Computer Systems Principles

C Pointers

UMASSCS
SCHOOL OF COMPUTER SCIENCE

Learning Objectives

- Learn about floating point number
- Learn about typedef, enum, and union
- Learn and understand pointers

FLOATING POINT NUMBER

IEEE Floating Point Standard

IEEE Floating Point Standard

- **sign bit**
- **exponent**
- **fraction**

IEEE Floating Point Standard

- **sign bit**
- **exponent**
- **fraction**

- $(-1)^s \times 2^E \times F$
  1. Sign bit s determines whether number is negative or positive
  2. Exponent E weights value by power of two
  3. Fractional part F normally a fractional value in range [1.0,2.0).

IEEE Floating Point Standard

- **sign bit**
- **exponent**
- **fraction**

- $(-1)^s \times 2^E \times F$
  1. Sign bit s determines whether number is negative or positive
  2. Exponent E weights value by power of two
  3. Fractional part F normally a fractional value in range [1.0,2.0).
     - Since F always starts with 1, we don't store it

IEEE Floating Point Standard

float

1 k=8 n=23

IEEE Floating Point Standard

float

1 k=8 n=23

double

1 k=11 n=52

IEEE Floating Point Standard

$(-1)^s \times 2^E \times F$

IEEE Floating Point Standard

$(-1)^s \times 2^E \times F$

- **when exp = 00..0, represent 0**

IEEE Floating Point Standard

$(-1)^s \times 2^E \times F$

- **when exp = 00..0, represent 0**
- **when exp = 11..1**
  - frac=00..0, represent ∞

IEEE Floating Point Standard

$(-1)^s \times 2^E \times F$

- **when exp = 00..0, represent 0**
- **when exp = 11..1**
  - frac=00..0, represent ∞
  - frac!=00..0, represent NaN or “Not a Number”

IEEE Floating Point Standard

$(-1)^s \times 2^E \times F$

- **when exp = 00..0, represent 0**
- **when exp = 11..1**
  - frac=00..0, represent ∞
  - frac!=00..0, represent NaN or “Not a Number”
- **Otherwise, E = exp-bias**
  - bias=2^{k-1}-1
  - float: k=8, bias=127

Encode 72.0f in IEEE floating point standard

$(-1)^s \times 2^E \times F$

Step 1: Write 72.0 in binary scientific notation

$72.0 = 100\ 1000 = 1.001000 \times 2^6$

Encode 72.0f in IEEE floating point standard

$(-1)^s \times 2^E \times F$

Step 1: Write 72.0 in binary scientific notation

72.0 = 100 1000 = 1.001000×2^6

Step 2: compute fractional value:

frac = 0010 0000 0000 0000 0000 0000 000

Encode 72.0f in IEEE floating point standard

$(-1)^s \times 2^E \times F$

Step 1: Write 72.0 in binary scientific notation

$72.0_{10} = 100\ 1000_2 = 1.001000 \times 2^6$

Step 2: compute fractional value:

frac = 0010 0000 0000 0000 0000 0000 000

Step 3: compute the exponent value:

E = exp – bias, exp = E+bias = 6 + 127 $= 133_{10} = 1000\ 0101_2$

Encode 72.0f in IEEE floating point standard

$(-1)^s \times 2^E \times F$

Step 1: Write 72.0 in binary scientific notation

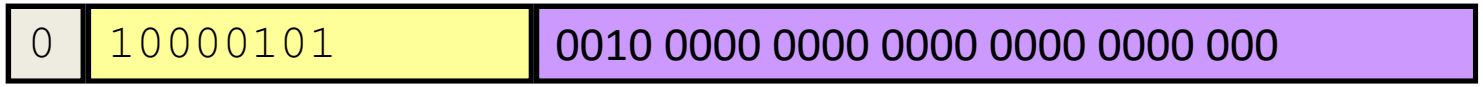
72.0 = 100 1000 = 1.001000×2^6

Step 2: compute fractional value:

frac = 0010 0000 0000 0000 0000 0000 000

Step 3: compute the exponent value:

E = exp – bias, exp = E+bias = 6 + 127 = 133 = 1000 0101



TYPEDEF

Typedef

- **Using structures is cumbersome**
  - Everything related has to include the “struct” keyword

Typedef

- **Using structures is cumbersome**
  - Everything related has to include the “struct” keyword
- **C allows you to give a type an alias**
  - `typedef` is a keyword in C
  - Give a meaningful name to an existing type

Typedef Example

- **Syntax**

```
typedef existing-type new-name

typedef unsigned char byte;
```

Typedef Example

Typedef Example

```
struct StudentRecord {
    char name[25];
    int  id;
    char gender;
    double gpa;
};
typedef struct StudentRecord StudentRecord;
```

Typedef Example

```
struct StudentRecord {
    char name[25];
    int  id;
    char gender;
    double gpa;
};
typedef struct StudentRecord StudentRecord;
struct StudentRecord s;
StudentRecord s;
```

Typedef Example

```
typedef struct  StudentRecord
{
    char name[25];
    int  id;
    char gender;
    double gpa;
} StudentRecord;
```

Typedef Example

```
typedef struct  StudentRecord
{
    char name[25];
    int  id;
    char gender;
    double gpa;
} StudentRecord;
```

Typedef Example

```
typedef struct  StudentRecord
{
    char name[25];
    int  id;
    char gender;
    double gpa;
} StudentRecord;
struct StudentRecord record;
```

Typedef Example

```
typedef struct  StudentRecord
{
    char name[25];
    int  id;
    char gender;
    double gpa;
} StudentRecord;
```

~~struct StudentRecord record;~~

Typedef Example

```
typedef struct  StudentRecord
{
    char name[25];
    int  id;
    char gender;
    double gpa;
} StudentRecord;
StudentRecord record;
```

ENUMERATION & UNION

Enumerations in C

- **What are enumerations?**
  - A convenient construct for associating names with constant values that have a type.

- **Syntax:**

```
enum Color { RED, GREEN, BLUE };
enum Color color = RED;
```

C Unions

- **What is a union?**
  - Like structures, but every field occupies the same region in memory!
  - The largest type in the union defines the total size of that union.
  - Use one member at a time

C Unions

- **What is a union?**
  - Like structures, but every field occupies the same region in memory!
  - The largest type in the union defines the total size of that union.
  - Use one member at a time
- **Example:**

```
union value {
  float f;
  int i;
  char s;
};
```

```
union value v;
v.f = 45.7;
v.i = 12;
v.s = 'X';
```

C Unions

Example:

```
#include <stdio.h>

union ufloat {
  float f;
  unsigned u;
};

int main()
{
   union ufloat u1;
   u1.f = 72.0f;
   printf("%X\n", u1.u);
}
```

iClicker question

```
union value { float f; int i; char s; };
struct value { float f; int I; char s; };
```

The sizes of the union and the struct are (on x64 with gcc):

A. union: 12 bytes, struct 12 bytes

B. union: 9 bytes, struct 12 bytes

C. union: 4 bytes, struct 9 bytes

D. union: 5 bytes, struct 12 bytes

E. union 4 bytes, struct 12 bytes

union.c example

- **Let us compile this example**
  - Compilers usually maintain information about variables, this example is the start of a data structure for doing this…
  - Note how the different types interpret the bits differently!
  - This example shows how character arrays and integers are interpreted differently!

C POINTER

C Pointers

What is a pointer?

C Pointers

What is a pointer?

A pointer is like a mailing address,
it tells you where something is **located**.

C Pointers

What is a pointer?

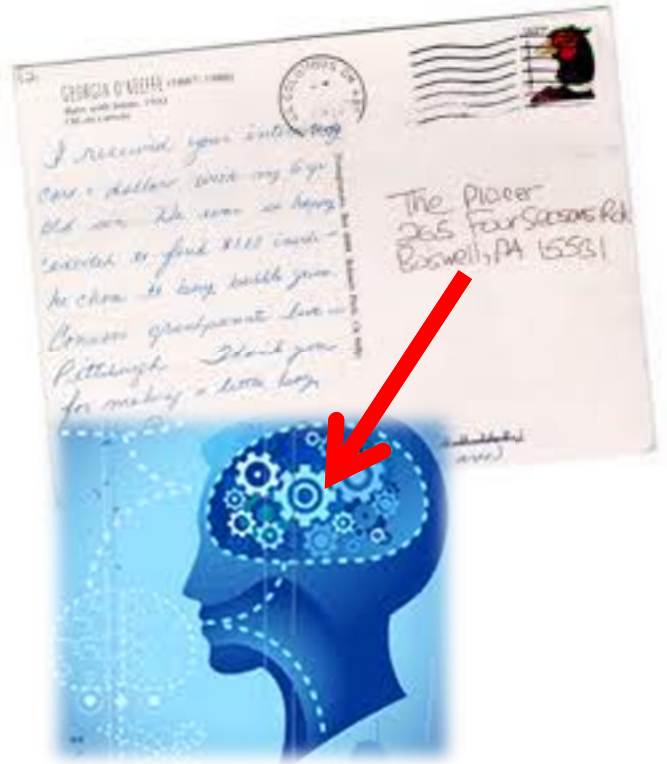
A pointer is like a mailing address,
it tells you where something is **located**.

Every object (including simple data types)
in Java and C reside in the **memory**
of the machine.

C Pointers

What is a pointer?

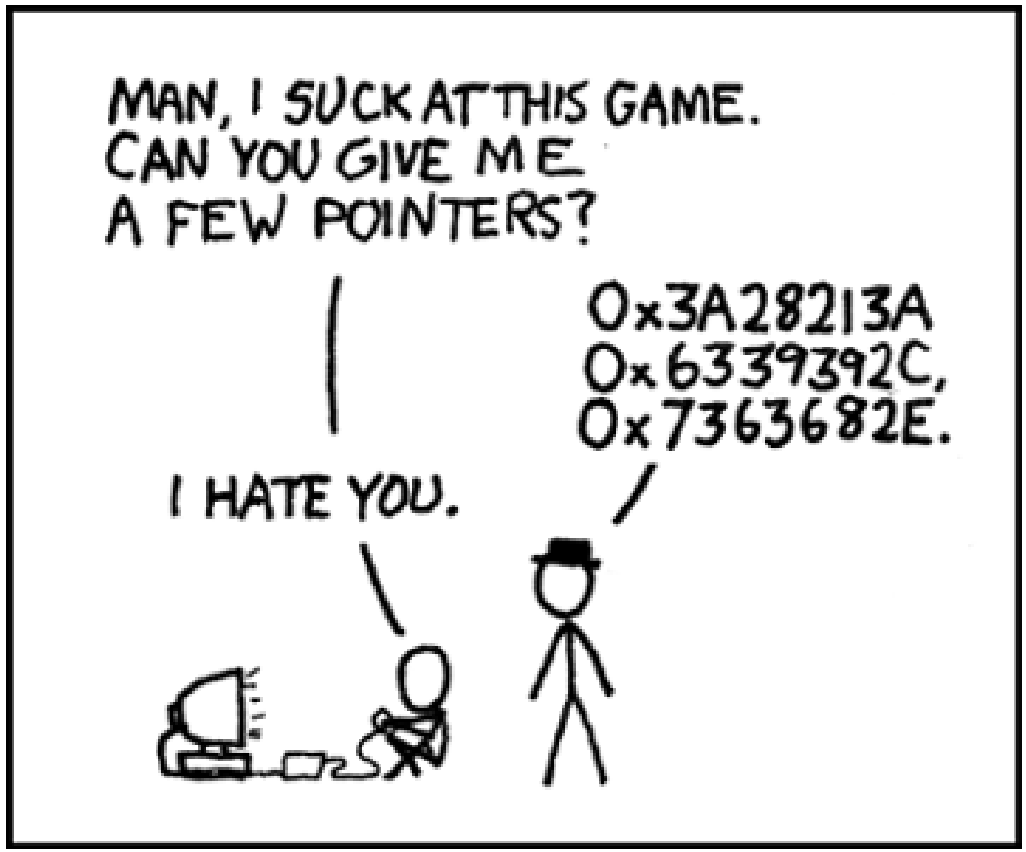
A pointer is like a mailing address,
it tells you where something is **located**.



Every object (including simple data types)
in Java and C reside in the **memory**
of the machine.

A **pointer** to an object is an “address” telling
You where the object is **located** in **memory**.

C Pointers


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

C Pointers

So why do I care about pointers?

C Pointers

So why do I care about pointers?

In Java, you do not have access to these pointers (or addresses).

In Java, you do **not** have **access to the address** of an object.

This provides **safety**!

C Pointers

So why do I care about pointers?

In Java, you do not have access to these pointers (or addresses).

In C, you do have **access to the address** of an object, which allows you to **interact with hardware.**

C Pointers

```
#include <stdio.h>

int main() {
  int *ptr;
  int *ptr2;
}
```

A pointer is denoted by ‘*’ and has a type.

The type is the kind of thing assumed to be at the location the pointer refers to.

C Pointers

```
#include <stdio.h>

int main() {
   int *ptr;
   int *ptr2;
   int x = 2;
   int y = 5;
}
```

C Pointers

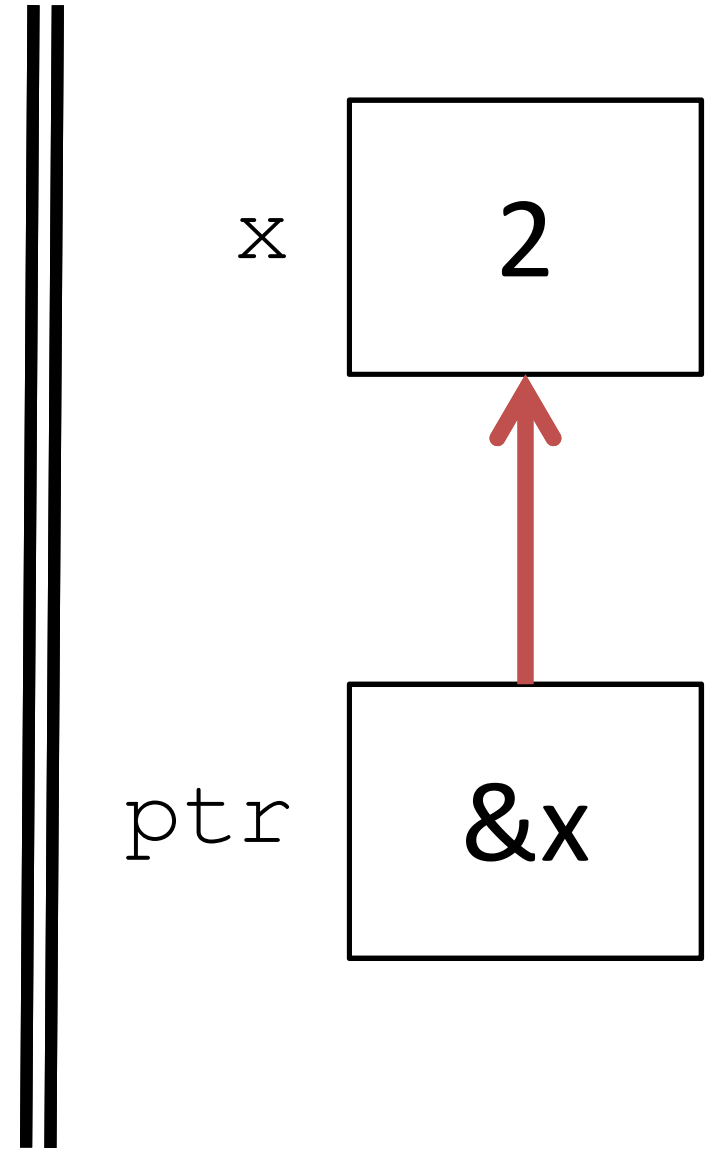
```
#include <stdio.h>

int main() {
   int *ptr;
   int *ptr2;
   int x = 2;
   int y = 5;
   ptr  = &x;
   ptr2 = &y;
}
```

You can assign an “address” to a pointer using the “address of” (&) operator.

A Visual…

```
int x = 2;
```

```
int ptr = &x;
```

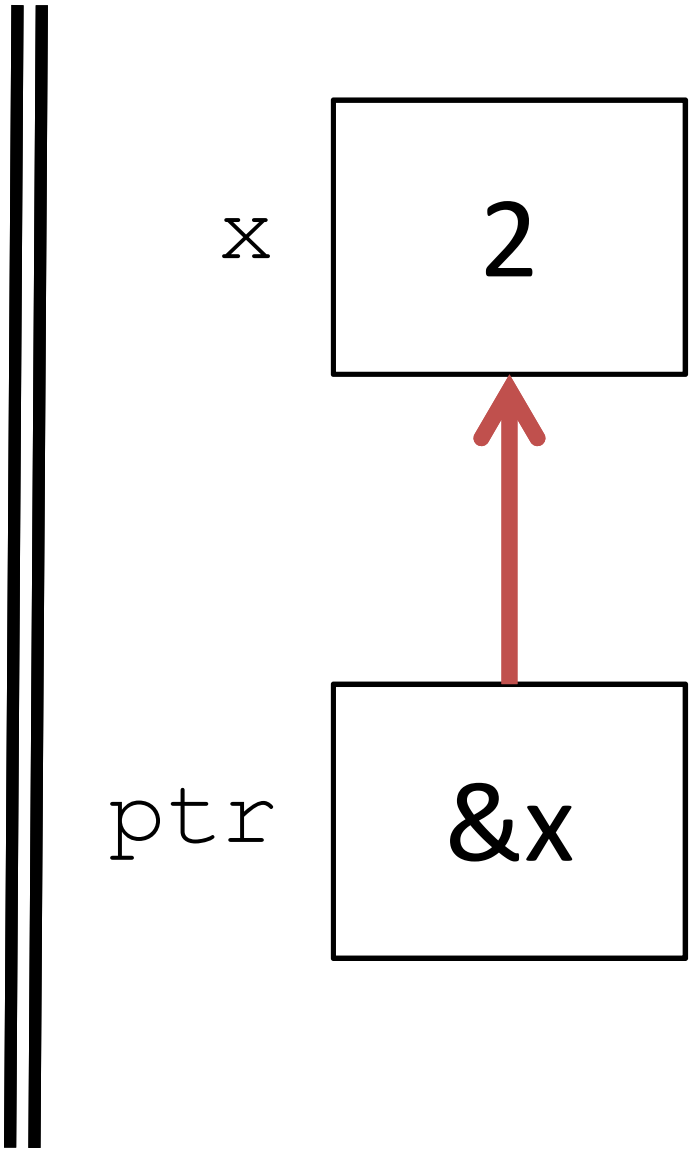


A Visual…

```
int x = 2;
```

A pointer can point to itself.

```
int ptr = &x;
```



C Pointers

```
#include <stdio.h>

int main() {
   int *ptr;
   int *ptr2;
   int x = 2;
   int y = 5;
   ptr  = &x;
   ptr2 = &y;
}
```

So, if `ptr` is a pointer that refers to a value in memory… How do we get the value?

C Pointers

```
#include <stdio.h>

int main() {
  int *ptr;
  int *ptr2;
  int x = 2;
  int y = 5;
  ptr  = &x;
  ptr2 = &y;
  printf("Value   : *ptr = %d\n", *ptr);
  printf("Address:  ptr = %d\n",  ptr);
}
```

You *dereference* (follow) the pointer! (Note the * !)

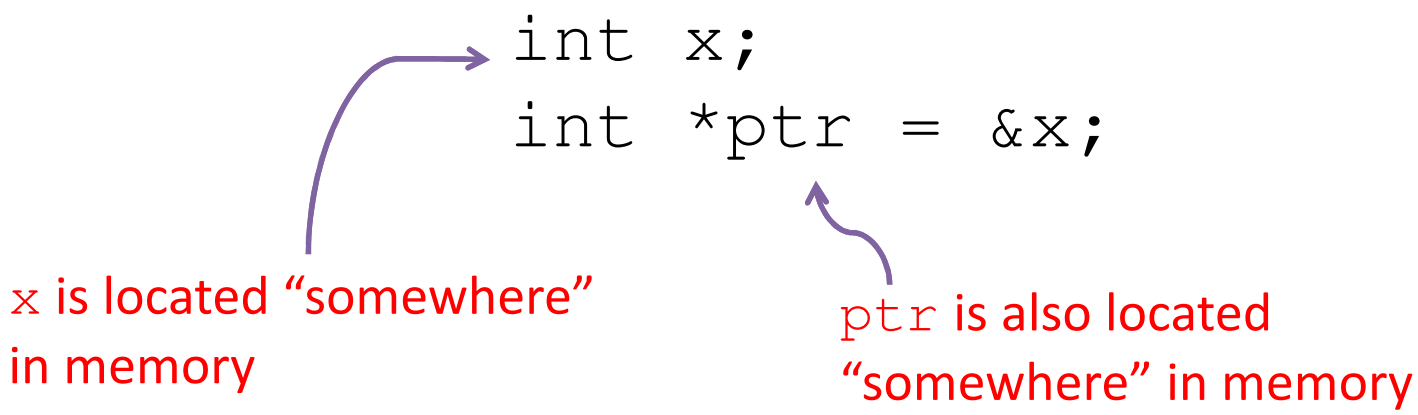
C Pointers

Imagine we have the following declarations…

```
int x;
int *ptr = &x;
```

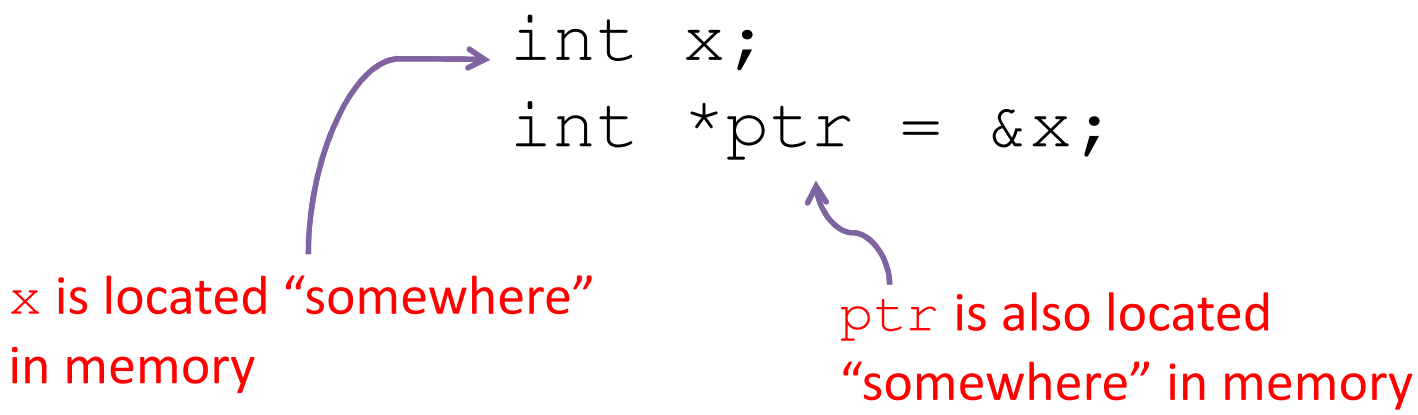
C Pointers

Imagine we have the following declarations…

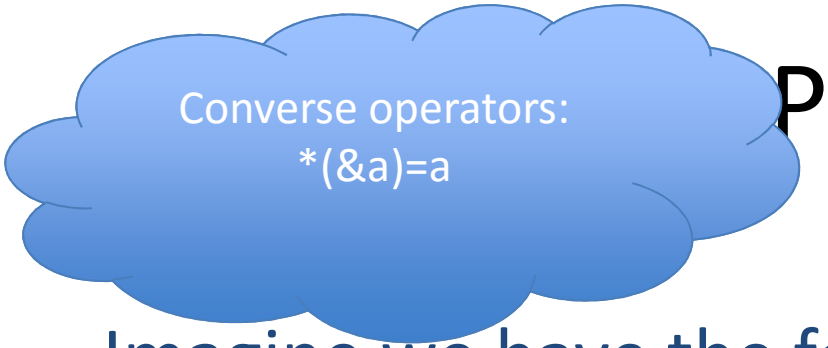


C Pointers

Imagine we have the following declarations…



`ptr` “points” to the location representing `x`.



Pointers

Imagine we have the following declarations…

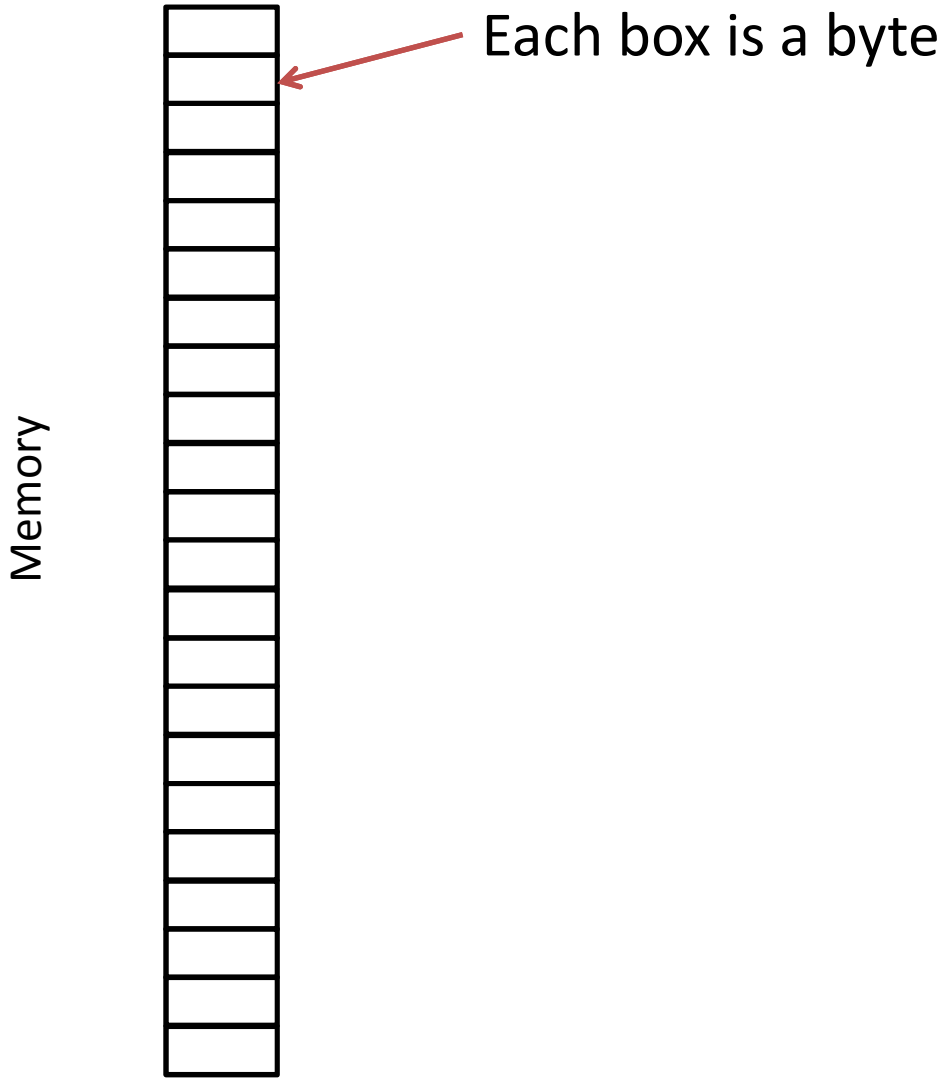
```
int x;
int *ptr = &x;
```

`x` is located "somewhere" in memory

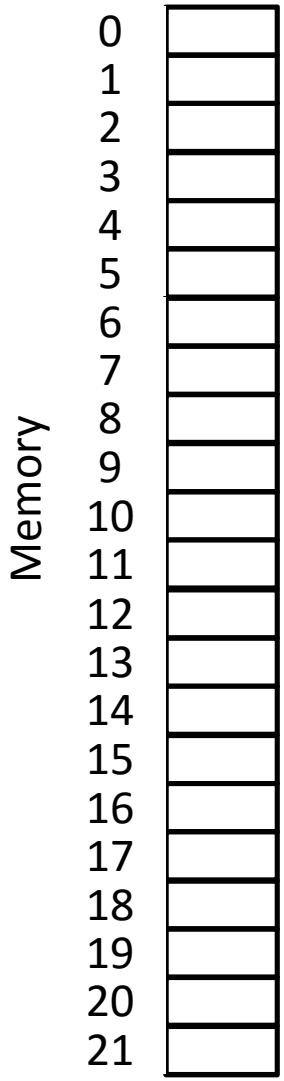
`ptr` is also located "somewhere" in memory

`ptr` "points" to the location representing `x`.

C Pointers



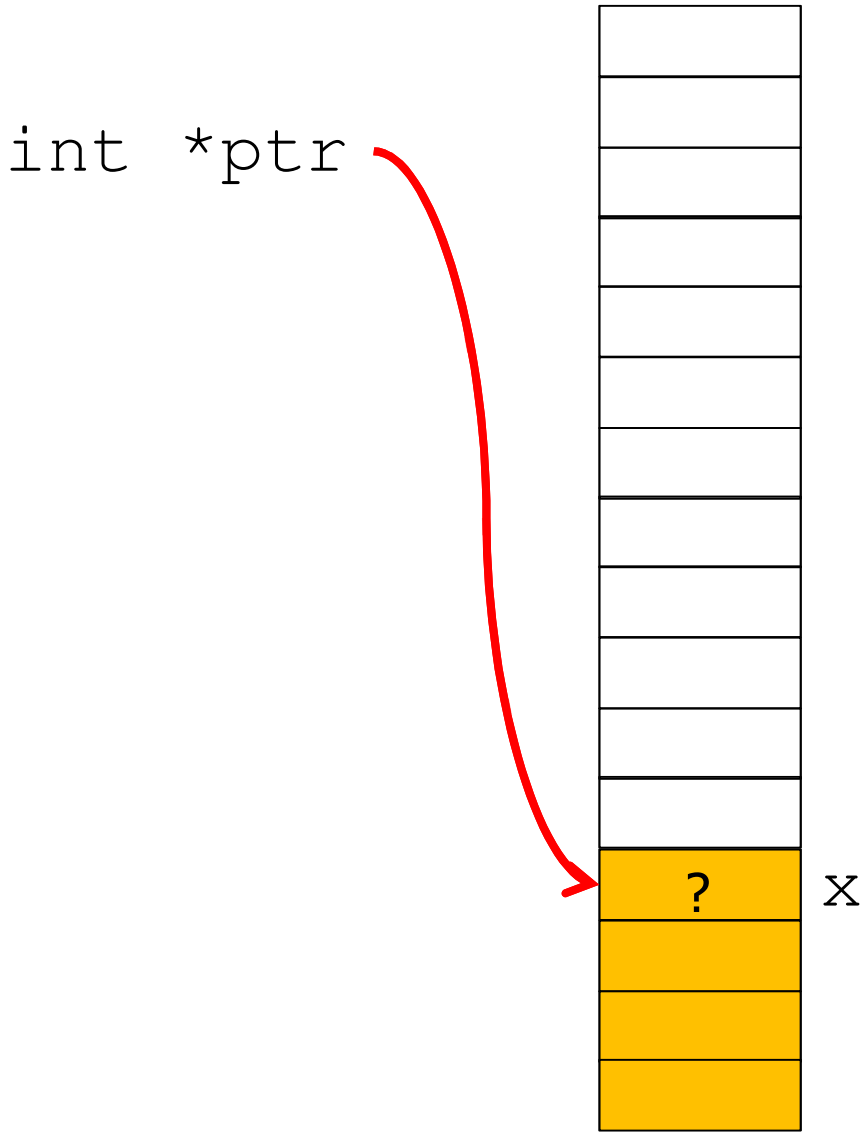
C Pointers



Each box is a byte
and has a location.

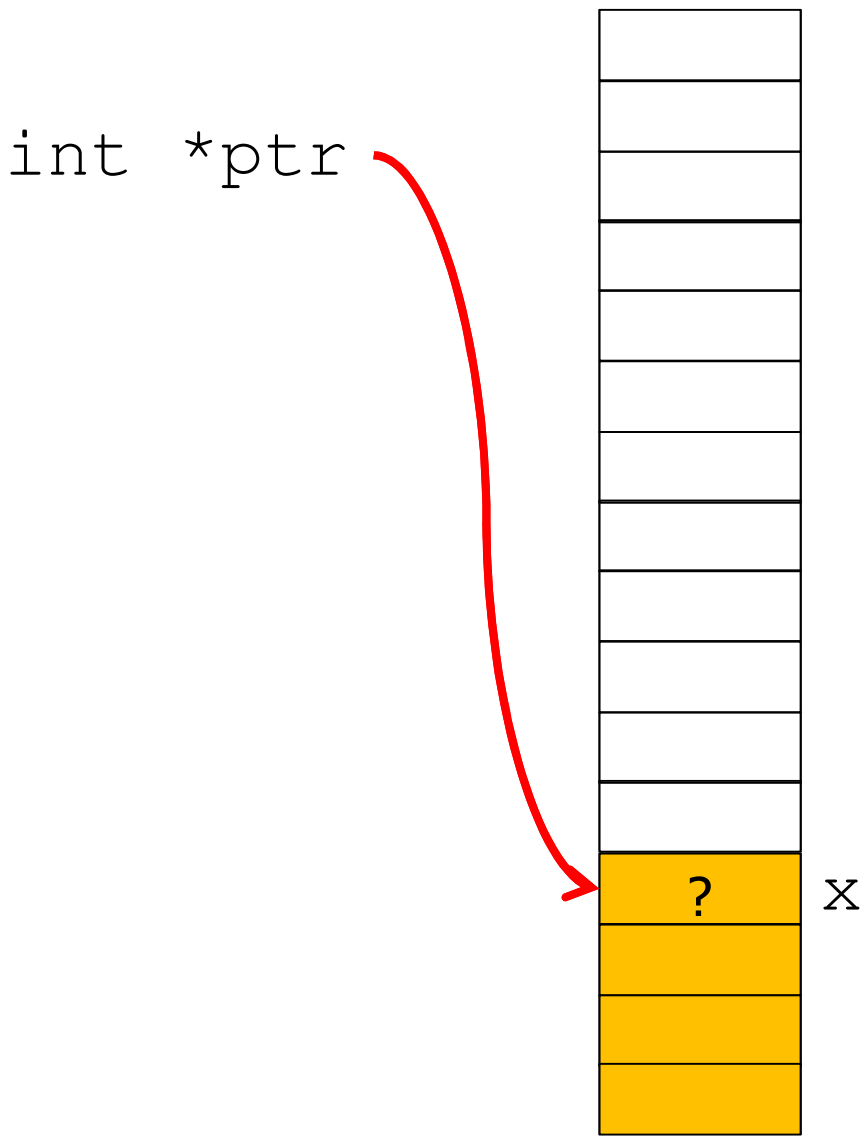
Memory is very much like a a giant character array!

C Pointers



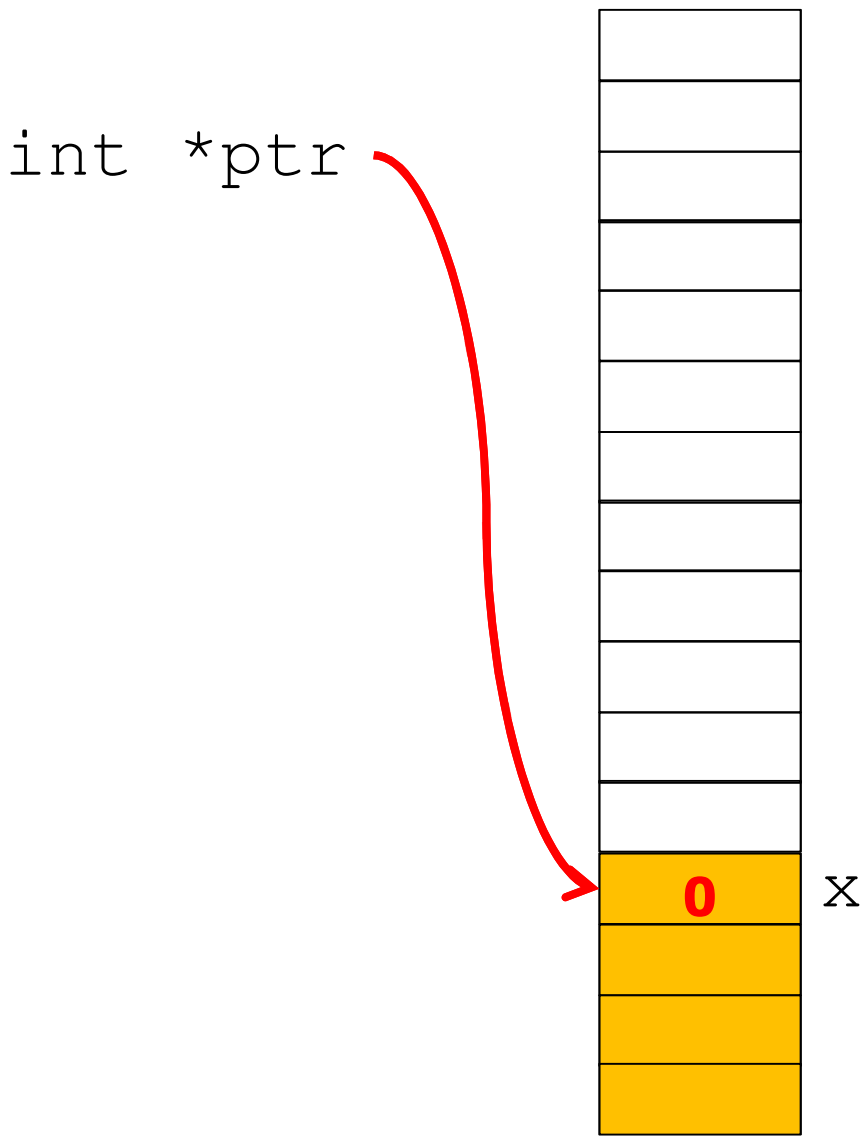

```
int x;
int *ptr = &x;
```

C Pointers



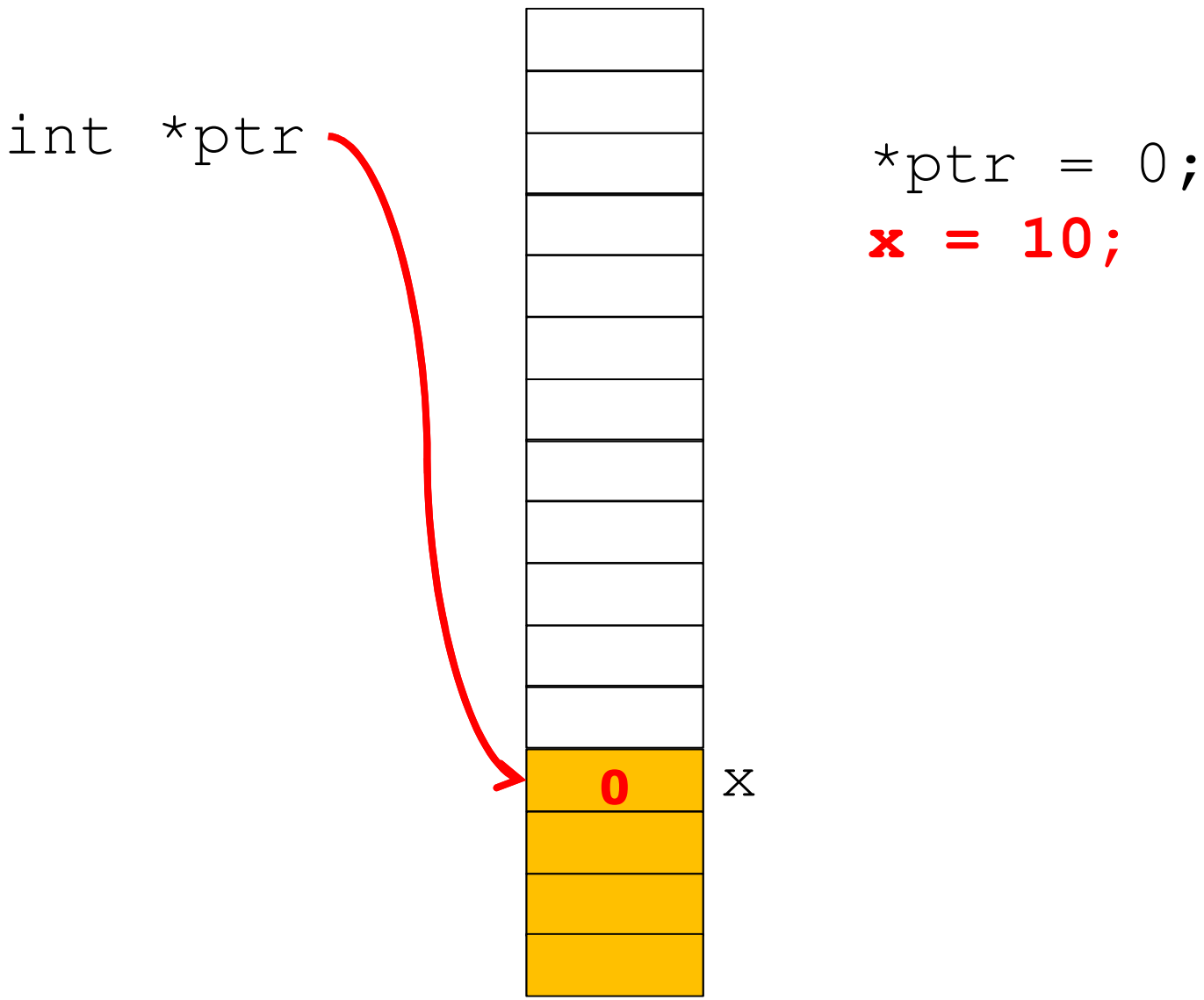

```
*ptr = 0;
```

C Pointers



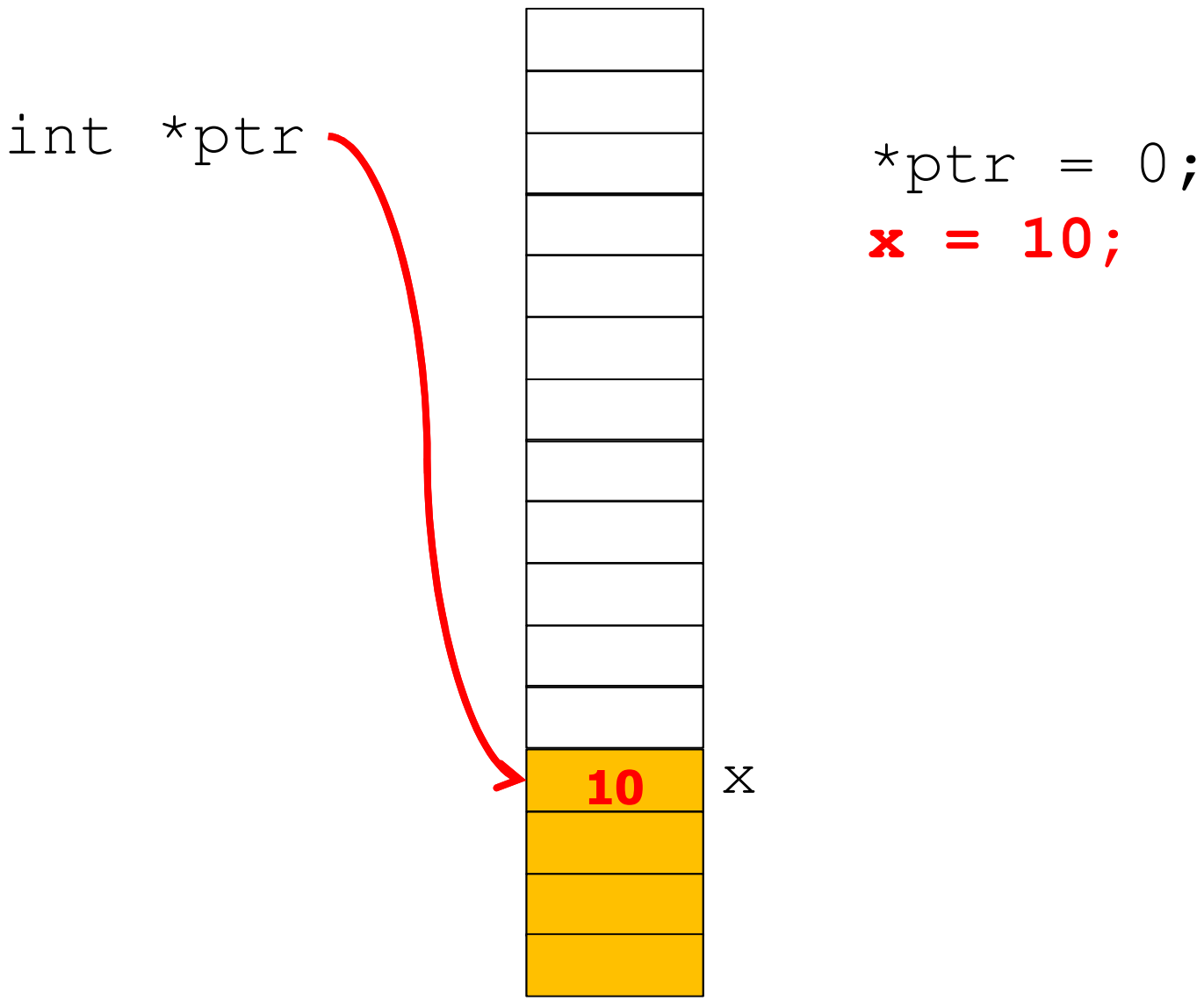

```
*ptr = 0;
```

C Pointers




```
*ptr = 0;
x = 10;
```

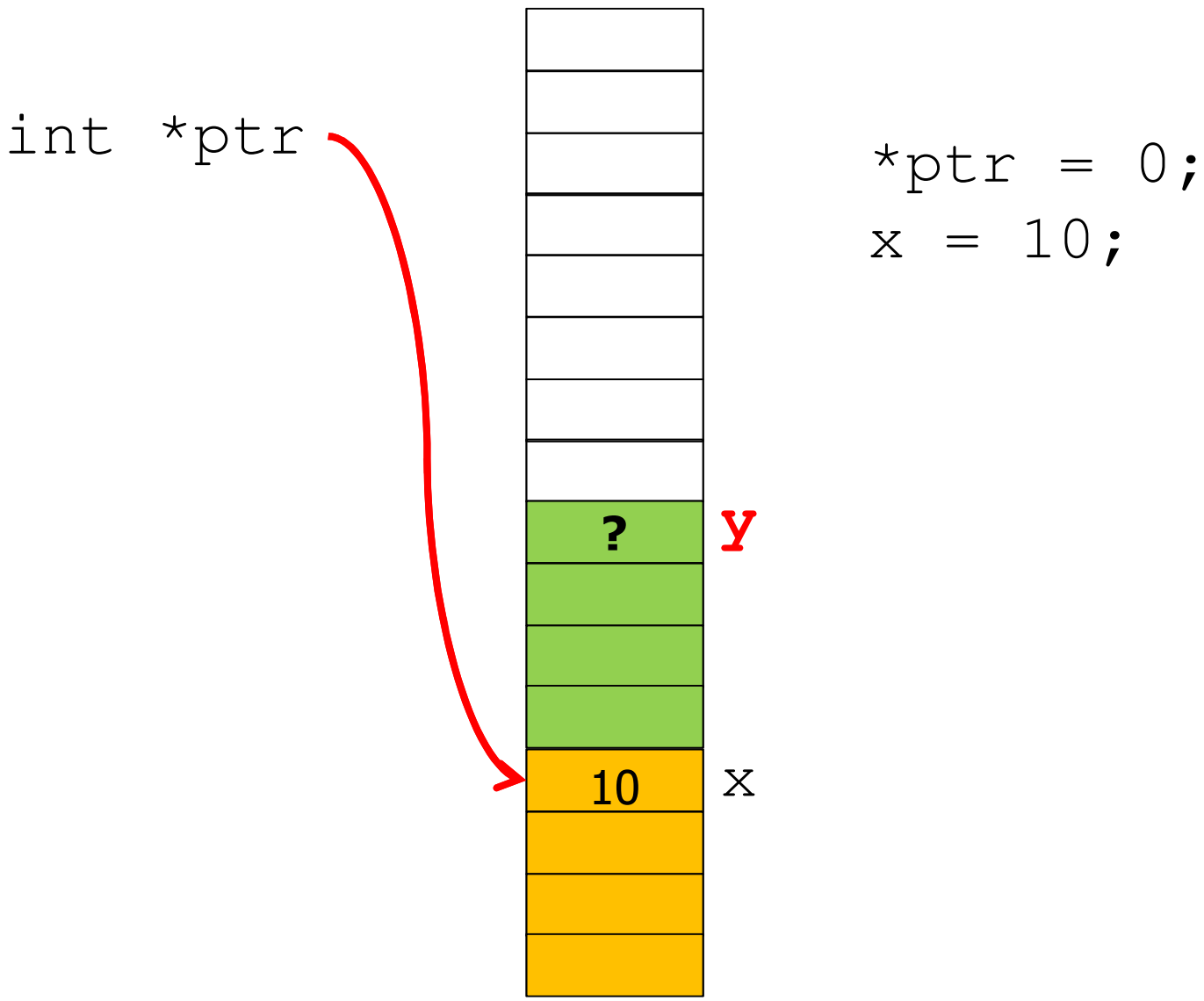
C Pointers



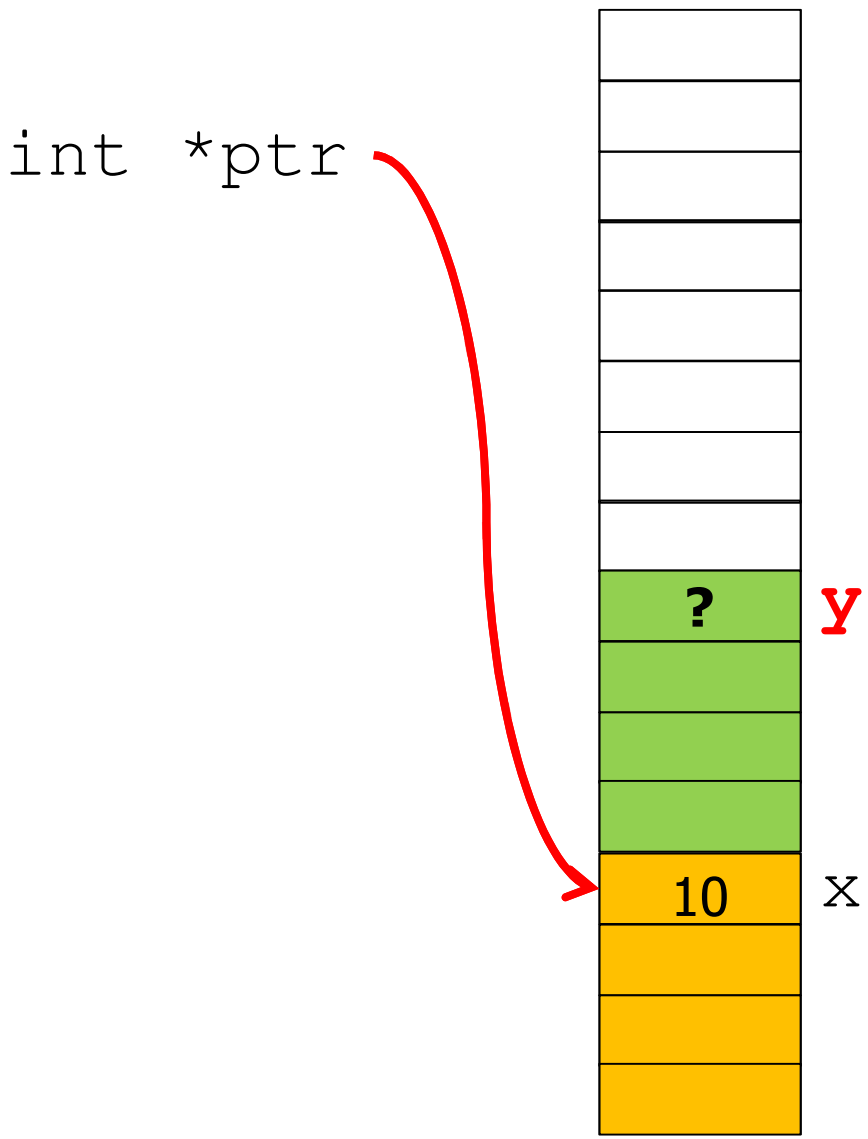

```
*ptr = 0;
```

x = 10;

C Pointers



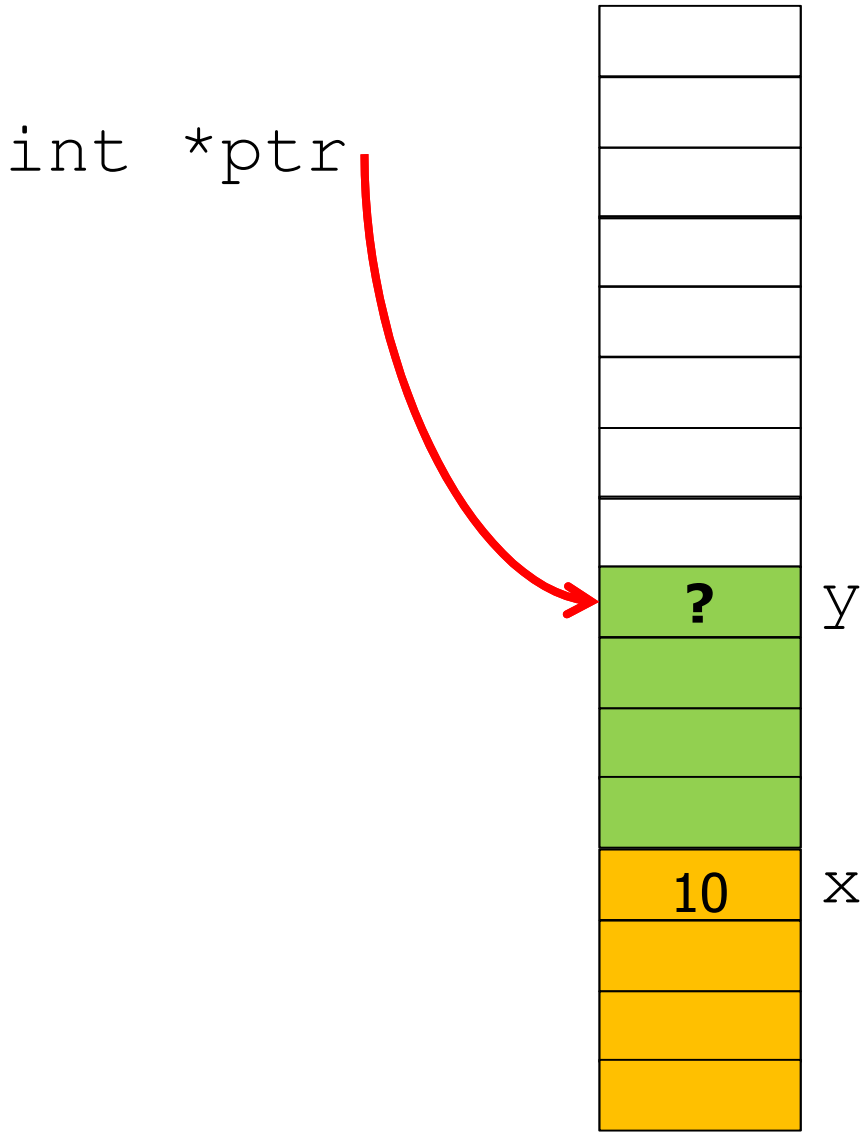
C Pointers




```
*ptr = 0;
x = 10;
ptr = &y;
```

What does this do?

C Pointers

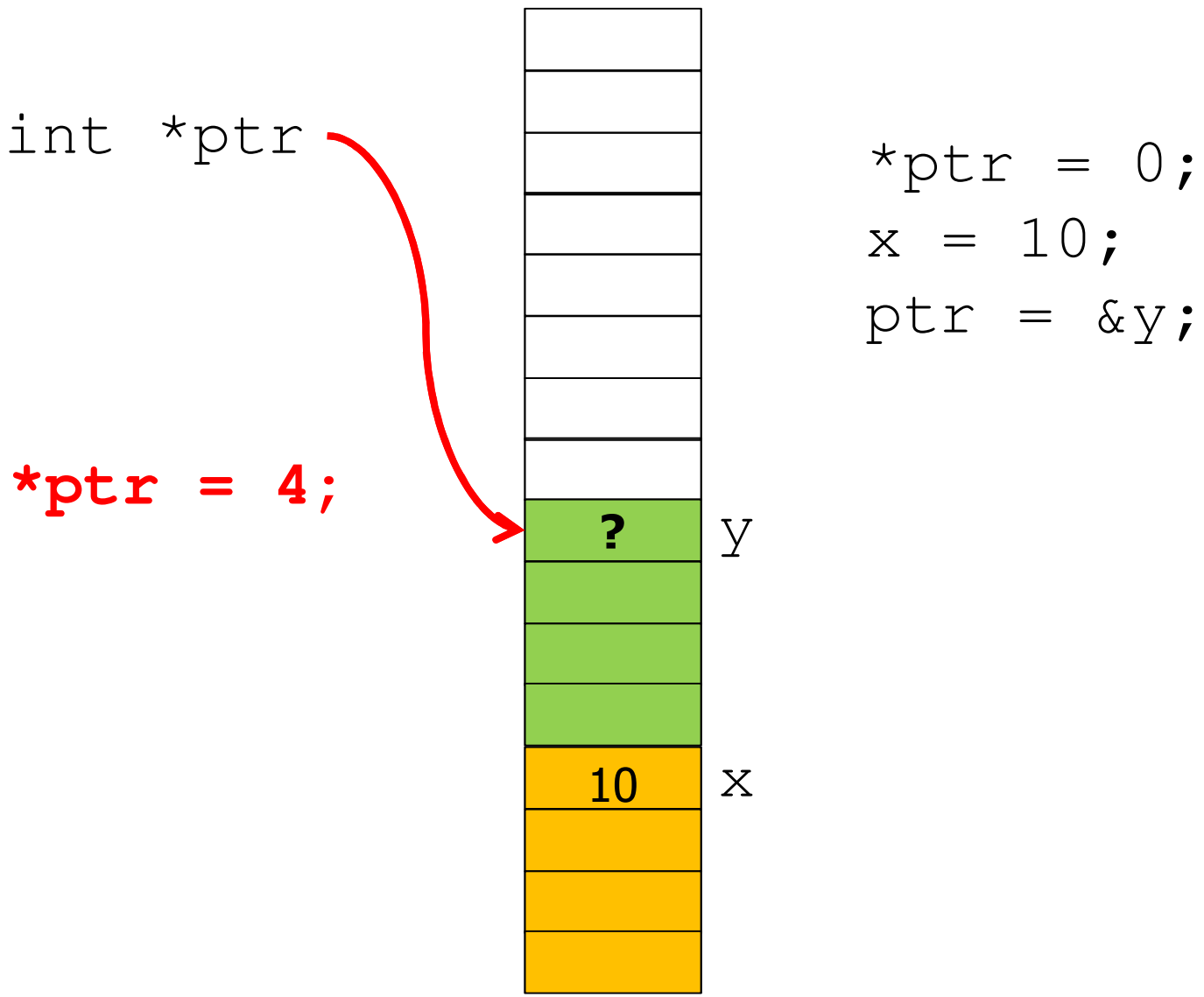



```
*ptr = 0;
x = 10;
ptr = &y;
```

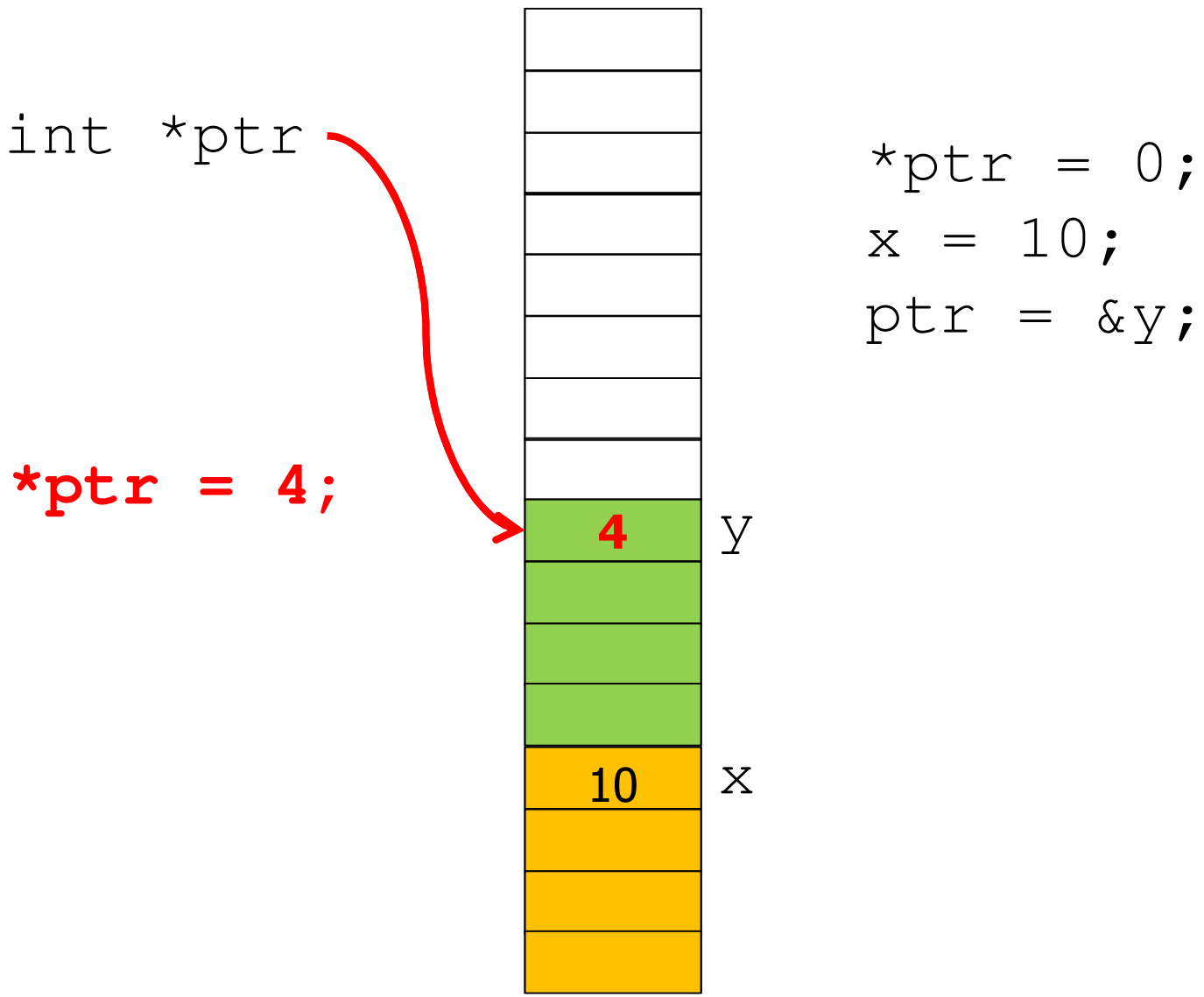
What does this do?

The pointer (ptr) is assigned a different address.

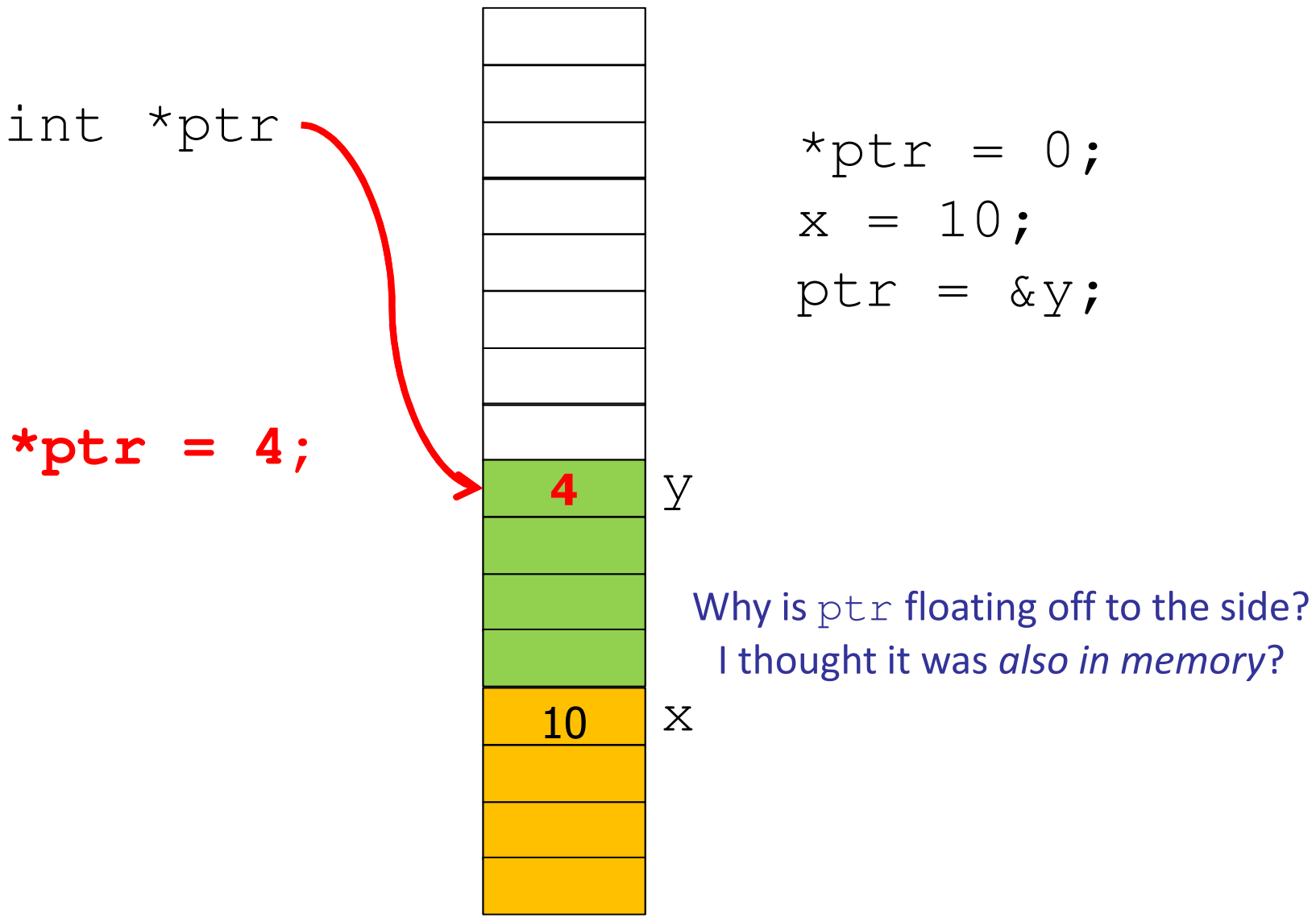
C Pointers



C Pointers



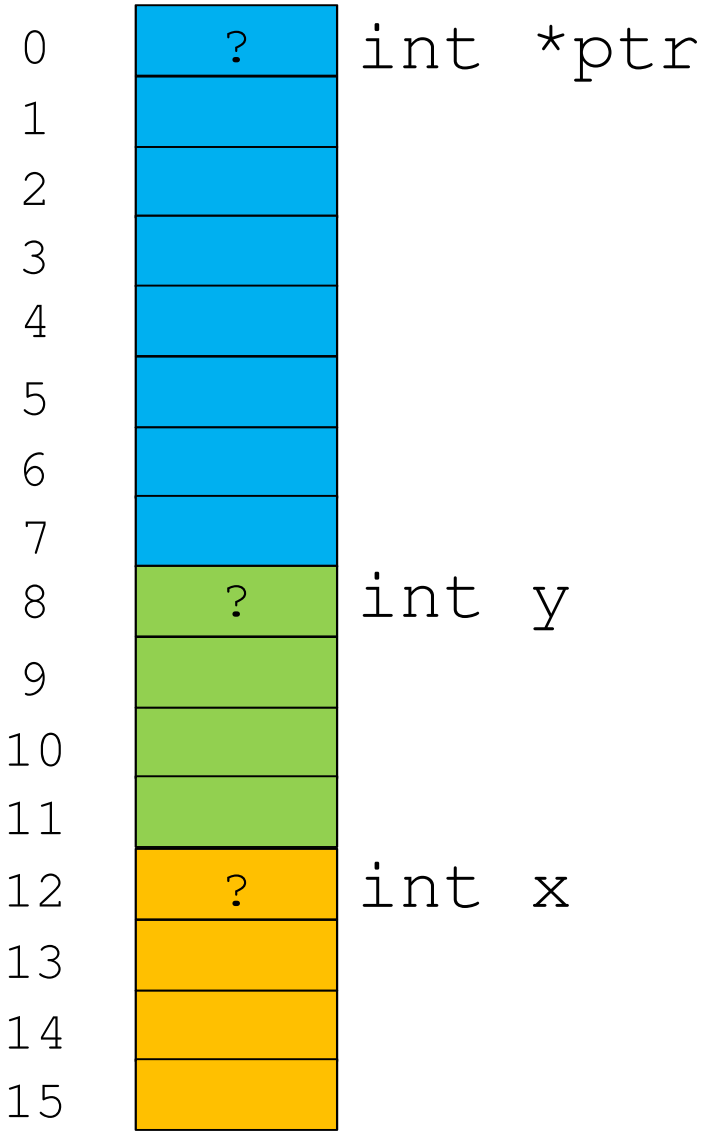
C Pointers


int *ptr
*ptr = 4;
4
y
10
x
*ptr = 0;
x = 10;
ptr = &y;
Why is ptr floating off to the side?
I thought it was *also in memory*?

C Pointers

Let us look at this a little more carefully…

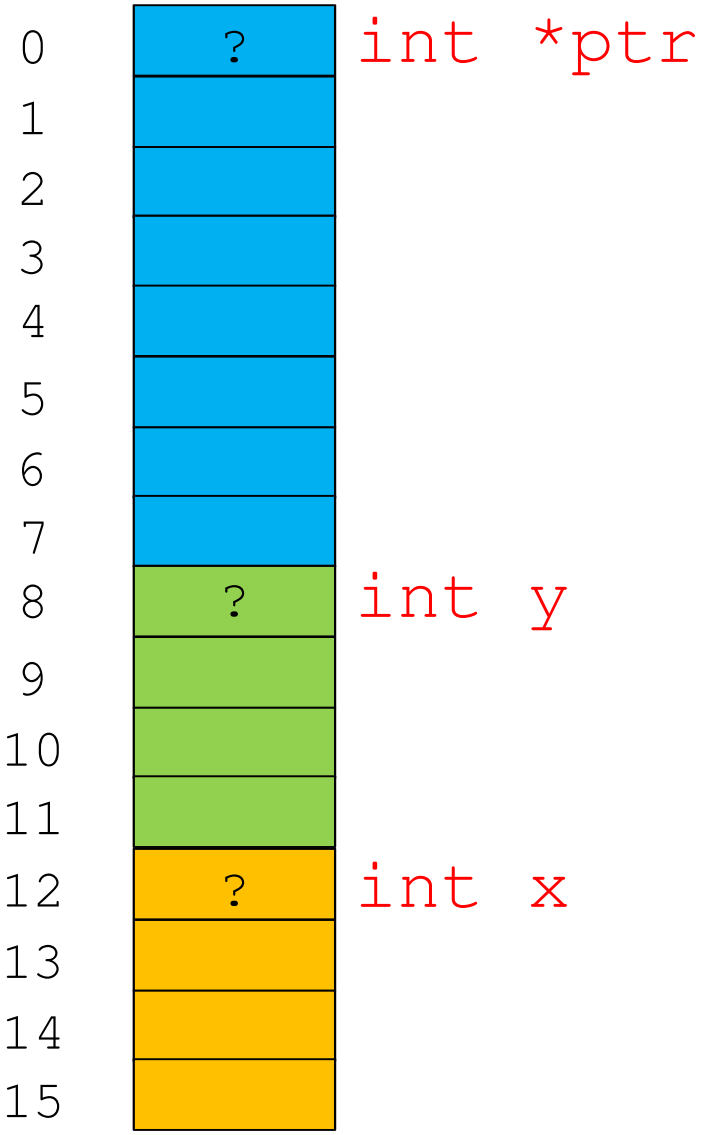
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

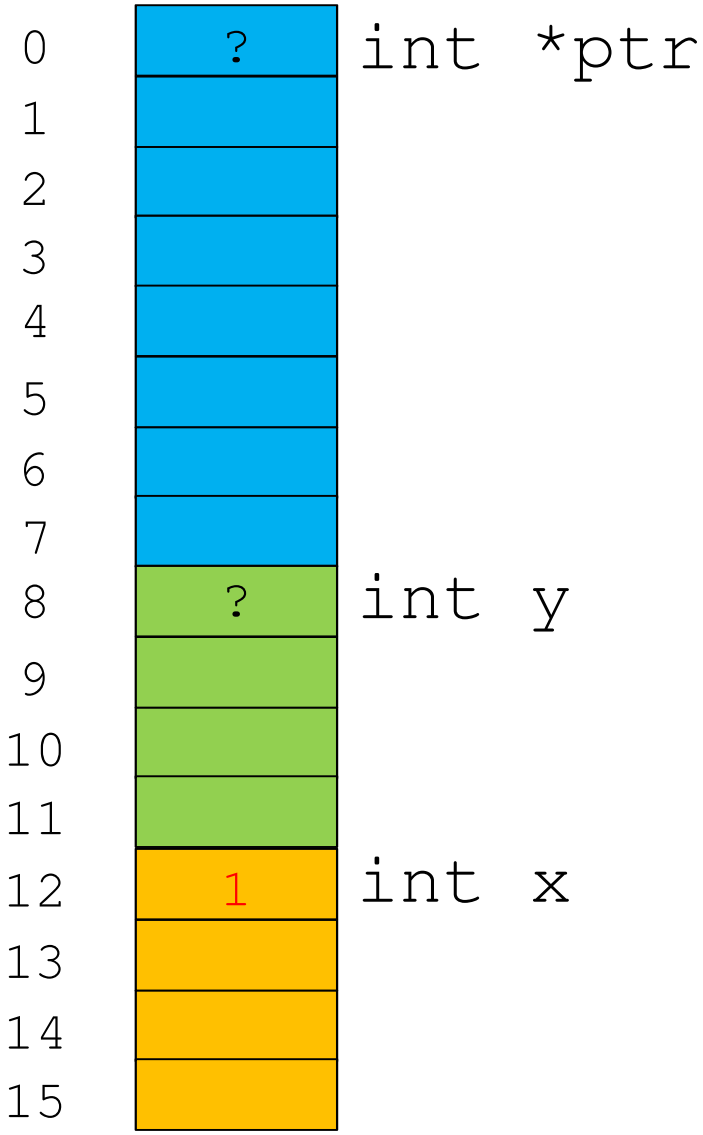
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

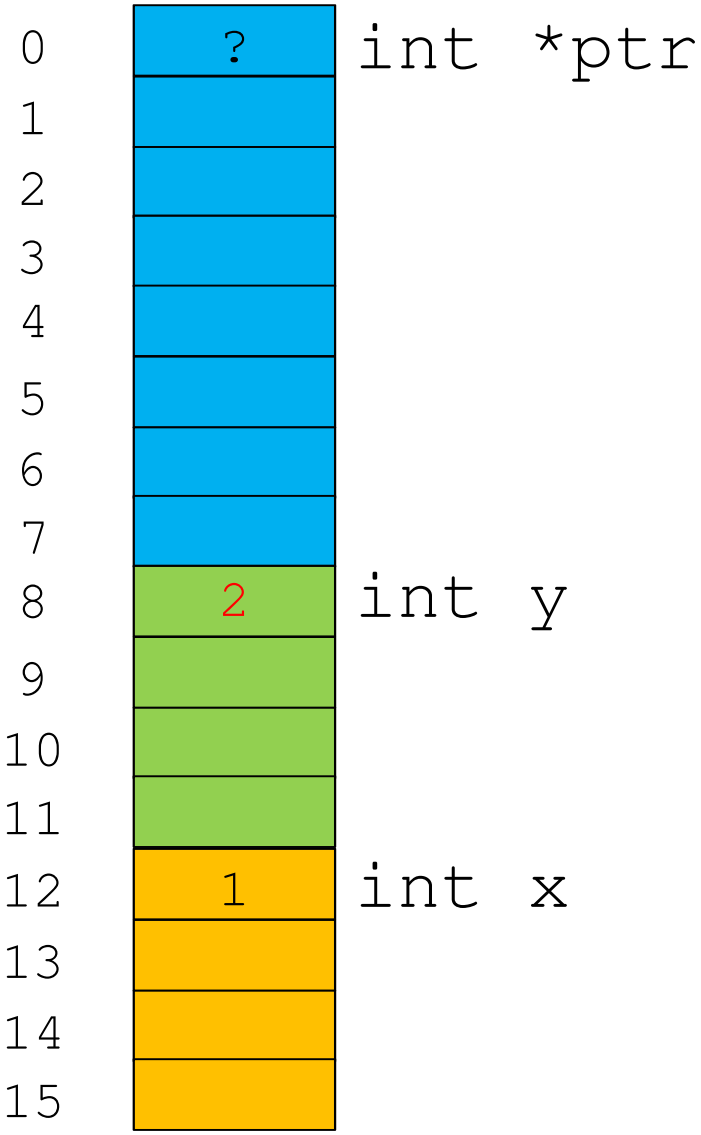
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

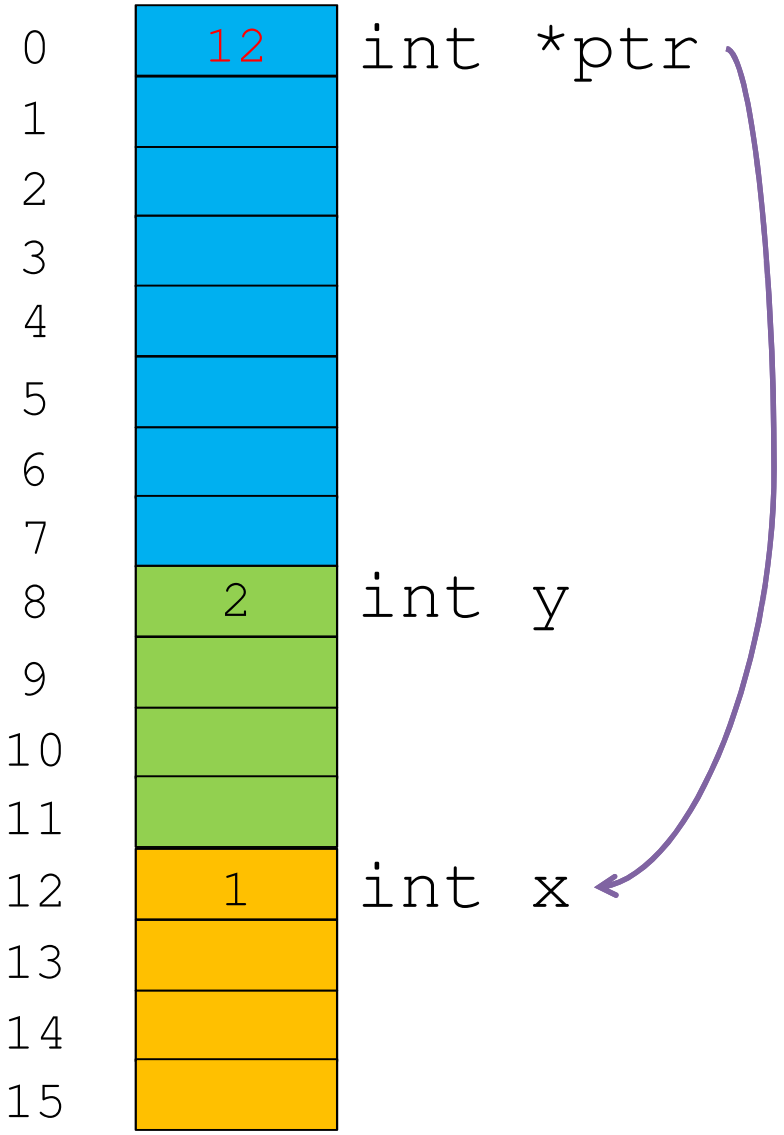
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

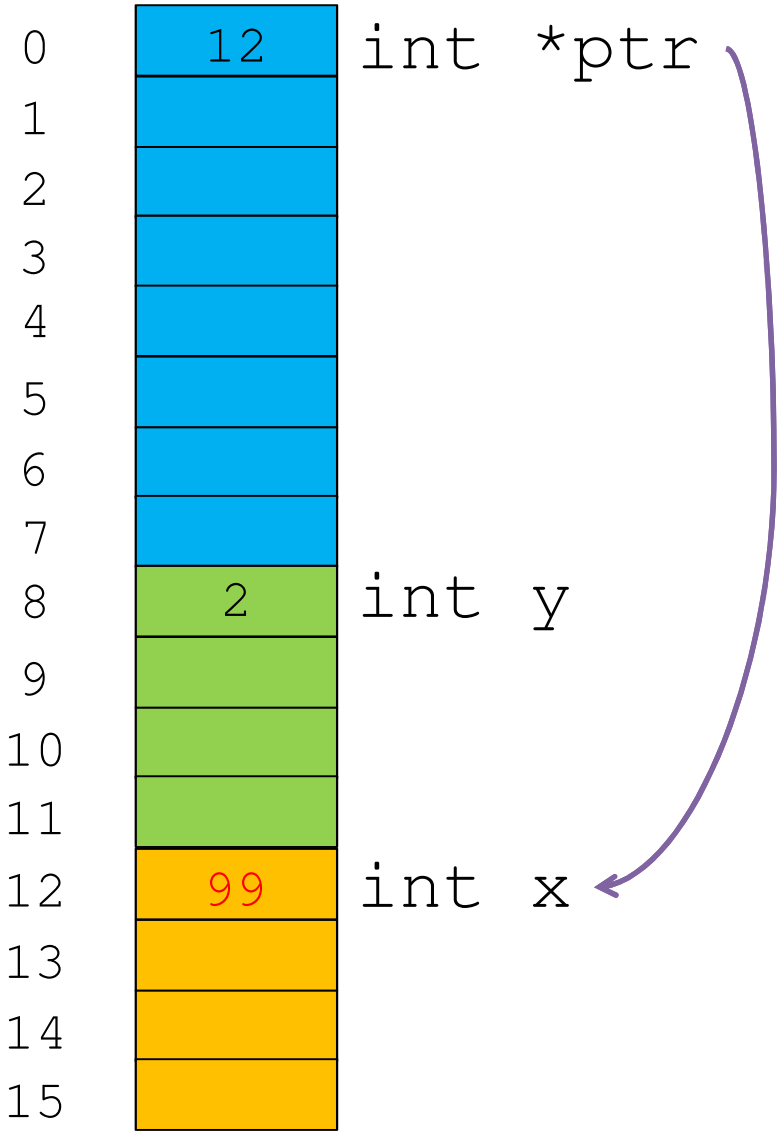
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

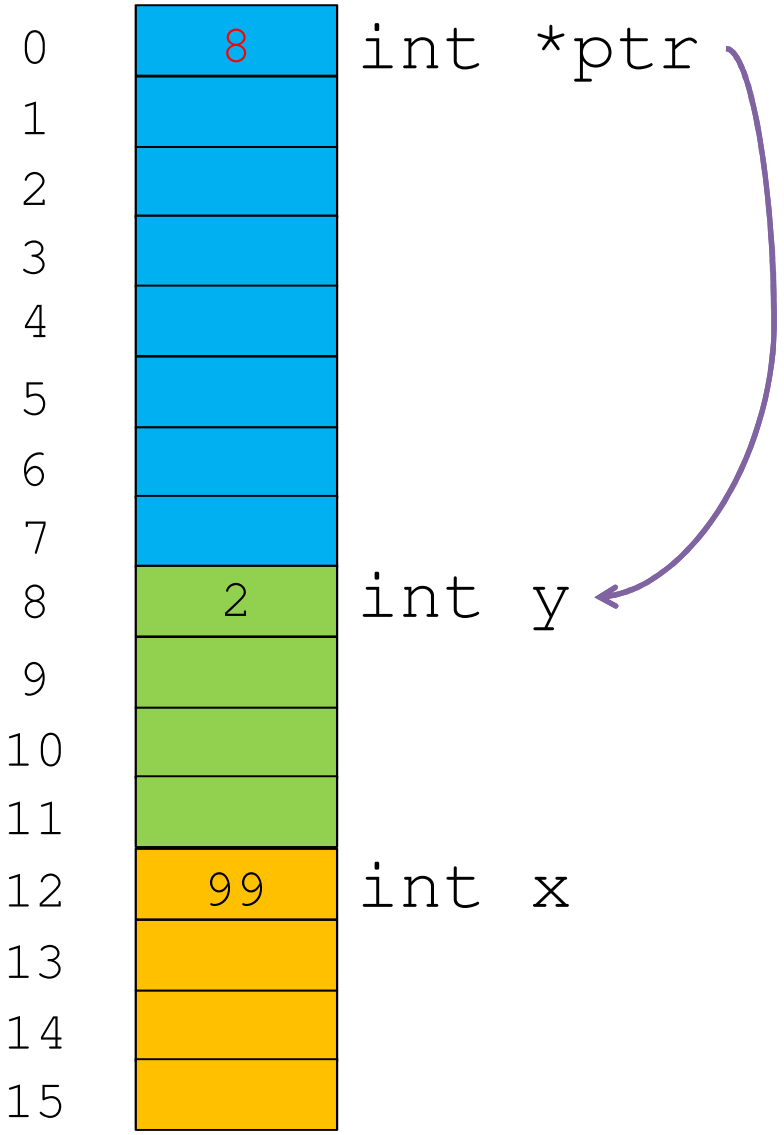
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

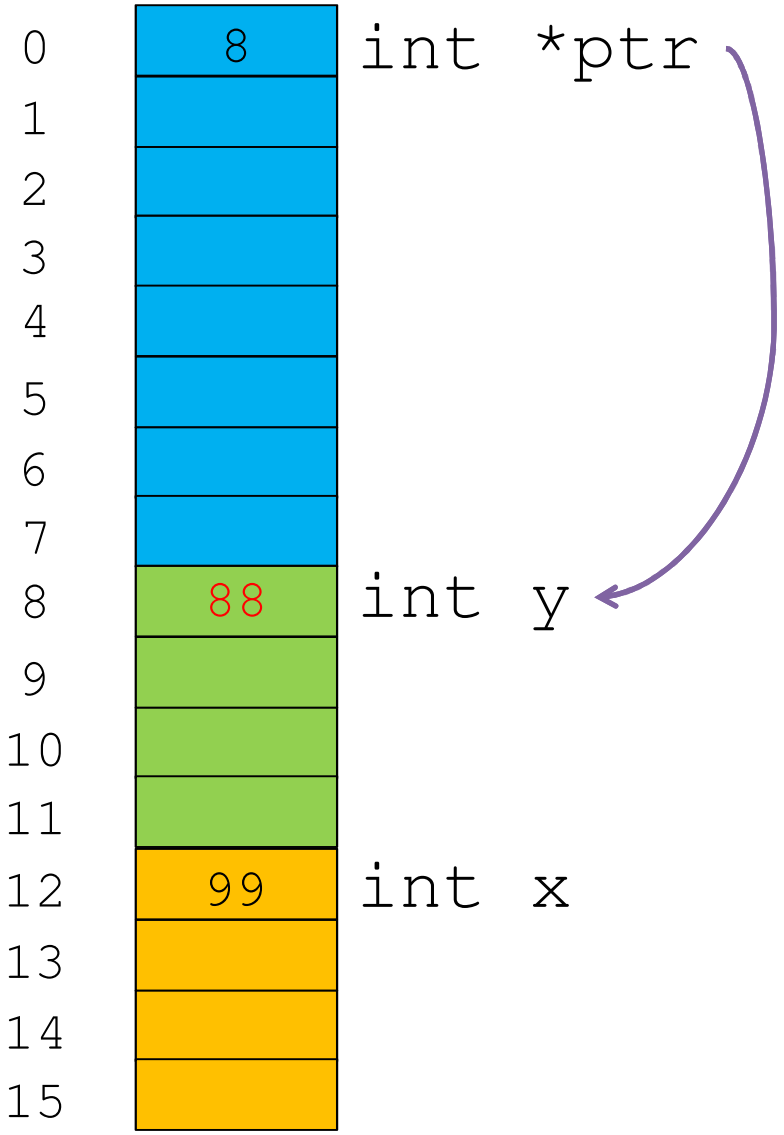
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

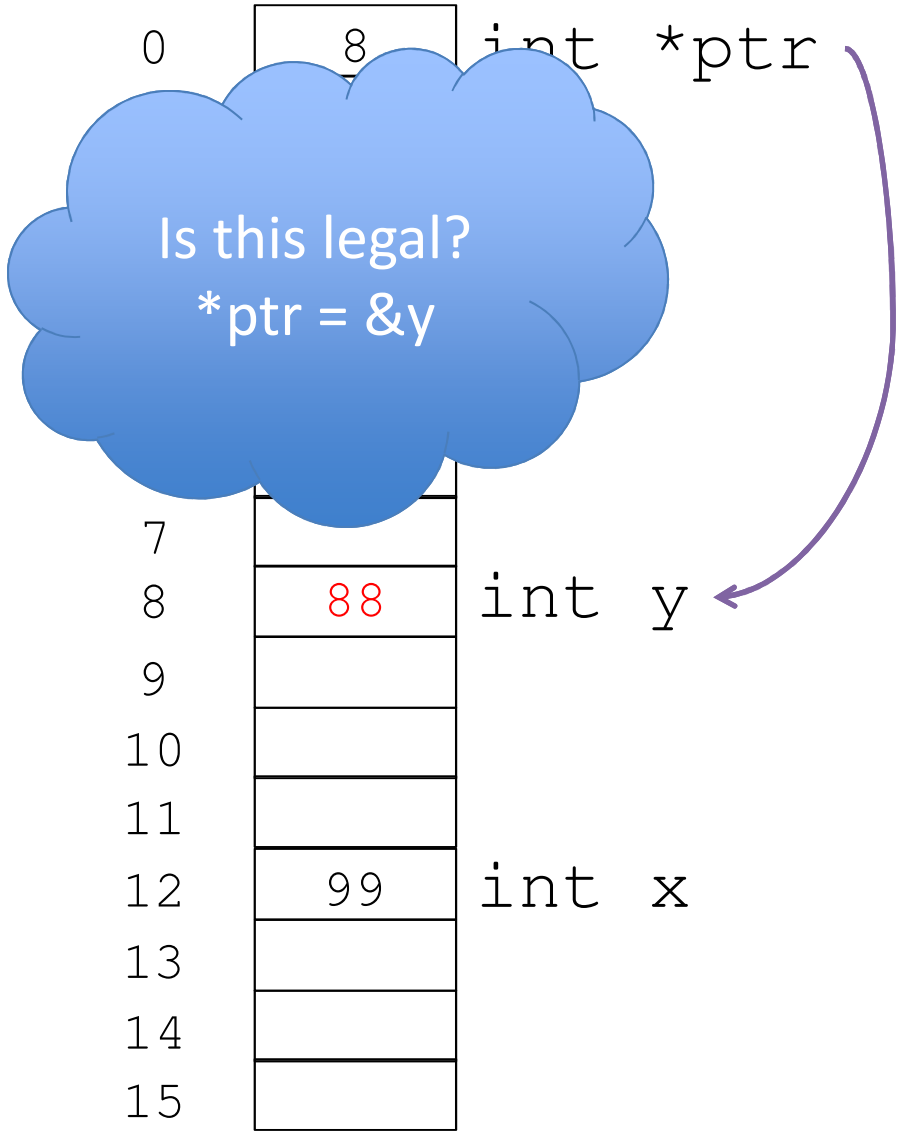
```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```



C Pointers

Let us look at this a little more carefully…

```
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```

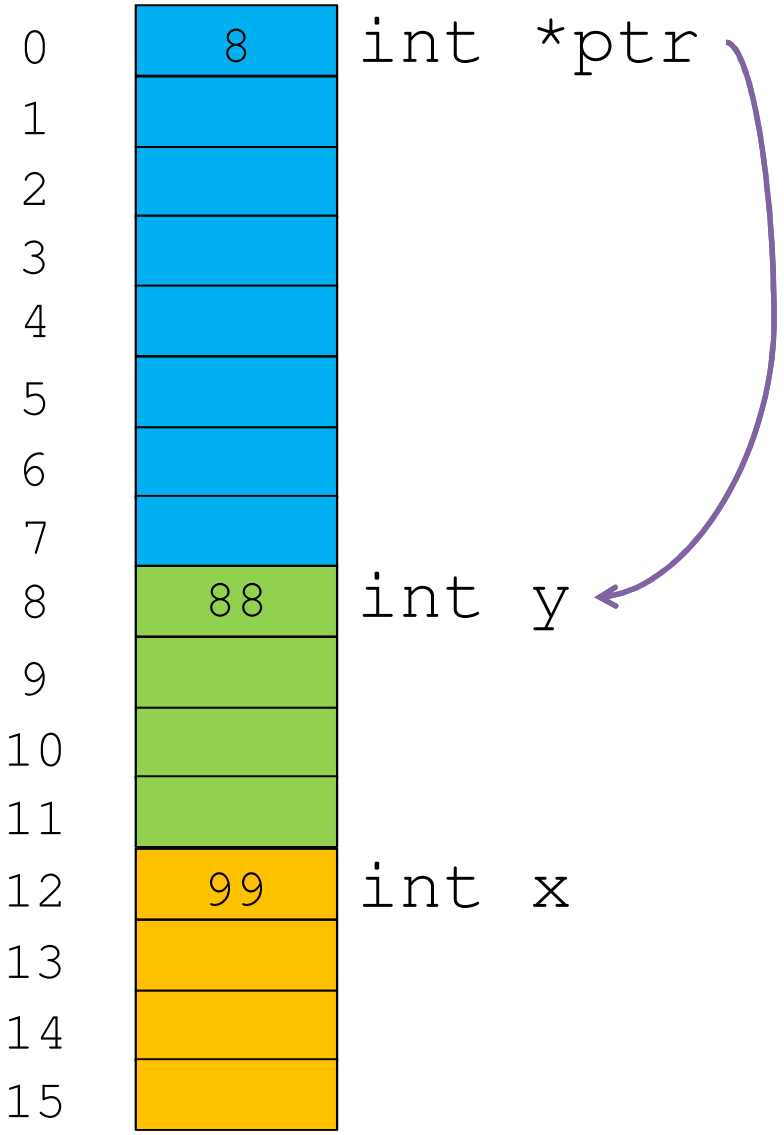


Pointer to pointer

Let us look at this a little more carefully…

What if we do this?

```
int **dptr = &ptr;
```

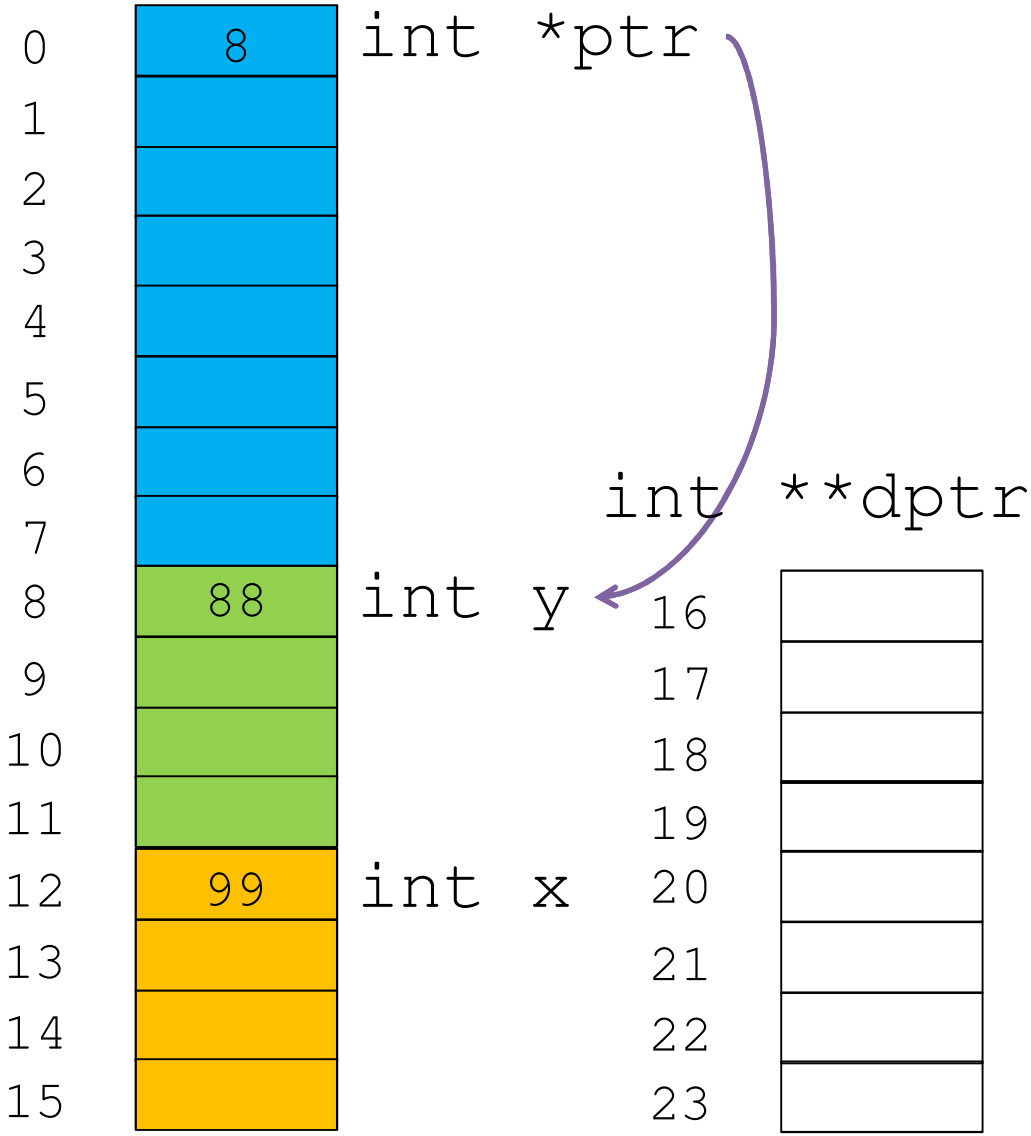


Pointer to pointer

Let us look at this a little more carefully…

What if we do this?

```
int **dptr = &ptr;
```

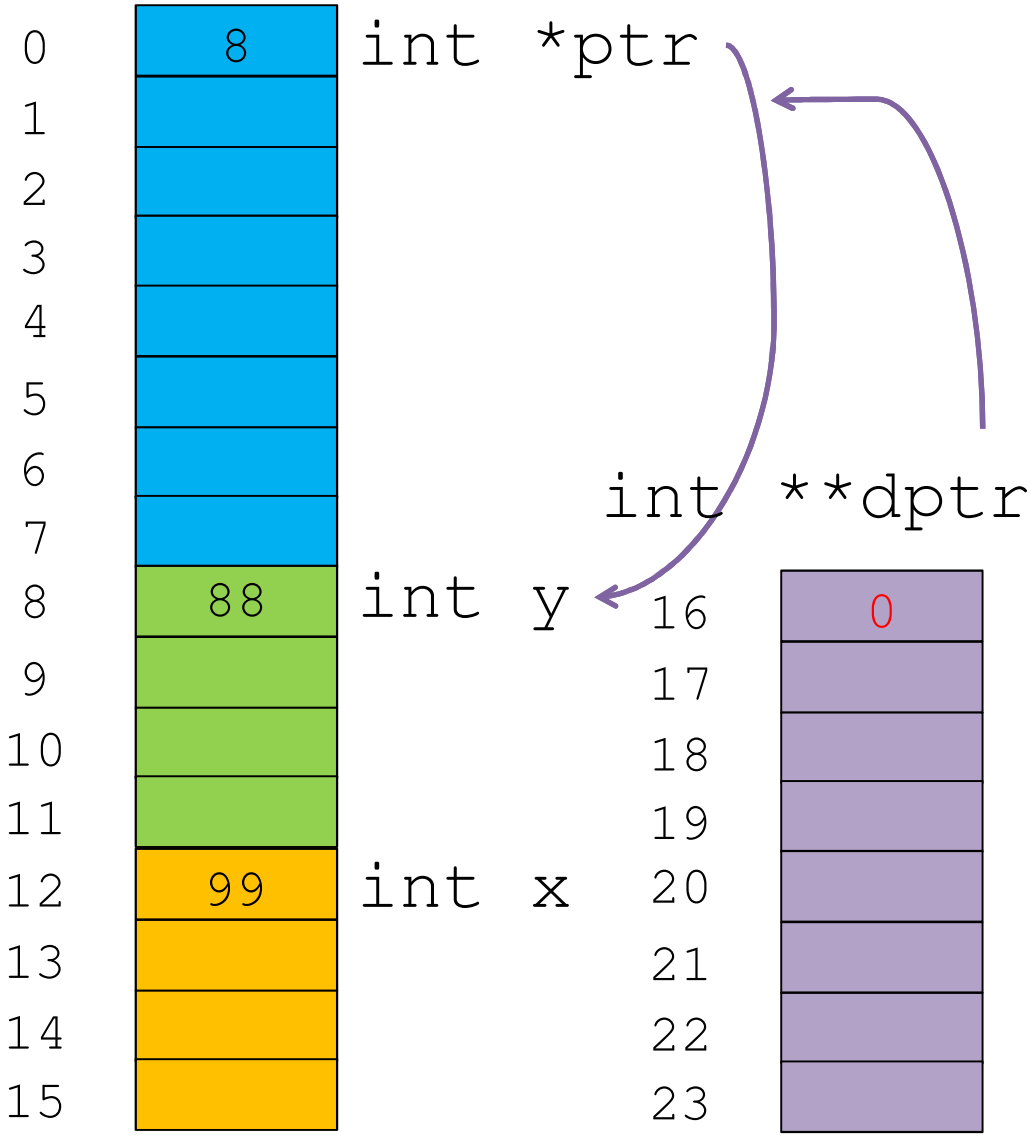


Pointer to pointer

Let us look at this a little more carefully…

What if we do this?

```
int **dptr = &ptr;
```



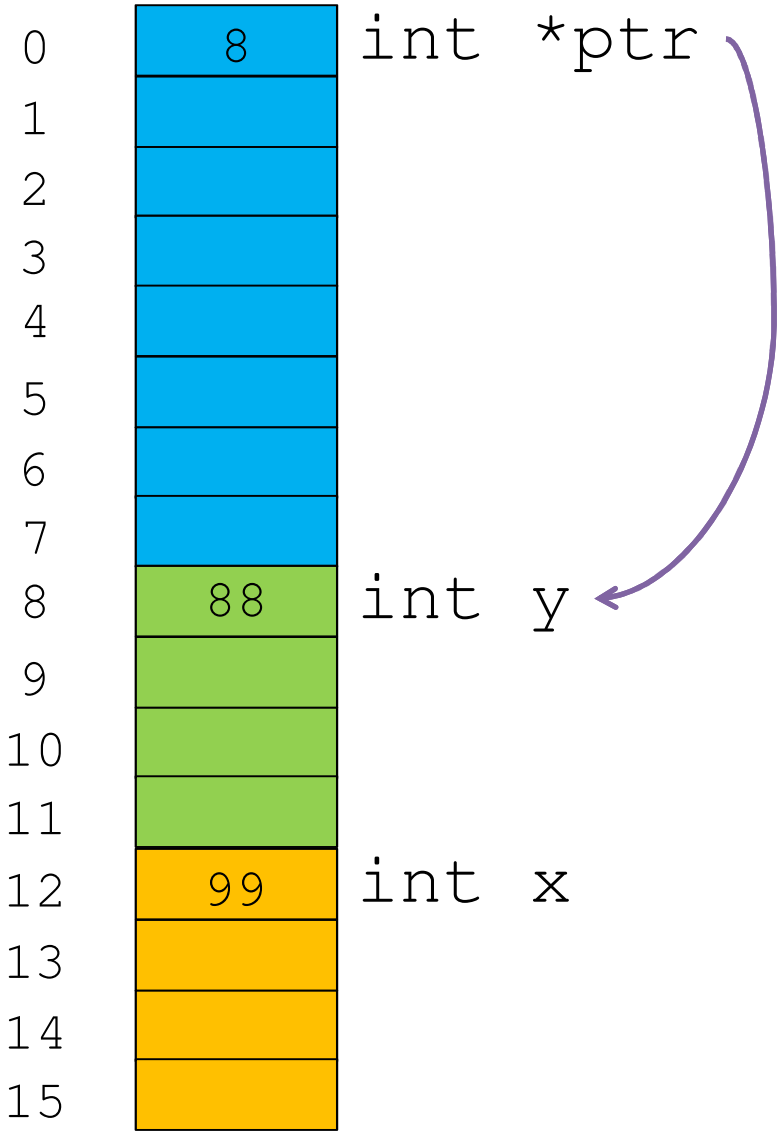
Pointer Arithmetic

Pointer Arithmetic

Let us look at this a little more carefully…

What does this mean?

```
ptr = ptr + 1;
```

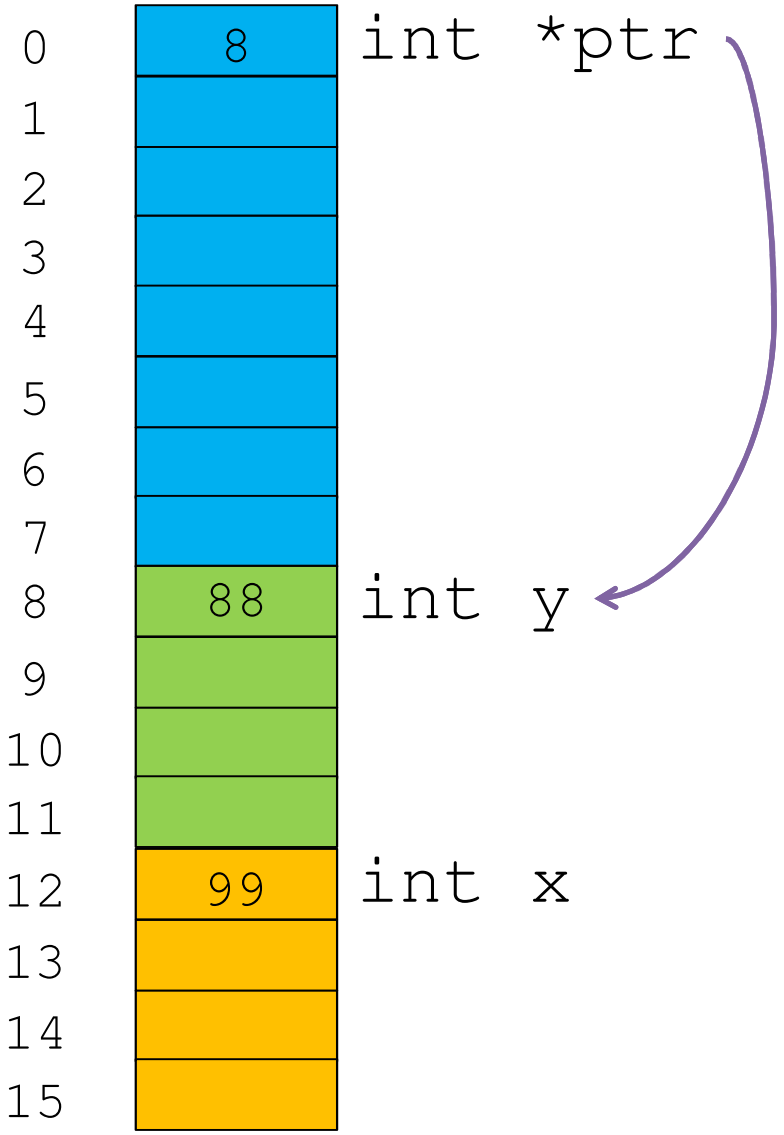


Pointer Arithmetic

Let us look at this a little more carefully…

What does this mean?

```
ptr = ptr + 1;
```

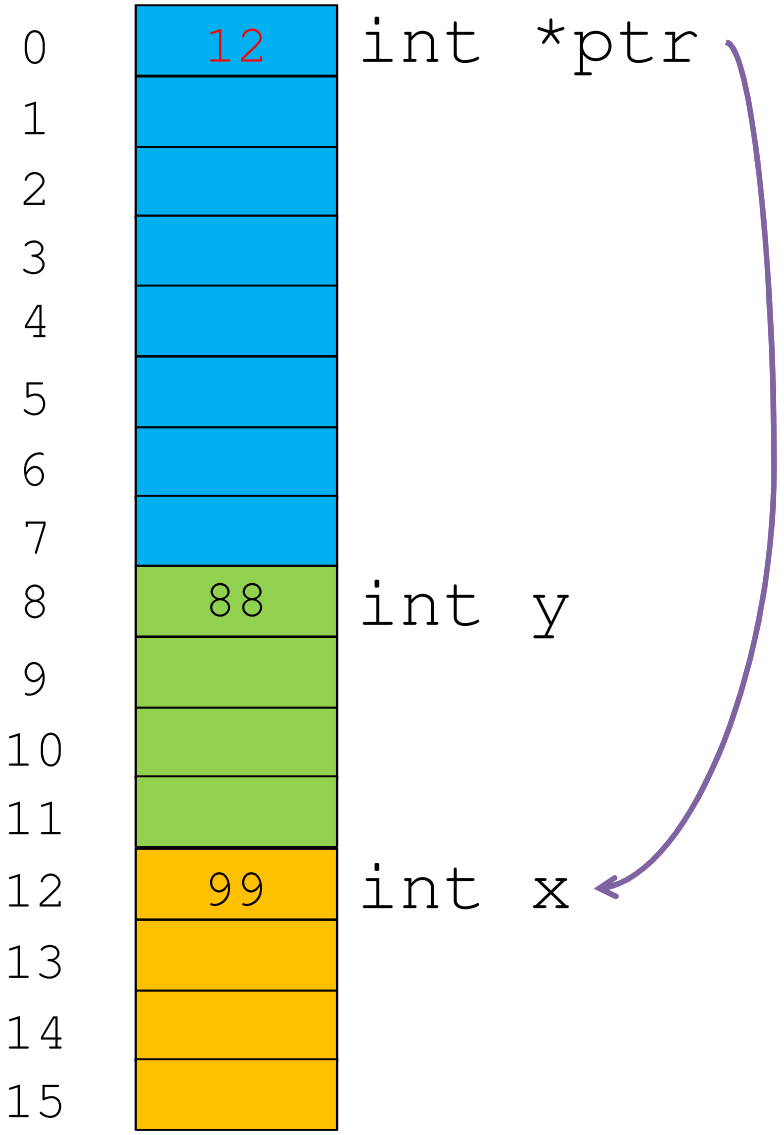


Pointer Arithmetic

Let us look at this a little more carefully…

What does this mean?

```
ptr = ptr + 1;
```

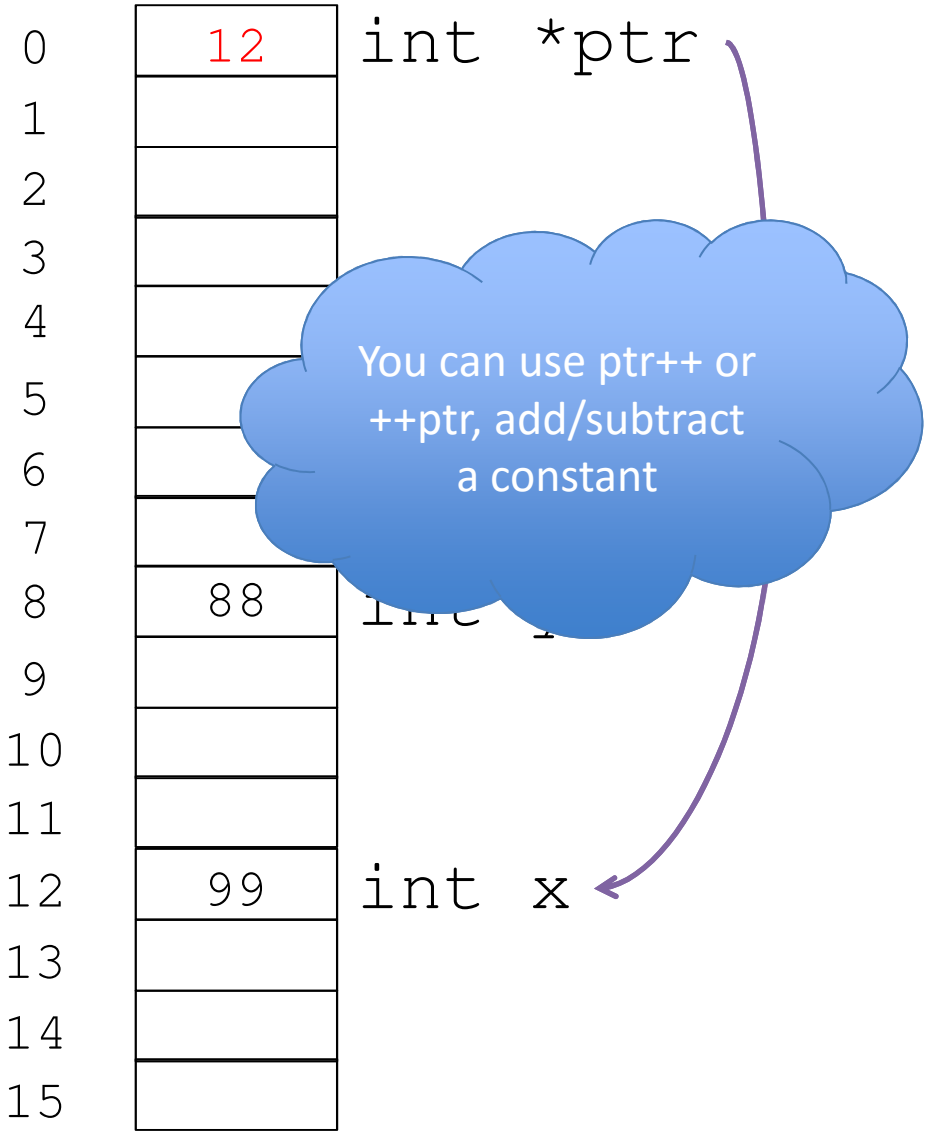


Pointer Arithmetic

Let us look at this a little more carefully…

What does this mean?

```
ptr = ptr + 1;
```



i-clicker question

What is the value of y after executing the following program?

```
{
    int x, y;
    int *p;
    p = &x;
    x = 5;
    *p = 23;
     y = x+1;
}
```

A. 5
B. 23
C. 6
D. 24

i-clicker question

What is the value of y after executing the following program?

```
{
    int x, y, a;
    int *p, *q;
    p = &a;
    q = &x;
    x = 5;
    *p = 23;
    y = x+1;
}
```

A. 5
B. 23
C. 6
D. 24

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;
- **Dereferencing: Access a location**
  - int x = *ptr; (read)

Pointer to Structures

```
struct point  {
      int x;
      int y;
};

struct point  {
      int x;
      int y;
};

struct point origin, *p;
p = &origin;
printf("origin is (%d,%d)\n", (*pp).x, pp->x);
```

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;
- **Dereferencing: Access a location**
  - int x = *ptr; (read)
  - x = ptr2->field; (read)

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;
- **Dereferencing: Access a location**
  - int x = *ptr; (read)
  - x = ptr2->field; (read)
  - *ptr = x; (write)

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;
- **Dereferencing: Access a location**
  - int x = *ptr; (read)
  - x = ptr2->field; (read)
  - *ptr = x; (write)
  - ptr2->field = x; (write)

Pointer operations

- **Referencing: Create a location**
  - type *p = &var;
- **Dereferencing: Access a location**
  - int x = *ptr; (read)
  - x = ptr2->field; (read)
  - *ptr = x; (write)
  - ptr2->field = x; (write)
- **Aliasing: Copy a pointer**
  - type *pa;
    pa = pb;