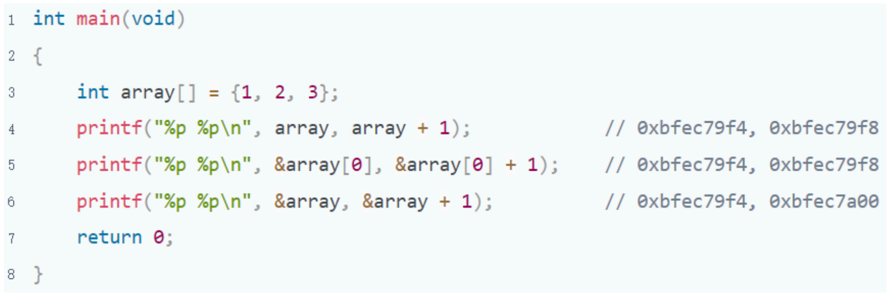
1. What can you learn from the following program?

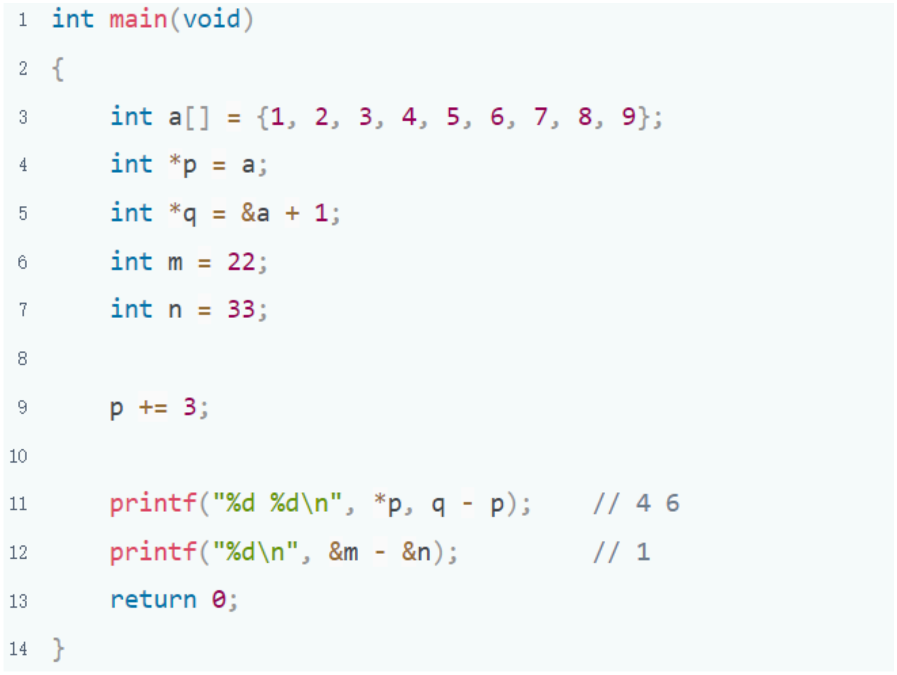


```c
int main(void)
{
    int array[] = {1, 2, 3};
    printf("%p %p\n", array, array + 1);            // 0xbfec79f4, 0xbfec79f8
    printf("%p %p\n", &array[0], &array[0] + 1);    // 0xbfec79f4, 0xbfec79f8
    printf("%p %p\n", &array, &array + 1);          // 0xbfec79f4, 0xbfec7a00
    return 0;
}
```

Understanding from the Given Program:

a. Array Name (**array**) Decays to Pointer:
    i. **array** is equivalent to **&array[0]** in pointer arithmetic.
b. Pointer Arithmetic Moves by Element Size:
    i. **array + 1** moves forward by **sizeof(int)**, not 1 byte.
c. **&array** Represents the Entire Array:
    i. **&array + 1** moves by the size of the full array, not just one element.
d. **array vs &array** Difference:
    i. **array** behaves like a pointer to **int**.
    ii. **&array** behaves like a pointer to **int[3]**.

2. Why does the following program have such an output? Is it reasonable?



```c
int main(void)
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *p = a;
    int *q = &a + 1;
    int m = 22;
    int n = 33;

    p += 3;

    printf("%d %d\n", *p, q - p);    // 4 6
    printf("%d\n", &m - &n);         // 1
    return 0;
}
```

a. Why does the program have such an output:

   i. int *p = a; means p initially points to a[0].

   ii. q = &a + 1;
      1. &a is the address of the whole array.
      2. &a + 1 moves one full array size ahead, moving &a + 1 jumps by 9 * sizeof(int).

   iii. Pointer Arithmetic:
      1. p initially pointed to a[0], so p += 3 moves it 3 positions forward.
      2. Now, p points to a[3], which contains 4.

   iv. First printf Statement:
      1. *p: Since p now points to a[3], *p prints 4.
      2. q - p: This calculates the distance between q and p:

         q = a + 9

         p = a + 3

         q - p = 6

   v. Second printf Statement :
      1. &m - &n computes the pointer difference between m and n in memory.
      2. &m - &n calculates the number of int-sized gaps between them that is 1.

b. Yes, this output is **reasonable** because:
   i. Pointer arithmetic on **p** and **q** works correctly.
   ii. Pointer subtraction **(q - p)** correctly computes the number of elements.
   iii. **&m - &n** correctly counts the number of **int-sized** memory slots between variables

3. Why does the following program have such an output?

```c
void fun(int array[])
{
    printf("The lengh of array in fun is %d\n", sizeof(array));
    return;
}

int main(void)
{
    int array[5] = {1, 2, 3, 4, 5};
    fun(array);
    printf("The lengh of array in main is %d\n", sizeof(array));

    return 0;
}
```

Why does the program have such an output:

a. **sizeof** in **main**
    i. **array** is a true array in **main**.
    ii. **sizeof(array)** returns the total number of bytes occupied by array.
    iii. Since array contains 5 integers and assuming sizeof(int) = 4:

sizeof(array) = 5*4 = 20

b. **sizeof in fun**:
    i. array inside fun is actually a pointer **(int *array)**.
    ii. **sizeof(array)** now returns the size of a pointer.
        1. 8 for 64 bit system
        2. 4 for 32 bit system

4. Why does the following program have such an output?

```c
int main(void)
{
    int array[5] = {1, 2, 3, 4, 5};
    int *p = array;

    printf("%p %p\n", &p, p);              // 0xbf84b118 0xbf84b11c
    printf("%p %p\n", &array, array);      // 0xbf84b11c 0xbf84b11c
    return 0;
}
```

Why does the program have such an output:

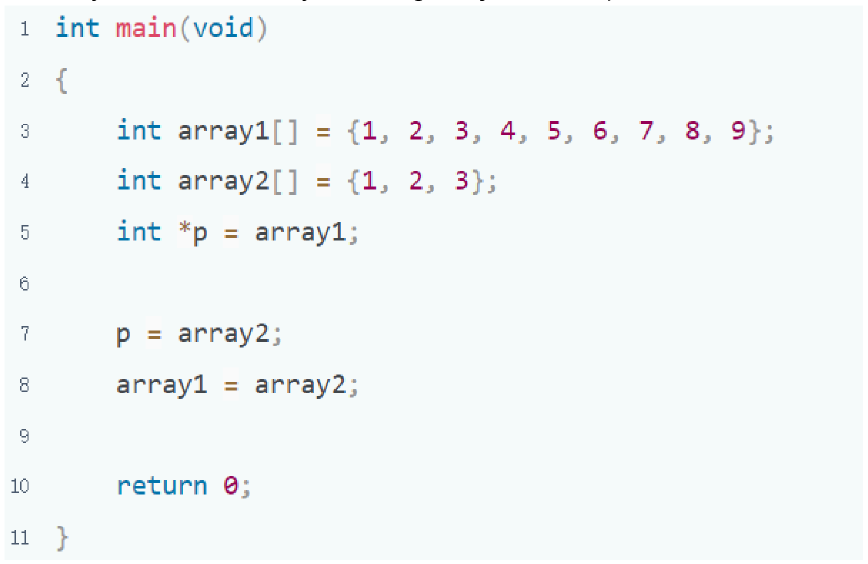
a. **Memory Representation**

Variable	Address	Value
array (first element)	0xbf84b11c	1
p (pointer)	0xbf84b118	0xbf84b11c

b. **printf("%p %p\n", &p, p);**
   i. **p** is a pointer variable that stores the address of **array[0]**.
   ii. **&p** gives the address where **p** itself is stored (separate from array).
   iii. **p** gives the address stored inside **p**, which is the first element of array

c. **printf("%p %p\n", &array, array)**;
   i. array itself is a constant pointer to its first element (**&array[0]**).
   ii. **&array** gives the address of the entire array, which is the same as array but interpreted differently.

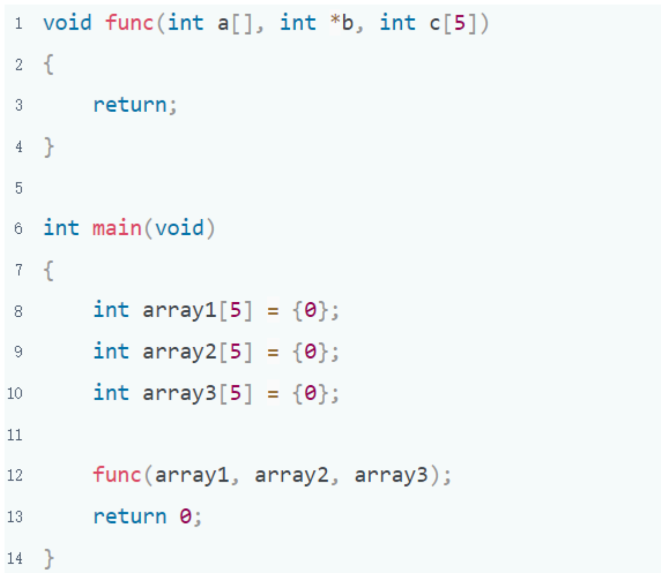
5. Why can not an array be assigned just like a pointer?



```c
int main(void)
{
    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int array2[] = {1, 2, 3};
    int *p = array1;

    p = array2;
    array1 = array2;

    return 0;
}
```

   a. The key reason an array cannot be assigned like a pointer is because an array name is not a modifiable lvalue. Pointers are variable so their value can be changed. Arrays Are Fixed.

6. What is the difference between a, b and c？

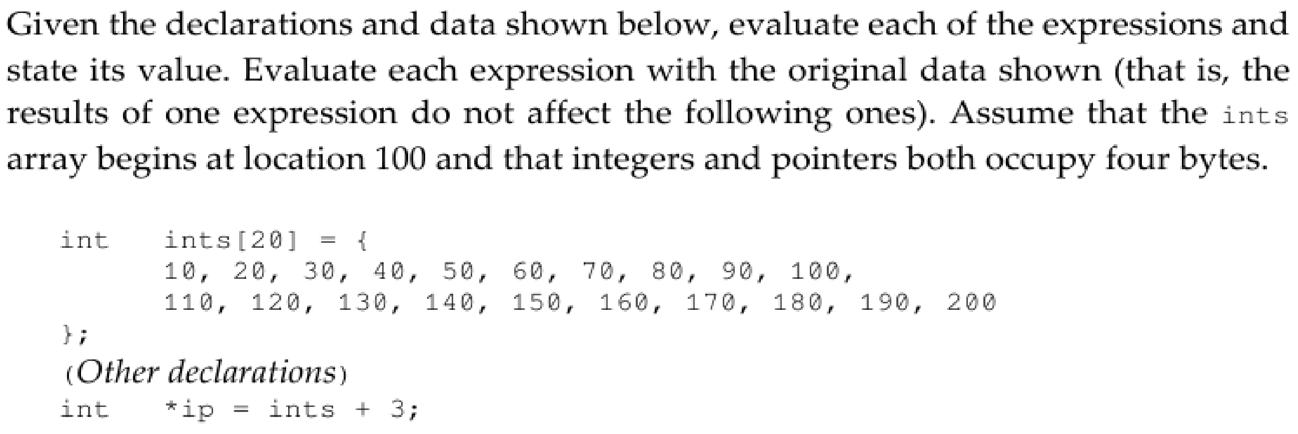


```c
void func(int a[], int *b, int c[5])
{
    return;
}

int main(void)
{
    int array1[5] = {0};
    int array2[5] = {0};
    int array3[5] = {0};

    func(array1, array2, array3);
    return 0;
}
```

This program demonstrates how array arguments behave when passed to a function in C.

a. All three parameters **(a[], *b, and c[5])** are treated as pointers.
b. Even though **a[], *b,** and **c[5]** appear different in function declaration, they are all treated as pointers when passed to **func()**.
c. This means **a, b**, and **c** inside **func()** all decay into **int *** (pointer to int).

7. Pointers-On-C.pdf 8.7 1



Given the declarations and data shown below, evaluate each of the expressions and state its value. Evaluate each expression with the original data shown (that is, the results of one expression do not affect the following ones). Assume that the `ints` array begins at location 100 and that integers and pointers both occupy four bytes.

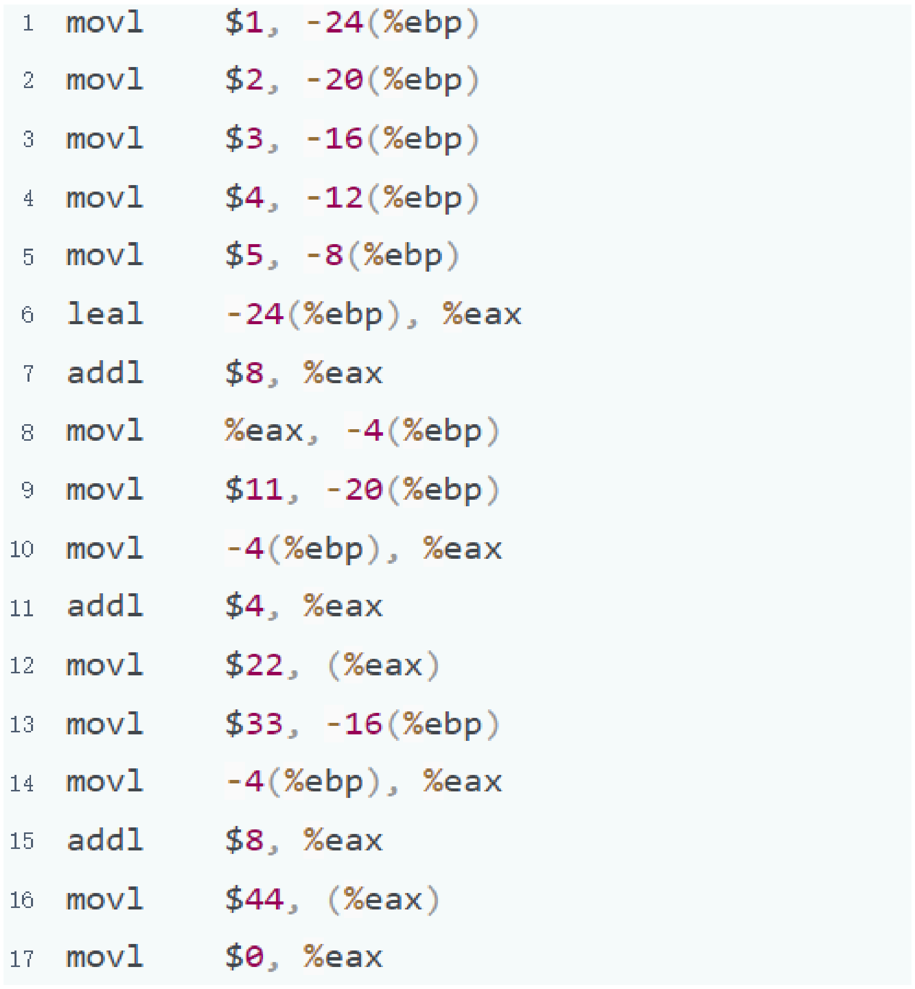
```
int    ints[20] = {
       10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
       110, 120, 130, 140, 150, 160, 170, 180, 190, 200
};
```
(Other declarations)
```
int    *ip = ints + 3;
```

a.

Expression	Value(Decimal)	Expression	Value(Decimal)
ints	100	ip	112
ints[4]	50	ip[4]	80
ints + 4	116	ip + 4	128
*ints + 4	14	*ip + 4;	44
*(ints + 4)	50	*(ip + 4)	80
ints[-2]	Undefined	ip[-2]	20
&ints	100	&ip	can't tell
&ints[4]	116	&ip[4]	128
&ints + 4	420	&ip + 4	can't tell
&ints[-2]	Undefined	&ip[-2]	104

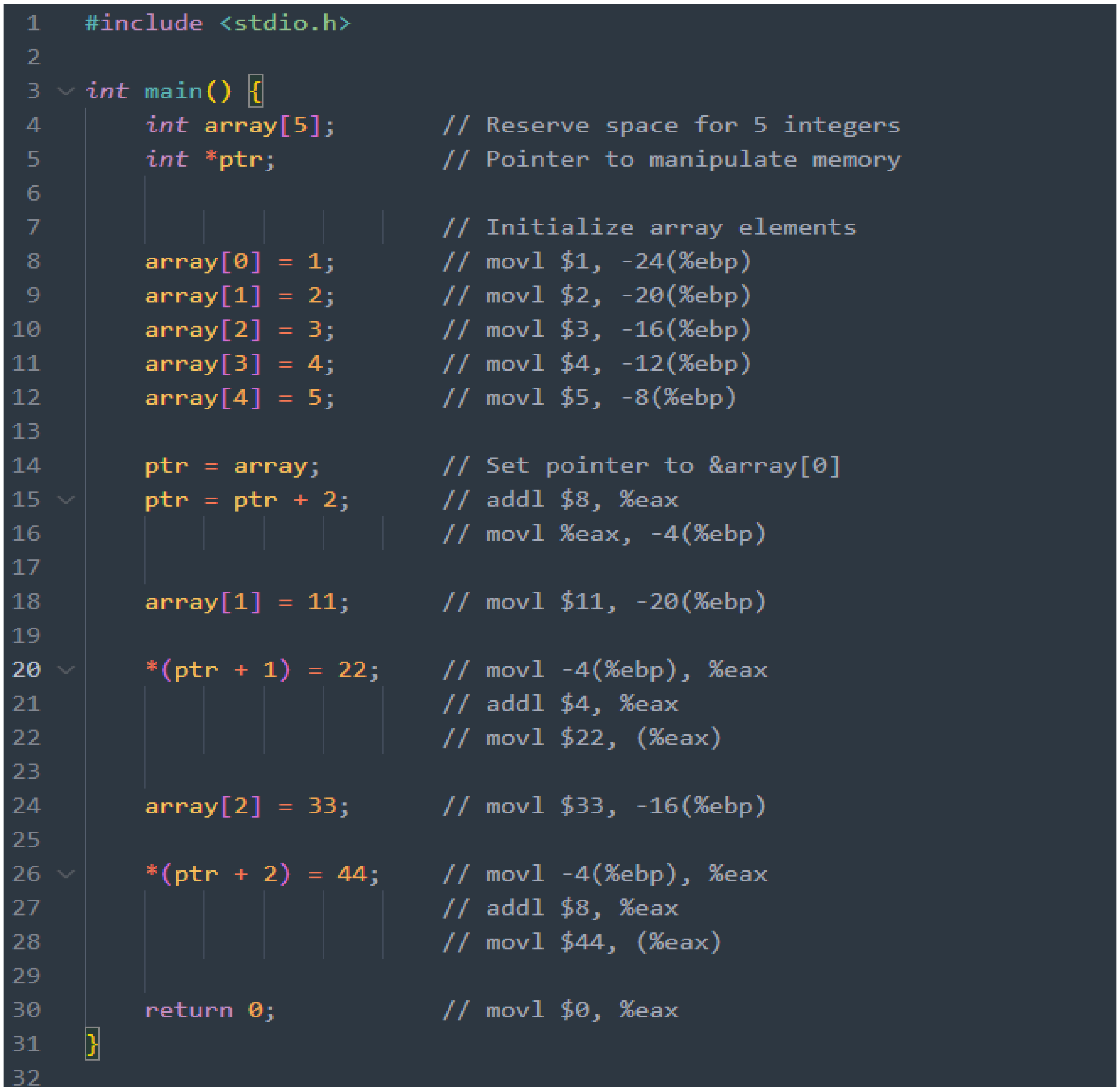
8. Here is a program, answer some questions

```c
int main(void)
{
    int a[] = {1, 2, 3, 4, 5};

    return 0;
}
```



```
movl    $1, -24(%ebp)
movl    $2, -20(%ebp)
movl    $3, -16(%ebp)
movl    $4, -12(%ebp)
movl    $5, -8(%ebp)
leal    -24(%ebp), %eax
addl    $8, %eax
movl    %eax, -4(%ebp)
movl    $11, -20(%ebp)
movl    -4(%ebp), %eax
addl    $4, %eax
movl    $22, (%eax)
movl    $33, -16(%ebp)
movl    -4(%ebp), %eax
addl    $8, %eax
movl    $44, (%eax)
movl    $0, %eax
```

a. Convert the following assembly language to C language

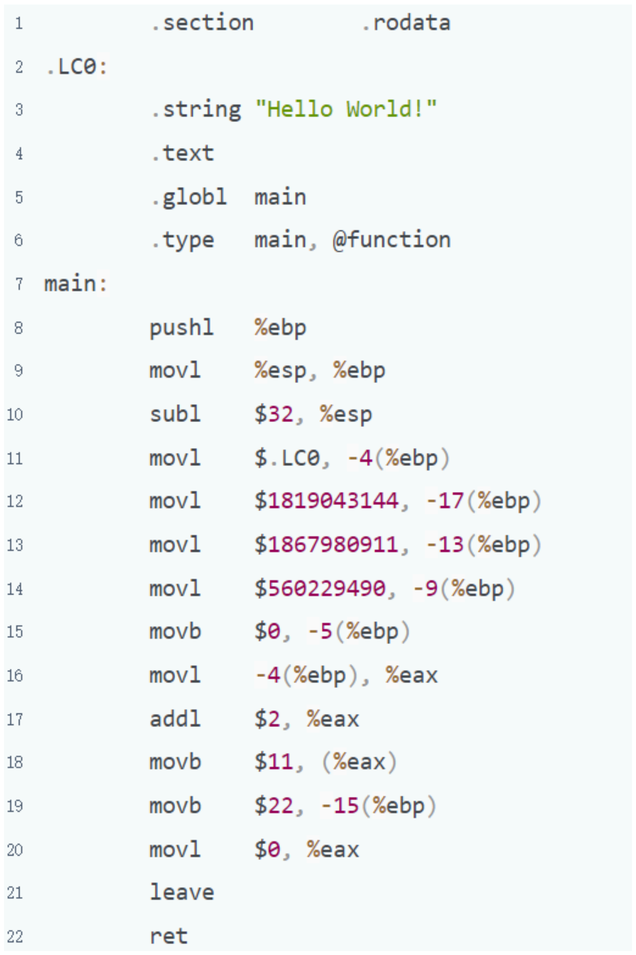


```c
#include <stdio.h>

int main() {
    int array[5];           // Reserve space for 5 integers
    int *ptr;               // Pointer to manipulate memory

                            // Initialize array elements
    array[0] = 1;           // movl $1, -24(%ebp)
    array[1] = 2;           // movl $2, -20(%ebp)
    array[2] = 3;           // movl $3, -16(%ebp)
    array[3] = 4;           // movl $4, -12(%ebp)
    array[4] = 5;           // movl $5, -8(%ebp)

    ptr = array;            // Set pointer to &array[0]
    ptr = ptr + 2;          // addl $8, %eax
                            // movl %eax, -4(%ebp)

    array[1] = 11;          // movl $11, -20(%ebp)

    *(ptr + 1) = 22;        // movl -4(%ebp), %eax
                            // addl $4, %eax
                            // movl $22, (%eax)

    array[2] = 33;          // movl $33, -16(%ebp)

    *(ptr + 2) = 44;        // movl -4(%ebp), %eax
                            // addl $8, %eax
                            // movl $44, (%eax)

    return 0;               // movl $0, %eax
}
```

b. Which are the direct/indirect references in this program?

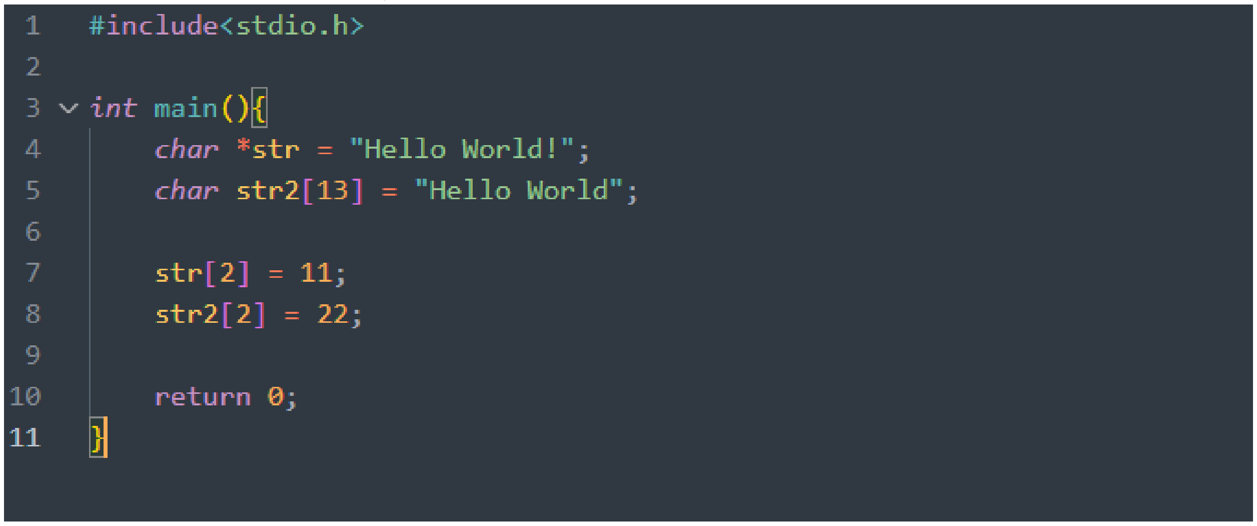
Direct Reference	**Indirect Reference**
array[0] = 1;	ptr = &array[2];
array[1] = 11;	*(ptr + 1) = 22;
array[2] = 33;	*(ptr + 2) = 44;
array[3] = 22;	
array[4] = 5;	

9. Convert the following assembly language to C language, What can you learn from the following program?



```
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $32, %esp
        movl    $.LC0, -4(%ebp)
        movl    $1819043144, -17(%ebp)
        movl    $1867980911, -13(%ebp)
        movl    $560229490, -9(%ebp)
        movb    $0, -5(%ebp)
        movl    -4(%ebp), %eax
        addl    $2, %eax
        movb    $11, (%eax)
        movb    $22, -15(%ebp)
        movl    $0, %eax
        leave
        ret
```

a. The C code for above assembly code:



```c
#include<stdio.h>

int main(){
    char *str = "Hello World!";
    char str2[13] = "Hello World";

    str[2] = 11;
    str2[2] = 22;

    return 0;
}
```

b. From the code, I can learn several important concepts related to **strings**, **memory manipulation**, and **pointer behavior** in C. Here's a breakdown of the key concepts:

   i. **String literals are read-only**, while **character arrays are modifiable**.

ii. **Pointer arithmetic and array indexing** are closely related in C, but pointer manipulation can be more error-prone.
iii. **Memory storage** matters — understand whether dealing with stack-allocated (modifiable) arrays or read-only data (like string literals).
iv. **Be cautious about undefined behavior** caused by modifying string literals