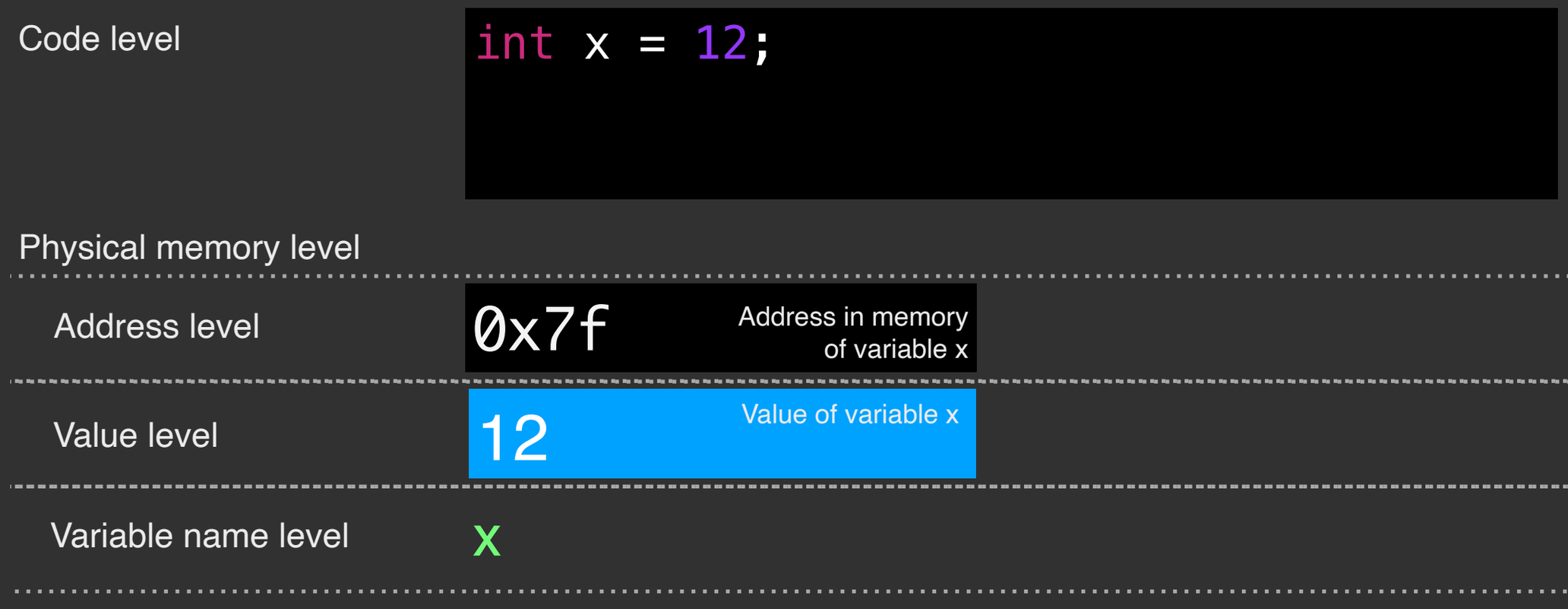
References & Pointers

Recap Variables & Memory

- If a variable is declared in the code, a certain amount of memory is allocated for it in the computer's memory

Recap Variables & Memory

- The amount of memory allocated depends on the variable's type, e.g., int32, int64, string, multiple data types in a user-defined class, …



Recap Variables & Memory

- The allocated memory block is clearly identified by an address and associated with the corresponding variable in code

The Reference Type

- A variable of type reference to a data type is an alias of another variable, it refers to an existing variable

Code level

```
int x = 12;

int& ref = x;
```

Physical memory level

Address level — `0x7f` — Address in memory of variable x

Value level — `12` — Value of variable x

Variable name level — x, ref

The Reference Type

- A variable of type reference to a data type is declared by adding the address-of-operator “&” to the data type like so:
  - int&, float&, myObject&, ofTexture&, …

Code level

```
int x = 12;

int& ref = x;
```

Physical memory level

Address level — 0x7f — Address in memory of variable x

Value level — 12 — Value of variable x

Variable name level — x, ref

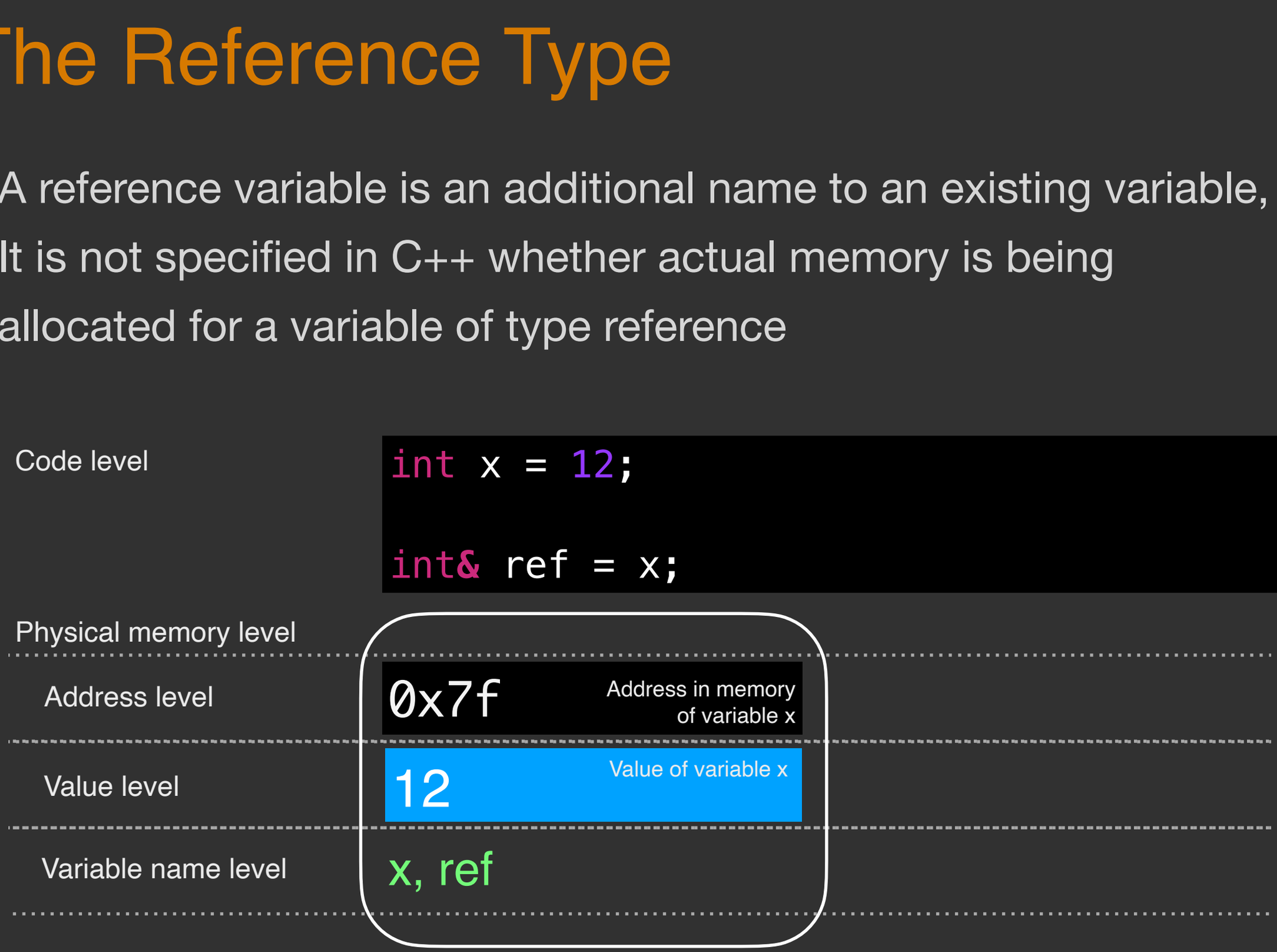
The Reference Type

- A reference variable is an additional name to an existing variable,
- It is not specified in C++ whether actual memory is being allocated for a variable of type reference

Code level

```
int x = 12;

int& ref = x;
```



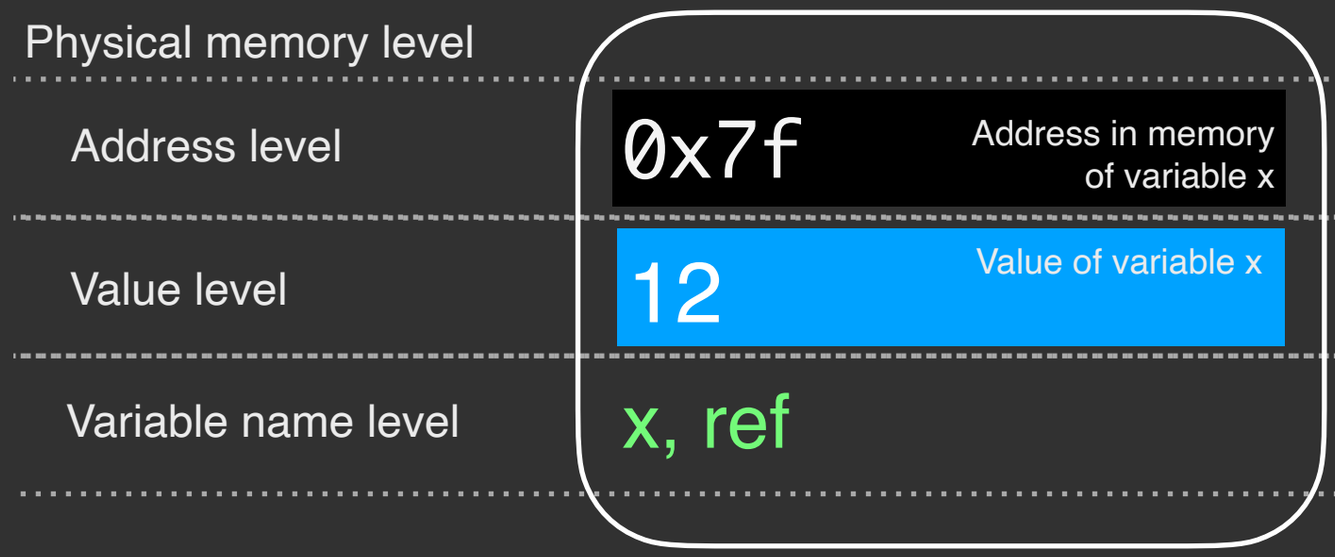
The Reference Type

- This is why a reference variable MUST always be initialized with the object it refers to
- A variable of type reference cannot stand alone

Code level

```
int x = 12;

int& ref = x;        int& refAlone;
```

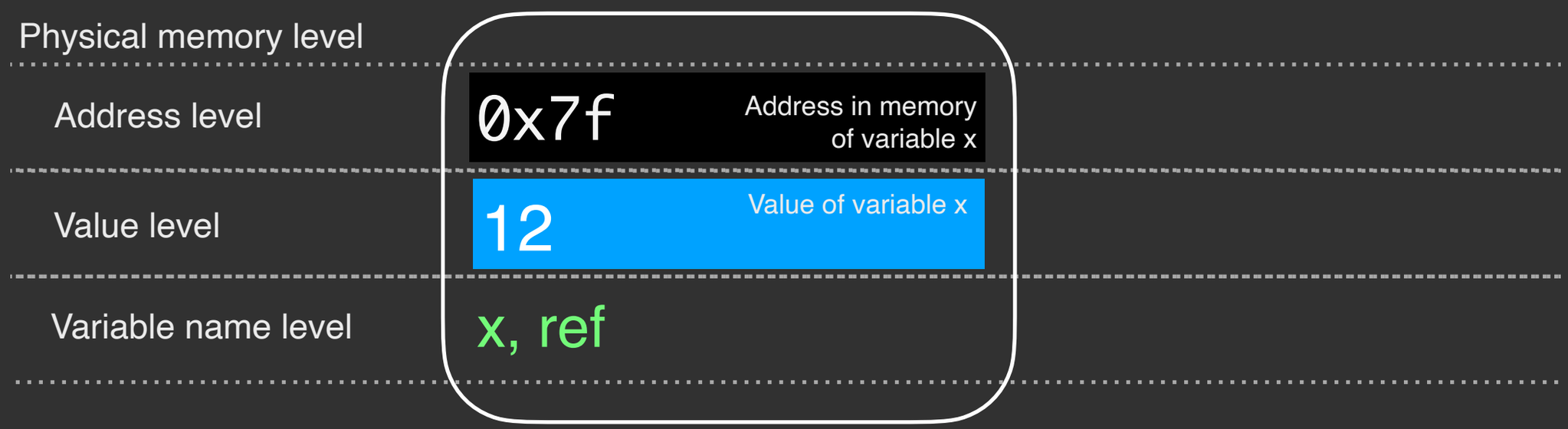


The Reference Type

- Any change to a variable of type reference will always result in a change of the original variable
- Reference variables can be used just like a normal variable

Code level

```
int x = 12;             ref += 5;
                    => // x (and ref)
int& ref = x;          // will both be 17
```



The Reference Type by Example

- Manipulating the reference variable directly affects the referenced variable & vice versa

```
43  int x = 5;
44  int &z = x; // z is another name for x
45  int &y;     // Error: reference must be initialized!
46
47  cout << x << endl; // -> prints 5
48  cout << z << endl; // -> prints 5
49
50  z = 9;      // same as x = 9;
51
52  cout << x << endl; // -> prints 9
53  cout << z << endl; // -> prints 9
```

Benefits of References

- Reference types support efficient data handling
- In particular, when used as function parameters they allow to access data without having
  - to initialize new variables and
  - to copy the variable's values
- Reference types support all kinds of data manipulation whenever the referenced variable cannot be accessed directly

The Pointer Type

- A variable of type pointer to a data type is storing the address value of an associated variable using &varName

Code level

```
int x = 12;

int* ptr = &x;
```

Physical memory level

Address level	0x7f — Address in memory of variable x	0x2f — Address in memory of variable ptr
Value level	12 — Value of variable x	0x7f — Value of variable ptr
Variable name level	x	ptr

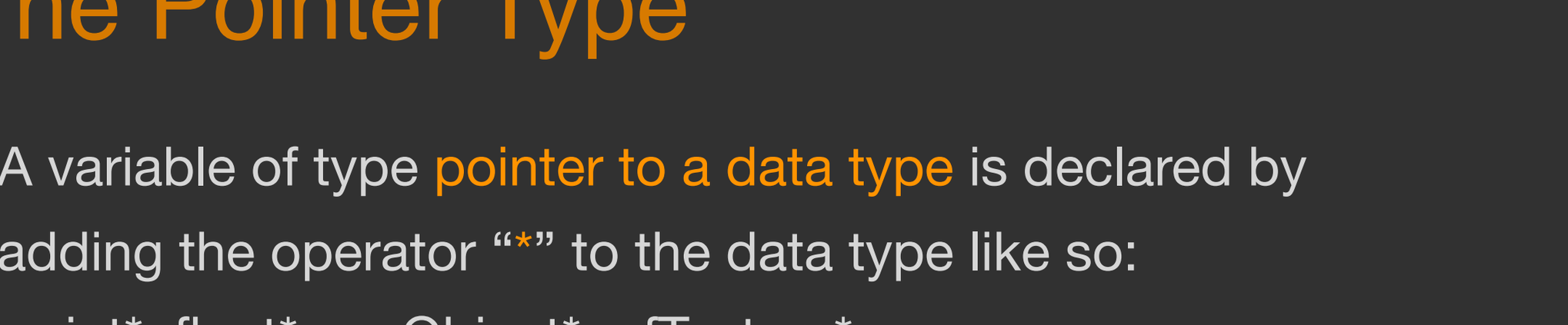
The Pointer Type

- A variable of type pointer to a data type is declared by adding the operator "*" to the data type like so:
  - int*, float*, myObject*, ofTexture*, …

Code level

```
int x = 12;

int* ptr = &x;
```



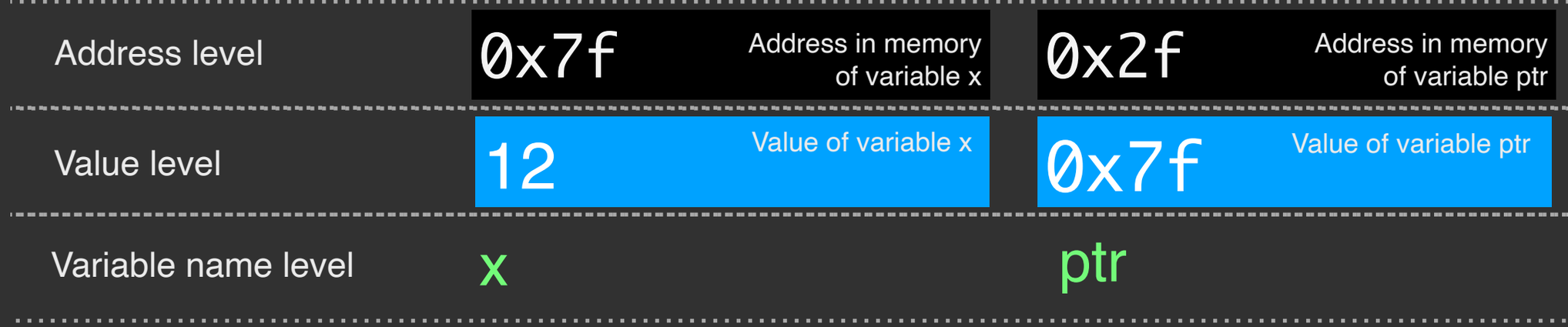
The Pointer Type

- The value of the variable the pointer points to can be de-referenced using the dereferencing-operator “*” which must be put in front of the variable’s name like so “*ptr”

Accessing the value of the variable ptr points to

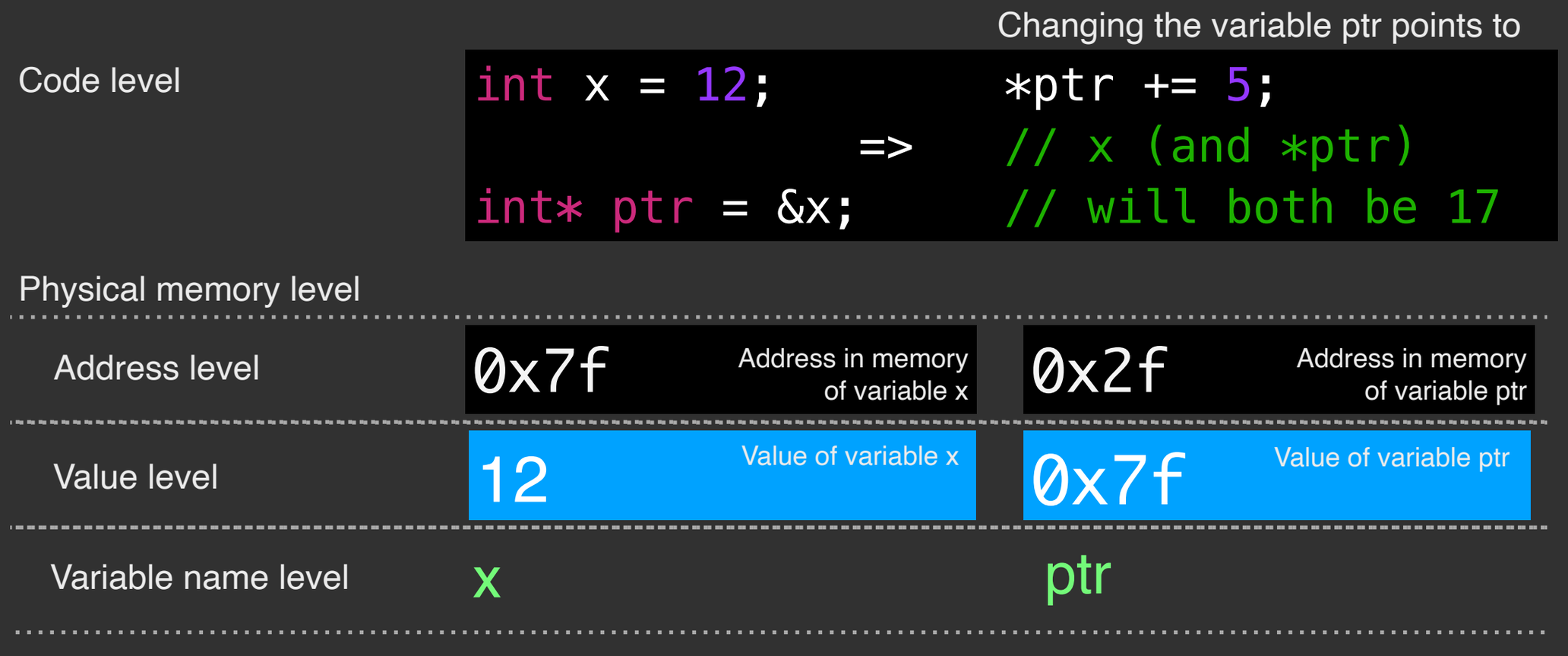
Code level

```
int x = 12;          cout << *ptr << endl;
                  =>
int* ptr = &x;    // prints 12
```



The Pointer Type

- The value of the variable the pointer points to can be de-referenced using the dereferencing-operator “*” which must be put in front of the variable’s name like so “*ptr”



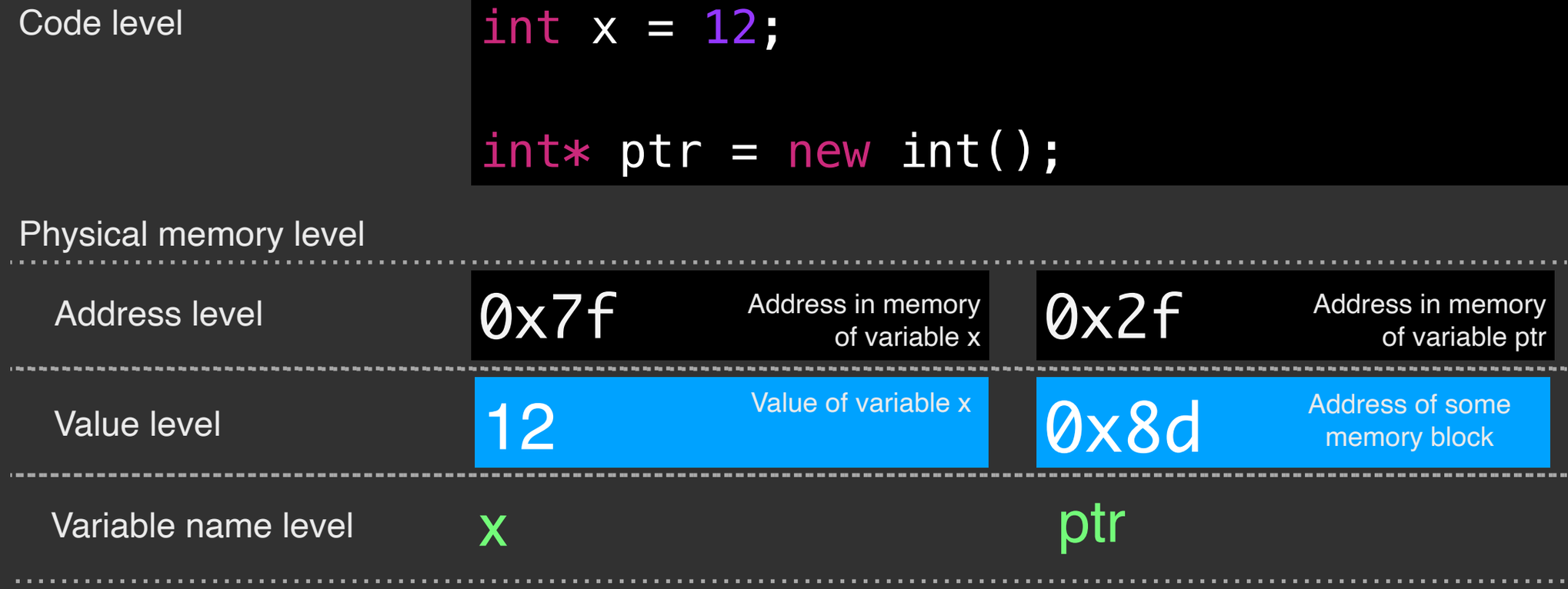
The Pointer Type

- A variable of type pointer to a data type can also be used to store the address of a newly (or dynamically) allocated block of memory using new <data type>( )

Code level

```
int x = 12;

int* ptr = new int();
```



The Pointer Type

- Using variable of type pointer to a data type can be tricky because the memory that was allocated dynamically with new must also be de-allocated once the program ends with delete

Code level

```
int x = 12;
// . . .
delete ptr;
```

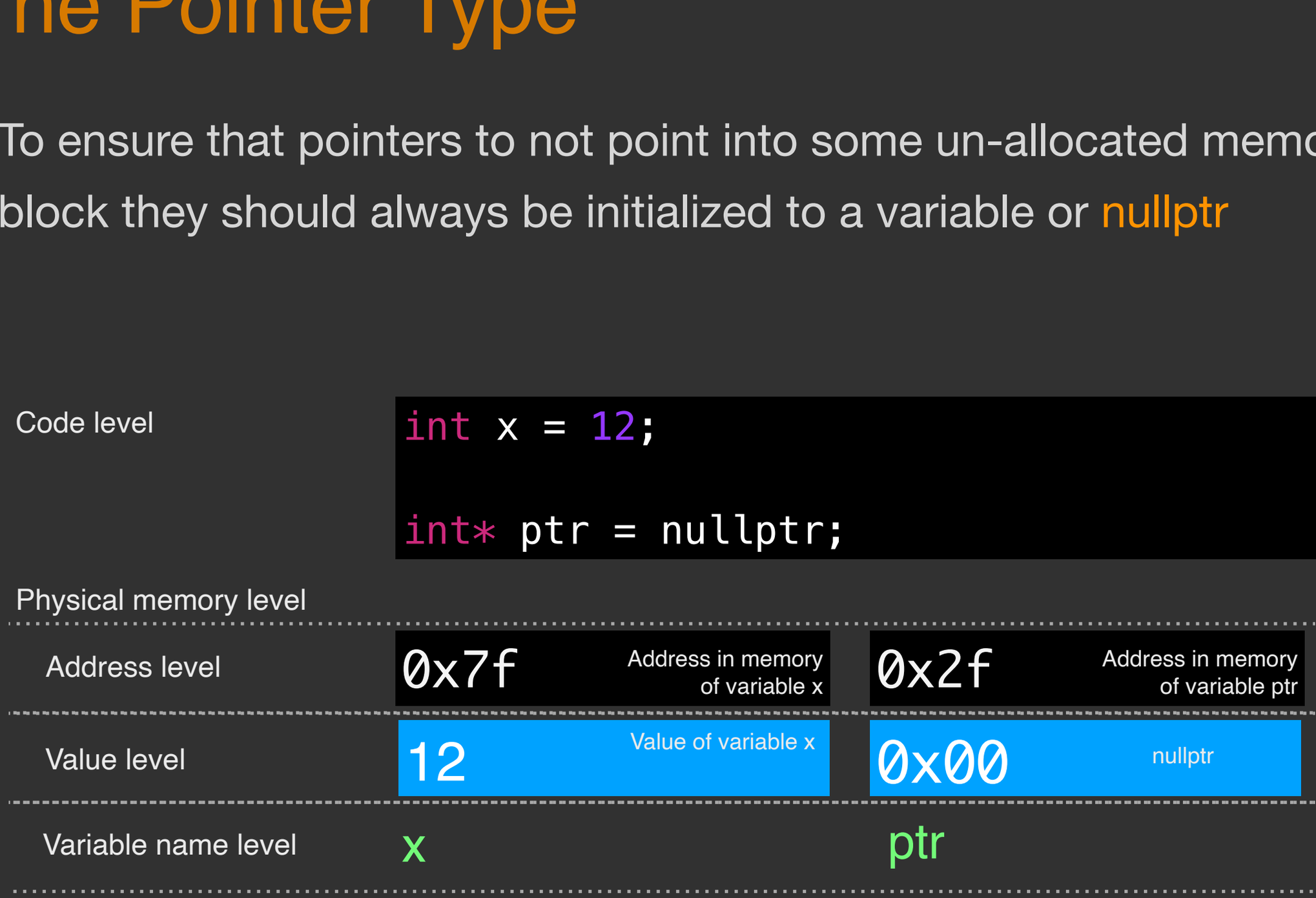
The Pointer Type

- To ensure that pointers to not point into some un-allocated memory block they should always be initialized to a variable or nullptr

Code level

```
int x = 12;

int* ptr = nullptr;
```



Benefits of Pointers

- Pointers in C++ have various application areas

- They are primarily used when the application program has to deal with dynamic memory allocation, i.e.,
  - the size of the memory that will be required to store the data can only be allocated during run-time
  - the size of the data is not known at compile-time

- Such data could be images, audio, user input

References & Pointers Compared

- Similar at first glance, there is a clear difference between the types:
  - References refer to the object they were initialized with
  - Pointers are individual objects that allow to manipulate memory

```
int x = 12;

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

Address level	`0x7f` Address in memory of variable x	`0x2f` Address in memory of variable ptr
Value level	`12` Value of variable x	`0x7f` Value of variable ptr
Variable name level	x, ref	ptr

References & Pointers by Example

- Let’s swap some values! We use two functions
  with different types of arguments to do the same thing:

```
8  #include <iostream>
9  using namespace std;
10
11 // Function Prototypes(required in C++)
12
13 // here * is not the dereference operator!
14 // int * -> indicates that the argument is a pointer pointing to an int
15 void p_swap(int *, int *);
16
17 // int & -> indicates the arguments are treated as references to ints
18 // here changing the value of an function argument will change
19 // the value of the variable initialized outside the function
20 void r_swap(int&, int&);
21
```

References & Pointers by Example

- In this program, the initially declared integer values are swapped in two different ways:

```
22
23 int main (void) {
24     int v = 5, x = 10;
25     cout << "v: " << v << " x: " << x << endl; // prints v: 5 x: 10
26     p_swap(&v, &x);                           // passing in addresses (pointers)
27     cout << "v: " << v << " x: " << x << endl; // prints v: 10 x: 15
28     r_swap(v, x);                             // passing in values
29     cout << "v: " << v << " x: " << x << endl; // prints v: 5 x: 10
30     return 0;
31 }
32
```

References & Pointers by Example

- Pointer Swap
  - Changing the values at memory address level requires to pass pointers that are being dereferenced

```
34 // uses the dereference operator to access the values
35 // belonging to the addresses passed in, changing v and x
36 void p_swap(int* a, int* b)
37 {
38     int temp;
39     temp = *a;  // store dereferenced value of a (5)
40     *a = *b;    // assign value at memory adress a to b (10)
41     *b = temp;  // assign stored value to b (5)
42 }
43
```

References & Pointers by Example

- Reference Swap
  - Changing the values of the external input variables v and x without using memory addresses at all

```
43
44 // reference swap:
45 // swaps the values of the original variables passed in
46 // without returning anything, changing v and x back
47 void r_swap(int& a, int& b)
48 {
49     int temp;
50     temp = a;   // store reference a to temp (10)
51     a = b;      // assign reference of b to a (5)
52     b = temp;   // assign original reference a to b (10)
53 }
54
```

Take Away

- References and pointers are dedicated data types in C++ introduced to increase the efficient management of data

- References should be considered as referring to another object, they are no individual object of themselves

- Pointers are pointing to memory addresses of variables and objects, they are individual objects that support the dynamic and manual allocation and management of data