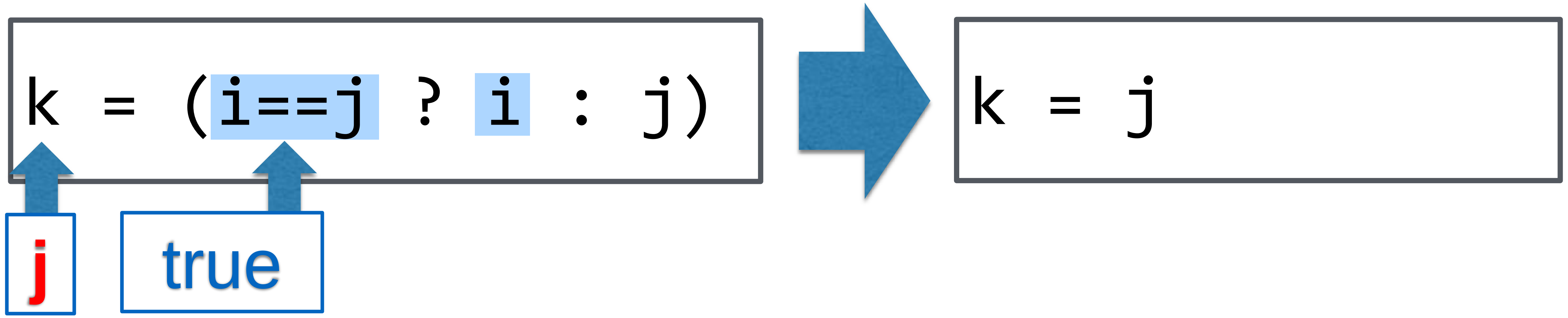


```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance: Integer Optimization Problem

1. Carry Provenance ✗

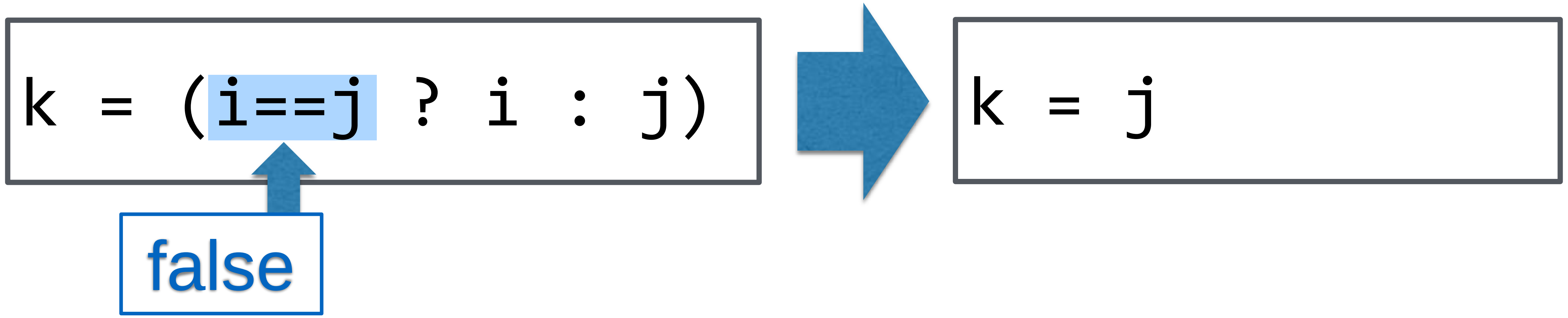


```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance:
Integer Optimization Problem

1. Carry Provenance



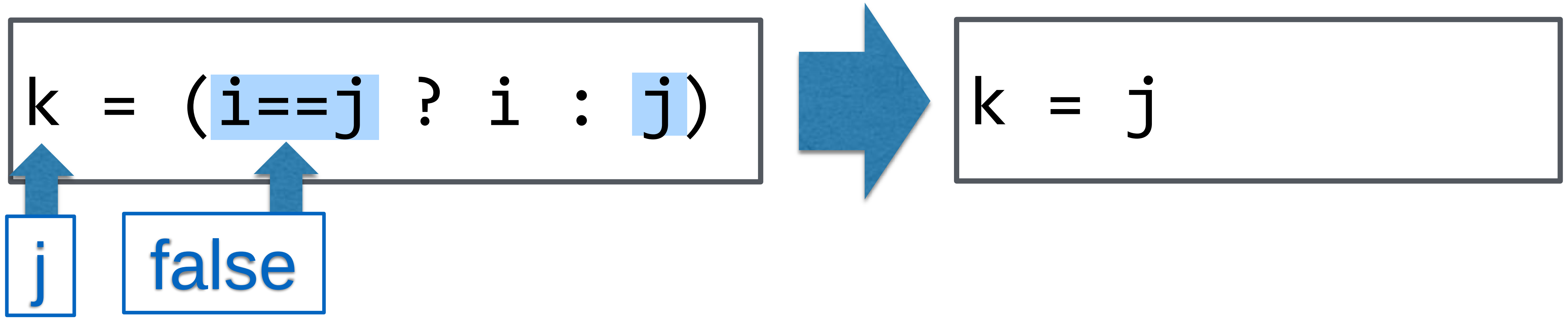
```
k = (i==j ? i : j)
```

false

```
k = j
```

Carry Provenance:
Integer Optimization Problem

1. Carry Provenance

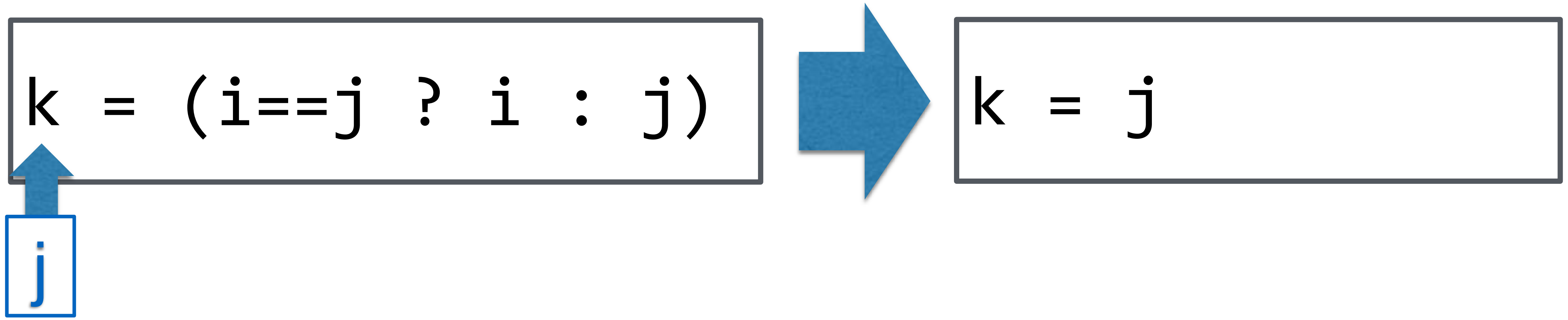


```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance:
Integer Optimization Problem

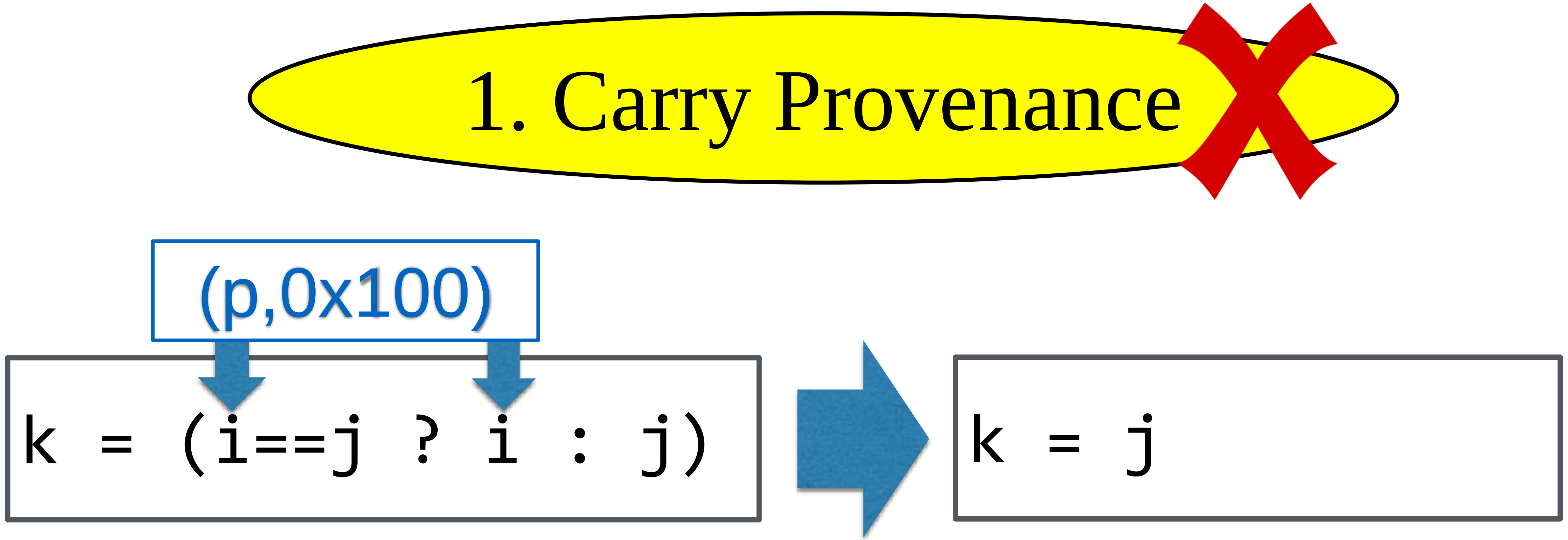
1. Carry Provenance



```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance:
Integer Optimization Problem



1. Carry Provenance

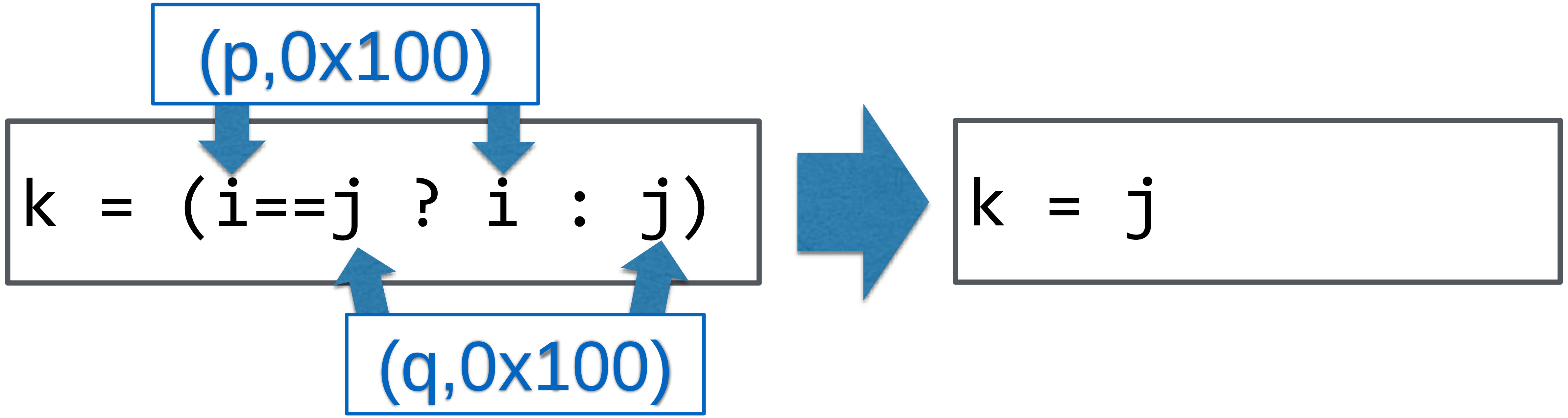
(p,0x100)

```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance: Integer Optimization Problem

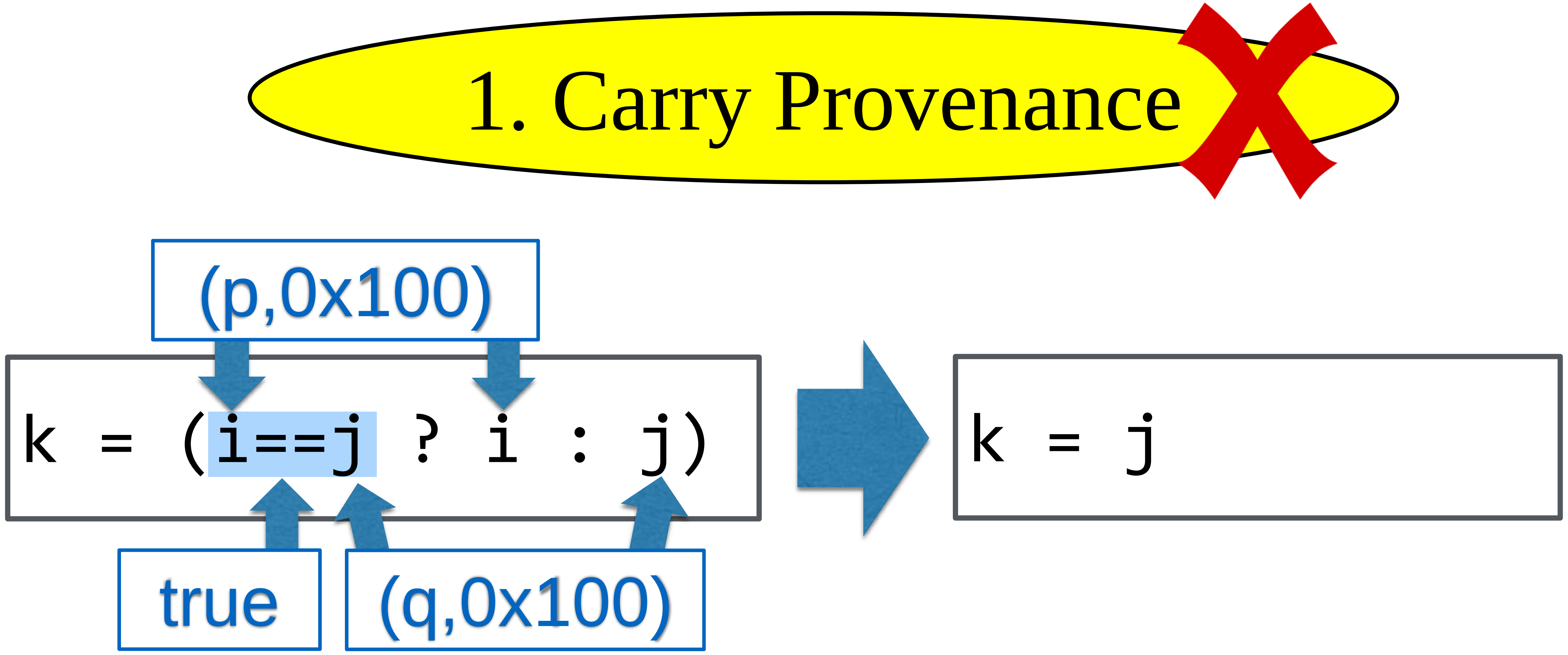
1. Carry Provenance ✗



```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance:
Integer Optimization Problem

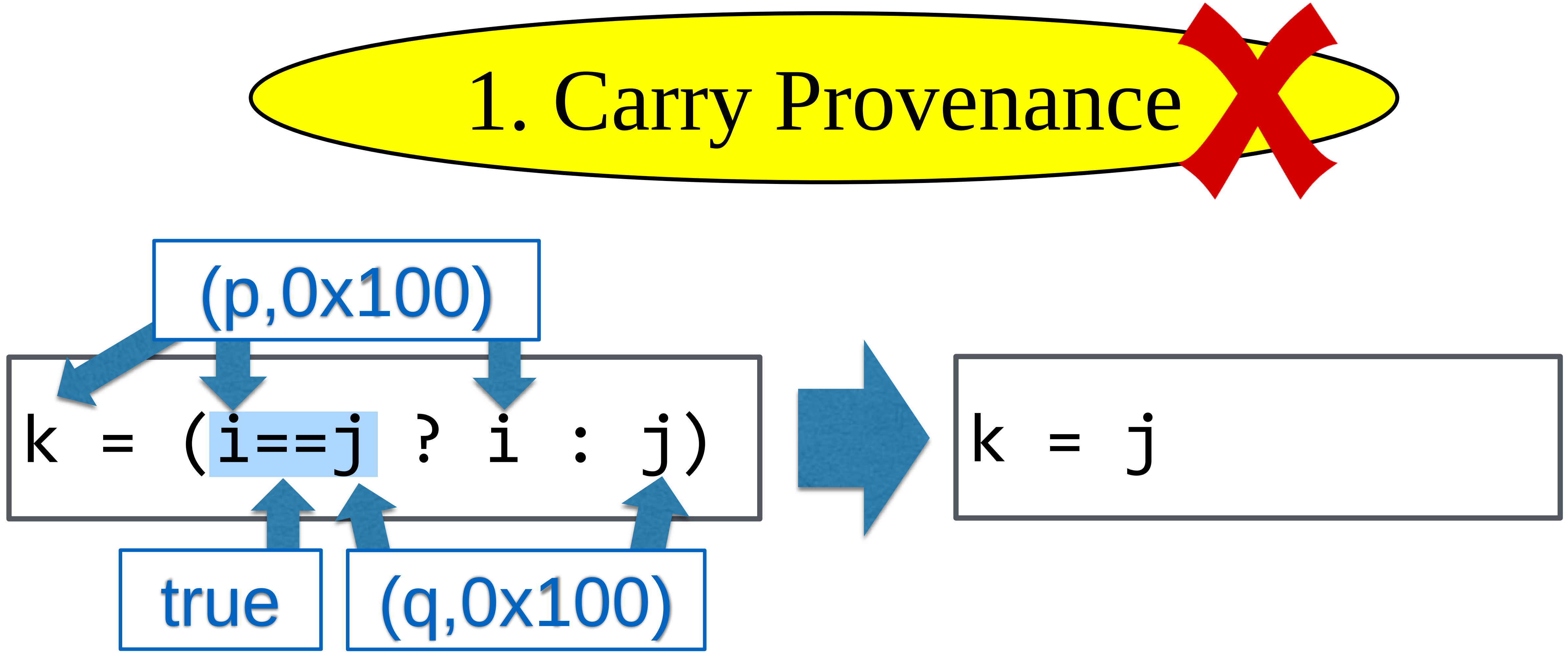


1. Carry Provenance

```
k = (i==j ? i : j)
```

```
k = j
```

Carry Provenance: Integer Optimization Problem

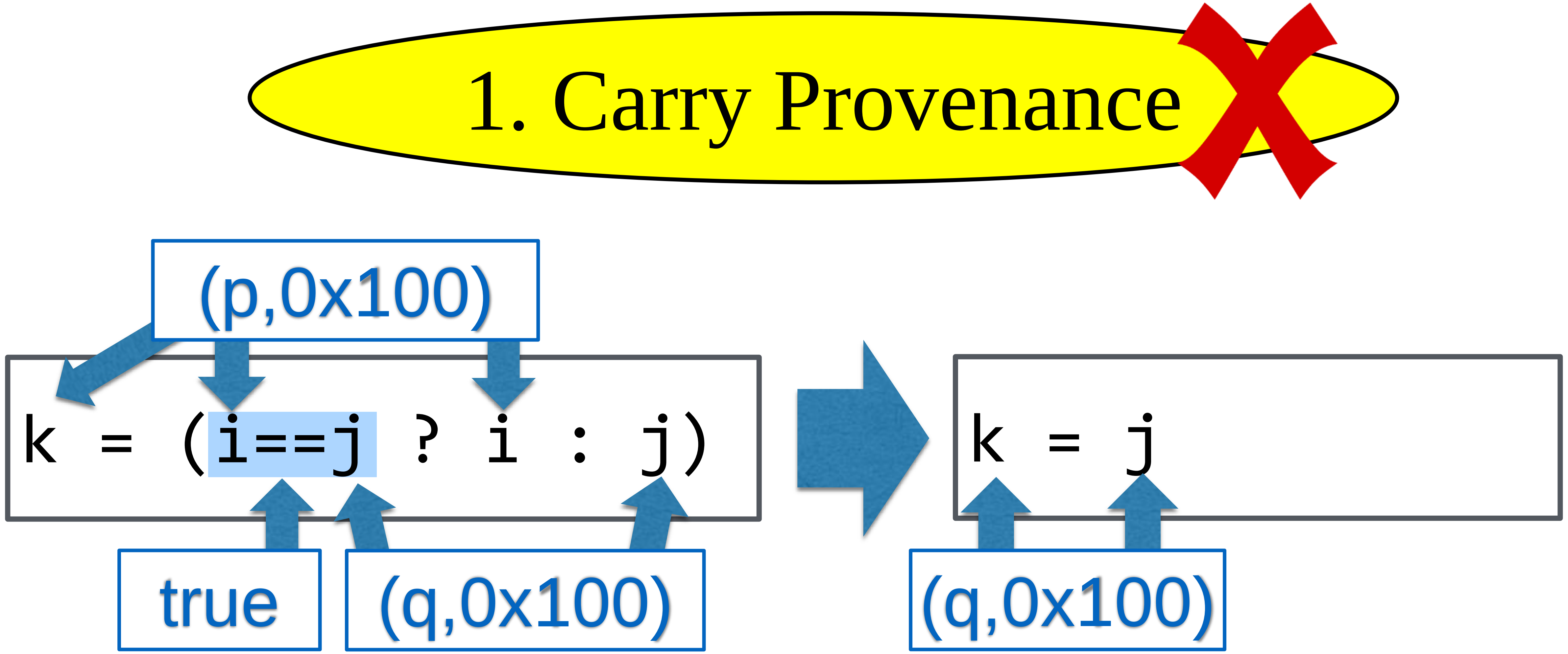


1. Carry Provenance

```
k = (i==j ? i : j)
```

```
k = j
```

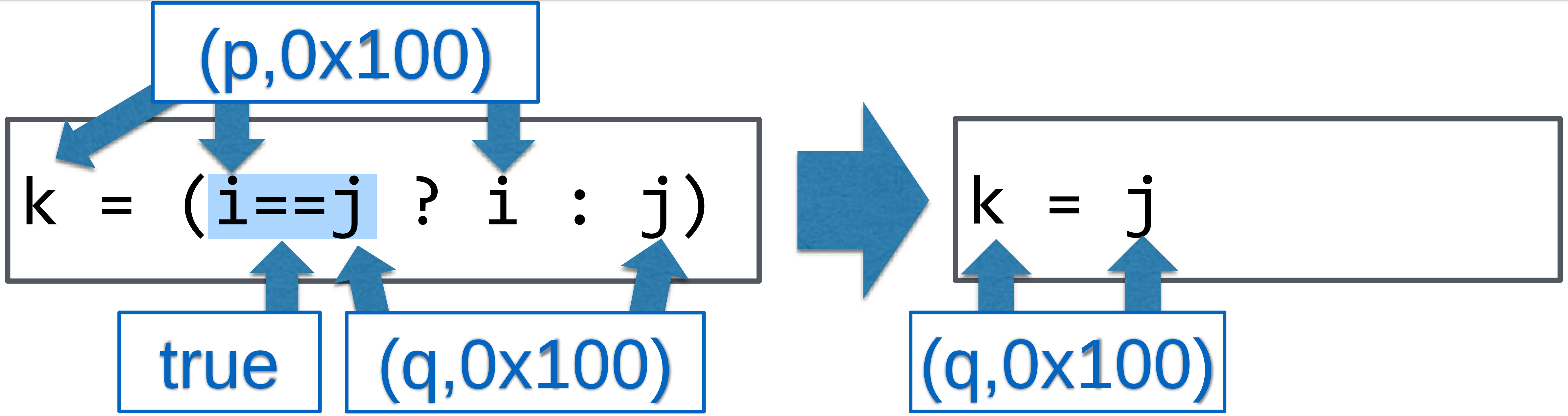
Carry Provenance: Integer Optimization Problem



Carry Provenance: Integer Optimization Problem

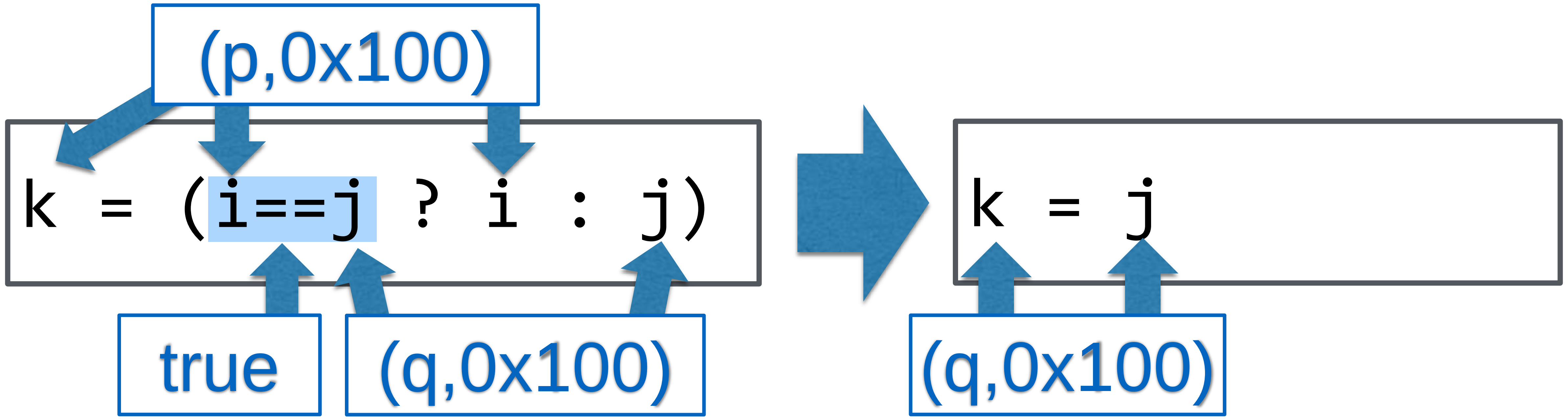
Problem

Integer optimizations may change provenance



Carry Provenance: Integer Optimization Problem

2. Drop Provenance ✓

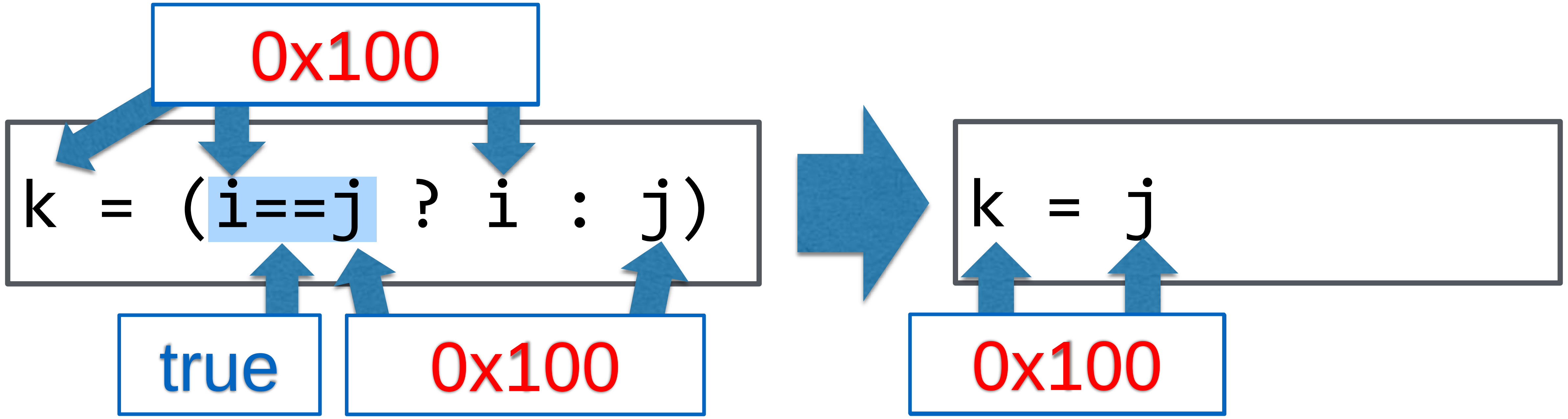


```
k = (i==j ? i : j)
```

```
k = j
```

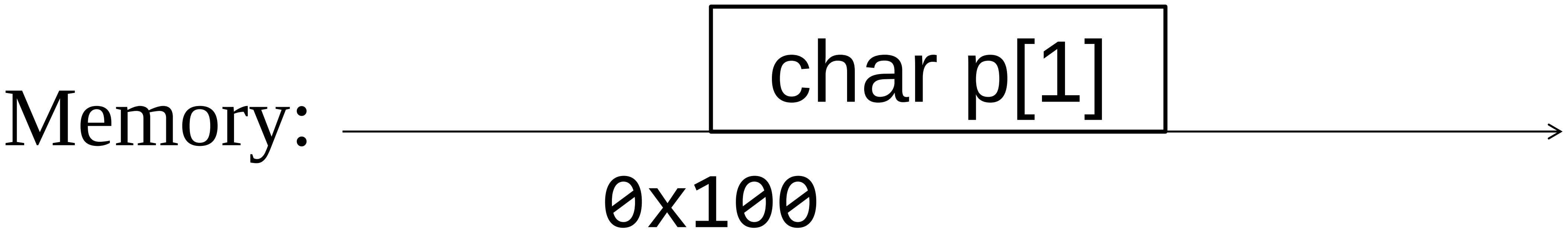
Carry Provenance: Integer Optimization Problem

2. Drop Provenance ✓



Problem 2

Integer → Pointer Casting?



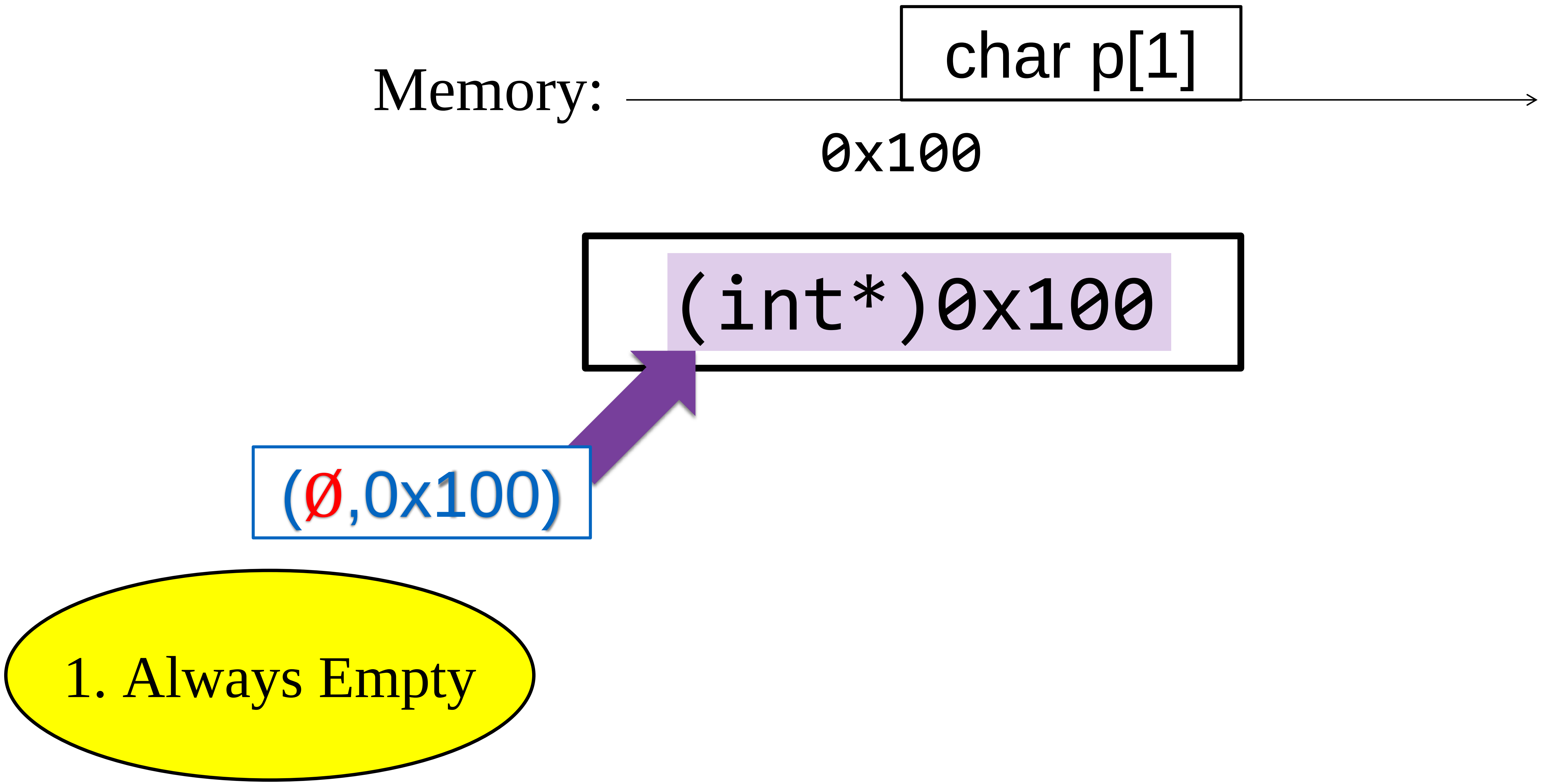
Memory:

char p[1]

0x100

```
(int*)0x100
```

Integer → Pointer Casting?



Memory:

char p[1]

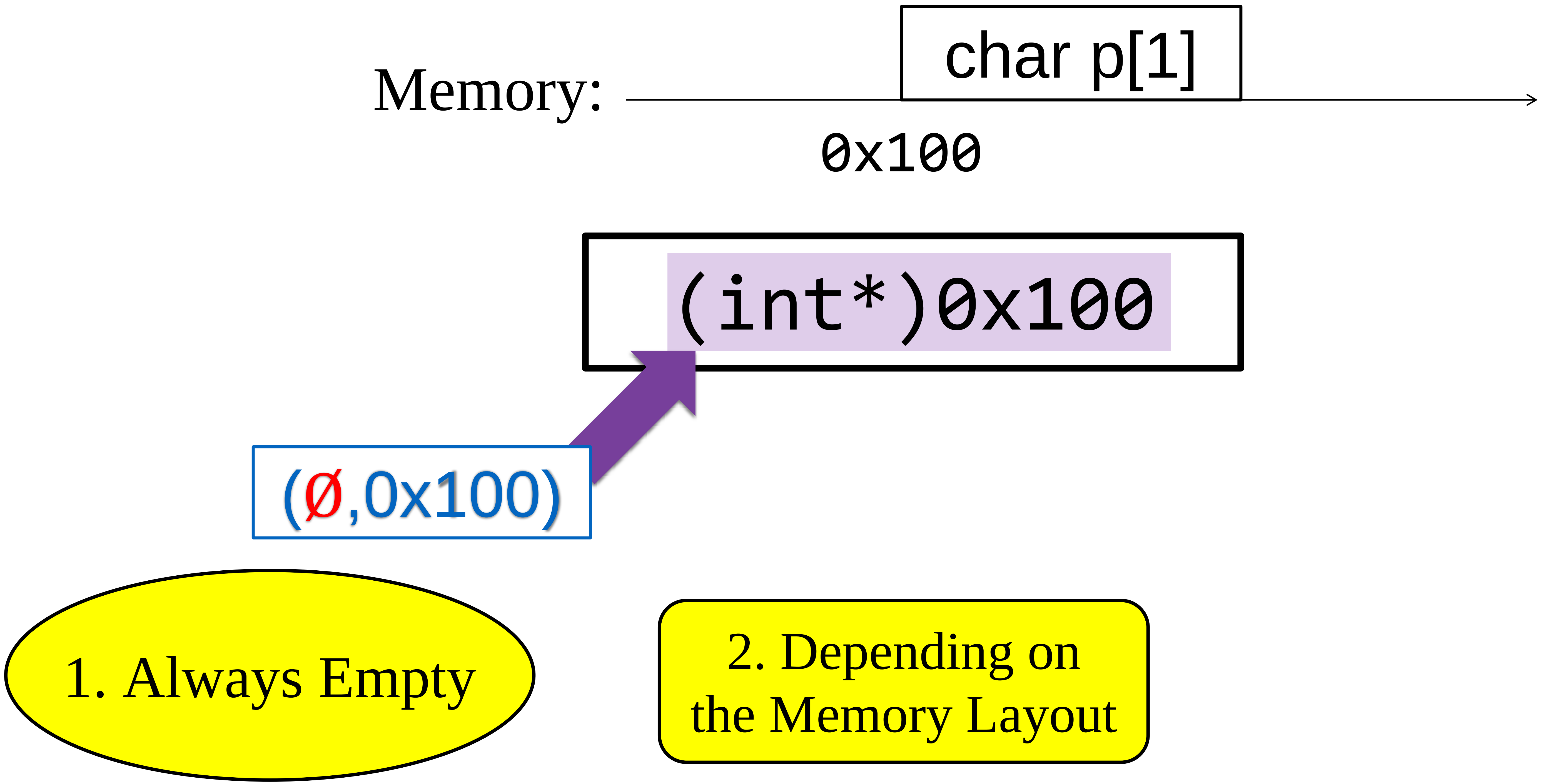
0x100

`(int*)0x100`

(Ø,0x100)

1. Always Empty

Integer → Pointer Casting?



Memory:

char p[1]

0x100

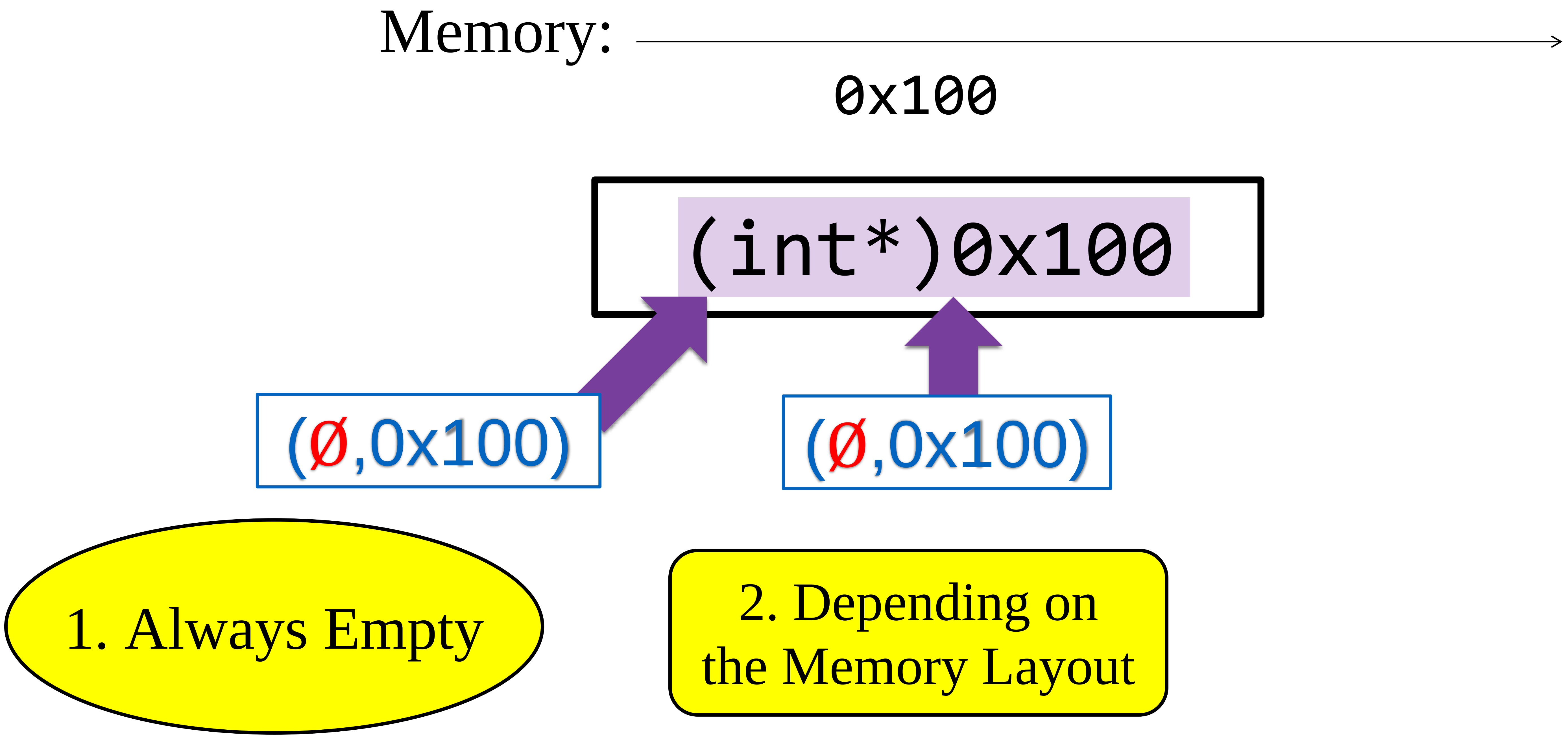
`(int*)0x100`

(Ø,0x100)

1. Always Empty

2. Depending on the Memory Layout

Integer → Pointer Casting?



Memory:

0x100

`(int*)0x100`

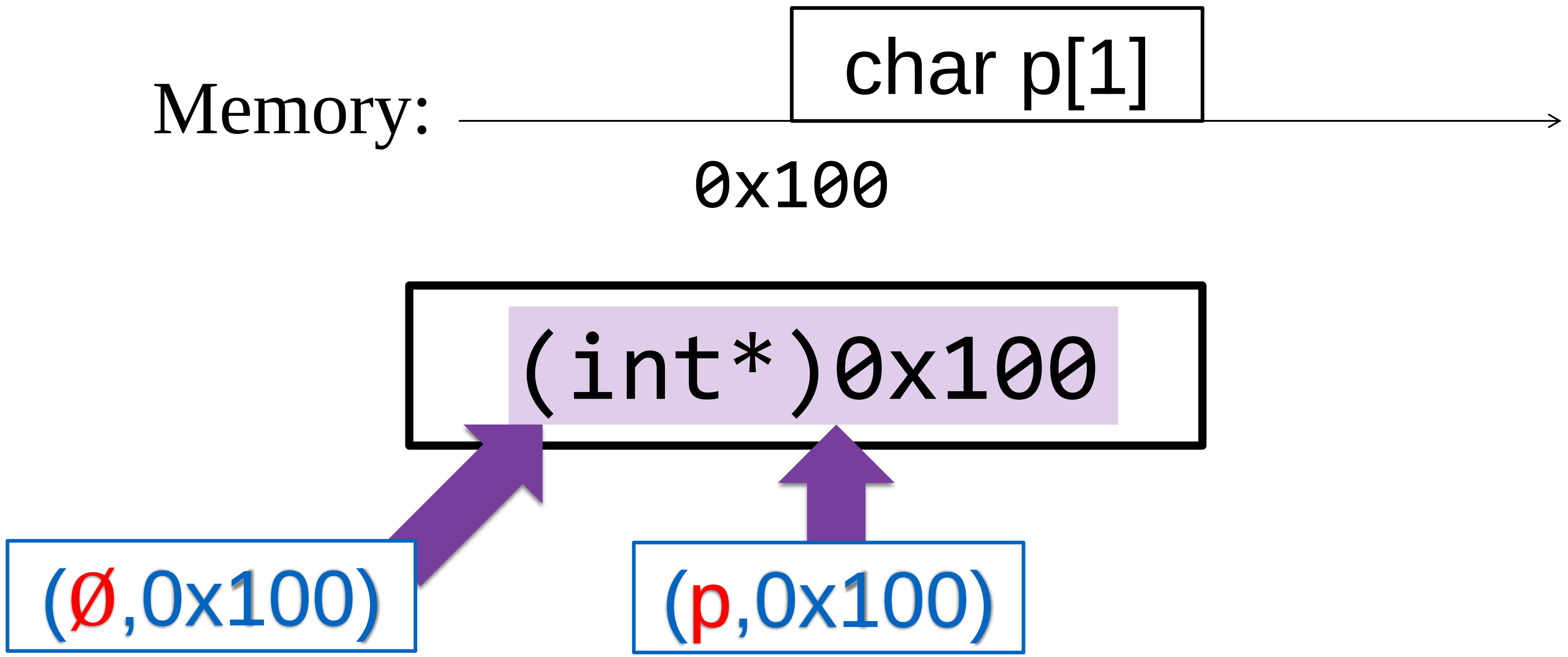
(Ø,0x100)

(Ø,0x100)

1. Always Empty

2. Depending on the Memory Layout

Integer → Pointer Casting?



Memory:

char p[1]

0x100

(int*)0x100

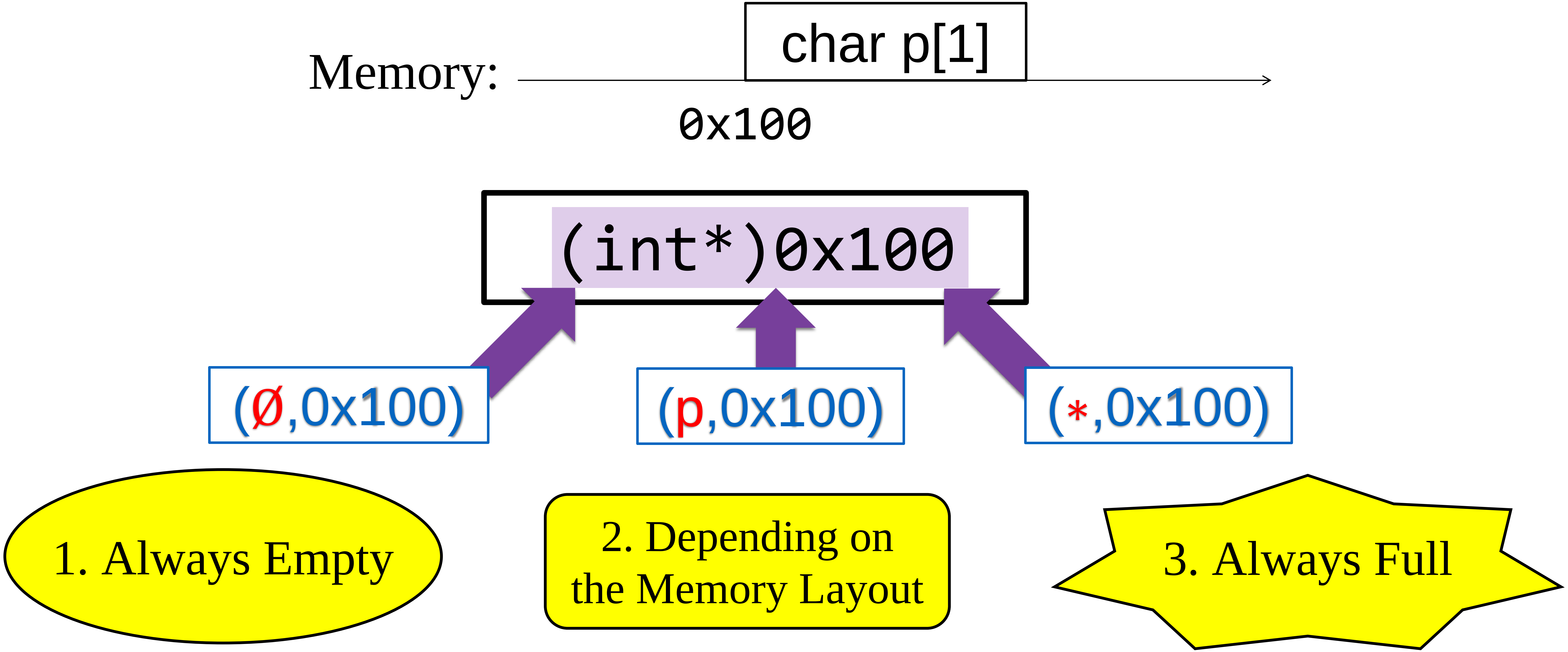
(Ø,0x100)

(p,0x100)

1. Always Empty

2. Depending on the Memory Layout

Integer → Pointer Casting?



Memory:

char p[1]

`0x100`

`(int*)0x100`

(Ø,0x100)

(p,0x100)

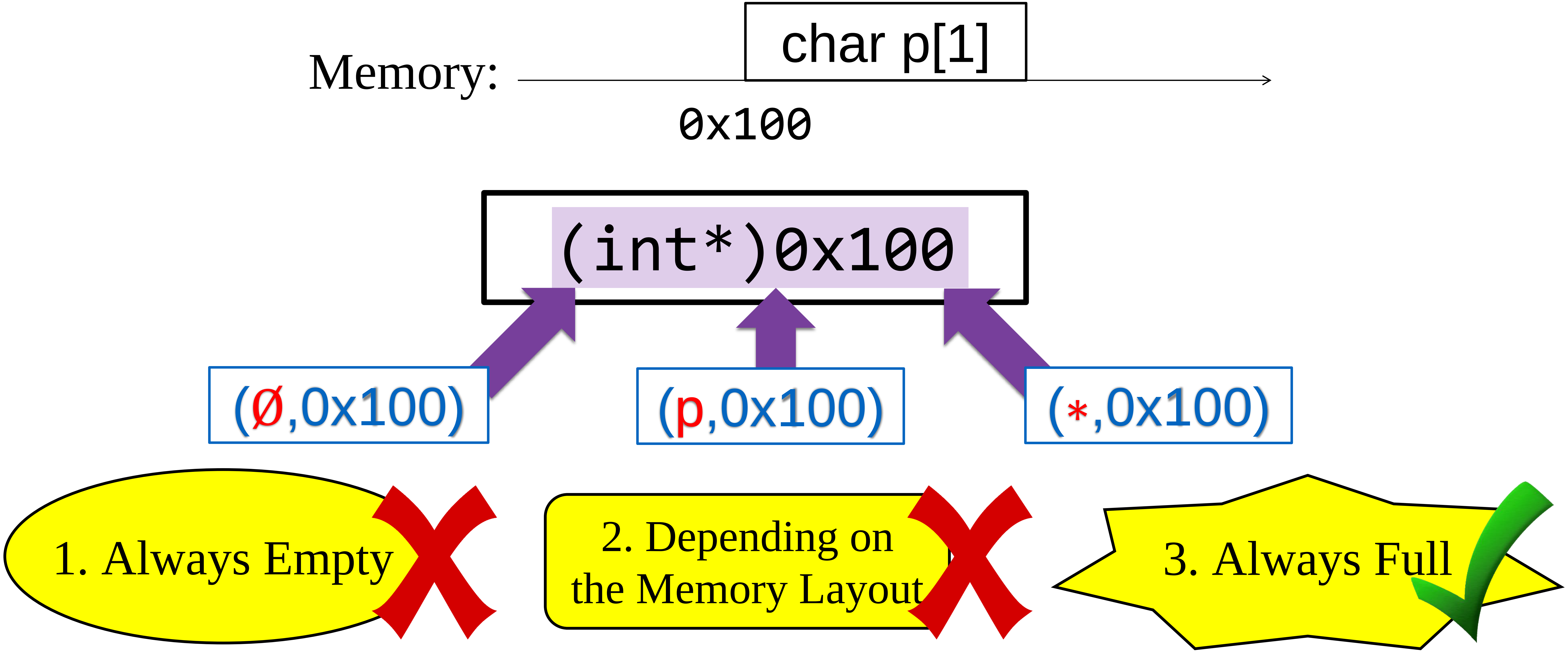
(*,0x100)

1. Always Empty

2. Depending on the Memory Layout

3. Always Full

Integer → Pointer Casting?



Memory:

char p[1]

0x100

`(int*)0x100`

(Ø,0x100)

(p,0x100)

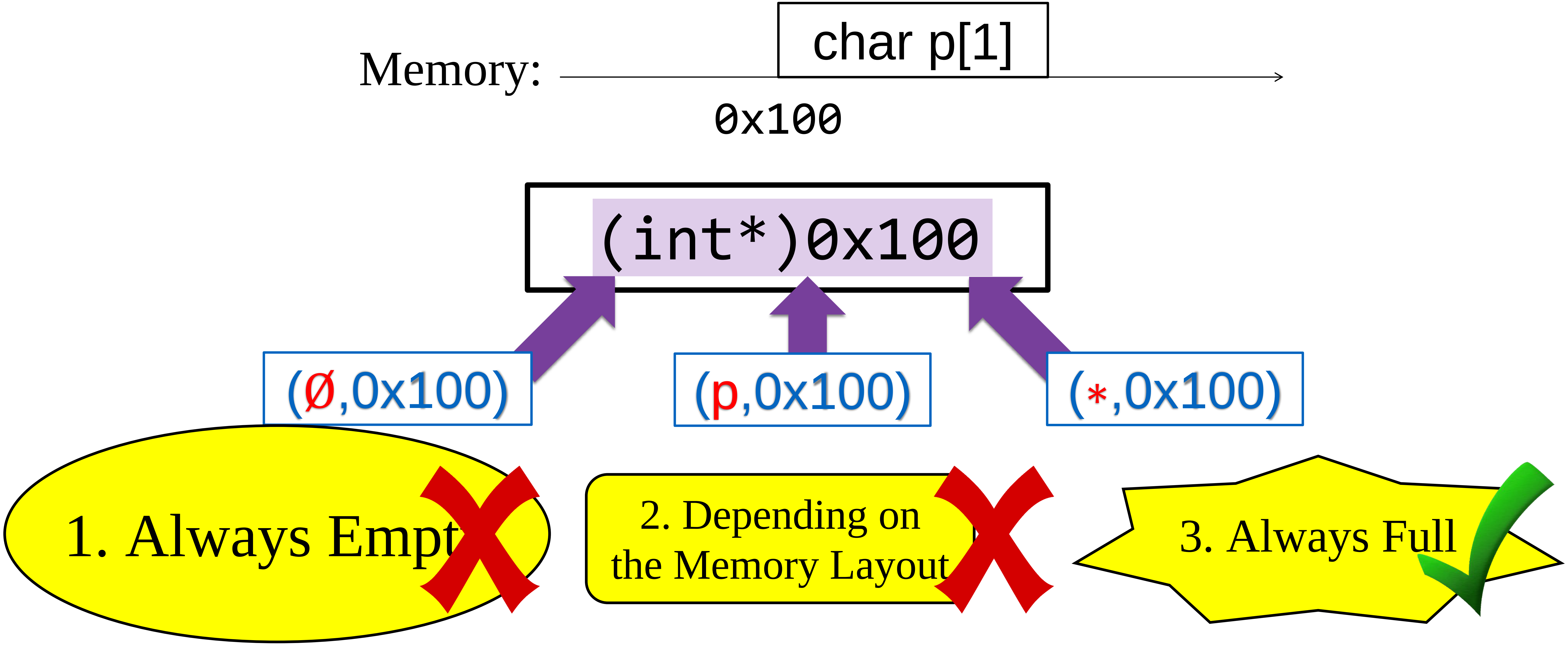
(*,0x100)

1. Always Empty ✗

2. Depending on the Memory Layout ✗

3. Always Full ✓

Integer → Pointer Casting?



Memory:

char p[1]

0x100

(int*)0x100

(Ø,0x100)

(p,0x100)

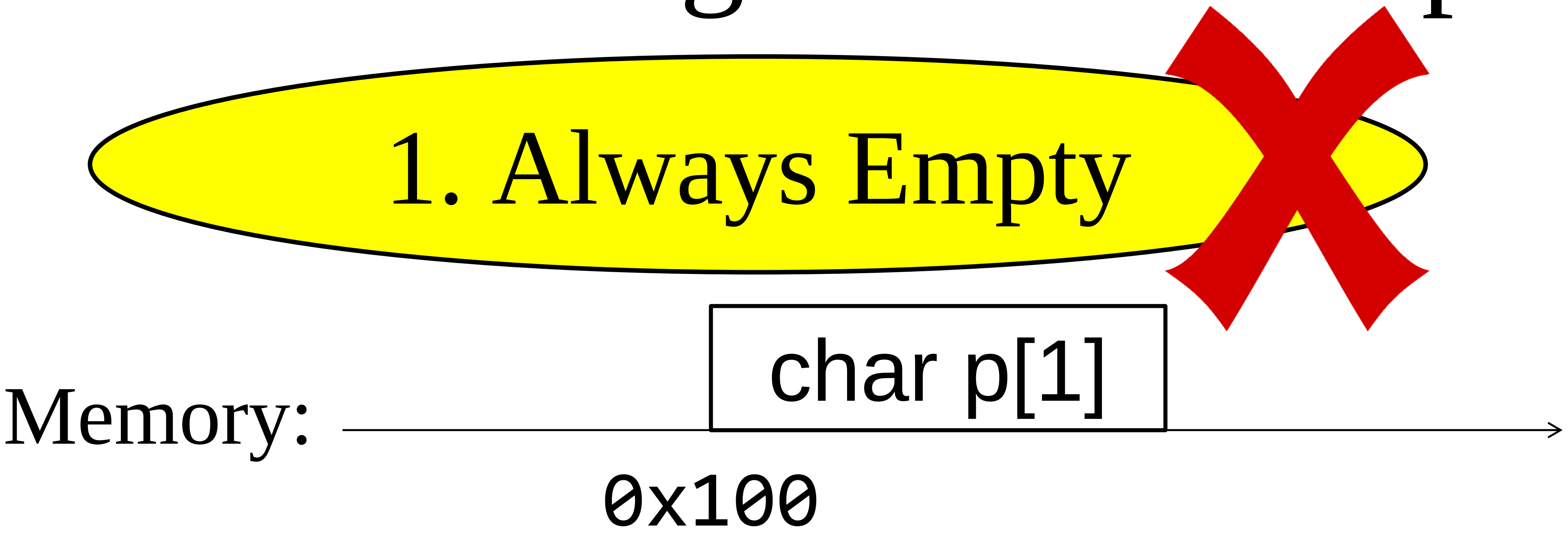
(*,0x100)

1. Always Empt

2. Depending on the Memory Layout

3. Always Full

Empty Provenance:
Pointer – Integer Round Trip



1. Always Empty

Memory:

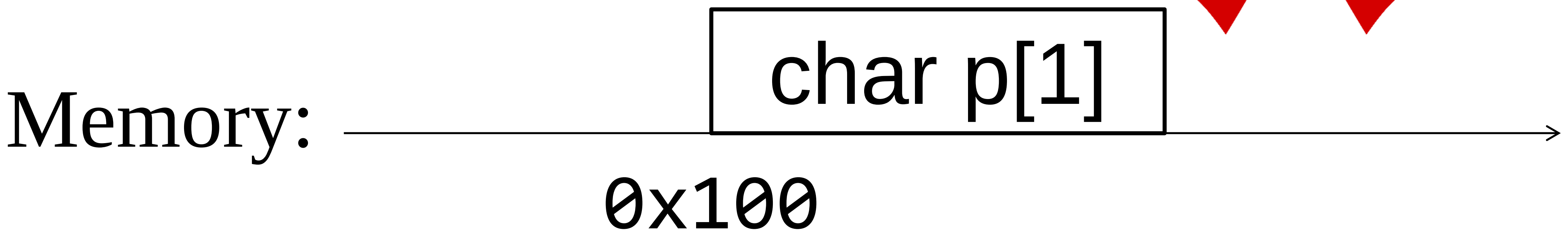
char p[1]

0x100

```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

Empty Provenance:
Pointer – Integer Round Trip

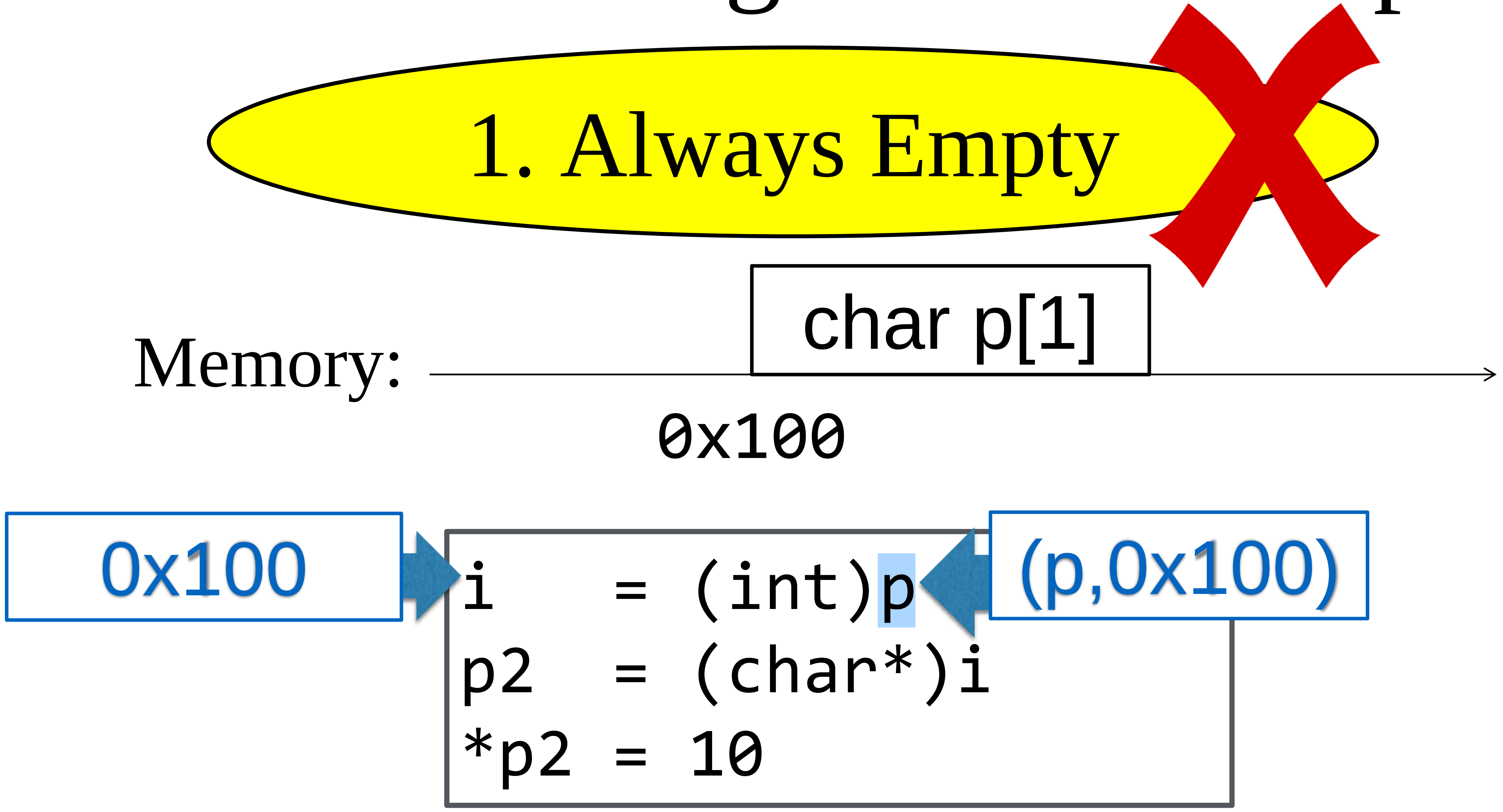
1. Always Empty



```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

(p,0x100)

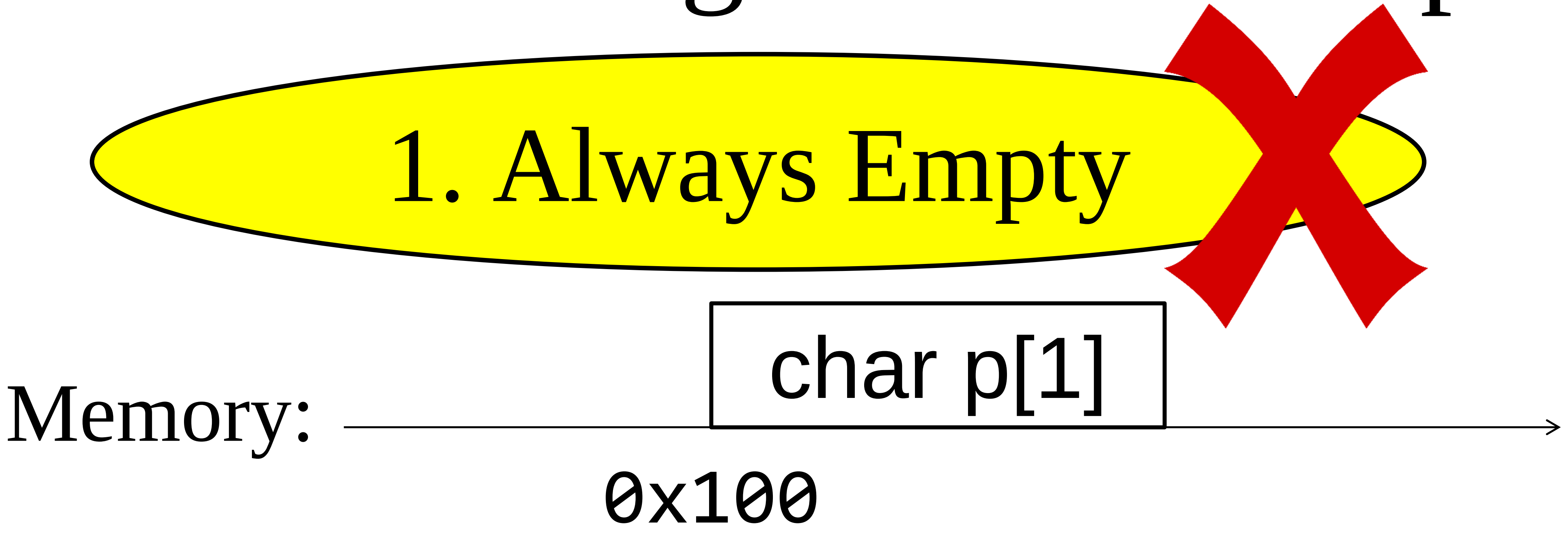
Empty Provenance:
Pointer – Integer Round Trip



1. Always Empty

```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

Empty Provenance: Pointer – Integer Round Trip



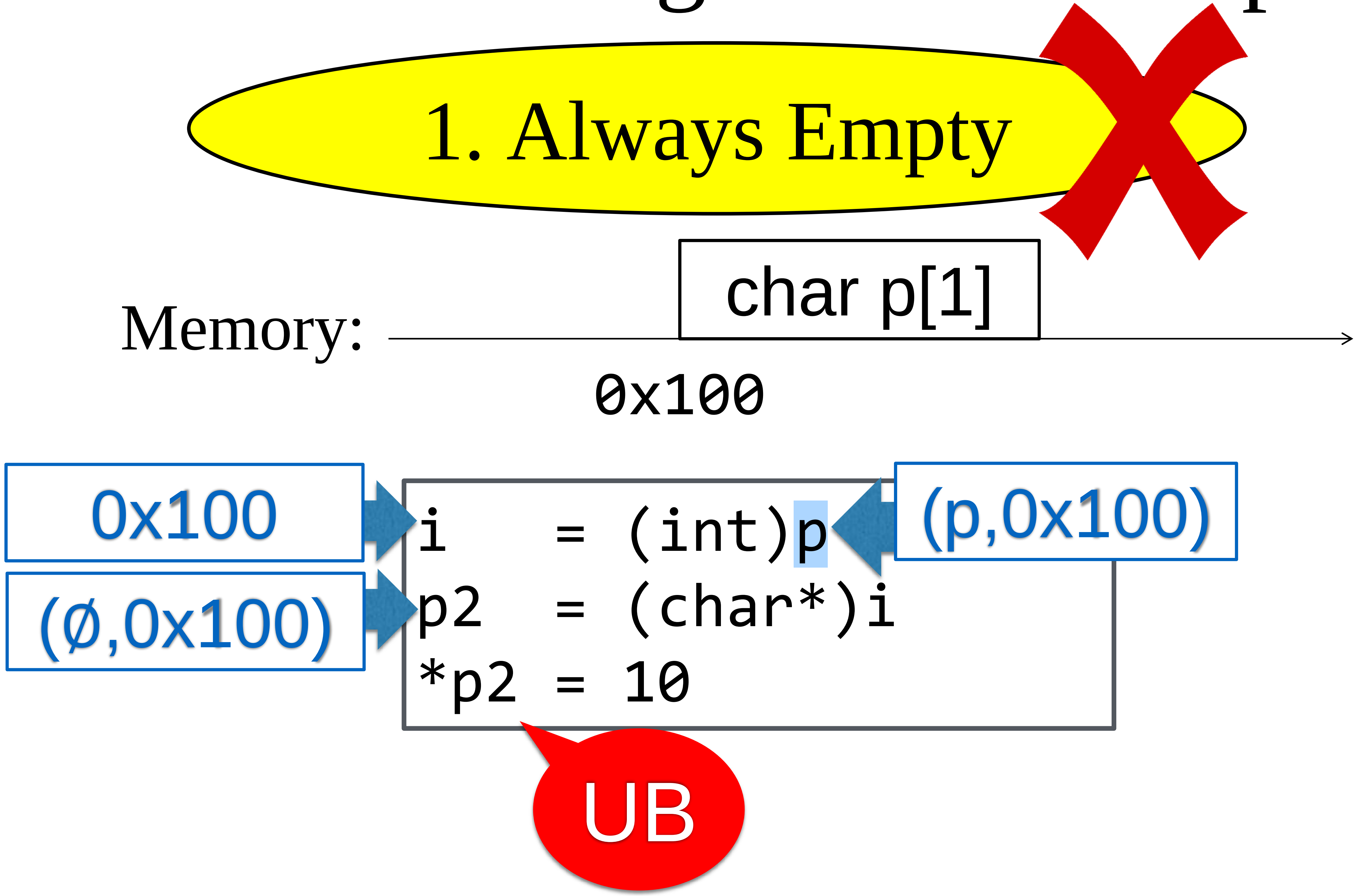
0x100

(∅,0x100)

```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

(p,0x100)

Empty Provenance:
Pointer – Integer Round Trip



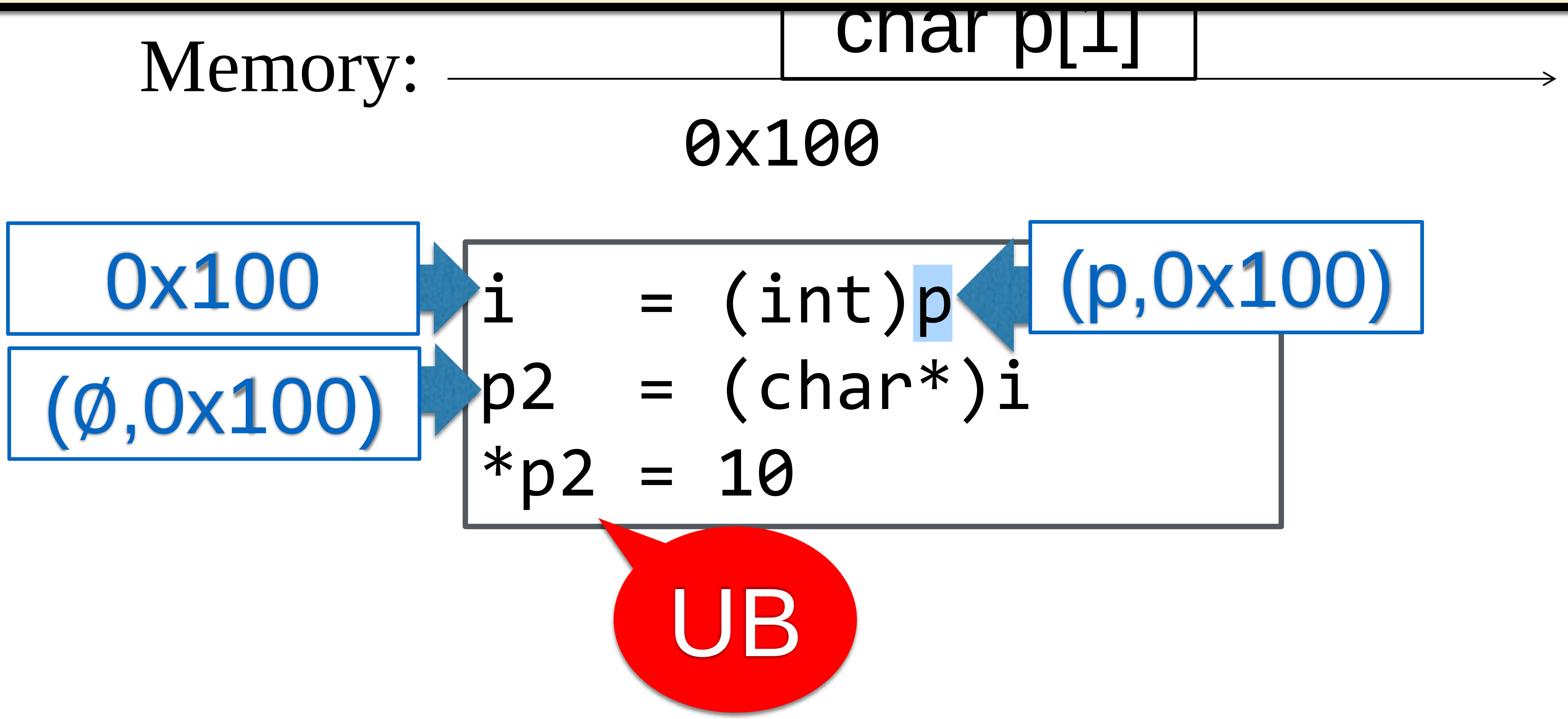
1. Always Empty

```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

Empty Provenance:
Pointer – Integer Round Trip

Problem

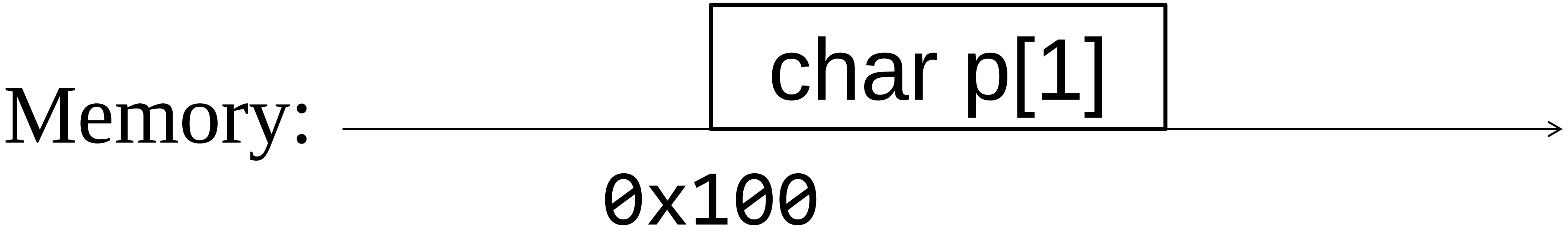
Common program patterns raise UB



```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

Empty Provenance:
Pointer – Integer Round Trip

3. Always Full ✓



Memory:

char p[1]

0x100

```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

0x100

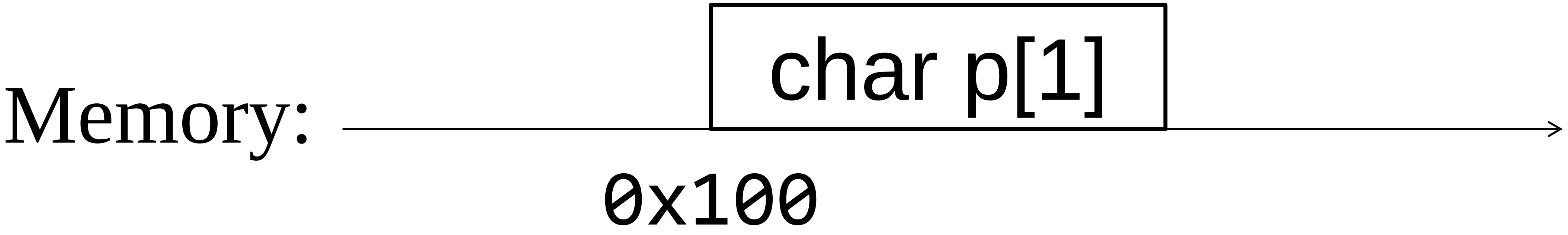
(∅,0x100)

(p,0x100)

UB

Empty Provenance: Pointer – Integer Round Trip

3. Always Full ✓



```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

0x100

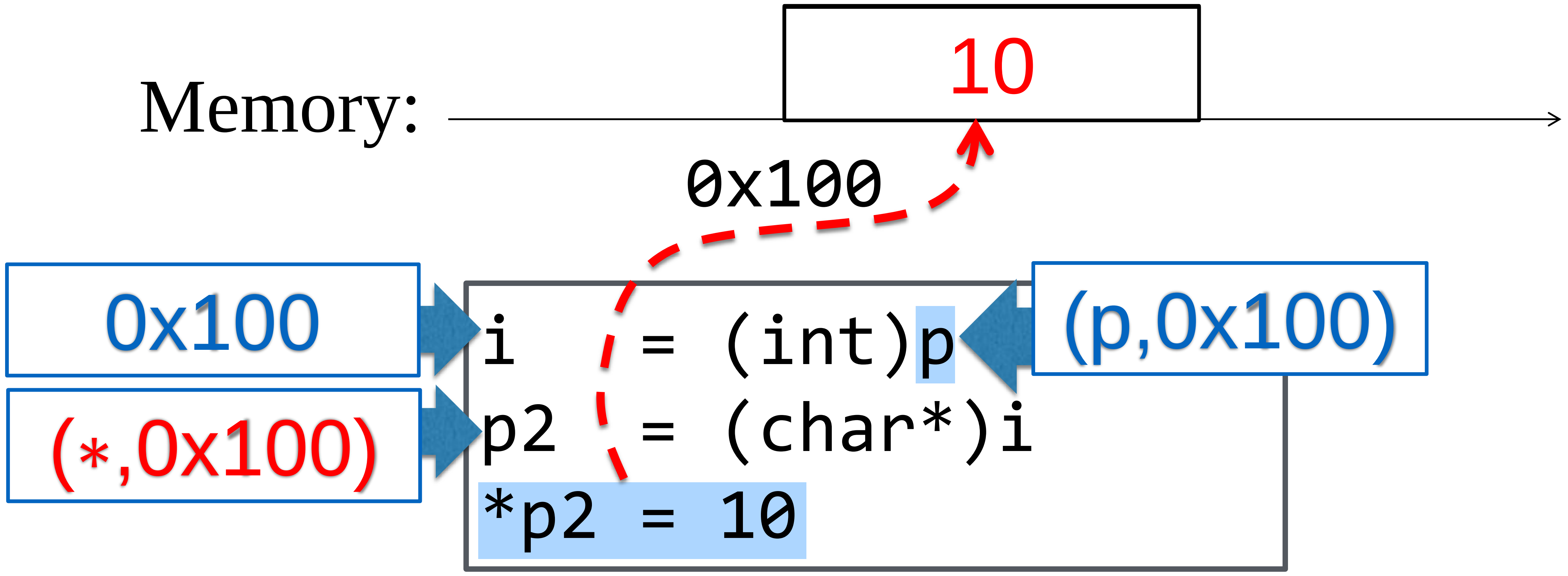
(*,0x100)

(p,0x100)

UB

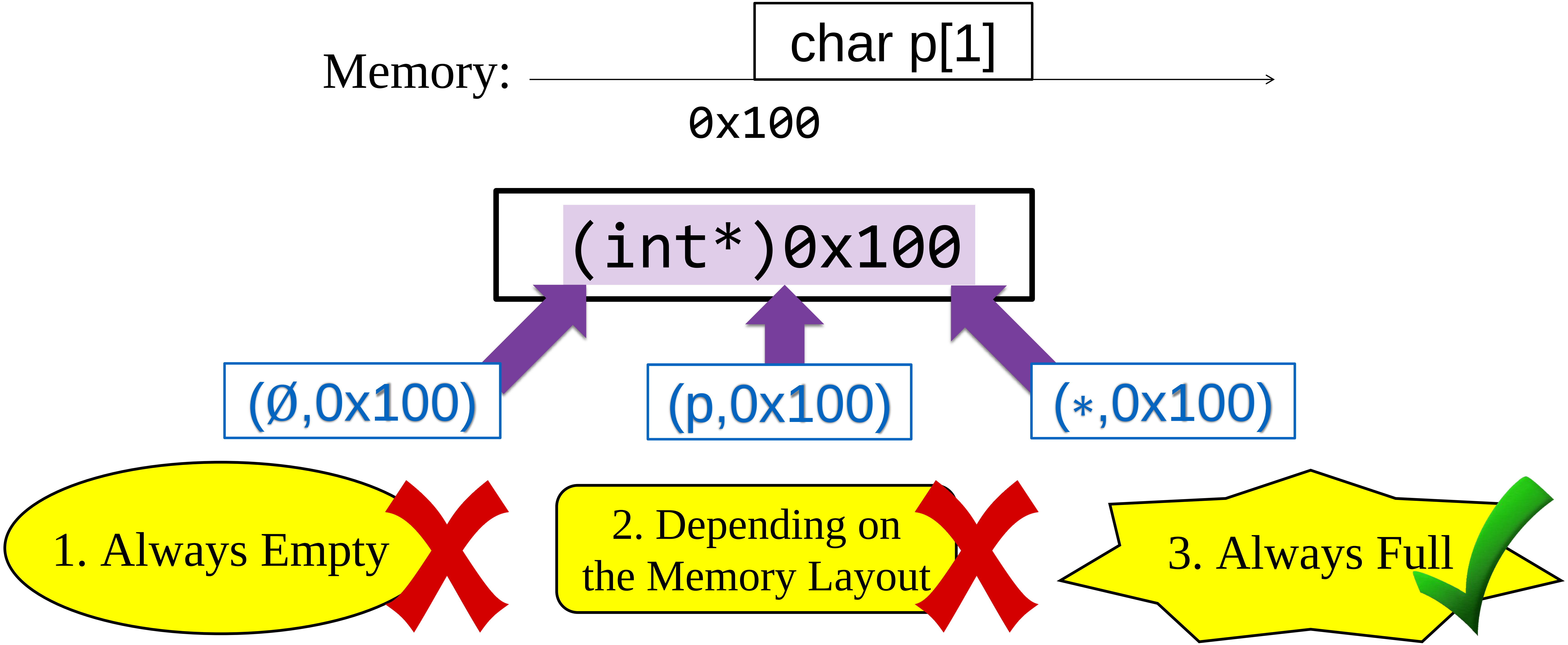
Empty Provenance:
Pointer – Integer Round Trip

3. Always Full



```
i   = (int)p
p2  = (char*)i
*p2 = 10
```

Integer → Pointer Casting?



Memory:

char p[1]

`0x100`

`(int*)0x100`

(Ø,0x100)

(p,0x100)

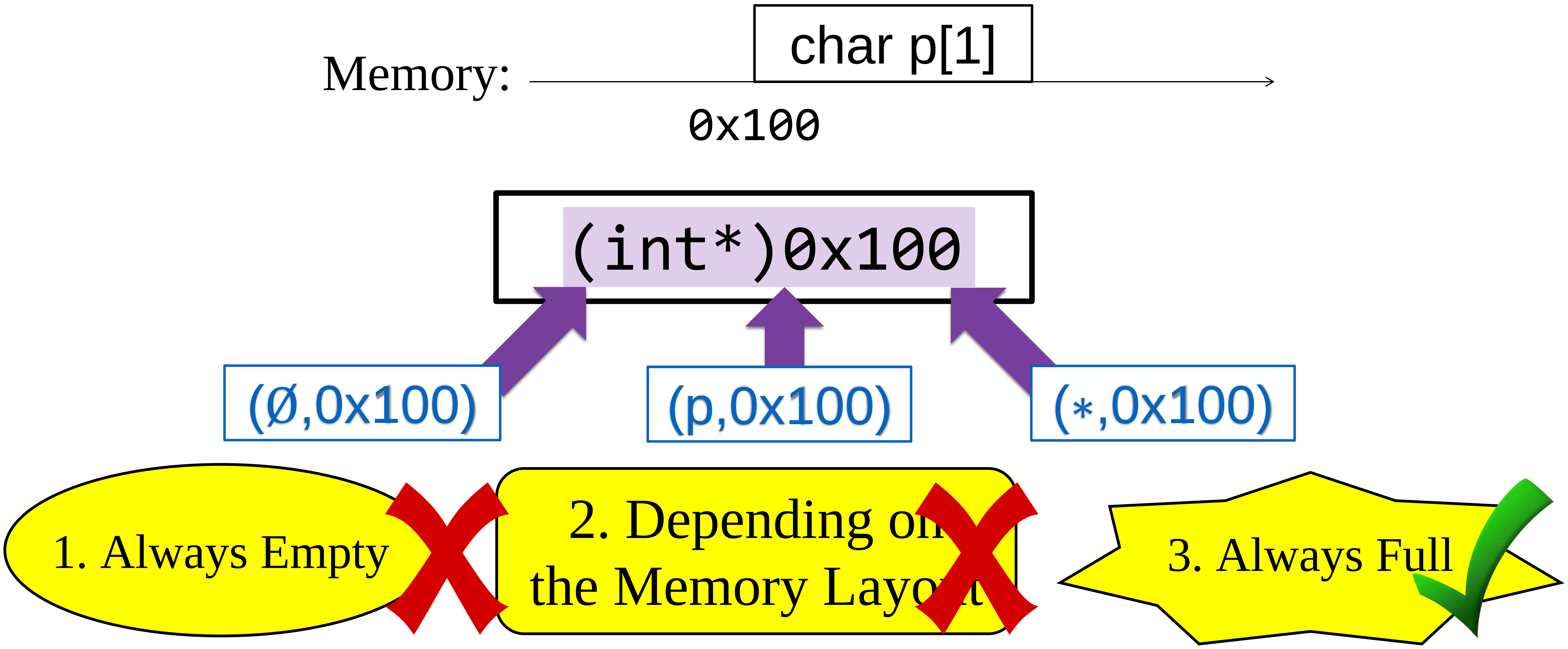
(*,0x100)

1. Always Empty ✗

2. Depending on the Memory Layout ✗

3. Always Full ✓

Integer → Pointer Casting?



Memory:

char p[1]

0x100

(int*)0x100

(Ø,0x100)

(p,0x100)

(*,0x100)

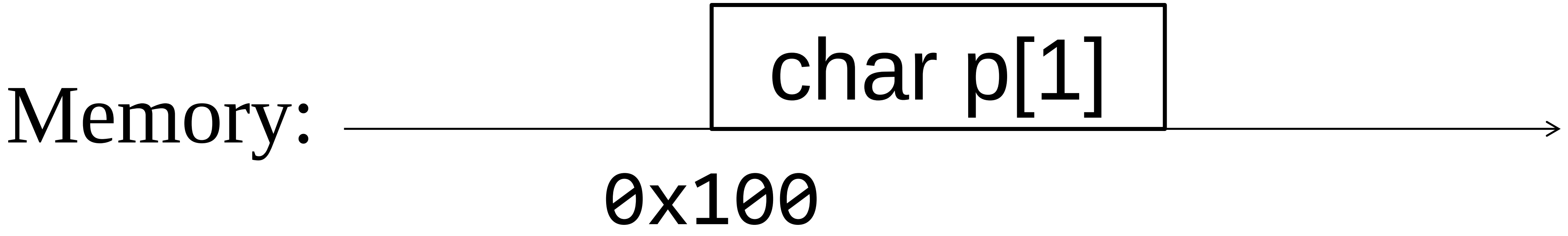
1. Always Empty

2. Depending on the Memory Layout

3. Always Full

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout



Memory:

char p[1]

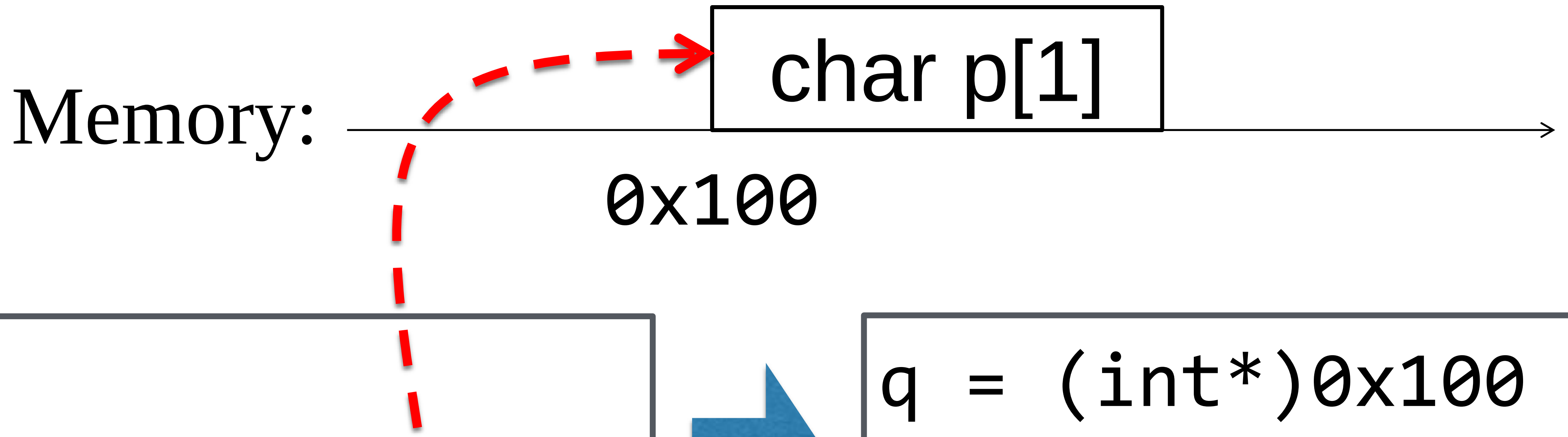
0x100

```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

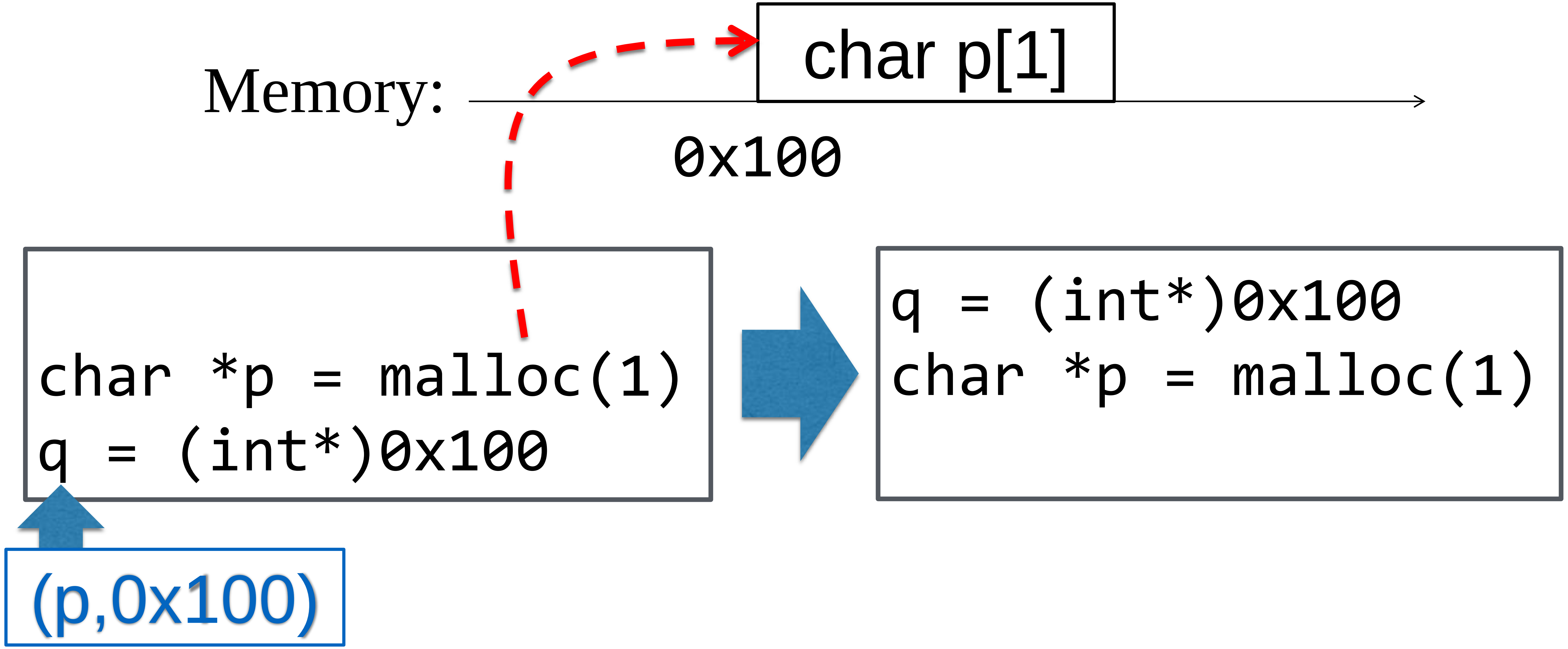


```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout



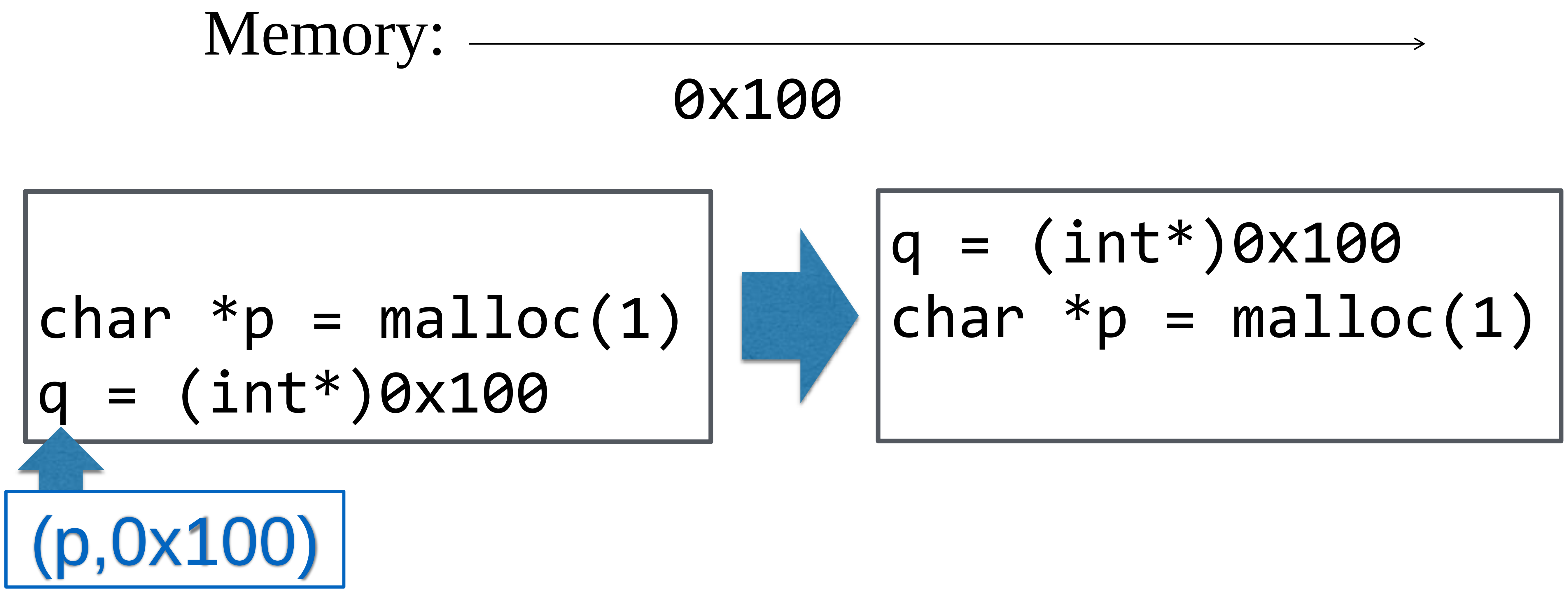
```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

(p,0x100)

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout



Memory:

0x100

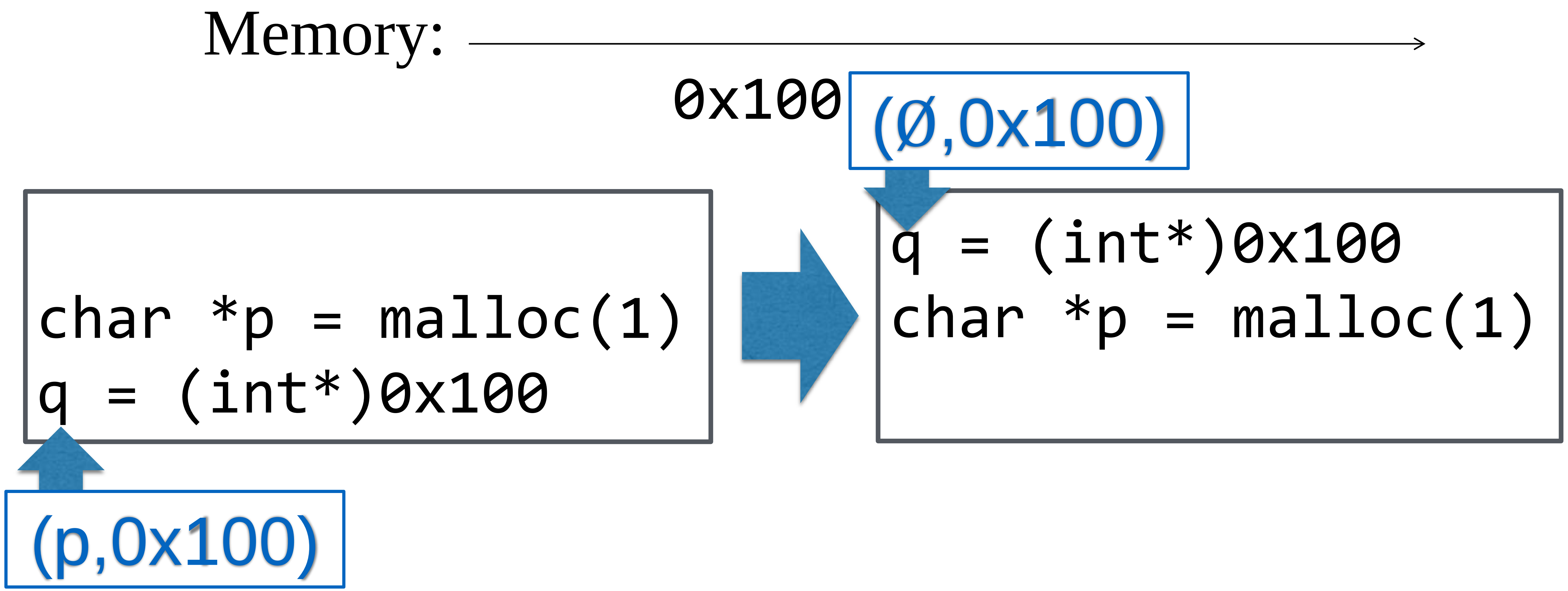
```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

(p,0x100)

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout ✗



```
char *p = malloc(1)
q = (int*)0x100
```

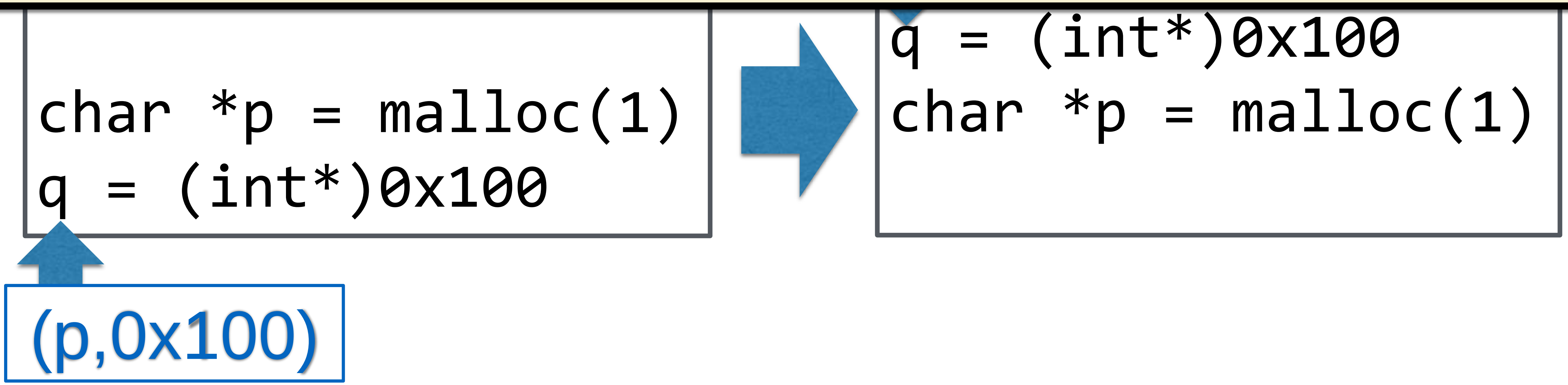
```
q = (int*)0x100
char *p = malloc(1)
```

Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

Problem

Movement of casts, or functions including them, is restricted

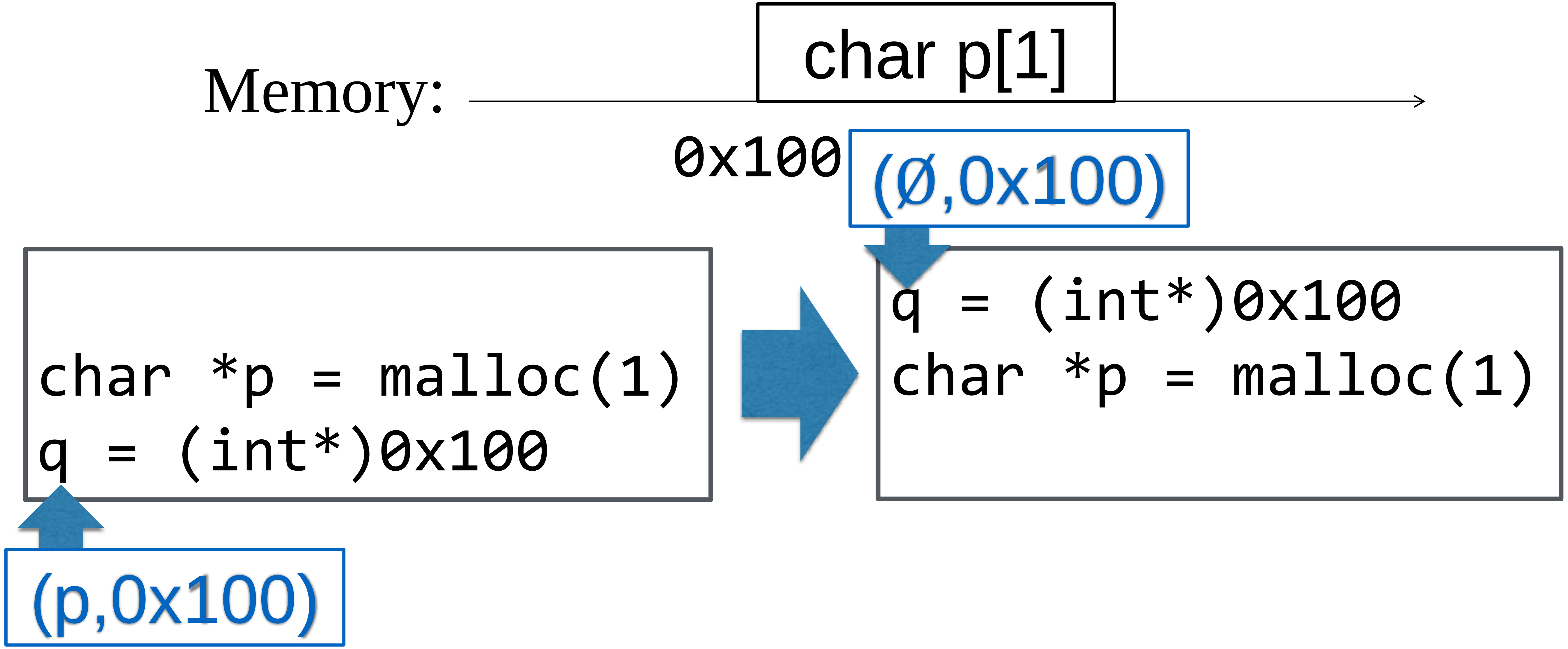


```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

Depending on the Memory Layout: Reordering

3. Always Full ✓

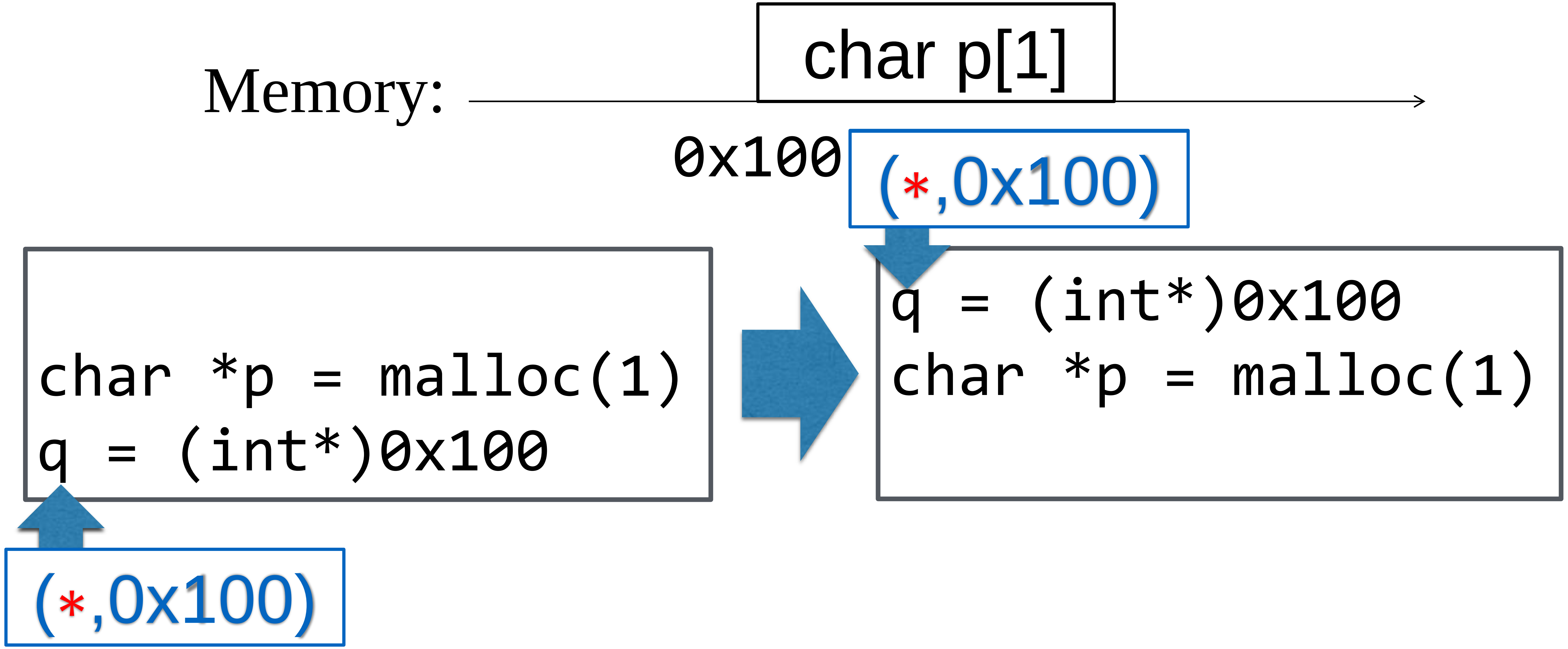


```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

Depending on the Memory Layout: Reordering

3. Always Full ✓



Memory:

char p[1]

0x100

(*,0x100)

```
char *p = malloc(1)
q = (int*)0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```

(*,0x100)

Problem 3

Problems with Full Provenance

Anyone can modify other’s local variables by

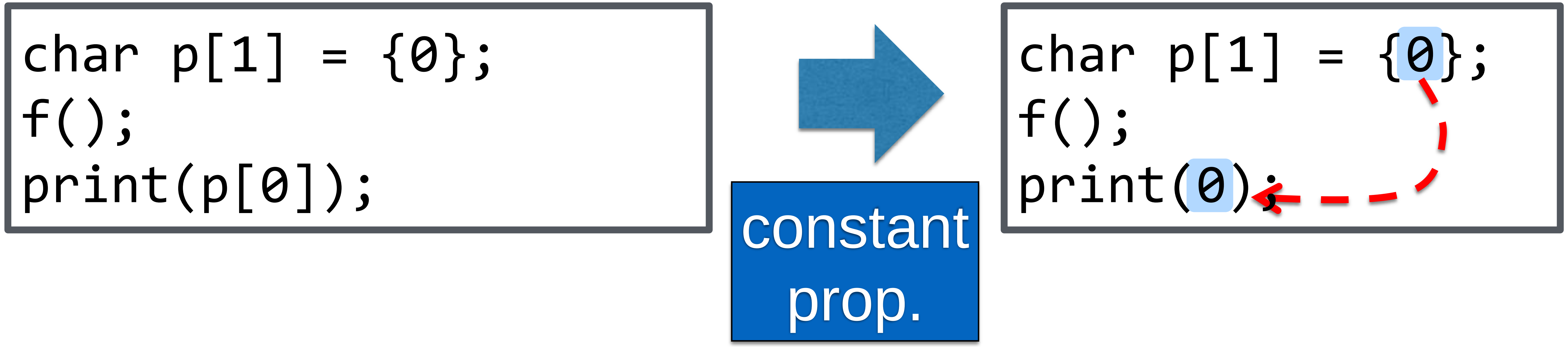
1. Guessing their addresses &
2. Acquiring full provenance via casting

```
char p[1] = {0};
f();
print(p[0]);
```

Problems with Full Provenance

Anyone can modify other’s local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



```
char p[1] = {0};
f();
print(p[0]);
```

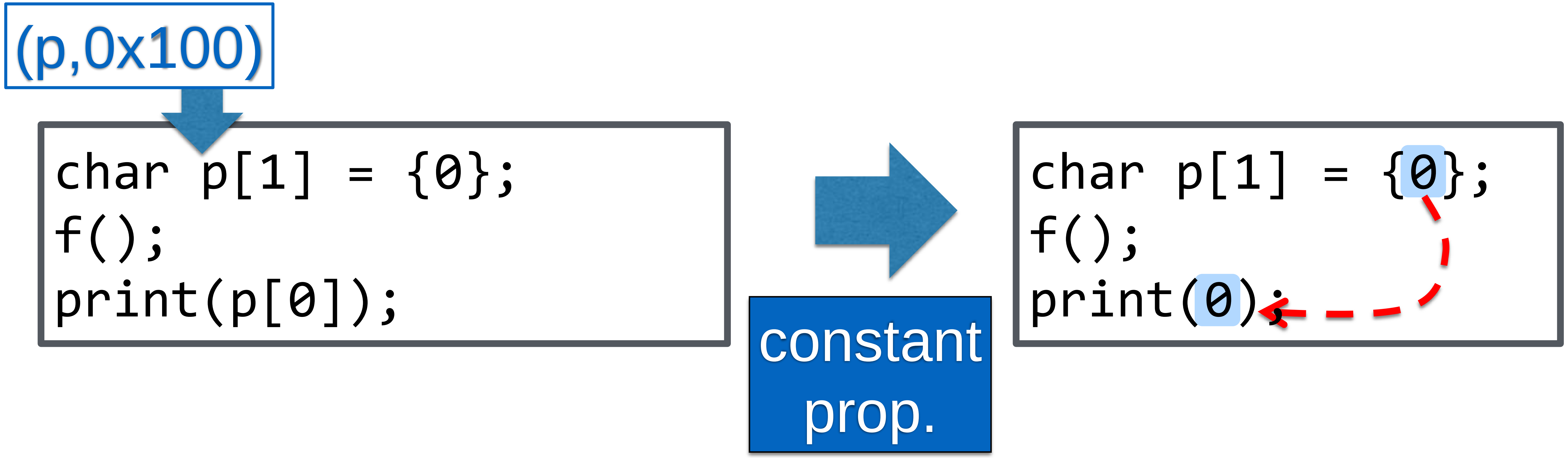
constant prop.

```
char p[1] = {0};
f();
print(0);
```

Problems with Full Provenance

Anyone can modify other’s local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



(p,0x100)

```
char p[1] = {0};
f();
print(p[0]);
```

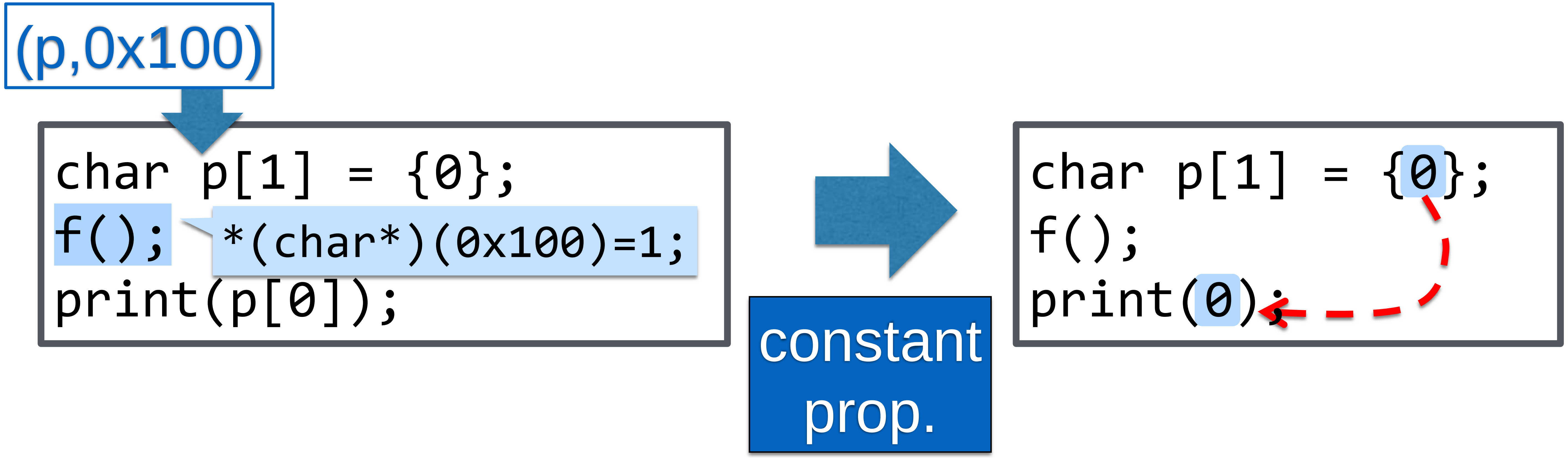
constant prop.

```
char p[1] = {0};
f();
print(0);
```

Problems with Full Provenance

Anyone can modify other's local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



(p,0x100)

```
char p[1] = {0};
f();
print(p[0]);
```

```
*(char*)(0x100)=1;
```

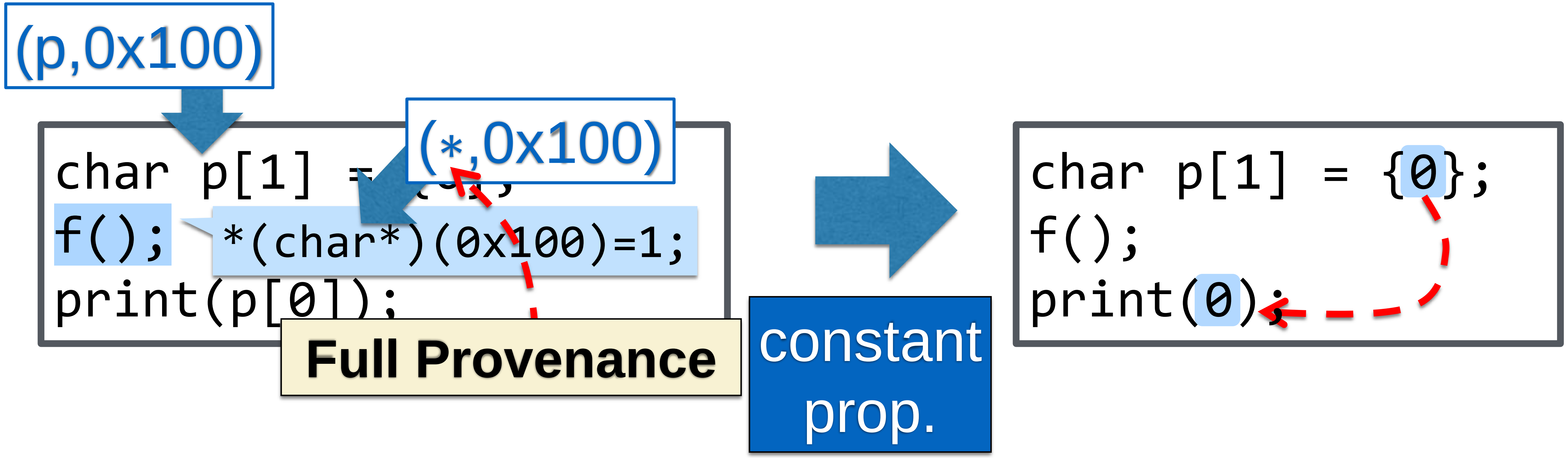
constant prop.

```
char p[1] = {0};
f();
print(0);
```

Problems with Full Provenance

Anyone can modify other's local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



```
char p[1] = {0};
f();
print(p[0]);
```

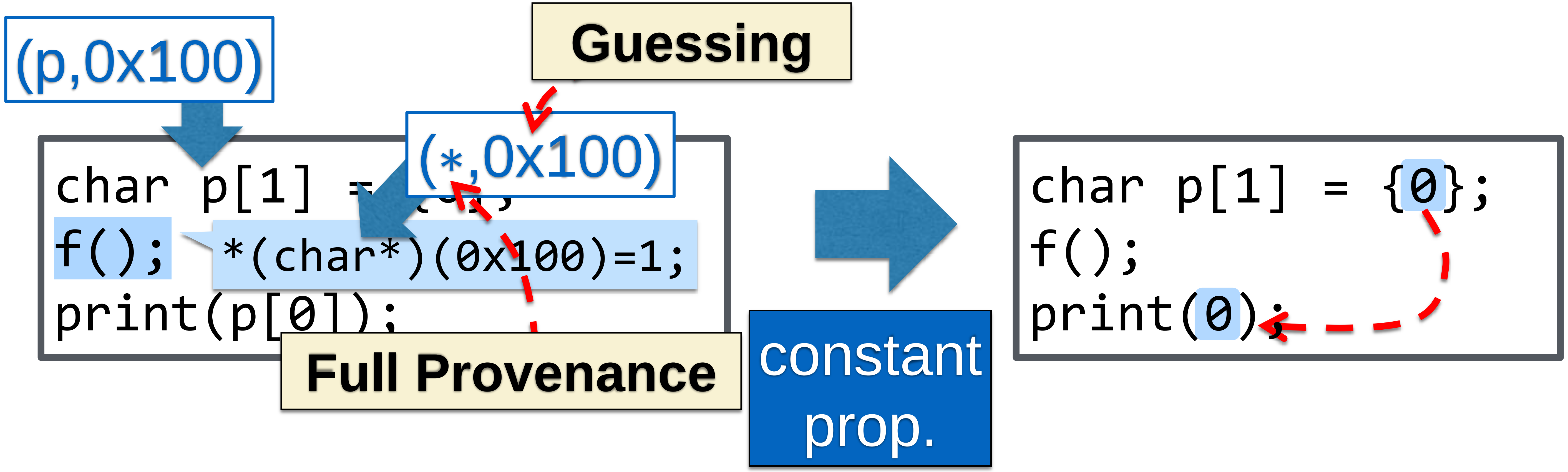
```
*(char*)(0x100)=1;
```

```
char p[1] = {0};
f();
print(0);
```

Problems with Full Provenance

Anyone can modify other’s local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



```
char p[1] = {0};
f();
print(p[0]);
```

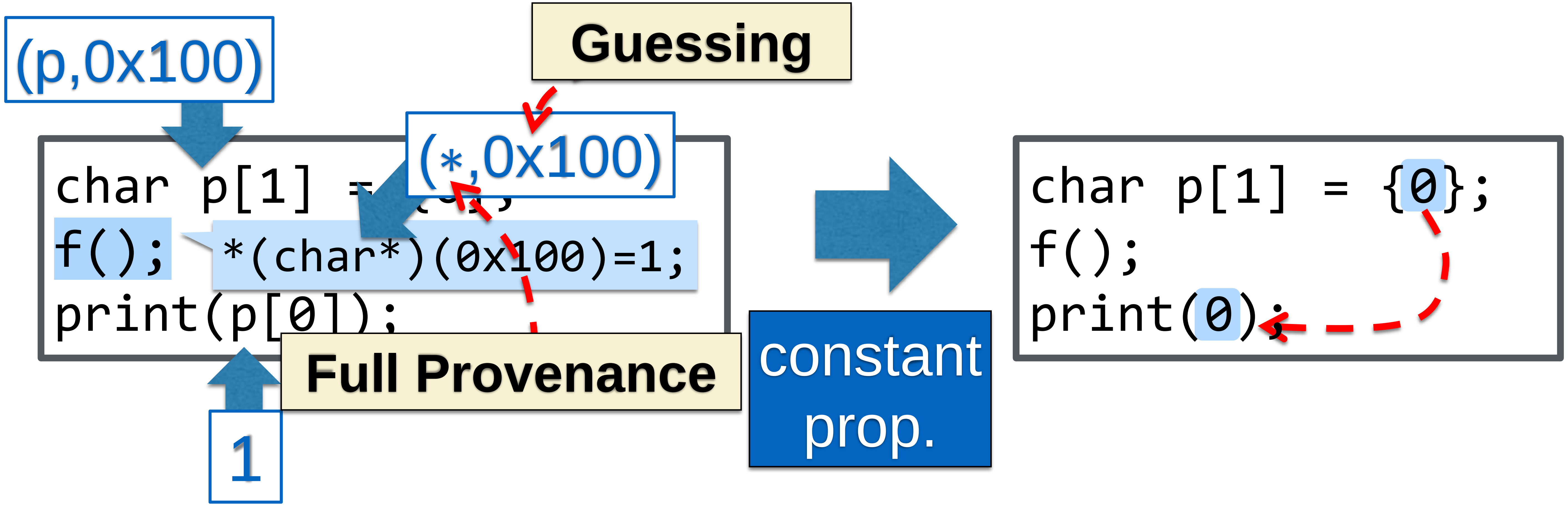
```
*(char*)(0x100)=1;
```

```
char p[1] = {0};
f();
print(0);
```

Problems with Full Provenance

Anyone can modify other’s local variables by

1. Guessing their addresses &
2. Acquiring full provenance via casting



```
char p[1] = {0};
f();
print(p[0]);
```

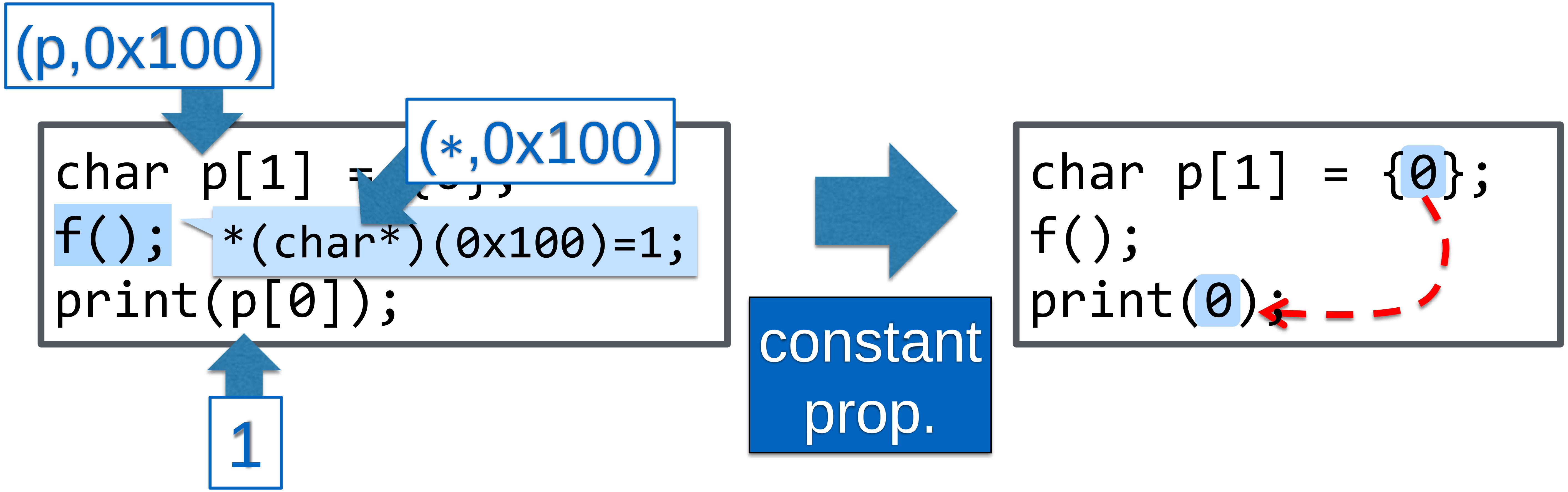
```
*(char*)(0x100)=1;
```

```
char p[1] = {0};
f();
print(0);
```

Our Solution

Basic Idea

Exploit Nondeterministic Allocation



```
char p[1] = {0};
f();
print(p[0]);
```

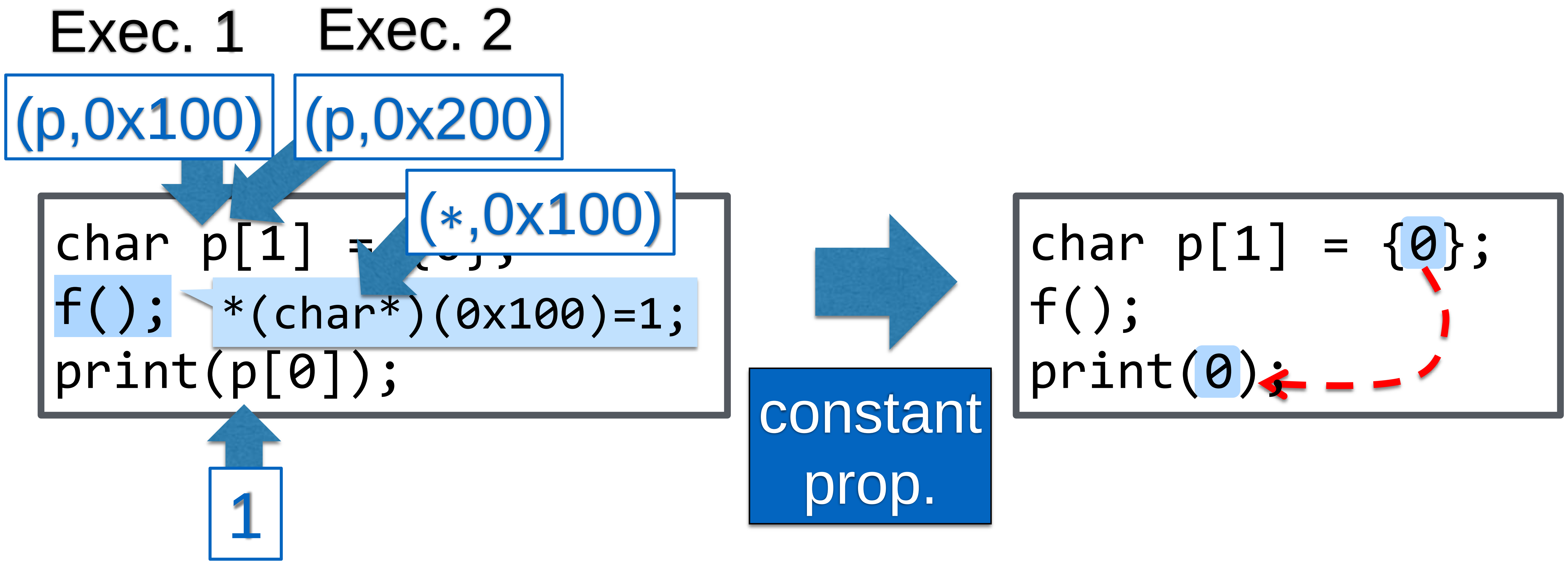
```
*(char*)(0x100)=1;
```

```
char p[1] = {0};
f();
print(0);
```

Our Solution

Basic Idea

Exploit Nondeterministic Allocation



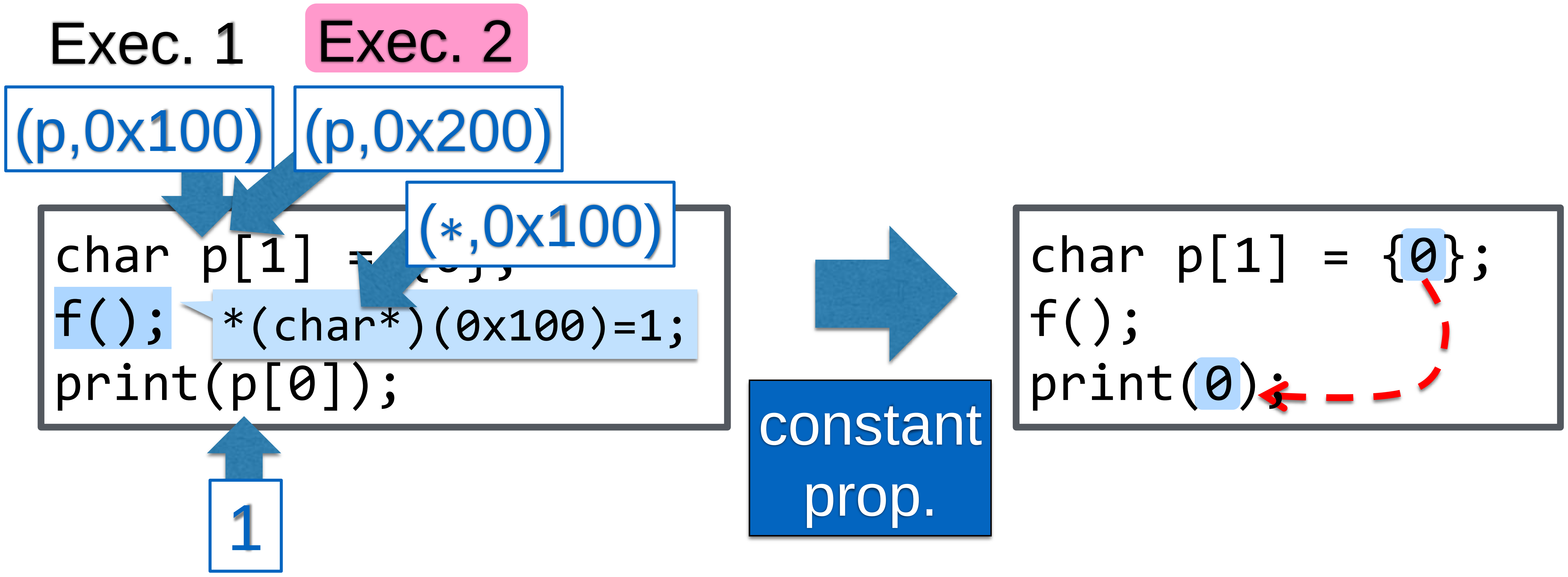
```
char p[1] = {0};
f();  *(char*)(0x100)=1;
print(p[0]);
```

```
char p[1] = {0};
f();
print(0);
```

Our Solution

Basic Idea

Exploit Nondeterministic Allocation



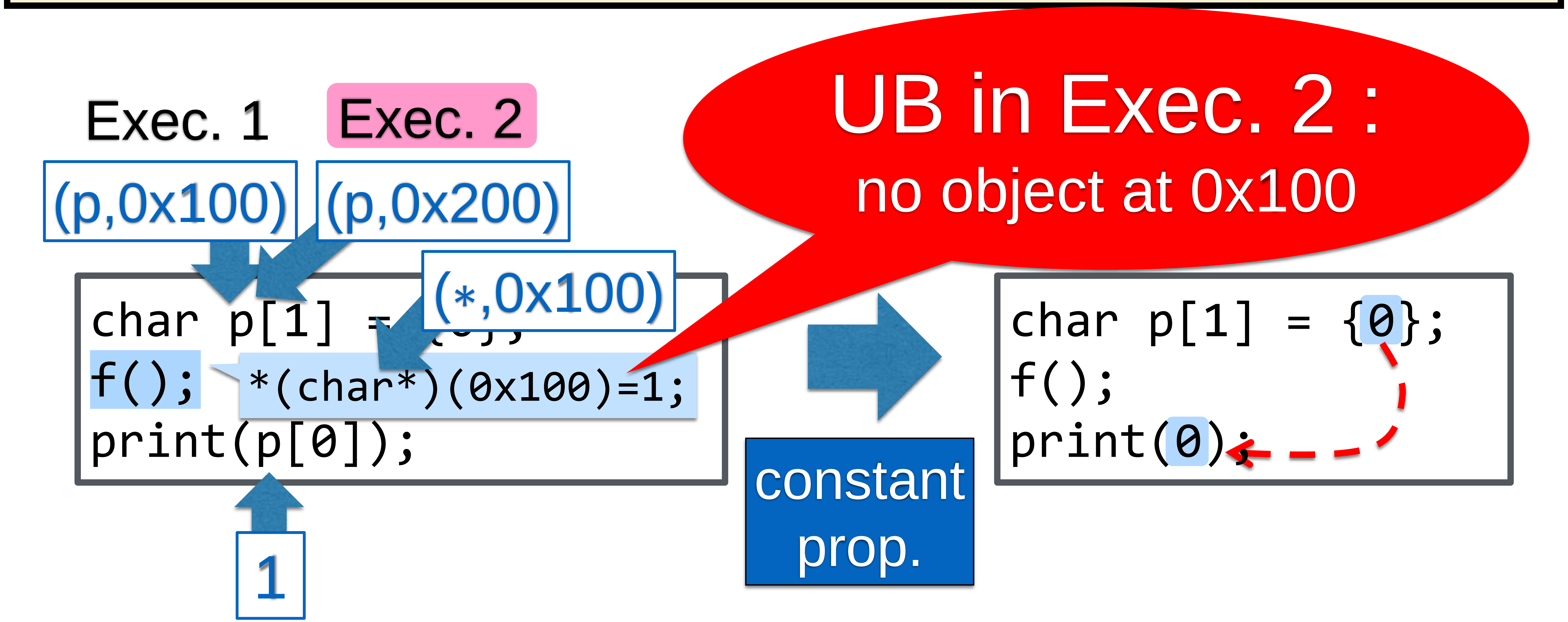
```
char p[1] = {0};
f();  *(char*)(0x100)=1;
print(p[0]);
```

```
char p[1] = {0};
f();
print(0);
```

Our Solution

Basic Idea

Exploit Nondeterministic Allocation

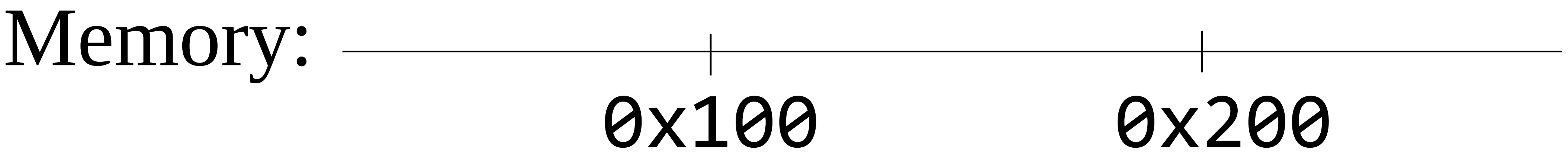


```
char p[1] = {0};
f();
print(p[0]);
```

```
*(char*)(0x100)=1;
```

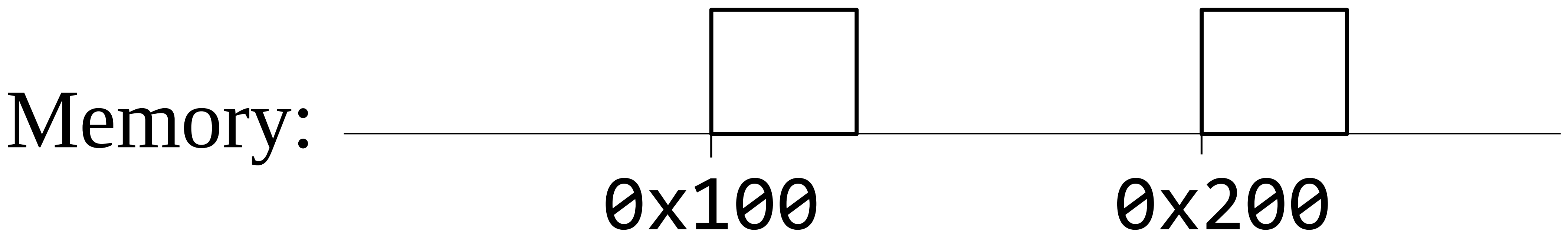
```
char p[1] = {0};
f();
print(0);
```

More Formally, Twin Allocation



```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

More Formally, Twin Allocation



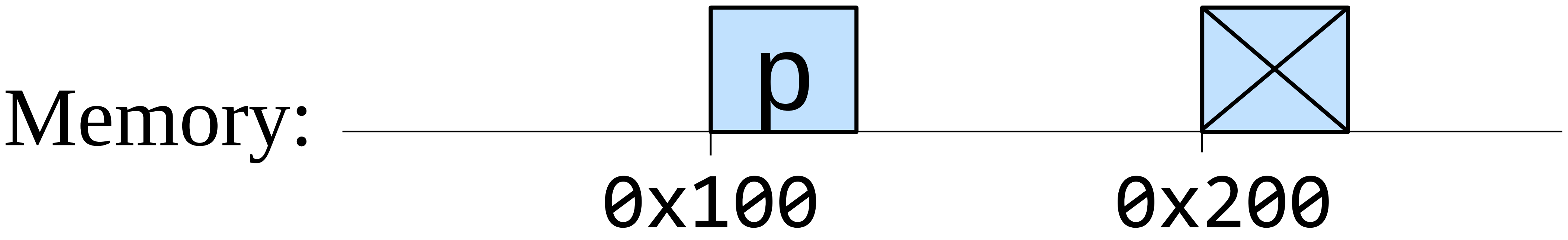
Memory:

0x100

0x200

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

More Formally, Twin Allocation

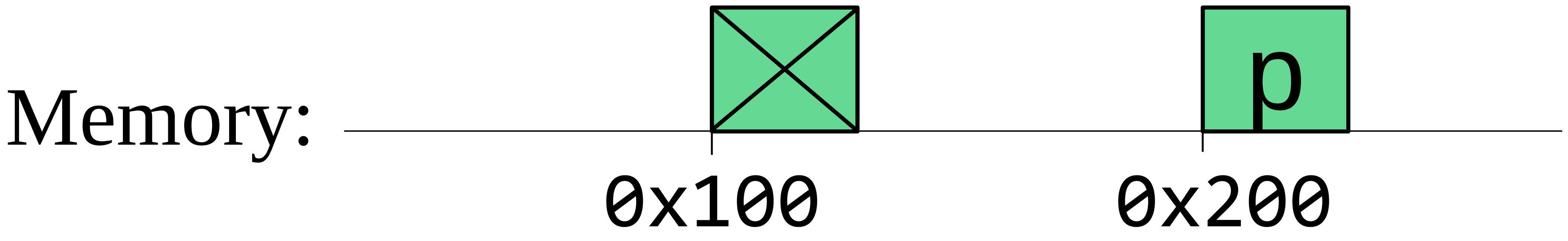


Exec. 1

(p,0x100)

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

More Formally, Twin Allocation

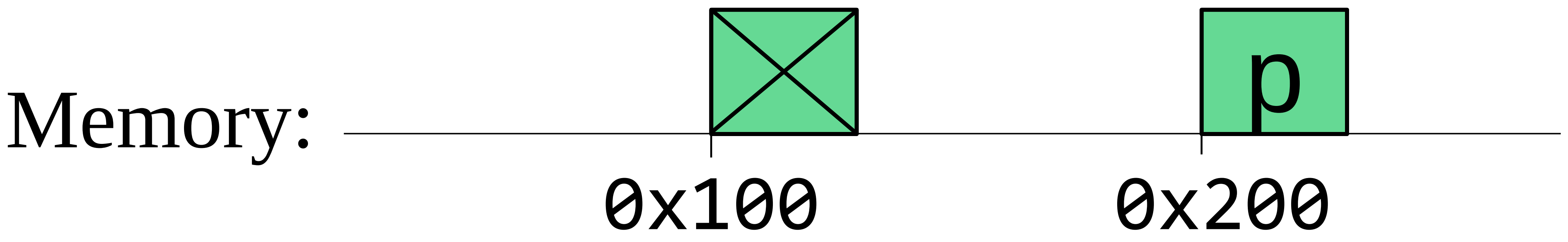


Exec. 1 Exec. 2

(p,0x100) (p,0x200)

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

More Formally, Twin Allocation



Exec. 1

Exec. 2

(p,0x100)

(p,0x200)

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

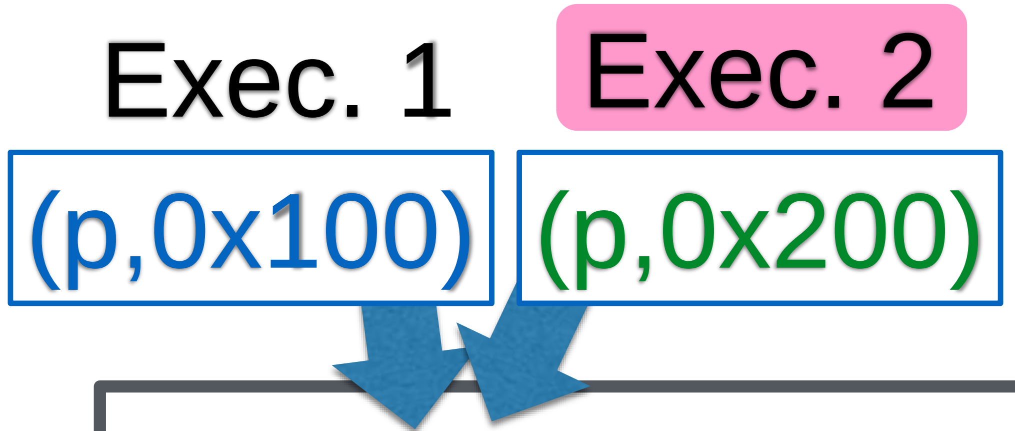
UB in Exec. 2 :

inaccessible at 0x100

More Formally, Twin Allocation

N.B.

This argument works only for unobserved addresses



Exec. 1

Exec. 2

(p,0x100)

(p,0x200)

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

UB in Exec. 2 :

inaccessible at 0x100

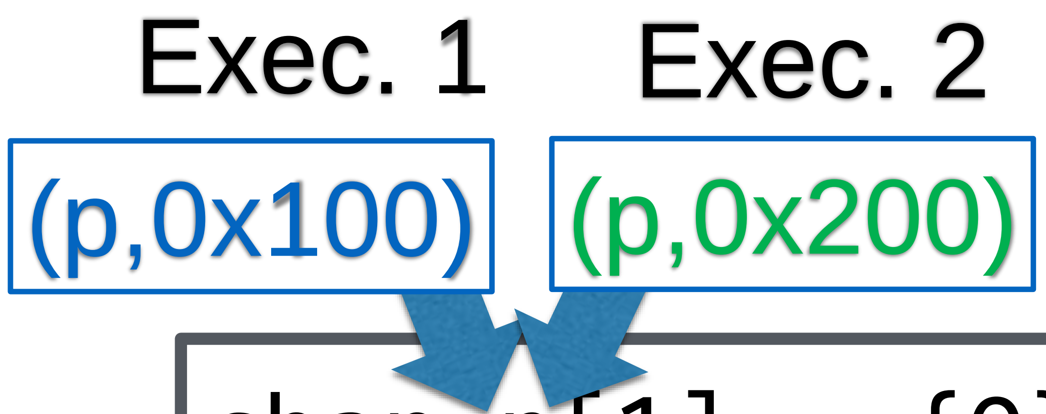
Example with Observed Address

```
char p[1] = {0};

*(char*)(0x100) = 1;

print(p[0]);
```

Example with Observed Address

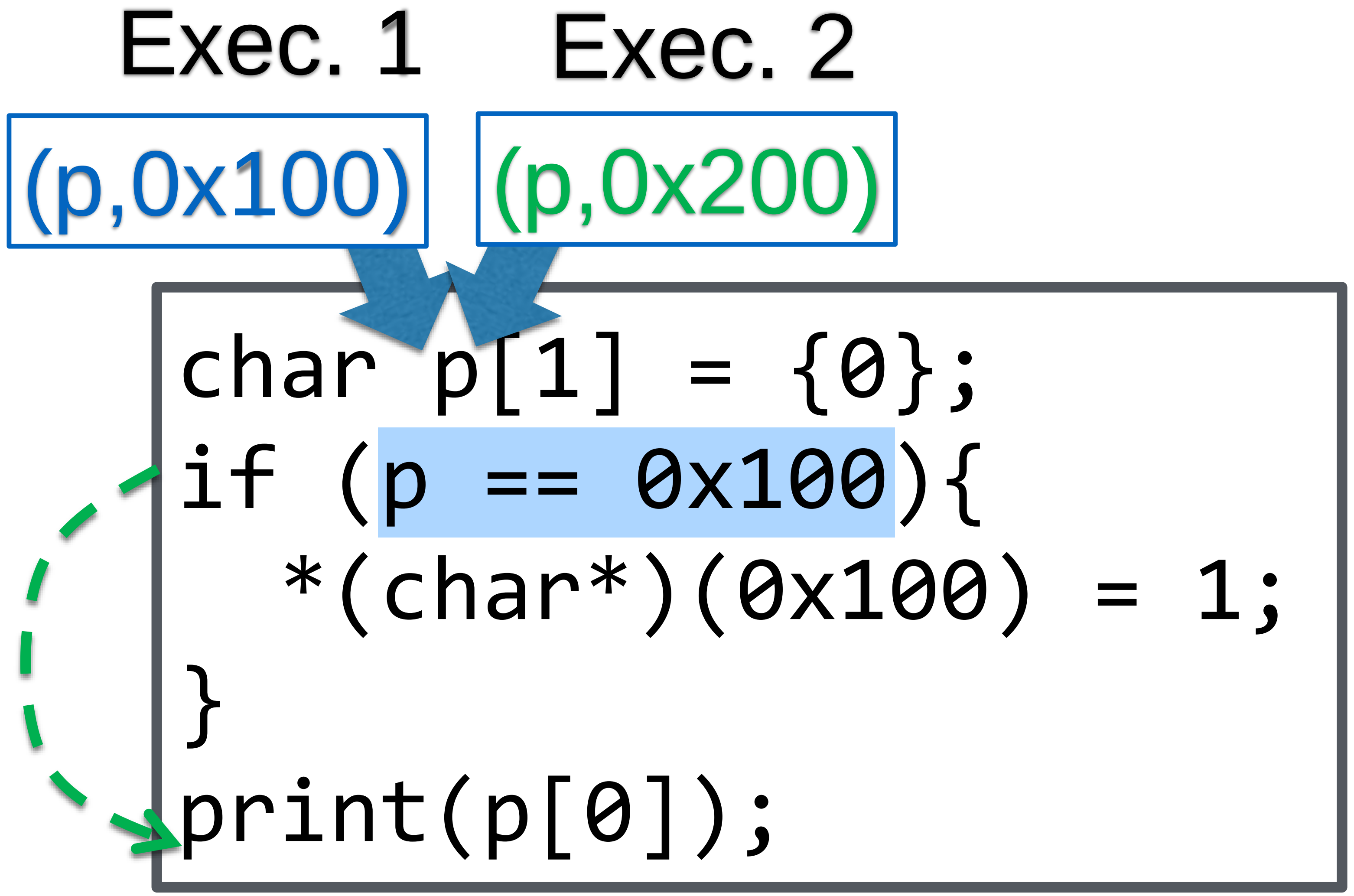
```
char p[1] = {0};
if (p == 0x100){
  *(char*)(0x100) = 1;
}
print(p[0]);
```

Example with Observed Address



```
char p[1] = {0};
if (p == 0x100){
  *(char*)(0x100) = 1;
}
print(p[0]);
```

Example with Observed Address



Exec. 1

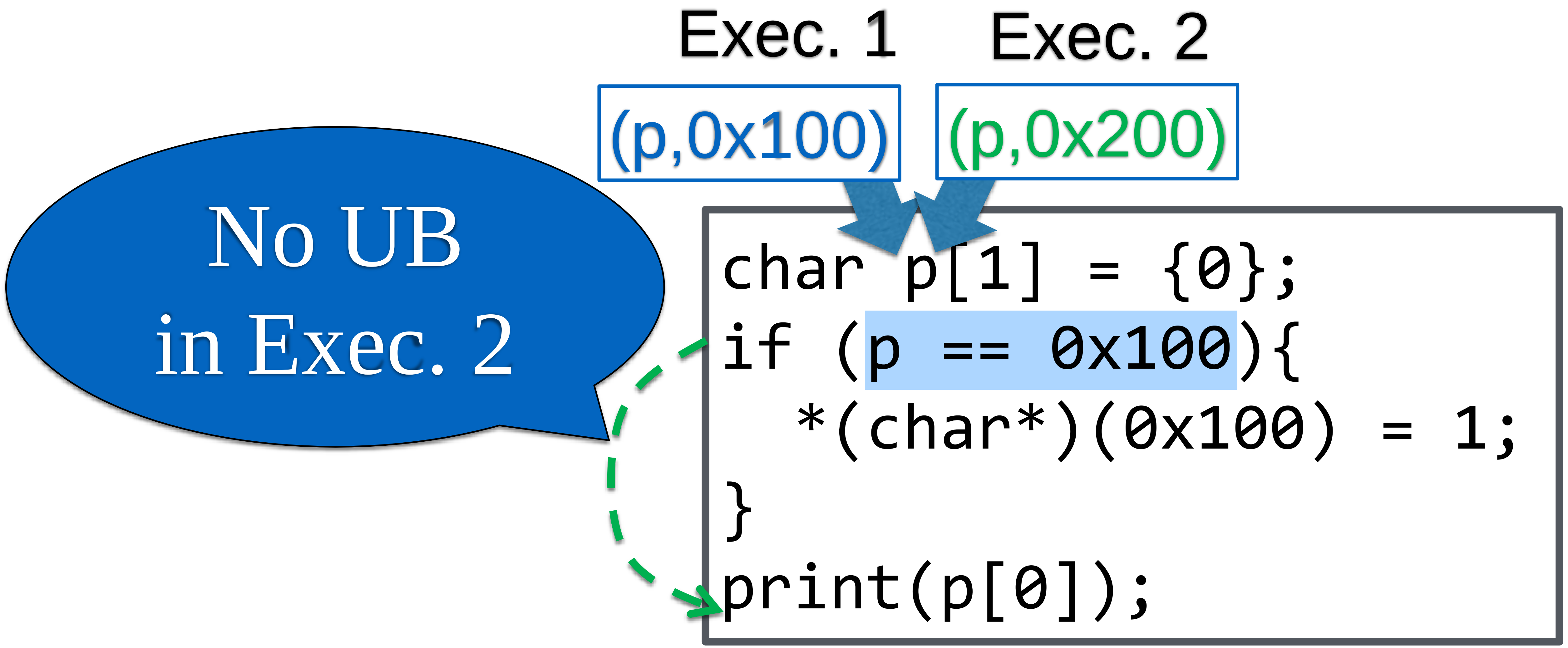
(p,0x100)

Exec. 2

(p,0x200)

```
char p[1] = {0};
if (p == 0x100){
  *(char*)(0x100) = 1;
}
print(p[0]);
```

Example with Observed Address



Exec. 1

Exec. 2

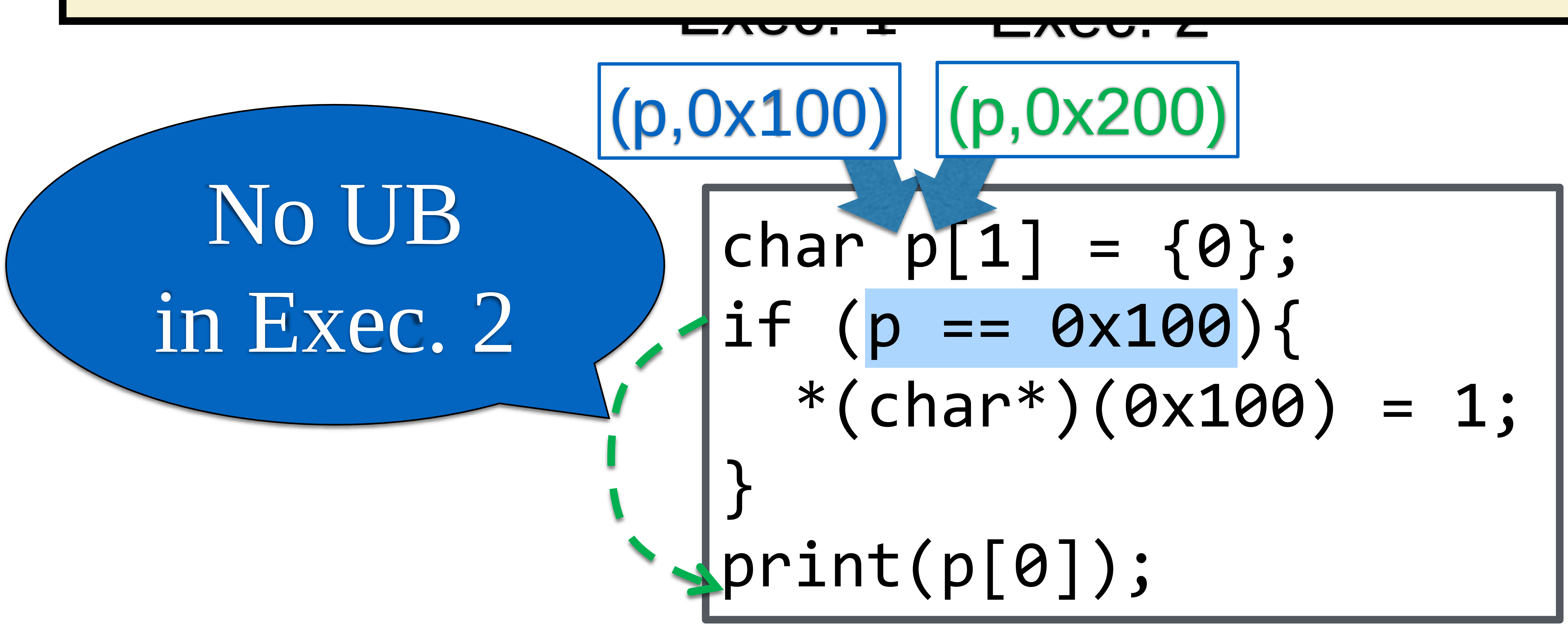
(p,0x100)

(p,0x200)

No UB
in Exec. 2

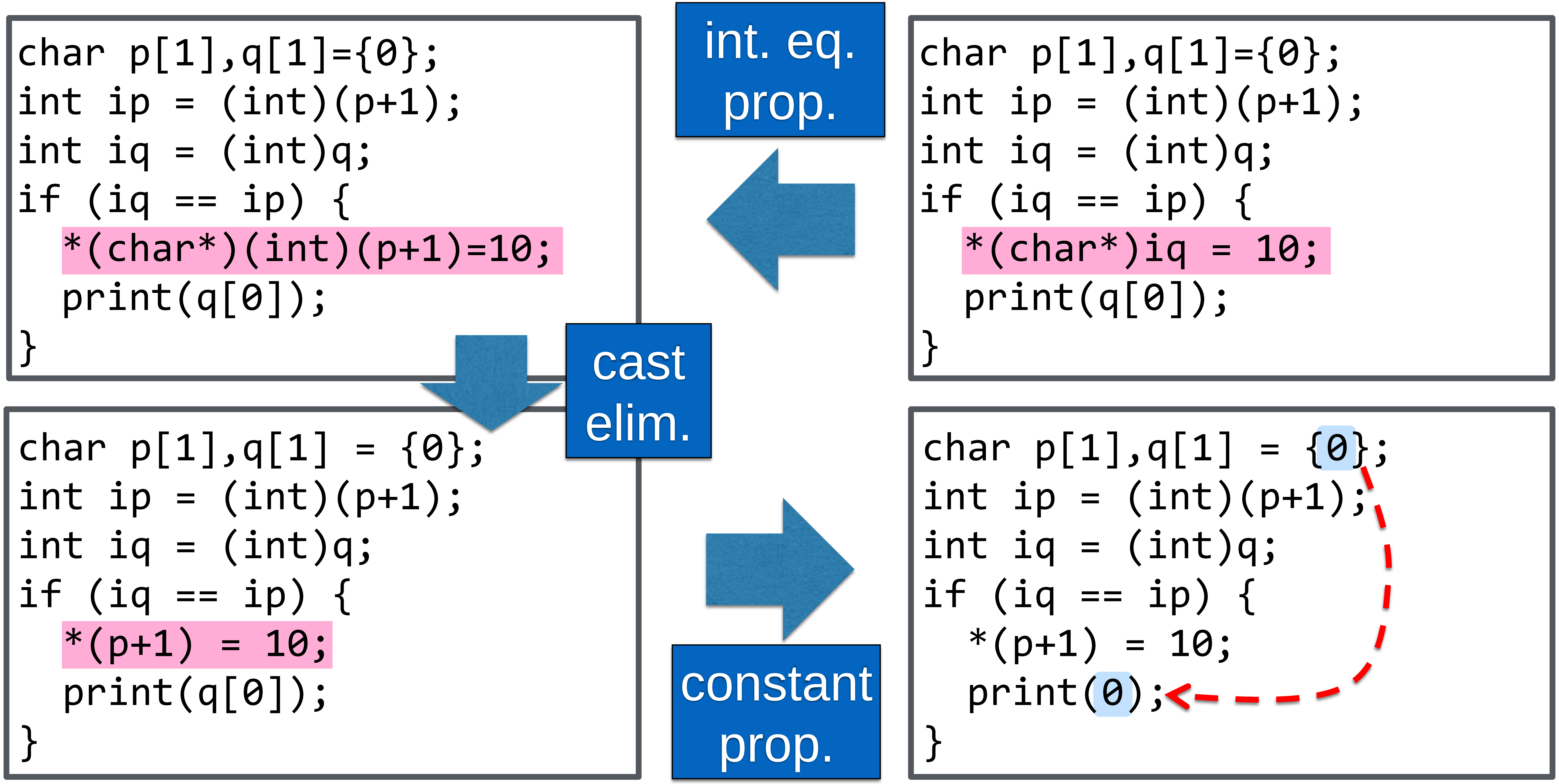
```
char p[1] = {0};
if (p == 0x100){
  *(char*)(0x100) = 1;
}
print(p[0]);
```

Consistent with common compilers' assumption: Observed variables can be modified by others



```
char p[1] = {0};
if (p == 0x100){
  *(char*)(0x100) = 1;
}
print(p[0]);
```

Miscompilation Revisited



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

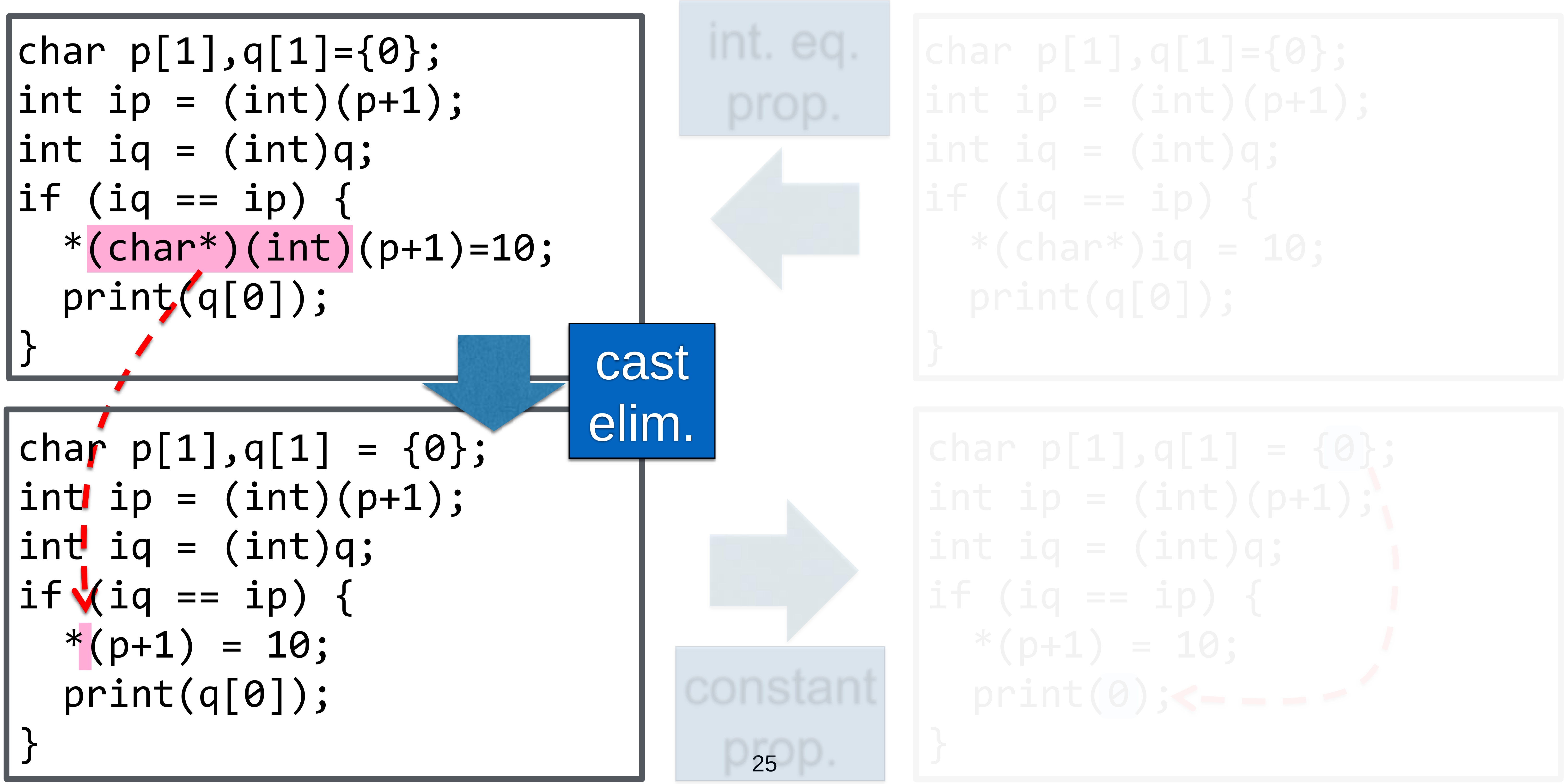
cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation Revisited



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

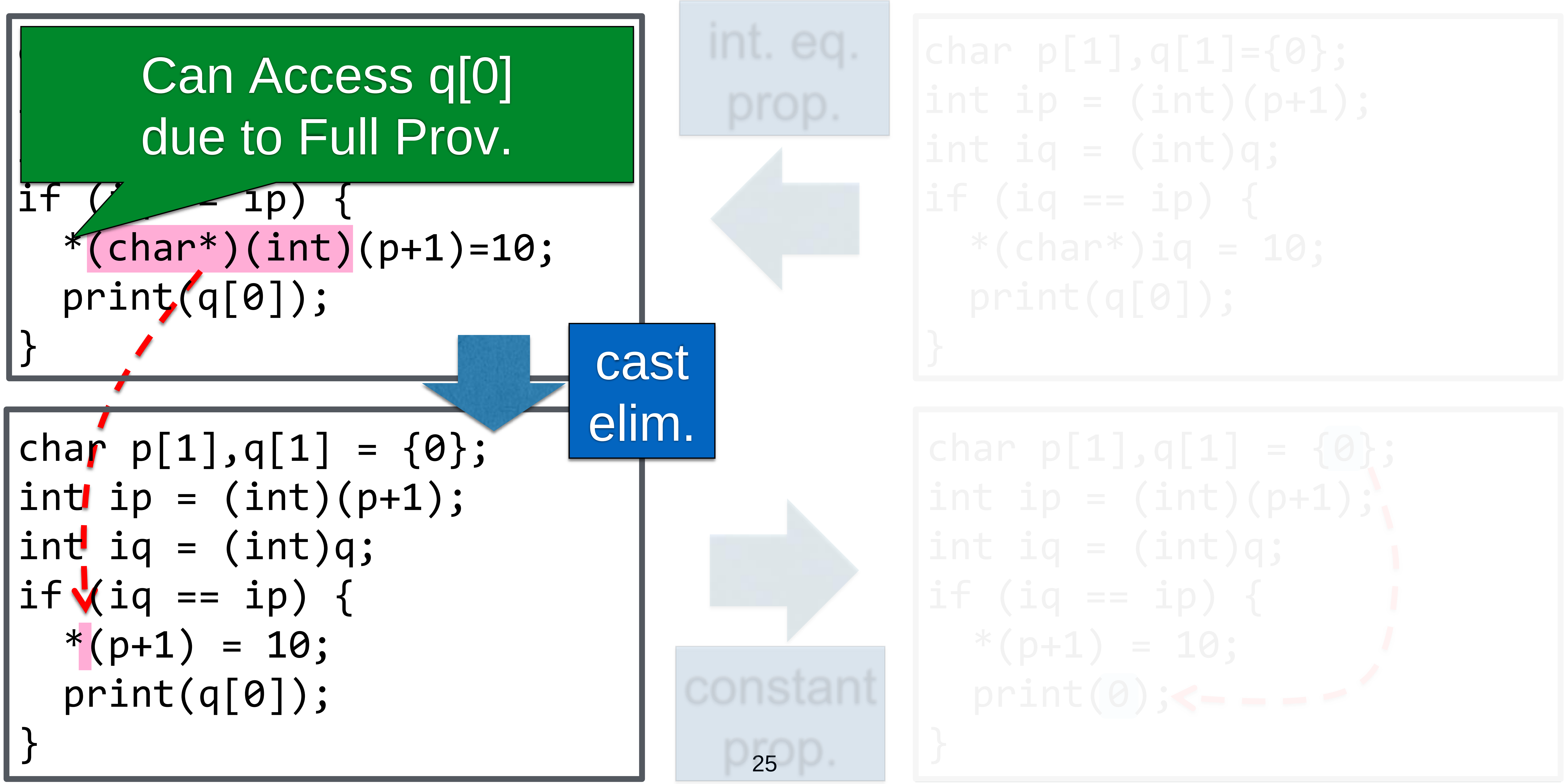
int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation Revisited



Can Access q[0] due to Full Prov.

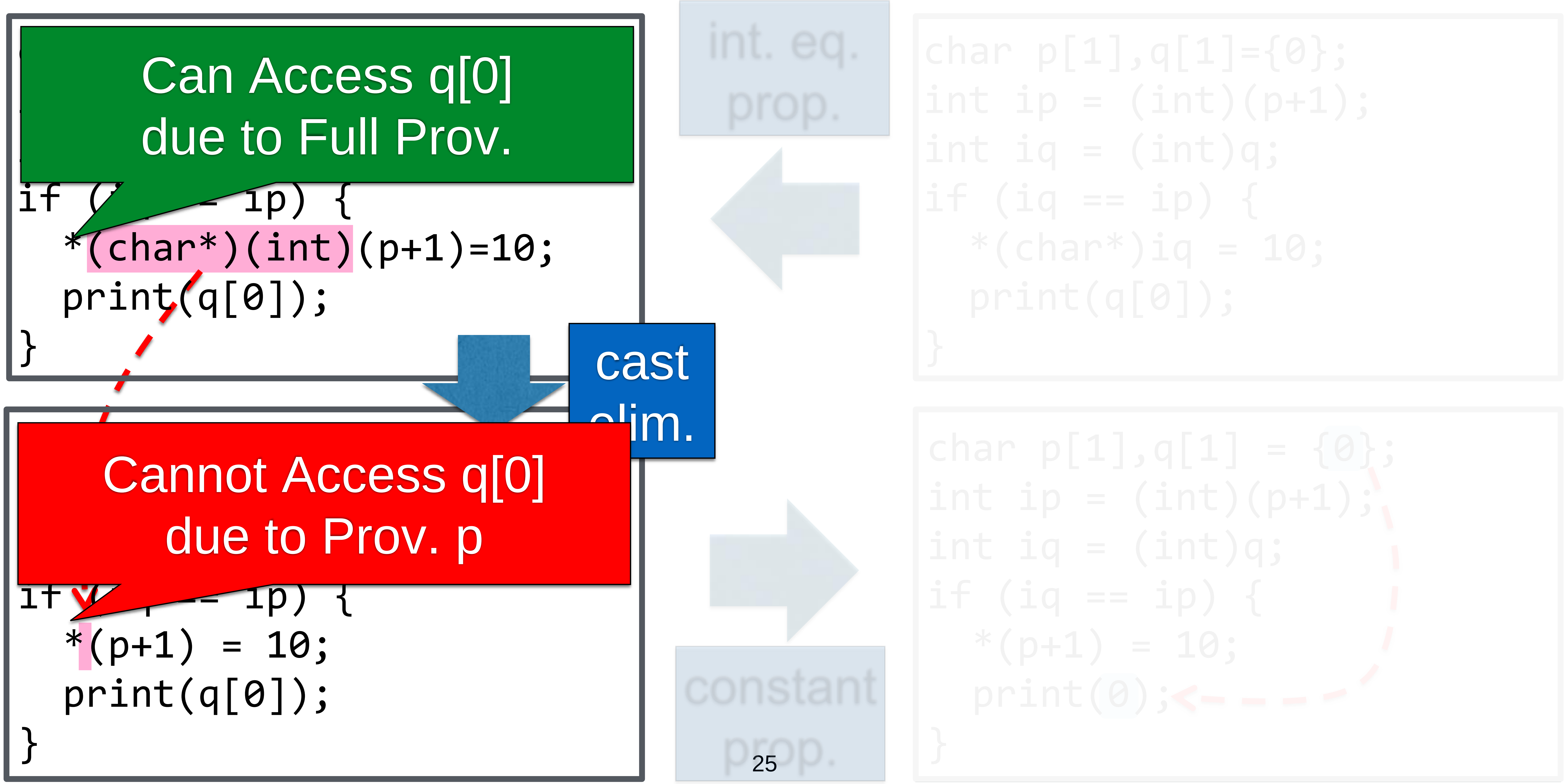
```
if (i      ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation Revisited



```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

Can Access q[0] due to Full Prov.

```
if (     ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

Cannot Access q[0] due to Prov. p

```
if (     ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation Revisited

Can Access q[0]

```
if (
  *(ch
  prin
}
```

Ca

due to Prov. p

```
if (    == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

Ptr → Int → Ptr Cast Elimination Is Unsound: A Potential Performance Issue

int. eq.

```
char p[1],q[1]={0};
```

```
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

constant prop.

Solution to the Cast Elim. Problem

Reducing # of Int ↔ Ptr Casts

- Most casts are introduced by compilers for convenience
- We recovered performance by reducing unnecessary casts
  - Int → Ptr: 95% removed
  - Ptr → Int: 75% removed

Solution to the Cast Elim. Problem

The paper includes more details
& a formal specification

Implementation & Evaluation

- We fixed LLVM 6.0 to be sound in our memory model

- We had to change only 1.7K LOC in total

- Benchmark Results

    - SPEC CPU2017 : <0.1% avg, <0.5% max slowdown

    - LLVM Nightly Tests : <0.1% avg , <3% max slowdown

- We verified key properties of our memory model in Coq

Conclusion

- We develop a memory model for IR which supports both low-level code & high-level optimizations
- We use full provenance & twin allocation to reconcile them
- Applying our model to LLVM has little impact on performance