1. The compiler will not report an error. But, depending on the compiler, it may give you a warning. The program may or may not crash depending on whether you have overwritten important information in memory.

2. What is passed is the base address of the array.

3. \( x = ++a[1] \); For this one, \( a[1] \) (which is 1) is incremented before getting assigned to \( x \). That is, what happens here is \( a[1]++ \) followed by \( x = a[1] \). So, \( x = 2 \), and \( a[1] \) becomes 2.
   \( y = a[1]++ \); Here, what happens is: \( y = a[1] \) and then \( a[1]++ \). So, \( y = 2 \) and \( a[1] \) becomes 3;
   \( z = a[x++] \); This is equivalent to \( z = a[x] \) followed by \( x++ \). This means \( z = z[2] = 15 \) and \( x \) becomes 3.
   The output is: 3  2  15

4. \(*(*(a+i)+j)\)

5. Nothing is wrong, the declaration is perfectly fine!

6. There is an error in this structure declaration. The structure emp contains a member e of the same type struct emp. At this stage compiler does not know the size of this structure. In the previous problem (#5 above), the compiler knows how big a pointer is.

7. a) 8
   b) 8
8.
Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

\[
\begin{align*}
\text{short } & \ y = -3; \\
\text{int } & \ y = sy; \\
\text{int } & \ x = -17; \\
\text{unsigned } & \ ux = x;
\end{align*}
\]

Note: You need not fill in entries marked with "-".

<table>
<thead>
<tr>
<th>Expression</th>
<th>Decimal Representation</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>000 000</td>
</tr>
<tr>
<td>-</td>
<td>-6</td>
<td>111 010</td>
</tr>
<tr>
<td>-</td>
<td>+18</td>
<td>01 0010</td>
</tr>
<tr>
<td>ux</td>
<td>47</td>
<td>101 111</td>
</tr>
<tr>
<td>y</td>
<td>-3</td>
<td>111 101</td>
</tr>
<tr>
<td>Tmax</td>
<td>+31</td>
<td>011 111</td>
</tr>
<tr>
<td>-TMin</td>
<td>-32</td>
<td>100 000</td>
</tr>
</tbody>
</table>
9.

```c
void foo(int x)
{
    printf("address of x is %p\n", &x);
    bar(x-1);
    return;
}

void bar(int y)
{
    printf("address of y is %p\n", &y);
    return;
}

void foo2()
{
    bar2();
}

void bar2()
{
    char buf[10];
    gets(buf);
}
```

A. Suppose we invoke the function `foo` many times. What is the relationship of the address of `x` and that of `y` in resulting `printf` statements?
(a) address of `x` is always less than address of `y`.
(b) address of `x` is always greater than address of `y`.
(c) address of `x` is sometimes less than address of `y` and sometimes greater than address of `y`.

B. The `gets` function reads a line from stdin into the buffer pointed to by `s` and does not check for buffer overrun. Suppose we invoke the function `foo2` and the user types in some line longer than 10 characters with the intent of exploiting the buffer overrun to execute malicious code by overwriting a return address. Which function would that return address have been pointing to if the attack had not occurred? (That is, which function would we have returned to).

`foo2()` because the buffer exists in bar2() stack frame, in an address “higher” than the return address of bar2 (so cannot over-write it), but “lower” than return address of foo2, so can over-write it.