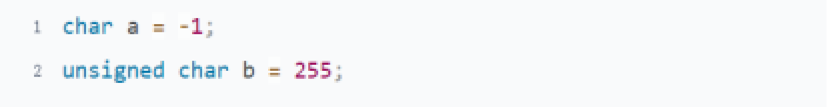
1. **What is the difference between the following 2 values in memory? Are they the same or not? Why?**



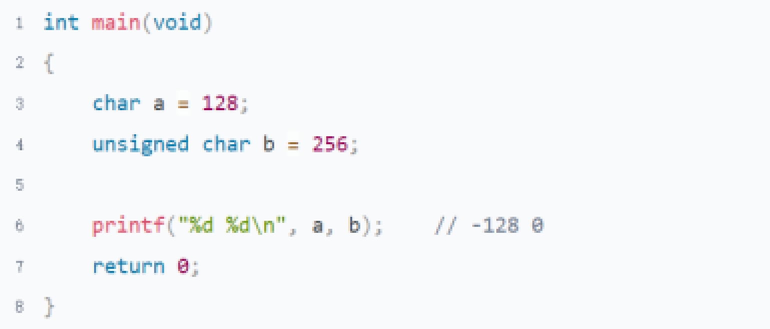
```c
char a = -1;
unsigned char b = 255;
```

   a. variable a is stored in memory as signed char and variable b is stored in memory as unsigned char but both are stored 11111111 in binary. They are identical in memory.

   b. In most systems, a char is a signed 8-bit integer This means that it can hold values from -128 to 127. When we assign -1 to a, it will be stored as 11111111 in binary.

   c. An unsigned char is an 8-bit integer that can hold values from 0 to 255. The value 255 in binary is 11111111, which is the highest value for an unsigned 8-bit integer.

2. **Try to explain why the result of the following program is that way. What happened?**



```c
int main(void)
{
    char a = 128;
    unsigned char b = 256;

    printf("%d %d\n", a, b);    // -128 0
    return 0;
}
```

   a. The result will be a = -128 and b = 0.

   b. The char type typically a signed 8 bit integer in most memory system and it’s range is -128 to 127. 128 exceeds the maximum value but its bit representation is 10000000 which is also equivalent to -128 in sign char type.

c. The unsigned char type is an 8-bit integer that can hold values in the range of 0 to 255. 256 exceeds the maximum value for an unsigned char, so overflow will be occurred. 256 binary representation is 100000000. So right most 8 bits 00000000 only will be stored in memory. So b will be 0

3. **What is an extension when storing a number? When does it occur?**

    a. **Extension** refers to the process of adjusting the size of a number while preserving its value, particularly when converting it between different **data types** or **bit-widths**. This is necessary when a number is stored in a type that has a larger bit-width than the original type, so it can fit correctly into the new type without altering the value.

    There are two common types of **extension**:

    i. **Sign Extension**: Sign extension occurs when a **signed number** is expanded to a larger bit-width. The goal is to preserve the sign.For example, **when a 16-bit signed integer is stored in a 32-bit variable.**

    ii. **Zero Extension**: Zero extension occurs when an **unsigned number** is extended to a larger bit-width. Unlike sign extension, zero extension fills the additional higher-order bits with zeros. For example, **when an 8-bit unsigned integer is stored in a 16-bit variable.**

4. **What's an integer promotion? What's the rule of it? What scenarios does it occur?**

    a. Integer promotion is an implicit type conversion in C where smaller integer types (e.g., char, short) are automatically converted to at least int or unsigned int when used in expressions.

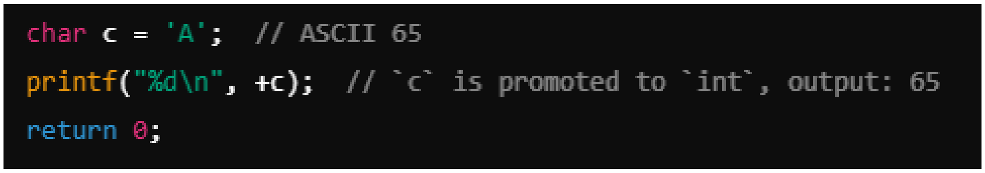
b. Rules of Integer Promotion:

   i. **Operands smaller than int are promoted to int :** This applies to: int, char, signed char, unsigned char short, unsigned short

   ii. **Only Affects Expressions, Not Assignments:** Promotions happen in expressions like arithmetic operations but do not change variable types or value.

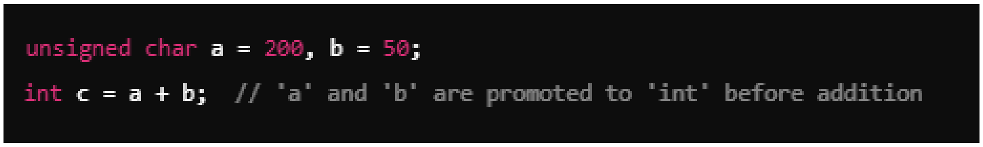
c. When Does Integer Promotion Occur:

   i. Unary Operations(+, -, ~)

      1. 

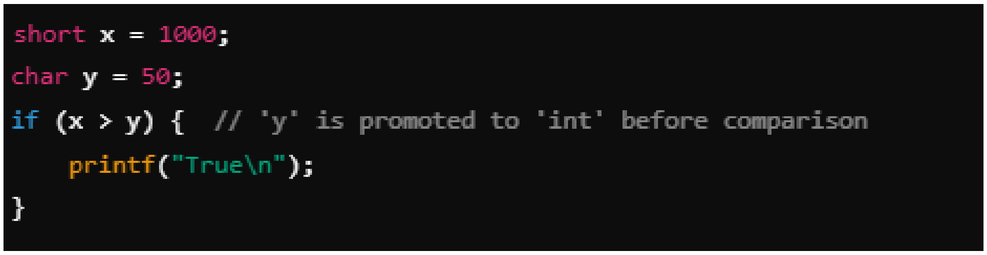
```
char c = 'A';  // ASCII 65
printf("%d\n", +c);  // `c` is promoted to `int`, output: 65
return 0;
```

   ii. Binary Operations (+, -, *, /, %)

      1. 

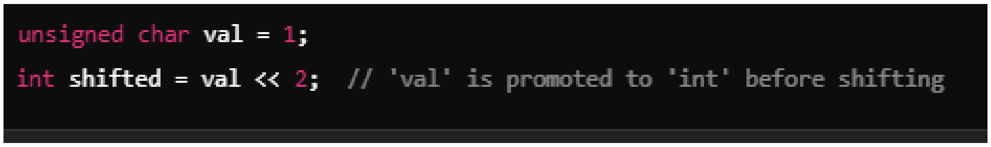
```
unsigned char a = 200, b = 50;
int c = a + b;  // 'a' and 'b' are promoted to 'int' before addition
```

   iii. Relational and Logical Operations (==, !=, <, >, &&, ||)

      1. 

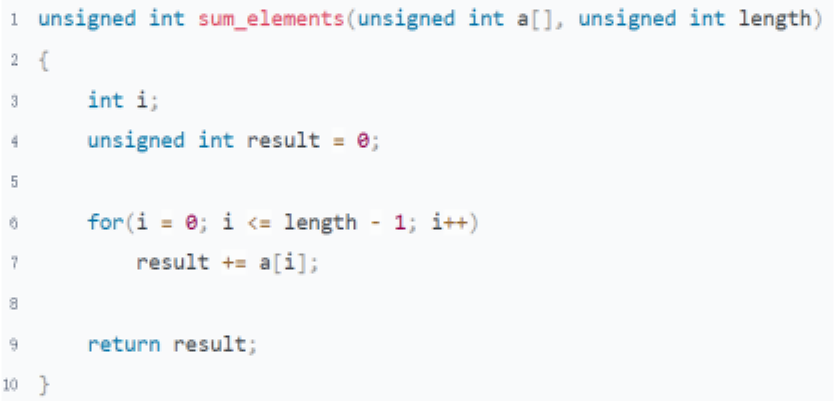
```
short x = 1000;
char y = 50;
if (x > y) {  // 'y' is promoted to 'int' before comparison
    printf("True\n");
}
```

   iv. Bitwise shift

      1. 

```
unsigned char val = 1;
int shifted = val << 2;  // 'val' is promoted to 'int' before shifting
```

5. **What happened in the following program when the argument length is 0? Why?**

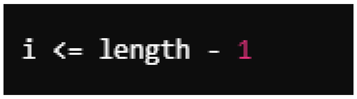


```c
unsigned int sum_elements(unsigned int a[], unsigned int length)
{
    int i;
    unsigned int result = 0;

    for(i = 0; i <= length - 1; i++)
        result += a[i];

    return result;
}
```

a. If the argument length is 0, then the program will be crashed.

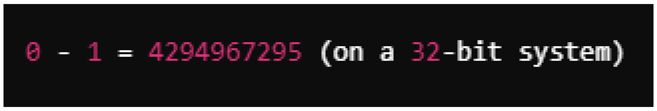
b. Step by step execution:

   i. The for loop condition is

      1. 

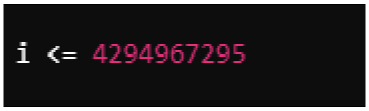
         ```
         i <= length - 1
         ```

   ii. The expression length - 1 can cause unsigned integer **underflow.**

      1. 

         ```
         0 - 1 = 4294967295 (on a 32-bit system)
         ```

   iii. The condition will be

      1. 

         ```
         i <= 4294967295
         ```

   iv. Since **i** is integer type, i always <= 4294967295, this loop runs indefinitely, leading to an out-of-bounds access of memory and potential crash