# Lecture 19

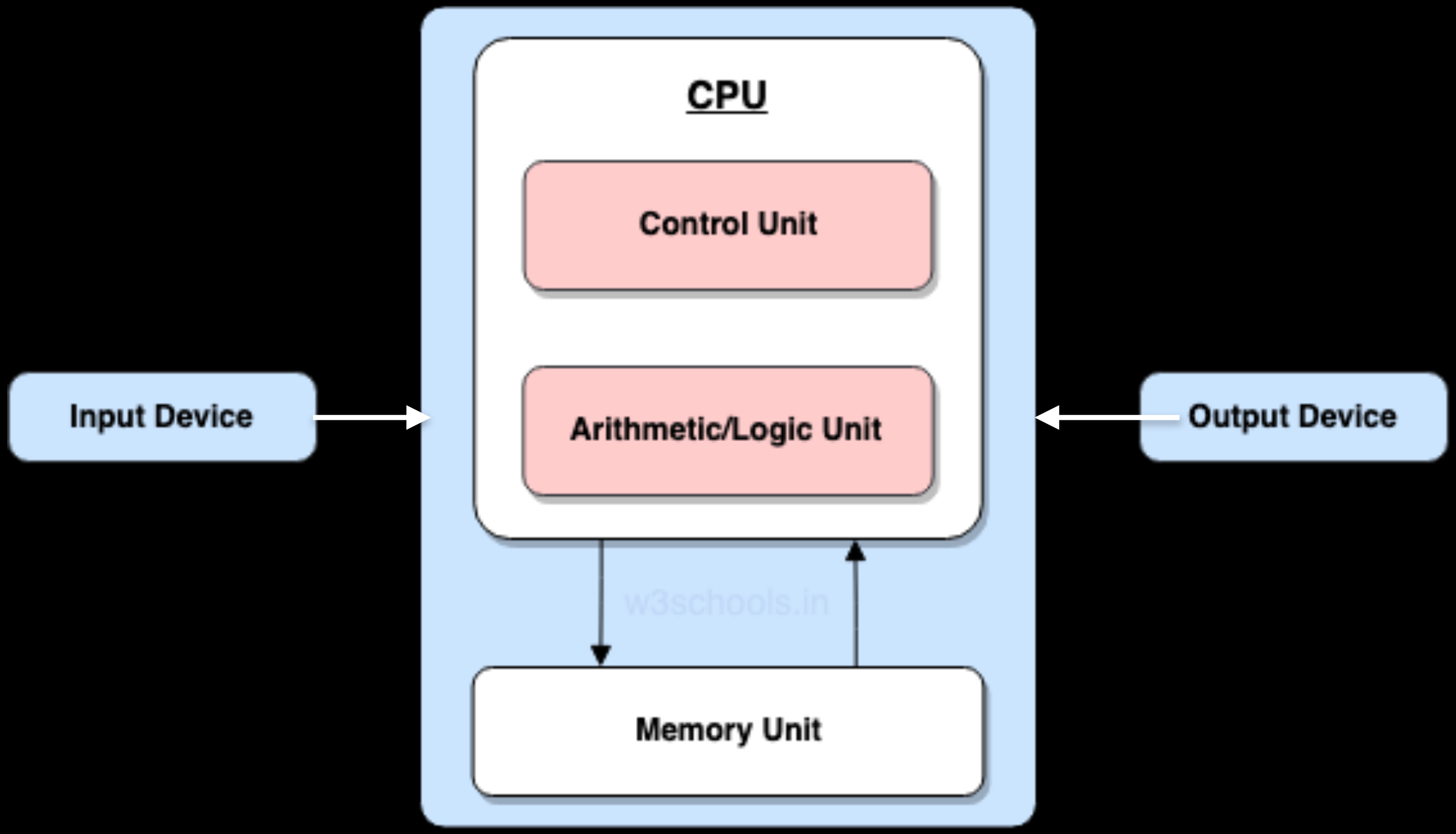
## Pointers

RECURSION 23

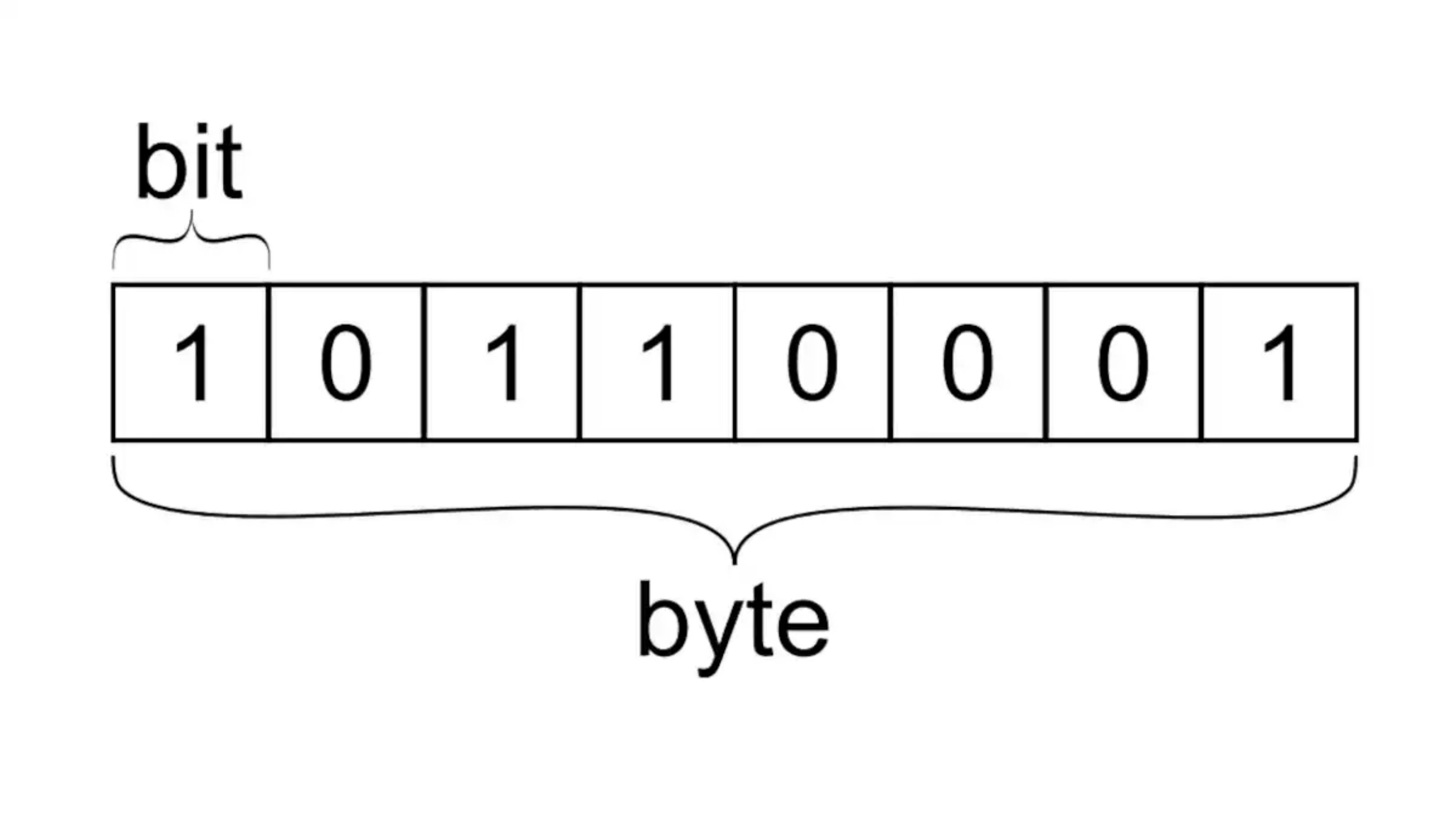
SIMPLIFIED CSE COURSE FOR ALL DEPARTMENTS

C & C++

# Computer Memory

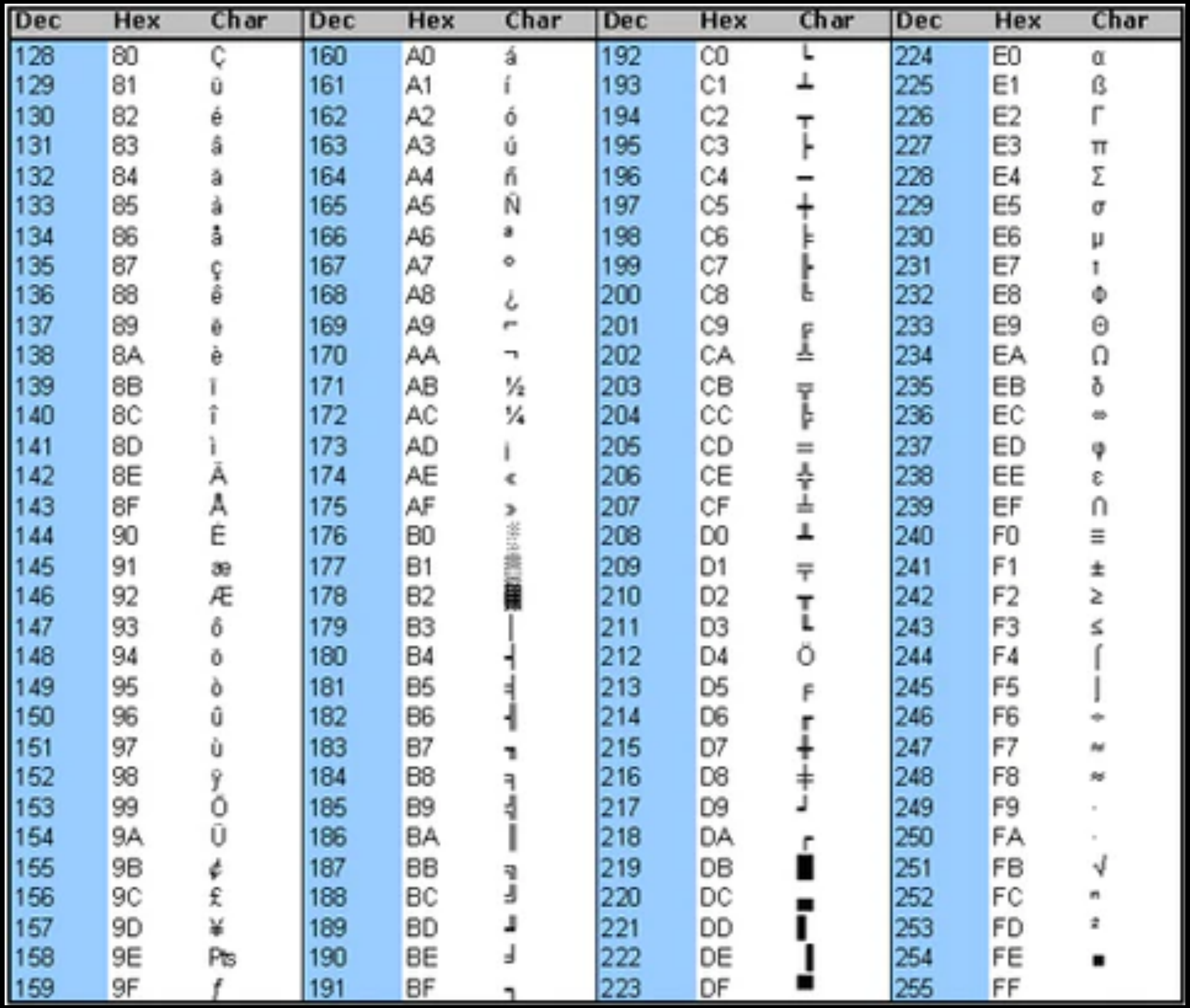


# Bit and Byte



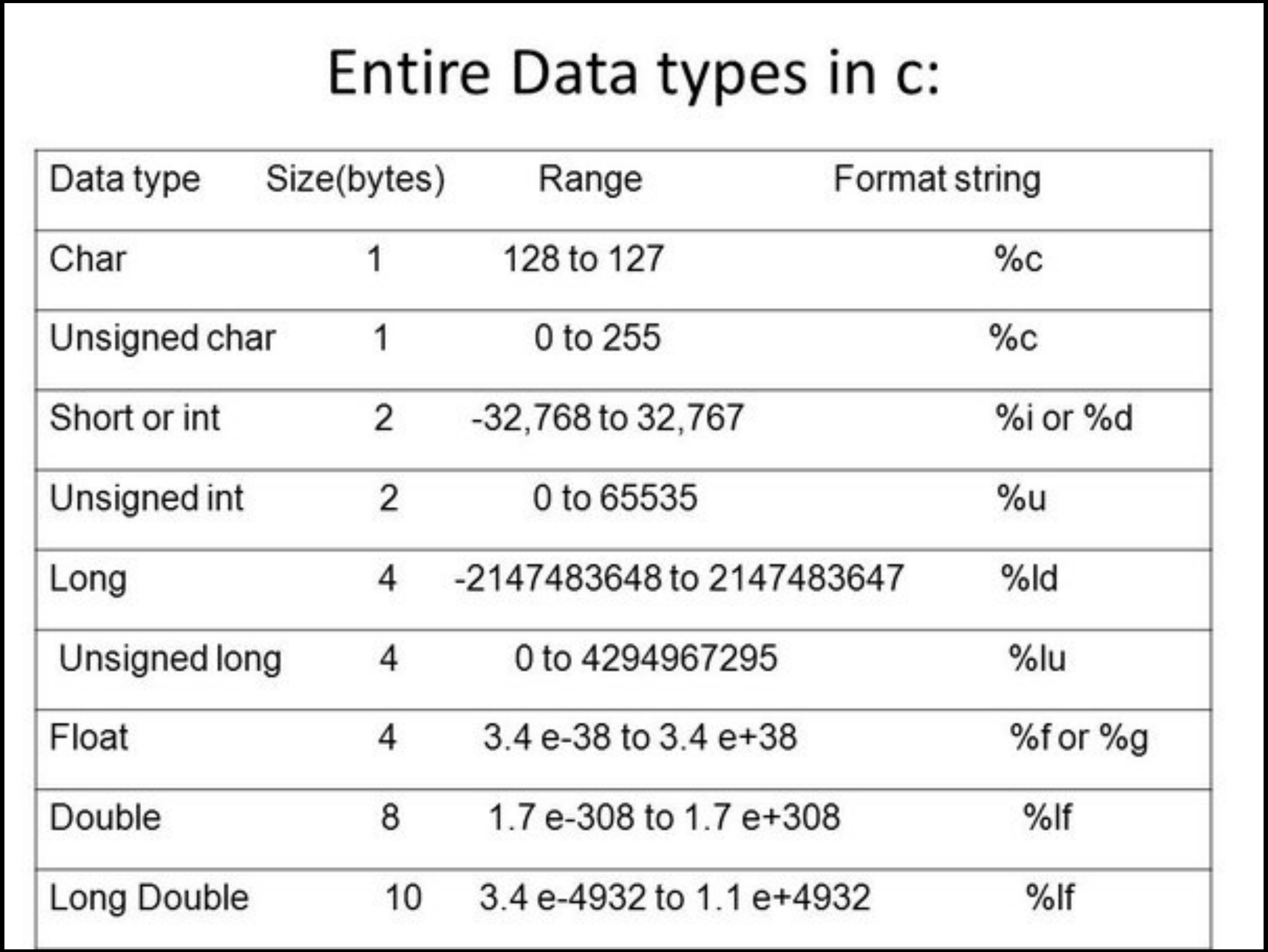
How many different things can be stored in a byte?

# ASCII Chart



| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | └ | 224 | E0 | α |
| 129 | 81 | ü | 161 | A1 | í | 193 | C1 | ┴ | 225 | E1 | ß |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | ┬ | 226 | E2 | Γ |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | ├ | 227 | E3 | π |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | ─ | 228 | E4 | Σ |
| 133 | 85 | à | 165 | A5 | Ñ | 197 | C5 | ┼ | 229 | E5 | σ |
| 134 | 86 | å | 166 | A6 | ª | 198 | C6 | ╞ | 230 | E6 | µ |
| 135 | 87 | ç | 167 | A7 | º | 199 | C7 | ╟ | 231 | E7 | τ |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | ╚ | 232 | E8 | Φ |
| 137 | 89 | ë | 169 | A9 | ⌐ | 201 | C9 | ╔ | 233 | E9 | Θ |
| 138 | 8A | è | 170 | AA | ¬ | 202 | CA | ╩ | 234 | EA | Ω |
| 139 | 8B | ï | 171 | AB | ½ | 203 | CB | ╦ | 235 | EB | δ |
| 140 | 8C | î | 172 | AC | ¼ | 204 | CC | ╠ | 236 | EC | ∞ |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | ═ | 237 | ED | φ |
| 142 | 8E | Ä | 174 | AE | « | 206 | CE | ╬ | 238 | EE | ε |
| 143 | 8F | Å | 175 | AF | » | 207 | CF | ╧ | 239 | EF | ∩ |
| 144 | 90 | É | 176 | B0 | ░ | 208 | D0 | ╨ | 240 | F0 | ≡ |
| 145 | 91 | æ | 177 | B1 | ▒ | 209 | D1 | ╤ | 241 | F1 | ± |
| 146 | 92 | Æ | 178 | B2 | ▓ | 210 | D2 | ╥ | 242 | F2 | ≥ |
| 147 | 93 | ô | 179 | B3 | │ | 211 | D3 | ╙ | 243 | F3 | ≤ |
| 148 | 94 | ö | 180 | B4 | ┤ | 212 | D4 | ╘ | 244 | F4 | ⌠ |
| 149 | 95 | ò | 181 | B5 | ╡ | 213 | D5 | ╒ | 245 | F5 | ⌡ |
| 150 | 96 | û | 182 | B6 | ╢ | 214 | D6 | ╓ | 246 | F6 | ÷ |
| 151 | 97 | ù | 183 | B7 | ╖ | 215 | D7 | ╫ | 247 | F7 | ≈ |
| 152 | 98 | ÿ | 184 | B8 | ╕ | 216 | D8 | ╪ | 248 | F8 | ≈ |
| 153 | 99 | Ö | 185 | B9 | ╣ | 217 | D9 | ┘ | 249 | F9 | · |
| 154 | 9A | Ü | 186 | BA | ║ | 218 | DA | ┌ | 250 | FA | · |
| 155 | 9B | ¢ | 187 | BB | ╗ | 219 | DB | █ | 251 | FB | √ |
| 156 | 9C | £ | 188 | BC | ╝ | 220 | DC | ▄ | 252 | FC | ⁿ |
| 157 | 9D | ¥ | 189 | BD | ╜ | 221 | DD | ▌ | 253 | FD | ² |
| 158 | 9E | Pts | 190 | BE | ╛ | 222 | DE | ▐ | 254 | FE | ■ |
| 159 | 9F | ƒ | 191 | BF | ┐ | 223 | DF | ▀ | 255 | FF | |

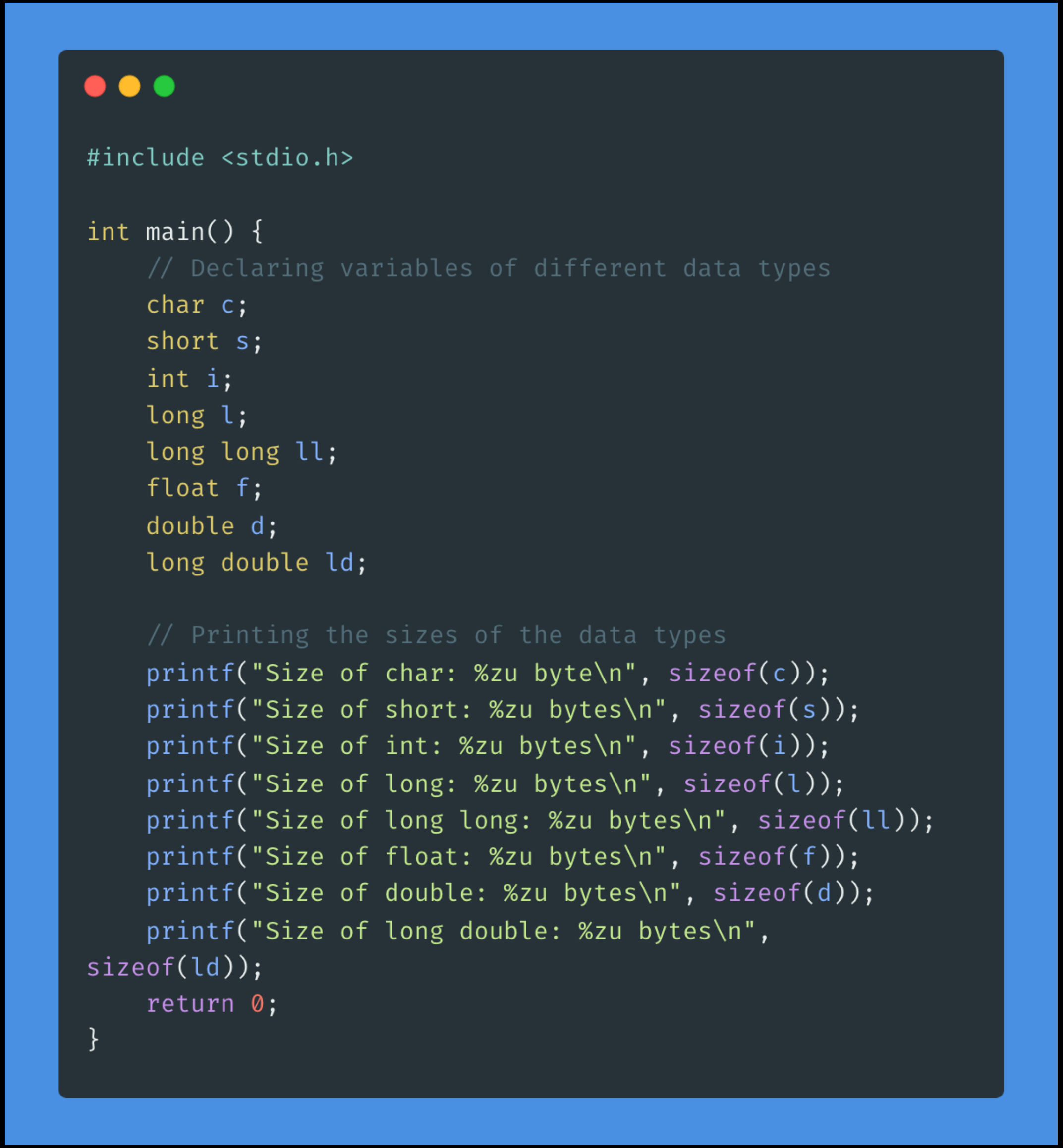
# Size of different datatypes



## Entire Data types in c:

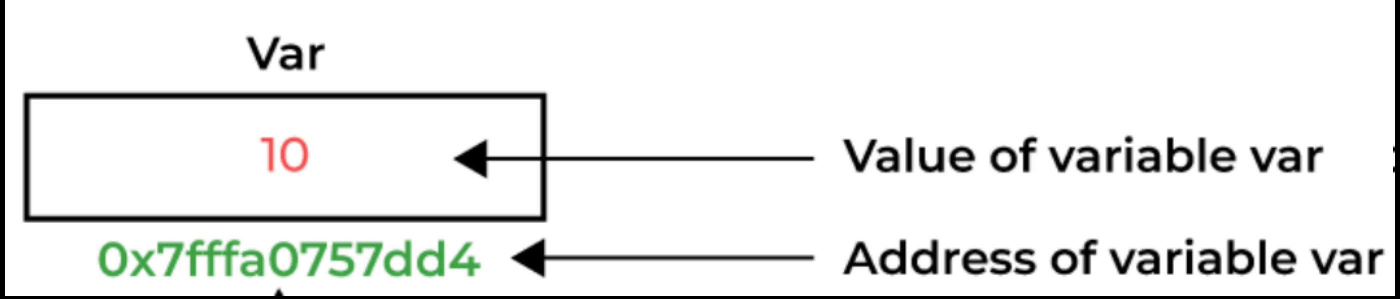
| Data type | Size(bytes) | Range | Format string |
|---|---|---|---|
| Char | 1 | 128 to 127 | %c |
| Unsigned char | 1 | 0 to 255 | %c |
| Short or int | 2 | -32,768 to 32,767 | %i or %d |
| Unsigned int | 2 | 0 to 65535 | %u |
| Long | 4 | -2147483648 to 2147483647 | %ld |
| Unsigned long | 4 | 0 to 4294967295 | %lu |
| Float | 4 | 3.4 e-38 to 3.4 e+38 | %f or %g |
| Double | 8 | 1.7 e-308 to 1.7 e+308 | %lf |
| Long Double | 10 | 3.4 e-4932 to 1.1 e+4932 | %lf |

# Size of different data types



```c
#include <stdio.h>

int main() {
    // Declaring variables of different data types
    char c;
    short s;
    int i;
    long l;
    long long ll;
    float f;
    double d;
    long double ld;

    // Printing the sizes of the data types
    printf("Size of char: %zu byte\n", sizeof(c));
    printf("Size of short: %zu bytes\n", sizeof(s));
    printf("Size of int: %zu bytes\n", sizeof(i));
    printf("Size of long: %zu bytes\n", sizeof(l));
    printf("Size of long long: %zu bytes\n", sizeof(ll));
    printf("Size of float: %zu bytes\n", sizeof(f));
    printf("Size of double: %zu bytes\n", sizeof(d));
    printf("Size of long double: %zu bytes\n",
sizeof(ld));
    return 0;
}
```

# How a variable is stored in memory



Var

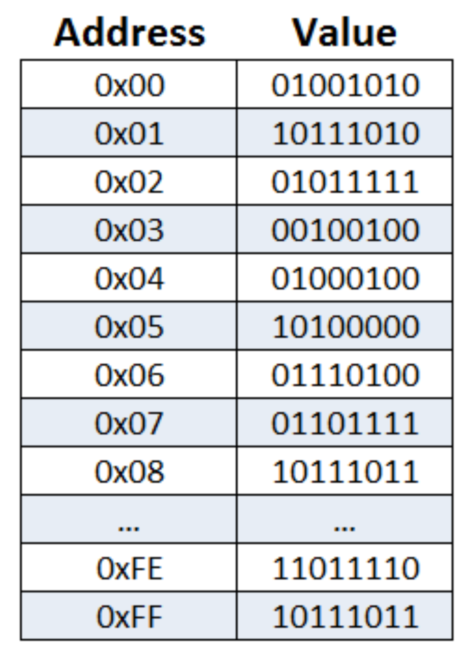
10

Value of variable var

0x7fffa0757dd4

Address of variable var

Every variable has a memory address assigned to it

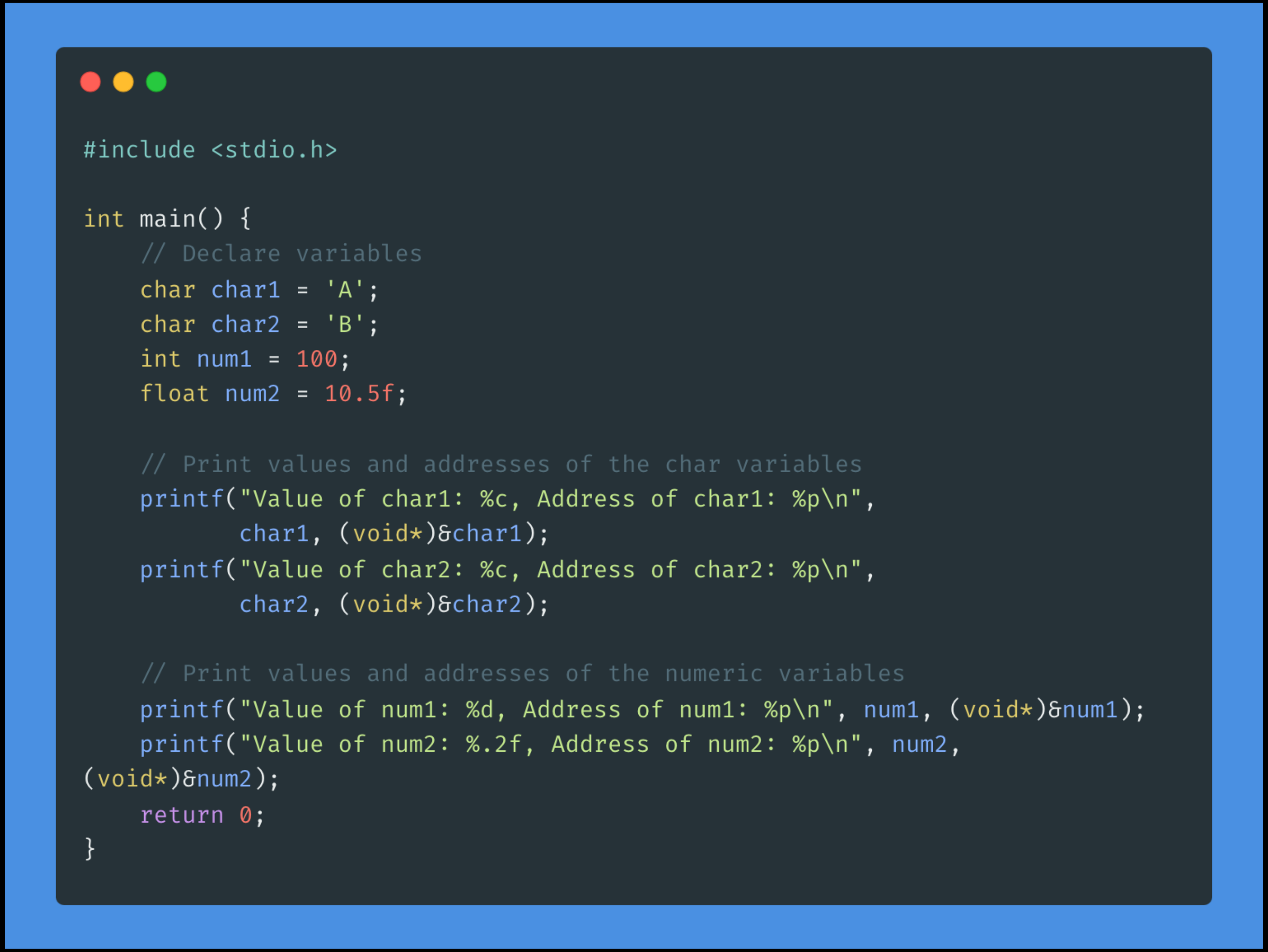


| Address | Value |
|---|---|
| 0x00 | 01001010 |
| 0x01 | 10111010 |
| 0x02 | 01011111 |
| 0x03 | 00100100 |
| 0x04 | 01000100 |
| 0x05 | 10100000 |
| 0x06 | 01110100 |
| 0x07 | 01101111 |
| 0x08 | 10111011 |
| ... | ... |
| 0xFE | 11011110 |
| 0xFF | 10111011 |

# How to access an address

Format Specifier: %p

&Variable_name

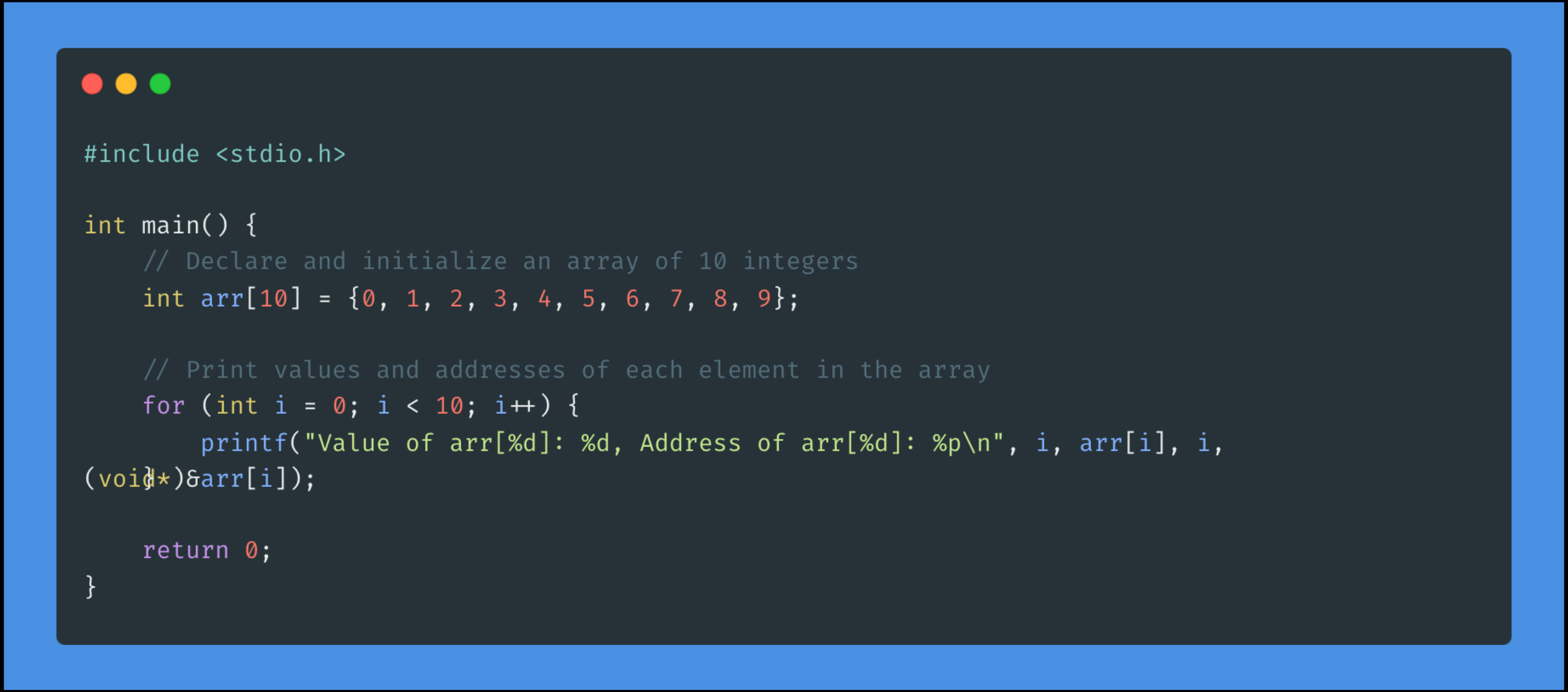


```c
#include <stdio.h>

int main() {
    // Declare variables
    char char1 = 'A';
    char char2 = 'B';
    int num1 = 100;
    float num2 = 10.5f;

    // Print values and addresses of the char variables
    printf("Value of char1: %c, Address of char1: %p\n",
           char1, (void*)&char1);
    printf("Value of char2: %c, Address of char2: %p\n",
           char2, (void*)&char2);

    // Print values and addresses of the numeric variables
    printf("Value of num1: %d, Address of num1: %p\n", num1, (void*)&num1);
    printf("Value of num2: %.2f, Address of num2: %p\n", num2,
(void*)&num2);
    return 0;
}
```

# How to access an address

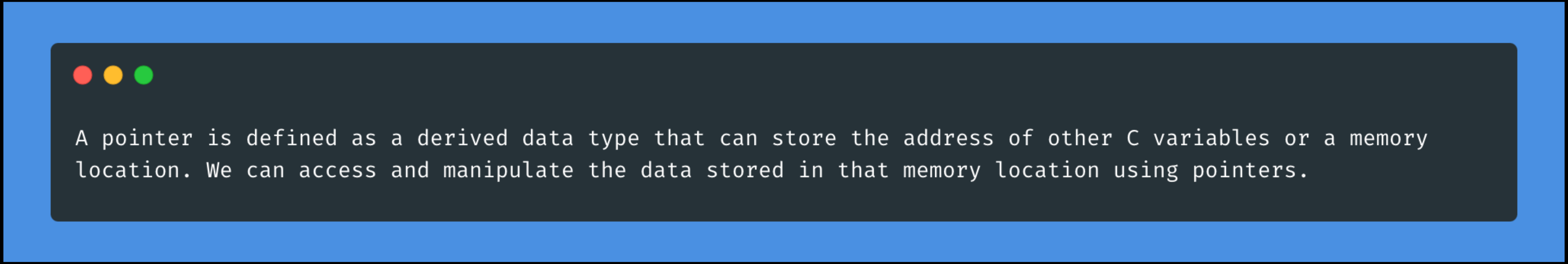
Format Specifier: %p

&Variable_name



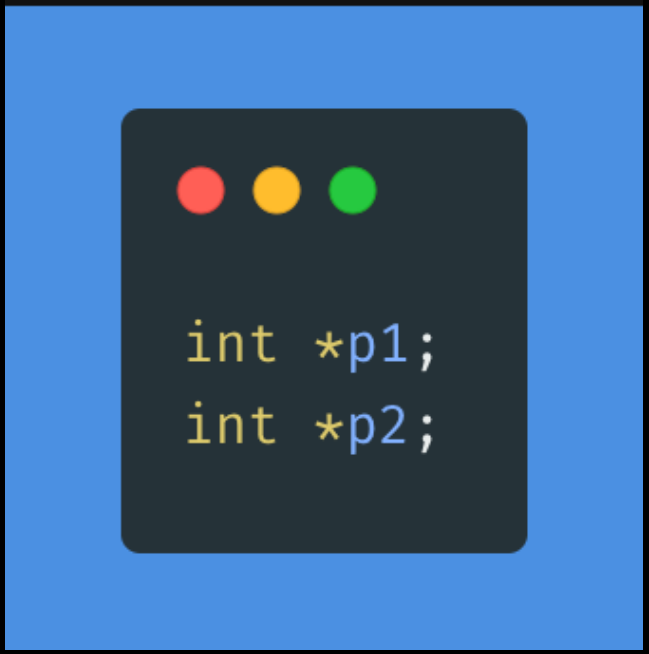
```c
#include <stdio.h>

int main() {
    // Declare and initialize an array of 10 integers
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    // Print values and addresses of each element in the array
    for (int i = 0; i < 10; i++) {
        printf("Value of arr[%d]: %d, Address of arr[%d]: %p\n", i, arr[i], i,
(void*)&arr[i]);
    }

    return 0;
}
```

# Pointer



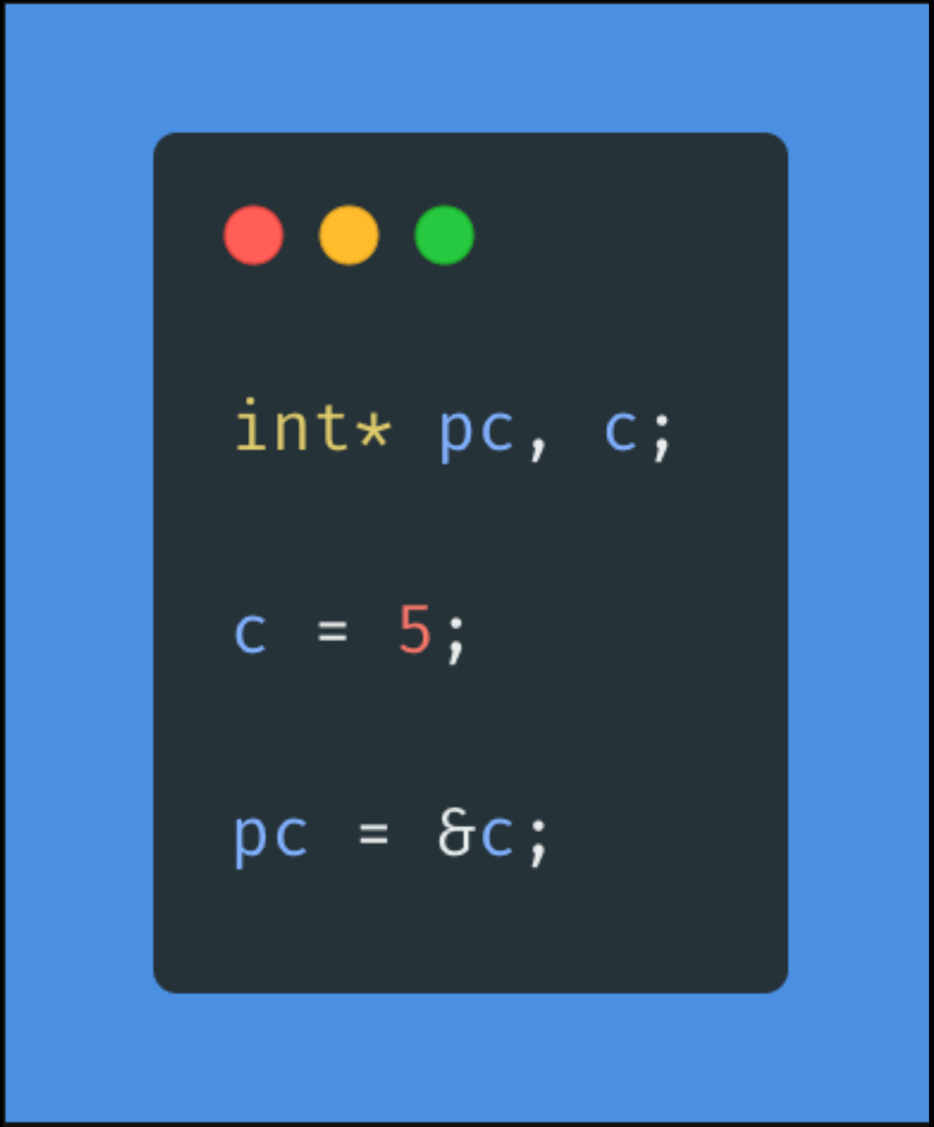
```
A pointer is defined as a derived data type that can store the address of other C variables or a memory
location. We can access and manipulate the data stored in that memory location using pointers.
```

# Declare Pointer



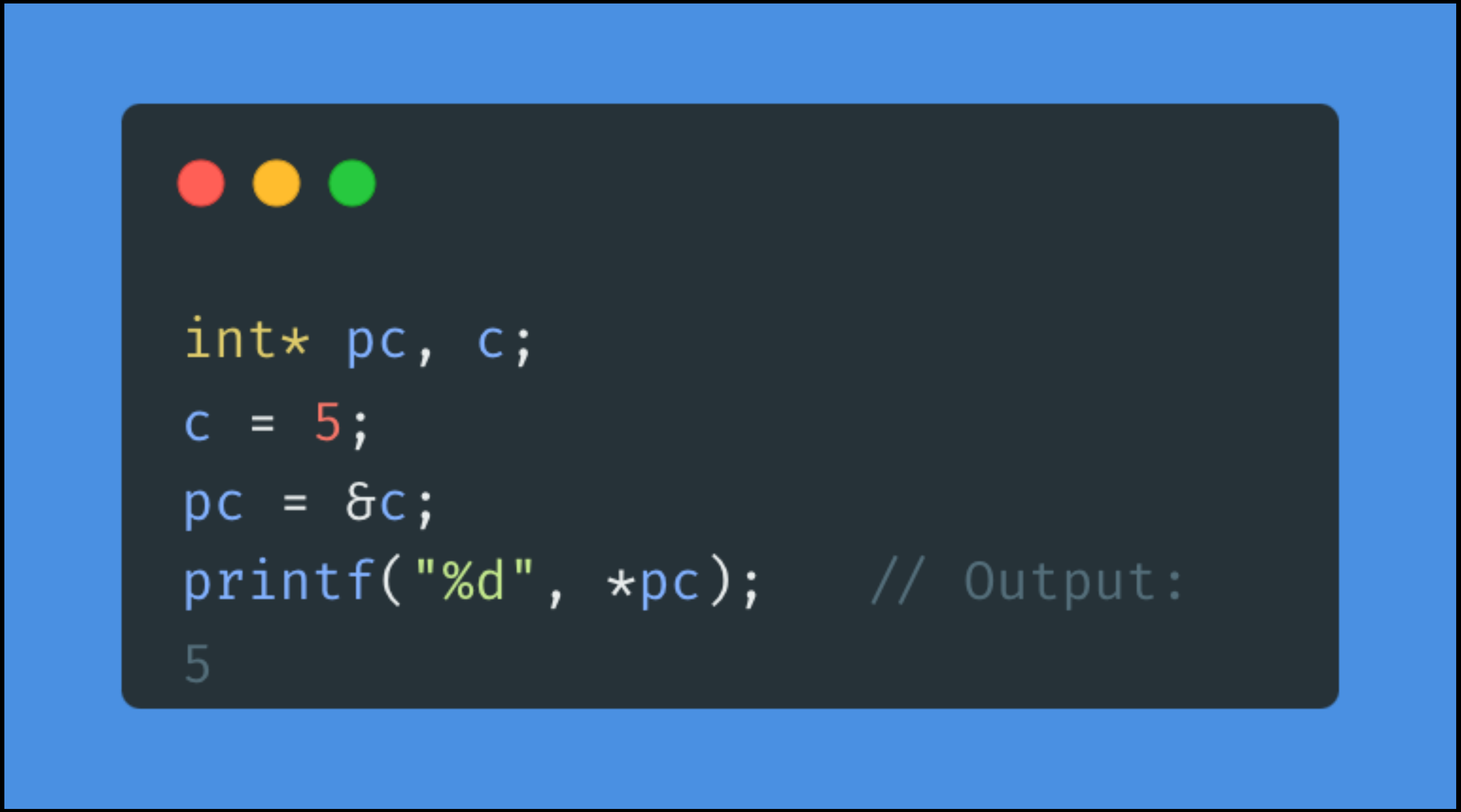
```
int *p1;
int *p2;
```

# Assigning addresses to Pointers



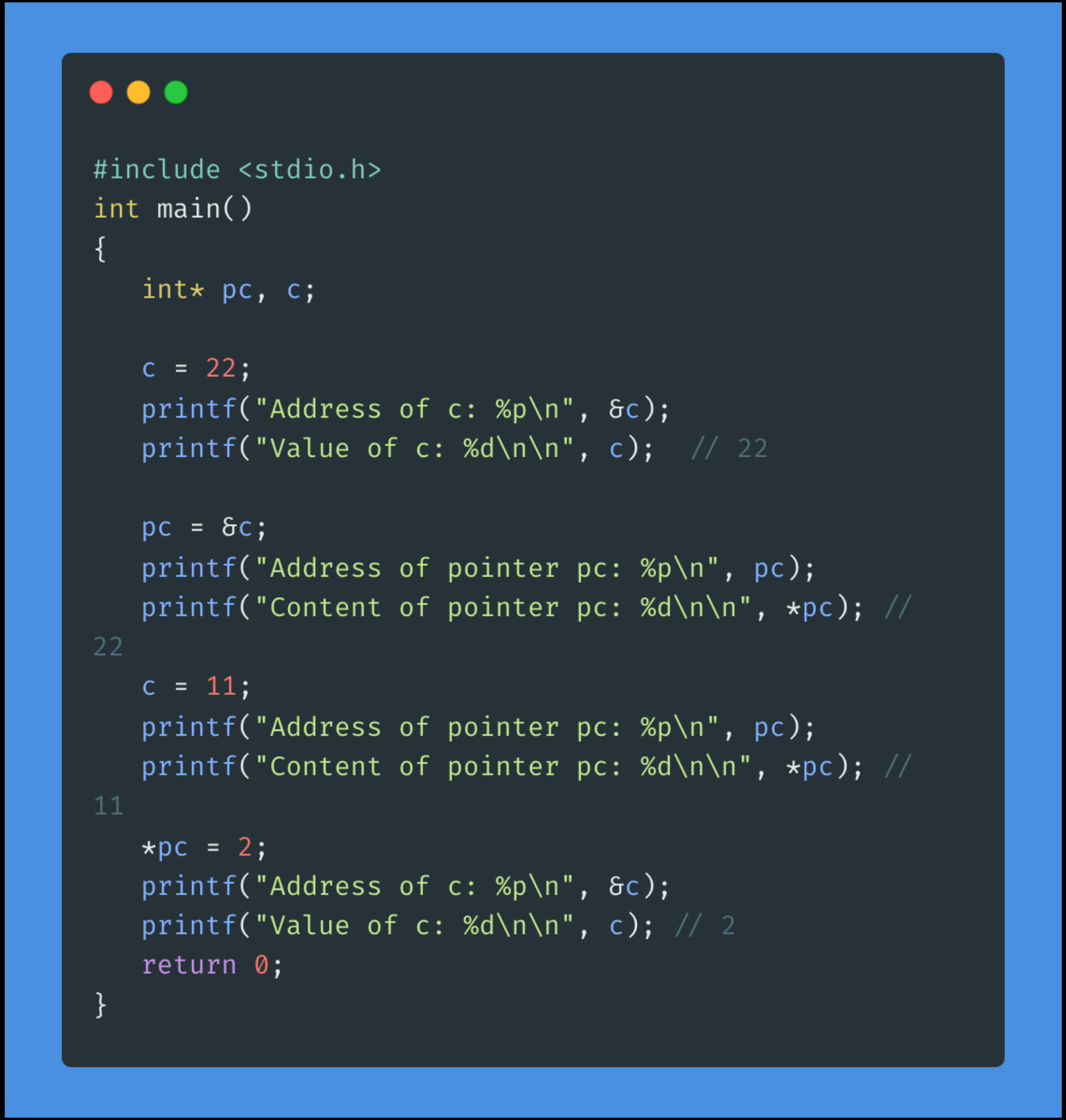
```
int* pc, c;

c = 5;

pc = &c;
```

# Get Value of Thing Pointed by Pointers



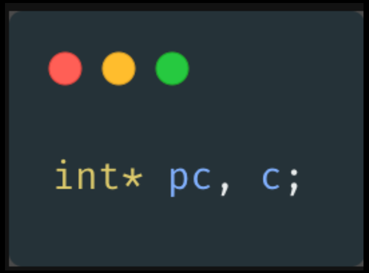
```c
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);    // Output:
5
```

# Example

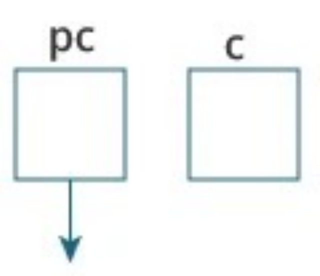


```
#include <stdio.h>
int main()
{
   int* pc, c;

   c = 22;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); //
22
   c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); //
11
   *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

# Explanation

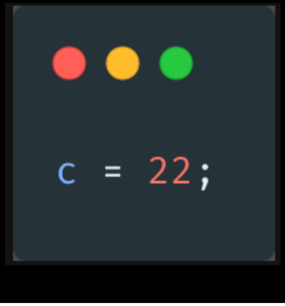


```
int* pc, c;
```

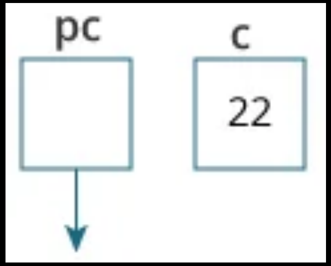


Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created.
Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

# Explanation

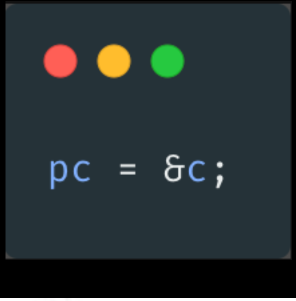


```
c = 22;
```

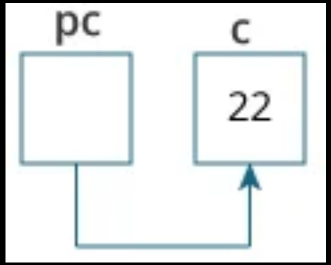


This assigns 22 to the variable c. That is, 22 is stored in the memory location of variable c.

# Explanation

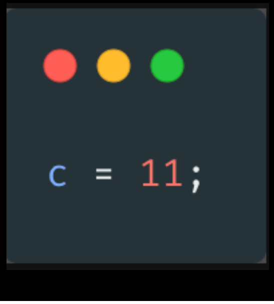


```
pc = &c;
```

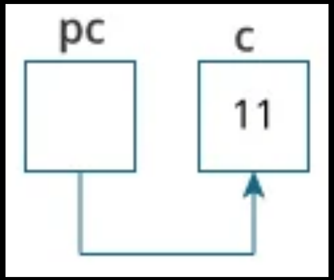


This assigns the address of variable c to the pointer pc.

# Explanation

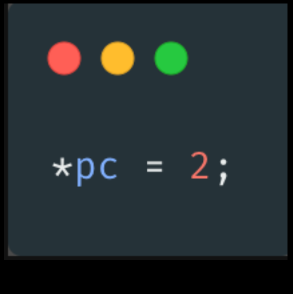


```
c = 11;
```

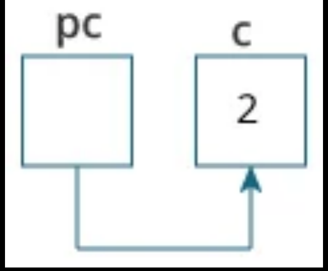


This assigns 11 to variable c.

# Explanation



```
*pc = 2;
```

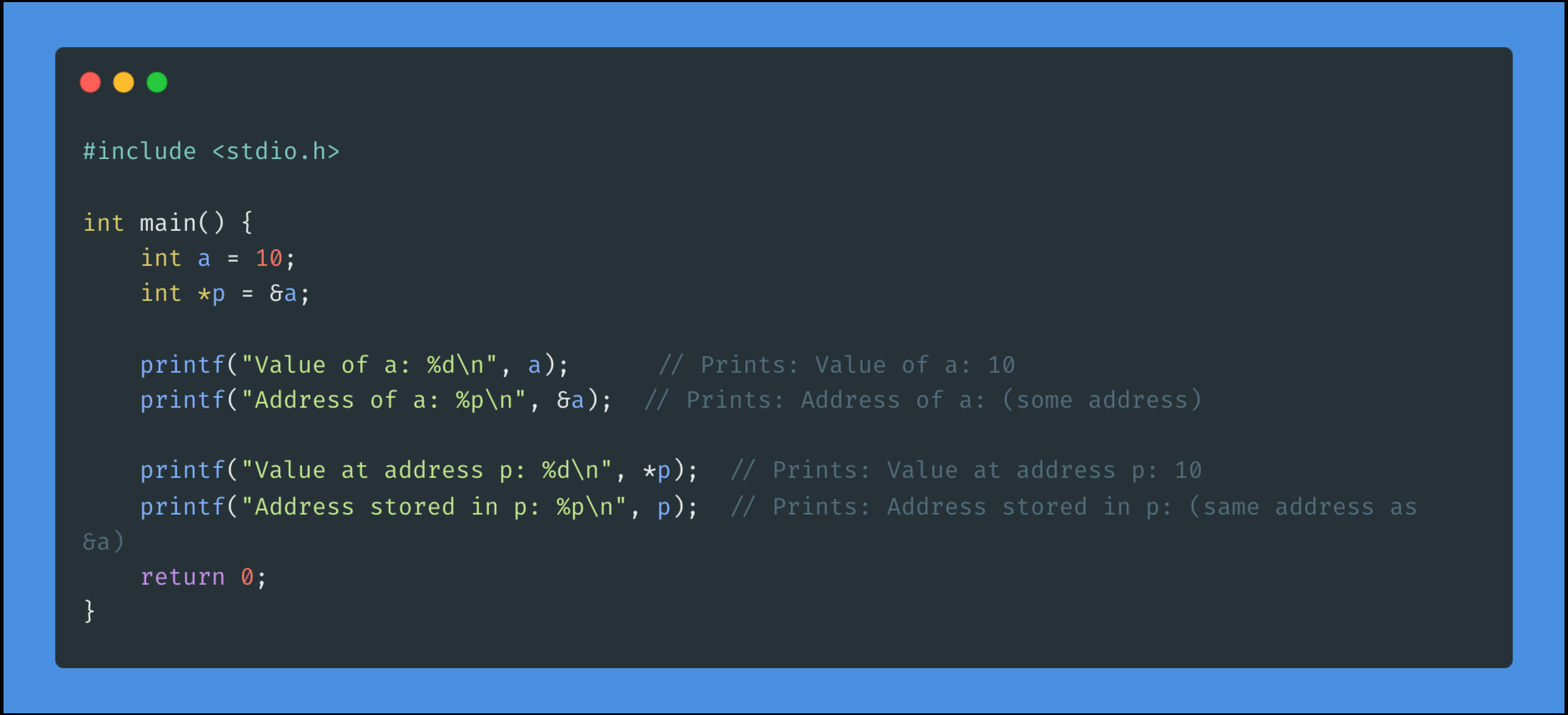


This change the value at the memory location pointed by the pointer pc to 2.

NOTE:

When we try to access with * operator it’s called dereferencing.
The * operator is called dereferencing operator.

# Working of Pointers

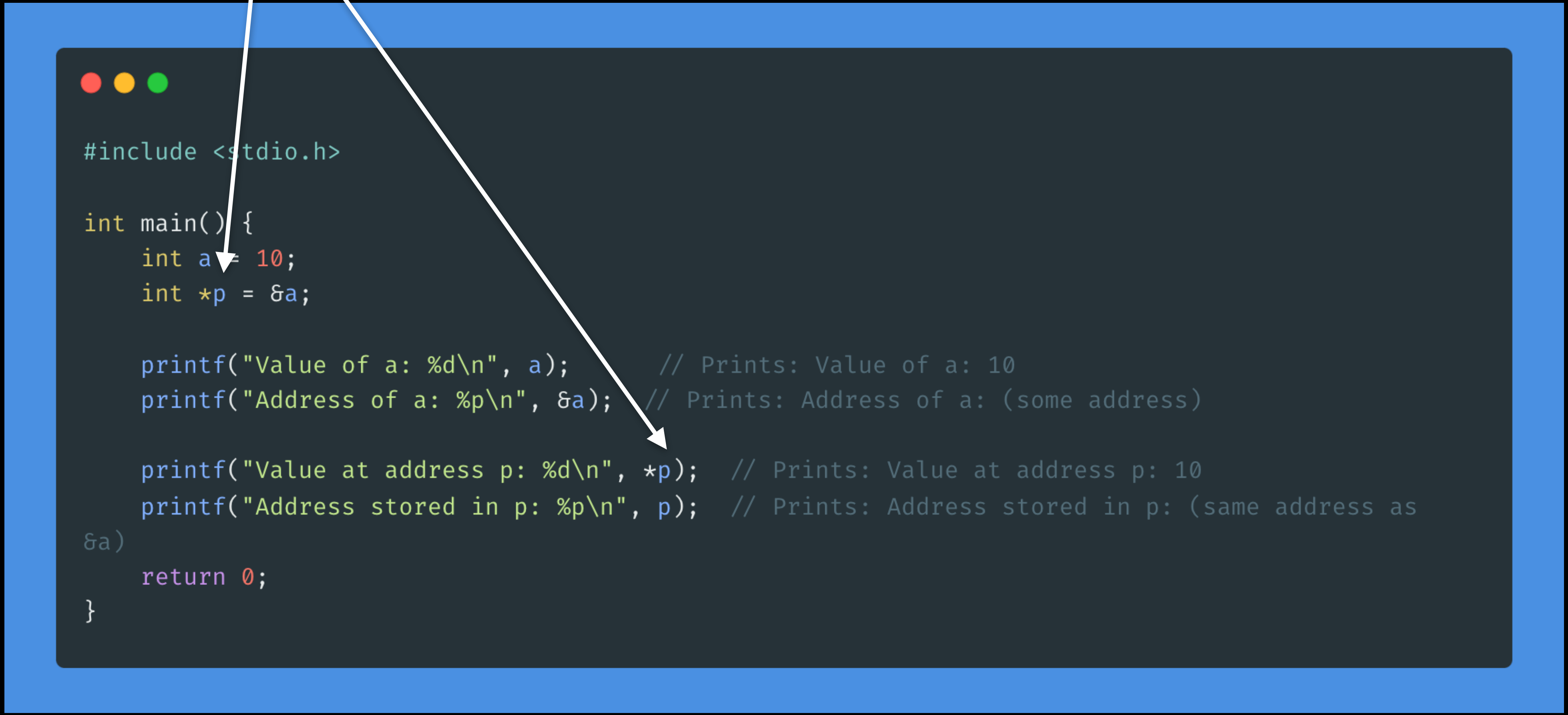


```c
#include <stdio.h>

int main() {
    int a = 10;
    int *p = &a;

    printf("Value of a: %d\n", a);      // Prints: Value of a: 10
    printf("Address of a: %p\n", &a);  // Prints: Address of a: (some address)

    printf("Value at address p: %d\n", *p);  // Prints: Value at address p: 10
    printf("Address stored in p: %p\n", p);  // Prints: Address stored in p: (same address as
&a)
    return 0;
}
```

# Working of pointers

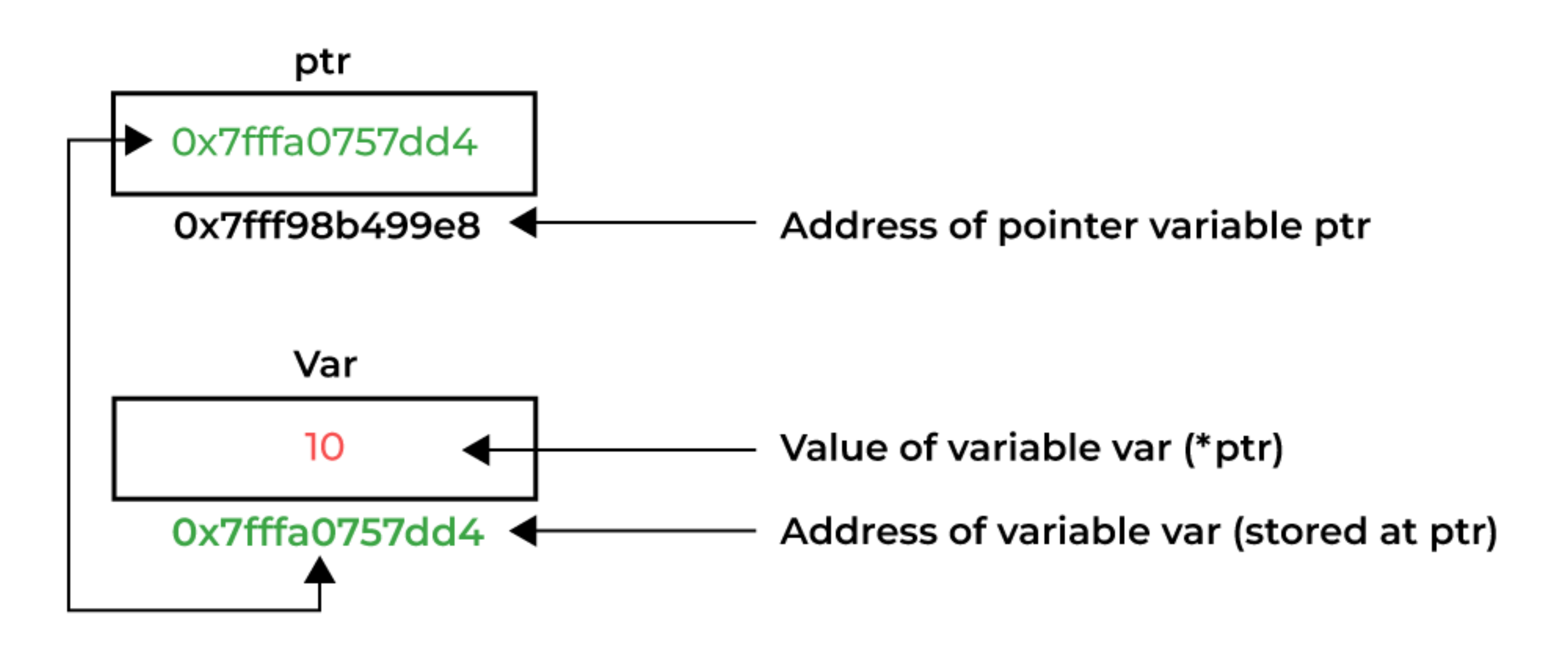
When Declaring : int *p = integer pointer p

When we access: *p = content of p

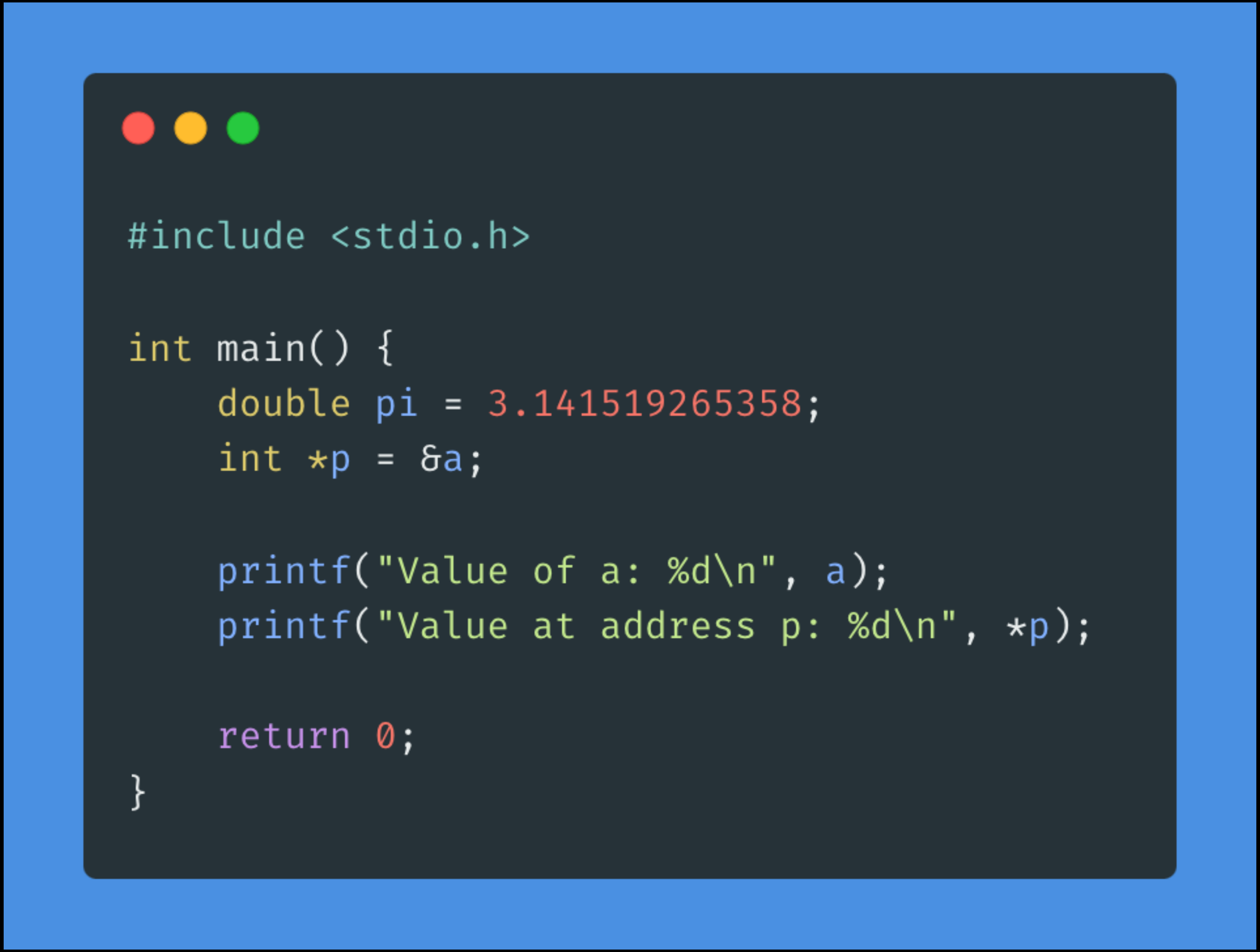


```
#include <stdio.h>

int main() {
    int a = 10;
    int *p = &a;

    printf("Value of a: %d\n", a);       // Prints: Value of a: 10
    printf("Address of a: %p\n", &a);   // Prints: Address of a: (some address)

    printf("Value at address p: %d\n", *p);  // Prints: Value at address p: 10
    printf("Address stored in p: %p\n", p);  // Prints: Address stored in p: (same address as
&a)
    return 0;
}
```

# Working of a pointer


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

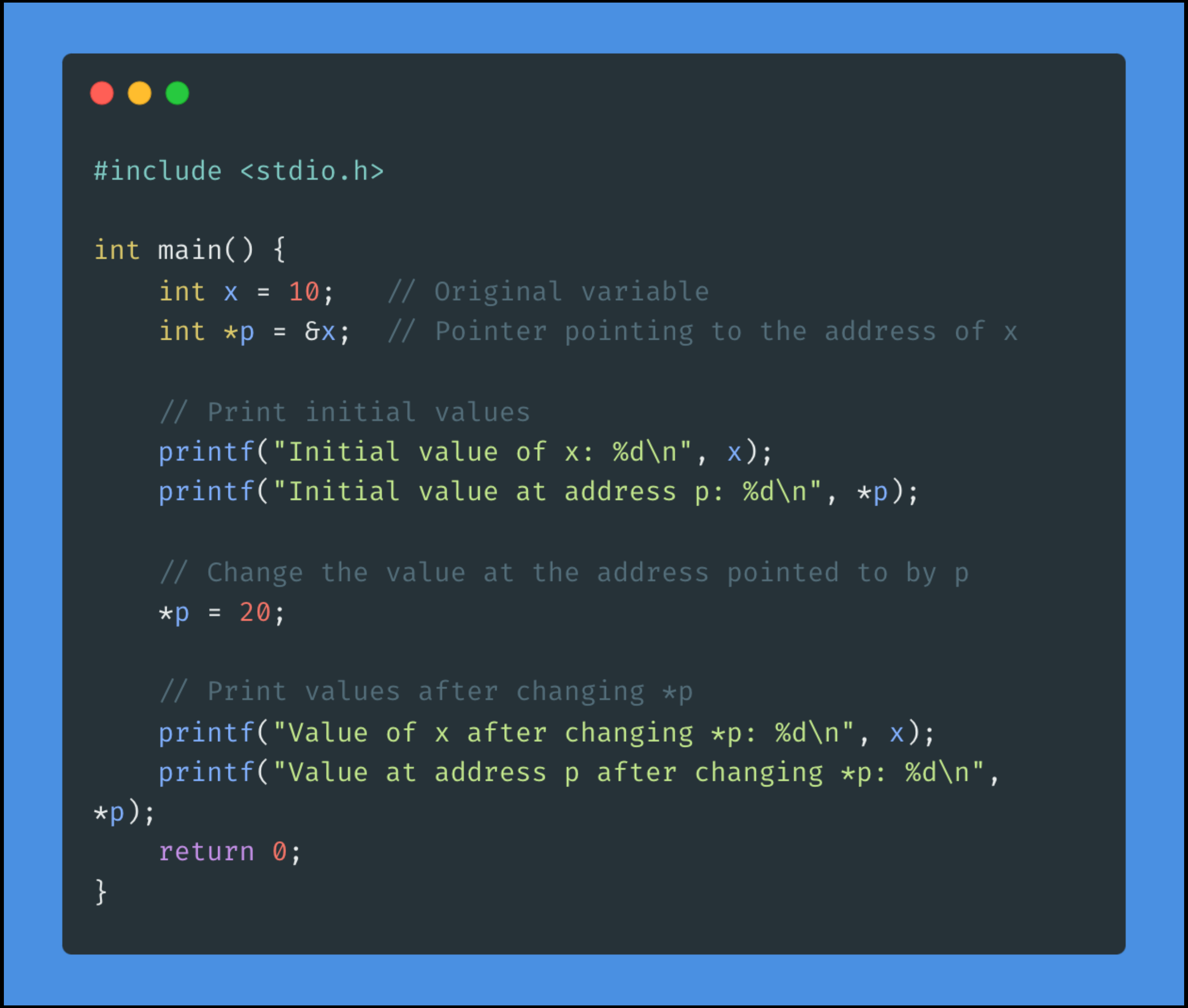
# Let’s write a wrong code



```c
#include <stdio.h>

int main() {
    double pi = 3.141519265358;
    int *p = &a;

    printf("Value of a: %d\n", a);
    printf("Value at address p: %d\n", *p);

    return 0;
}
```

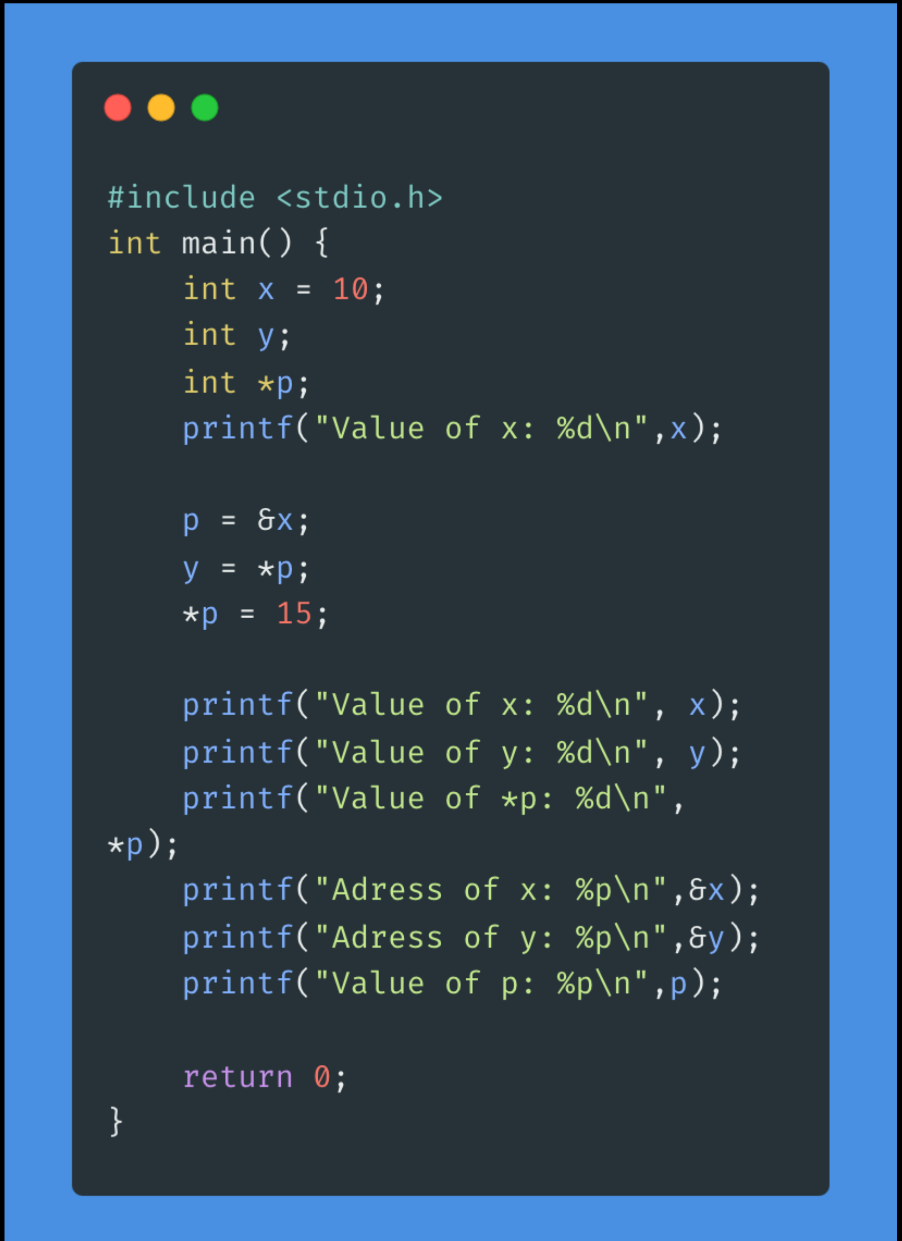
What’s the problem with this code?

# Effect of changing content of p



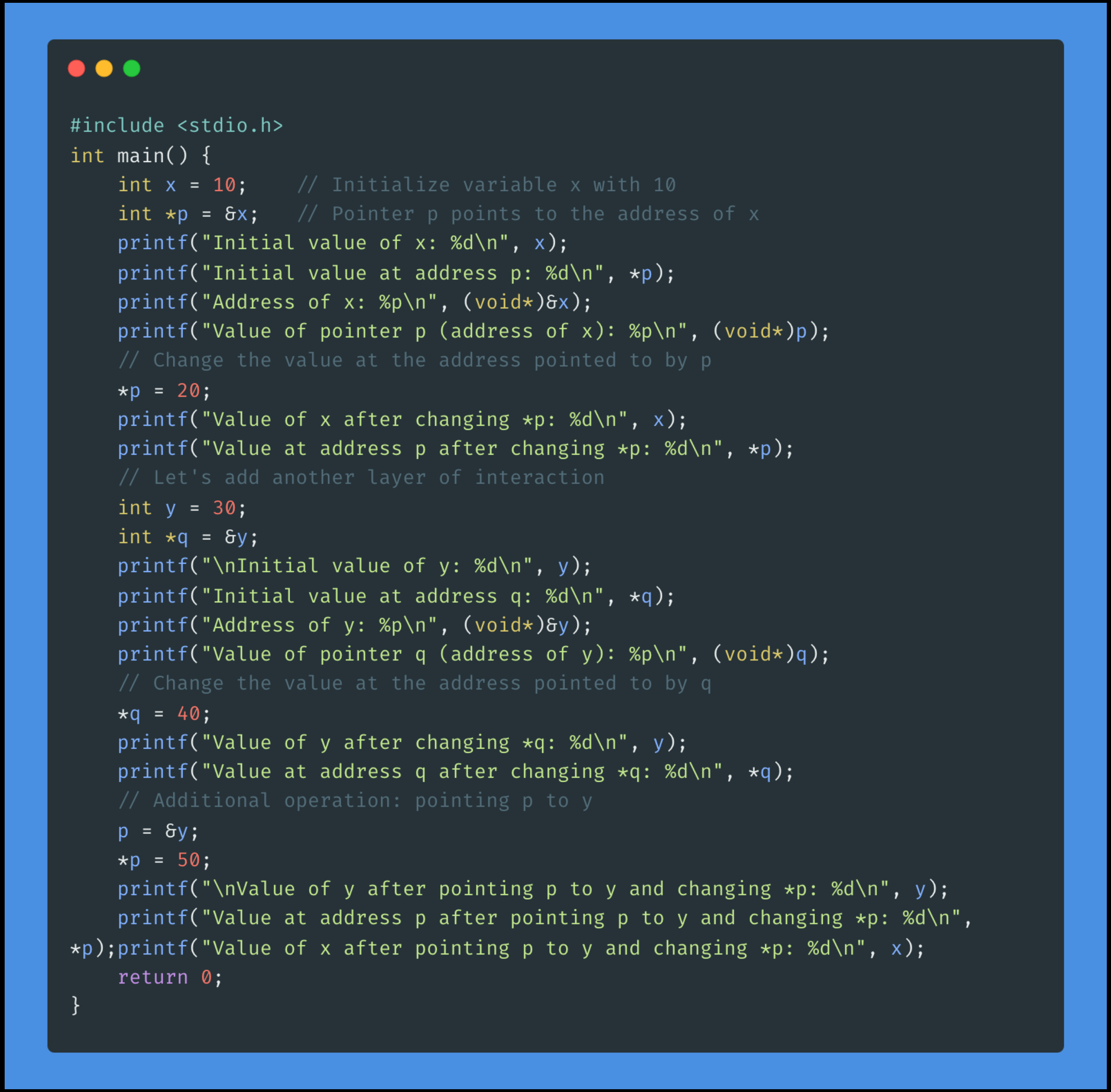
```c
#include <stdio.h>

int main() {
    int x = 10;   // Original variable
    int *p = &x;  // Pointer pointing to the address of x

    // Print initial values
    printf("Initial value of x: %d\n", x);
    printf("Initial value at address p: %d\n", *p);

    // Change the value at the address pointed to by p
    *p = 20;

    // Print values after changing *p
    printf("Value of x after changing *p: %d\n", x);
    printf("Value at address p after changing *p: %d\n",
*p);
    return 0;
}
```

# Example Code



```c
#include <stdio.h>
int main() {
    int x = 10;
    int y;
    int *p;
    printf("Value of x: %d\n",x);

    p = &x;
    y = *p;
    *p = 15;

    printf("Value of x: %d\n", x);
    printf("Value of y: %d\n", y);
    printf("Value of *p: %d\n",
*p);
    printf("Adress of x: %p\n",&x);
    printf("Adress of y: %p\n",&y);
    printf("Value of p: %p\n",p);

    return 0;
}
```

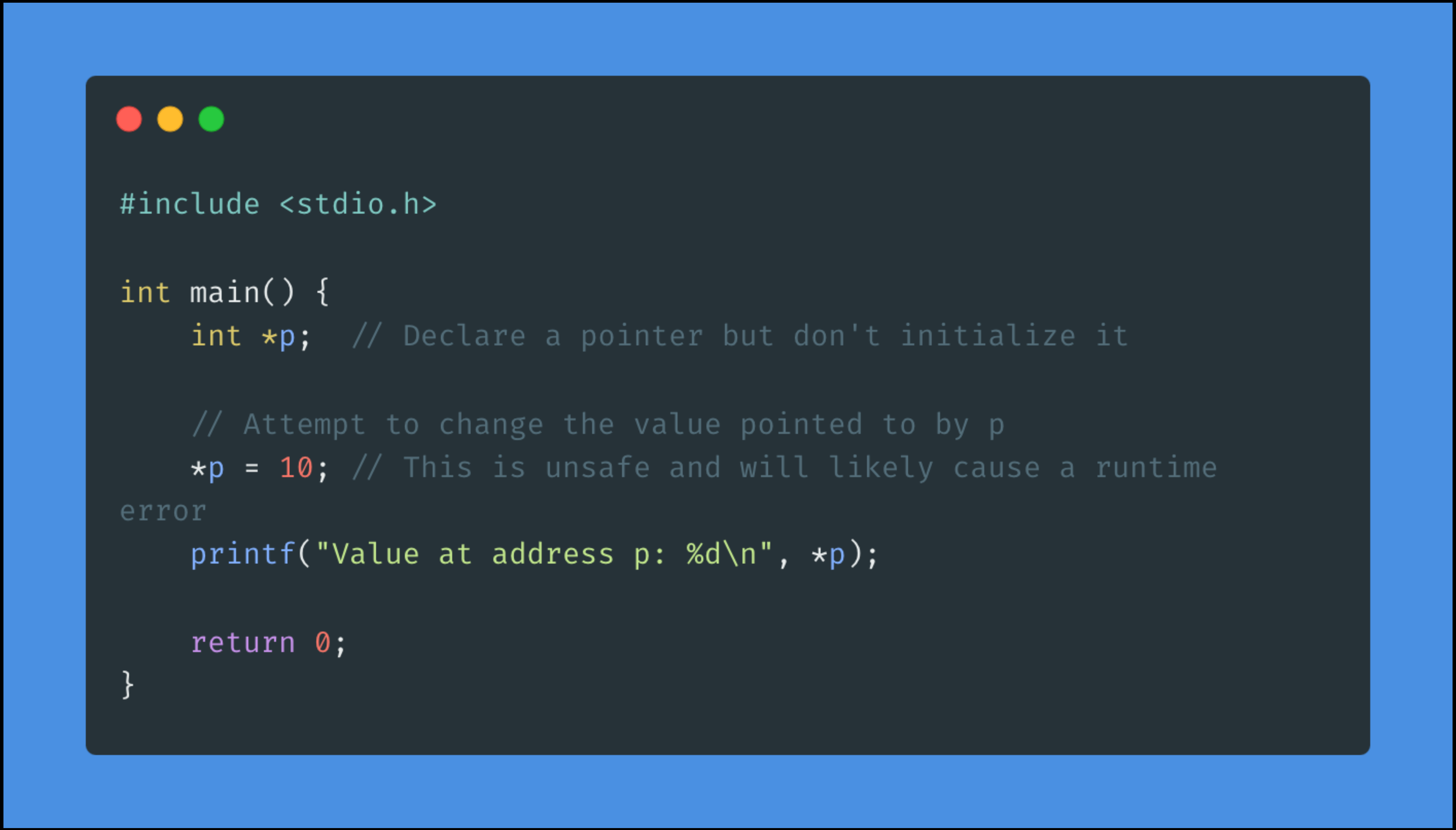
# Guess the output!



```c
#include <stdio.h>
int main() {
    int x = 10;    // Initialize variable x with 10
    int *p = &x;   // Pointer p points to the address of x
    printf("Initial value of x: %d\n", x);
    printf("Initial value at address p: %d\n", *p);
    printf("Address of x: %p\n", (void*)&x);
    printf("Value of pointer p (address of x): %p\n", (void*)p);
    // Change the value at the address pointed to by p
    *p = 20;
    printf("Value of x after changing *p: %d\n", x);
    printf("Value at address p after changing *p: %d\n", *p);
    // Let's add another layer of interaction
    int y = 30;
    int *q = &y;
    printf("\nInitial value of y: %d\n", y);
    printf("Initial value at address q: %d\n", *q);
    printf("Address of y: %p\n", (void*)&y);
    printf("Value of pointer q (address of y): %p\n", (void*)q);
    // Change the value at the address pointed to by q
    *q = 40;
    printf("Value of y after changing *q: %d\n", y);
    printf("Value at address q after changing *q: %d\n", *q);
    // Additional operation: pointing p to y
    p = &y;
    *p = 50;
    printf("\nValue of y after pointing p to y and changing *p: %d\n", y);
    printf("Value at address p after pointing p to y and changing *p: %d\n",
*p);printf("Value of x after pointing p to y and changing *p: %d\n", x);
    return 0;
}
```

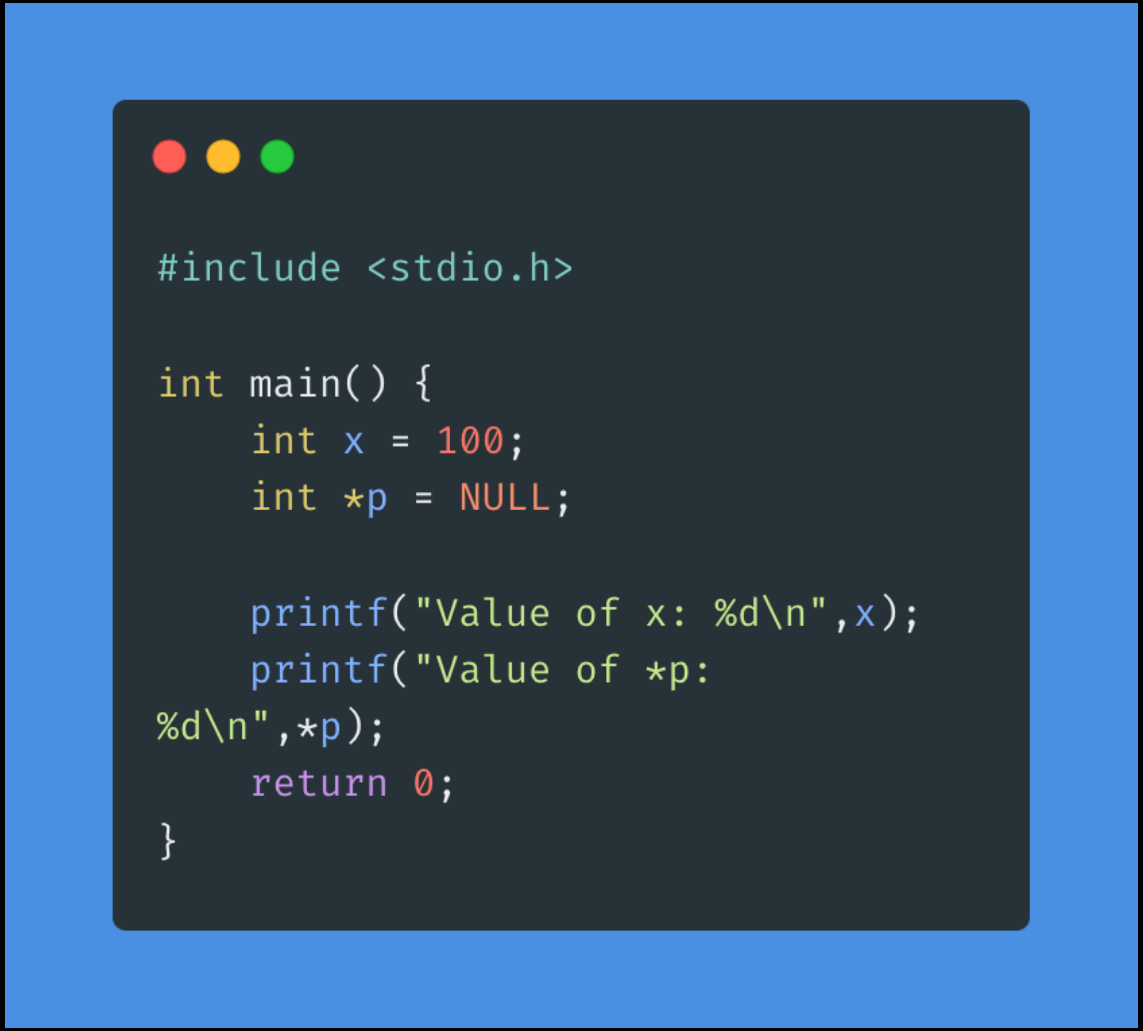
# Let’s write a wrong code

What’s the problem with this code?

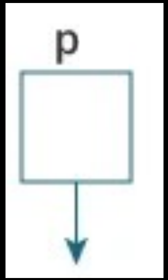


```
#include <stdio.h>

int main() {
    int *p;  // Declare a pointer but don't initialize it

    // Attempt to change the value pointed to by p
    *p = 10; // This is unsafe and will likely cause a runtime
error
    printf("Value at address p: %d\n", *p);

    return 0;
}
```

# Null Pointer

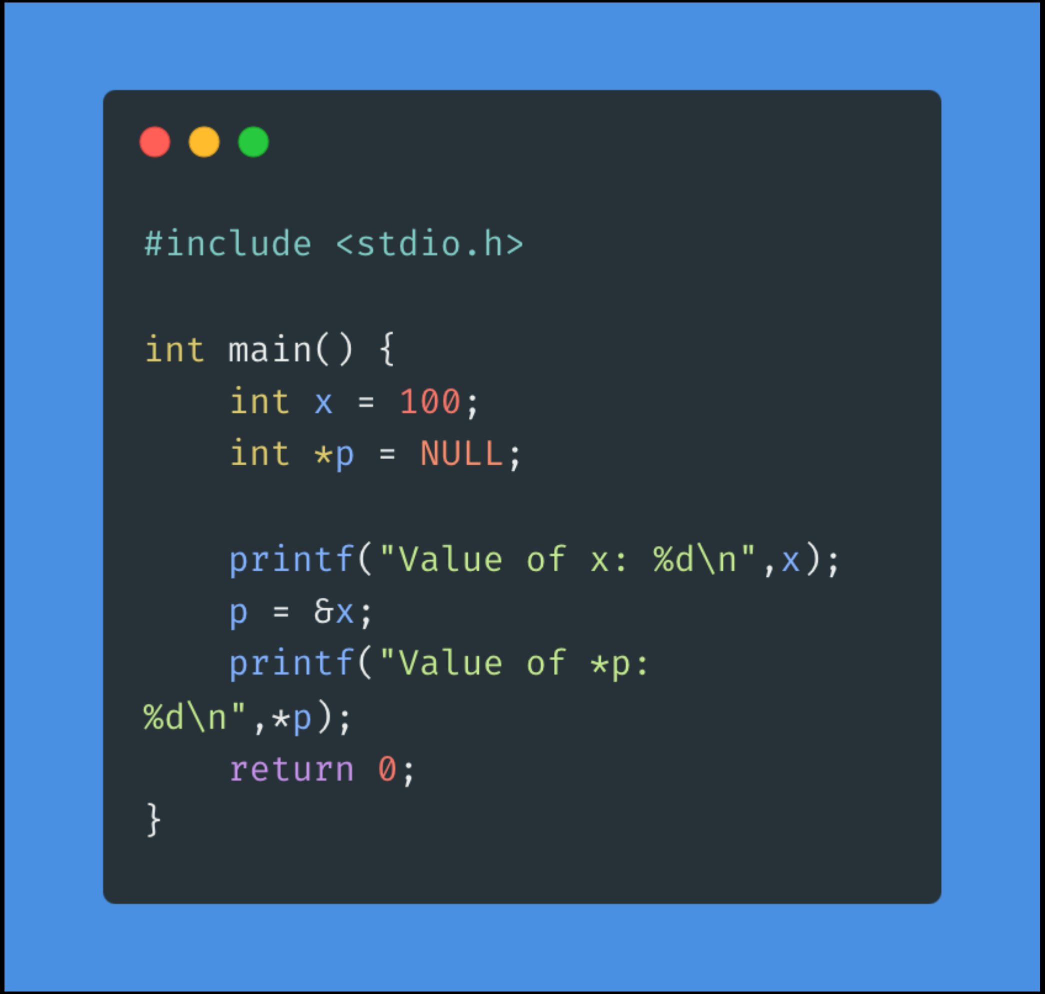


```
#include <stdio.h>

int main() {
    int x = 100;
    int *p = NULL;

    printf("Value of x: %d\n",x);
    printf("Value of *p:
%d\n",*p);
    return 0;
}
```



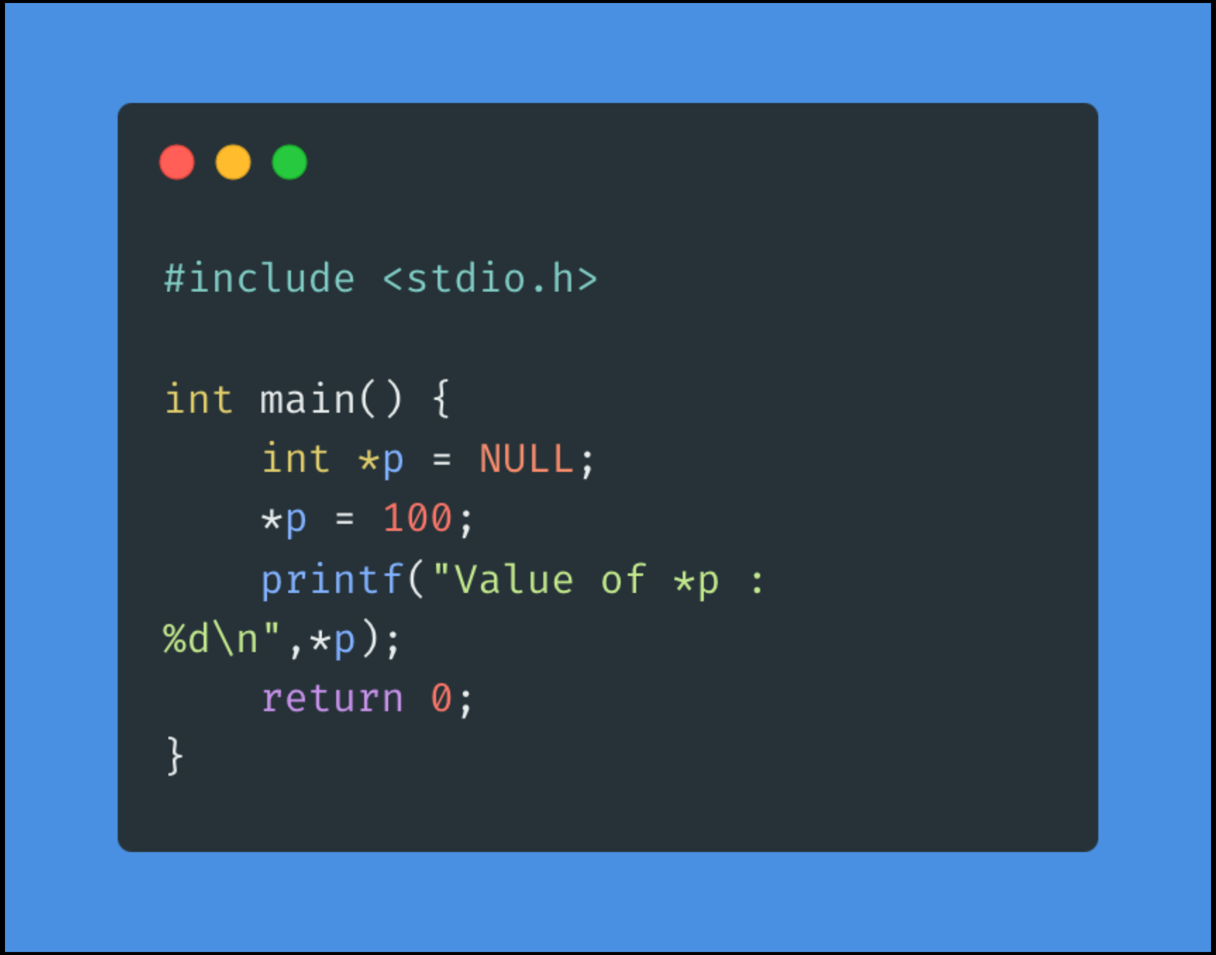
Segmentation Fault!!

# Null Pointer



```c
#include <stdio.h>

int main() {
    int x = 100;
    int *p = NULL;

    printf("Value of x: %d\n",x);
    p = &x;
    printf("Value of *p:
%d\n",*p);
    return 0;
}
```

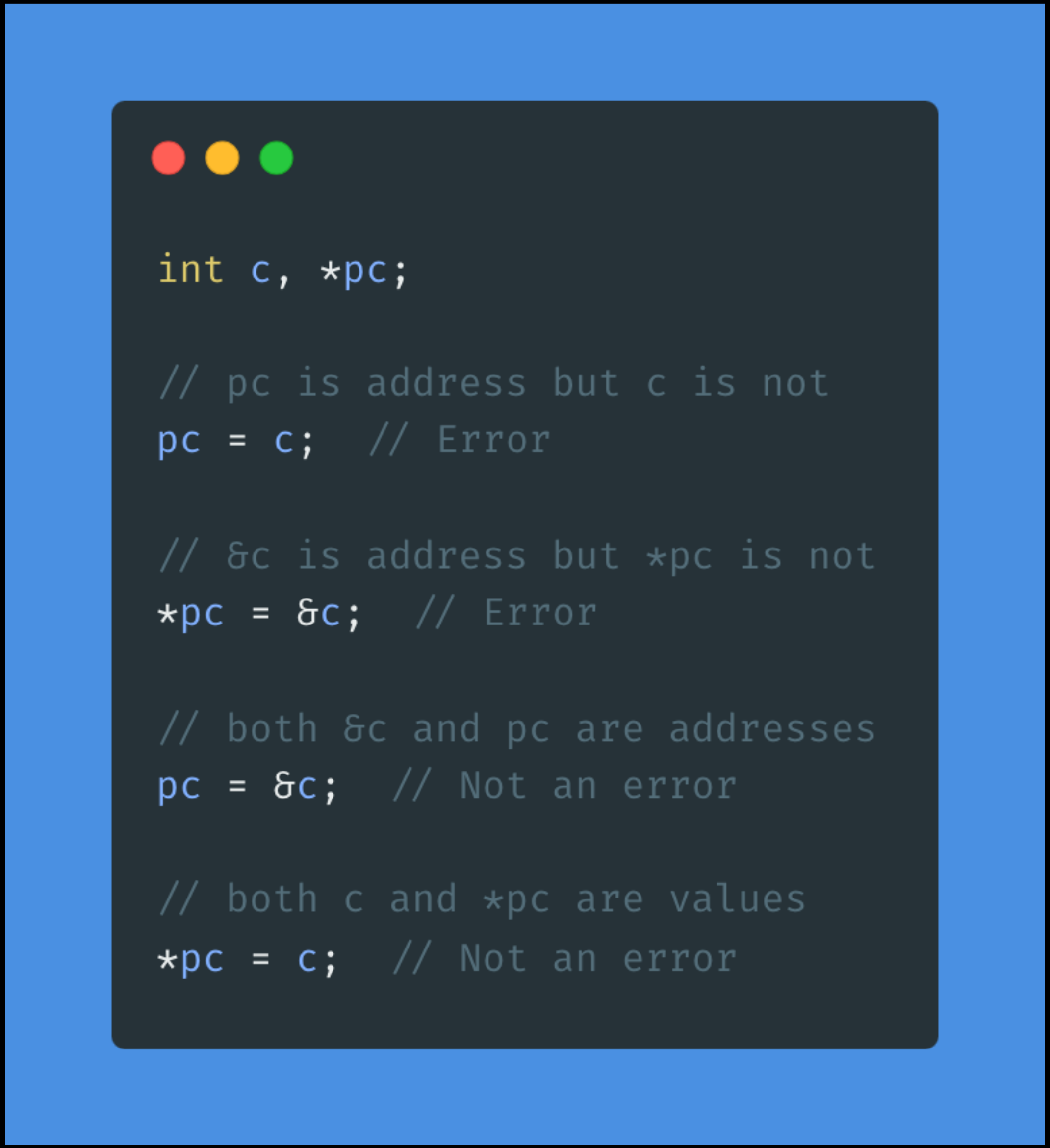
# Null Pointer



```
#include <stdio.h>

int main() {
    int *p = NULL;
    *p = 100;
    printf("Value of *p :
%d\n",*p);
    return 0;
}
```

Segmentation Fault!!

# Common mistakes with pointer



```c
int c, *pc;

// pc is address but c is not
pc = c;  // Error

// &c is address but *pc is not
*pc = &c;  // Error

// both &c and pc are addresses
pc = &c;  // Not an error

// both c and *pc are values
*pc = c;  // Not an error
```