A C programmer is working with a little-endian machine with 8 bits in a byte and 4 bytes in a word. The compiler supports unaligned access and uses 1, 2 and 4 bytes to store `char`, `short` and `int` respectively. The programmer writes the following definitions (below right) to access values in main memory (below left):

<table>
<thead>
<tr>
<th>Address</th>
<th>Byte offset</th>
<th>int **i=(int **)0x04;</th>
<th>short **pps=(short **)0x1c;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0 00 00 00</td>
<td>0x04</td>
<td>0x1c</td>
</tr>
<tr>
<td>0x08</td>
<td>61 72 62 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td>33 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>78 0c 00 00</td>
<td>struct i2c {</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td>08 00 00 00</td>
<td>int i;</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>01 00 4c 03</td>
<td>char *c;</td>
<td></td>
</tr>
<tr>
<td>0x1c</td>
<td>18 00 00 00</td>
<td><em>)p=(struct i2c</em>)0x10;</td>
<td></td>
</tr>
</tbody>
</table>

(a) Write down the values for the following C expressions:

```
**i p->c[2] &(*pps)[1] ++p->i
```

(b) Explain why the code shown below, when executed, will print the value 420.

```c
#include<stdio.h>

#define init_employee(X,Y) {(X),(Y),wage_emp}
typedef struct Employee Em;
struct Employee {int hours,salary;int (*wage)(Em*)};
int wage_emp(Em *ths) {return ths->hours*ths->salary;}

#define init_manager(X,Y,Z) {(X),(Y),wage_man,(Z)}
typedef struct Manager Mn;
struct Manager {int hours,salary;int (*wage)(Mn*)};int bonus;);
int wage_man(Mn *ths) {return ths->hours*ths->salary+ths->bonus;}

int main(void) {
    Mn m = init_manager(40,10,20);
    Em *e = (Em *) &m;
    printf("%d\n",e->wage(e));
    return 0;
}
```

(c) Rewrite the C code shown in part (b) using C++ primitives and give *four* reasons why your C++ solution is better than the C one.