1. First consider the declarations. The declaration `char dd` sets aside a box for a character and we are trying to figure out what is in that box. The declaration `char *cc="Aardvark"` sets aside 9 consecutive boxes in memory,

'A'	'a'	'r'	'd'	'v'	'a'	'r'	'k'	'\0'

and `cc` points to the first box, the one containing the character `'A'`; `cc+1` points to the second box, the one containing the character `'a'`; and so on.

As an aside, `cc+9` points to a very definite location in memory: it points to the box one after the `\0`. We just have no control over what is in that location. I have met a few compilers for which this box will be the box where `dd` is stored although most of them put the box for `dd` just before the boxes where `"Aardvark"` is stored. For a few more compilers, the `"Aardvark"` will be stored in a special "strings" area and will have no relation to the box for `dd` at all.

With this out of the way, consider the statement `dd=*(++cc);`. The first thing that happens is that the pointer `cc` is incremented and so now points to the second box. Then the value in that box is copied to the `dd` box, so `dd='a'`. For the other statement `dd=++(*cc);`, the value to which `cc` points (i.e. the value in the first box, or the character `'A'`) is incremented and this value is then stored in the `dd` box. Hence `dd='A'+1='B'`. A side effect of this incrementation is that the string now reads `"Bardvark"`.

Here are a couple of related comments. The statement `dd=(*cc)++;` would take the value pointed to by `cc`, namely `'A'` and put it in `dd`, BUT it then increments the value pointed to by `cc` so the string reads `"Bardvark"`. Finally, the statement `dd=*(cc++);` does the following: it sets `dd='A'` and then increments the pointer `cc` so AFTER the statement is finished, `cc` points to the box containing the `'a'`.

2. The declaration `char cc[10]` sets aside ten consecutive boxes in memory, `cc[0]` through `cc[9]`. The contents of those boxes are unspecified after the declaration. The statement `cc[0]='H';` puts the character `'H'` into the first box and so on. After all the program before the printf statement is finished, we have a collection of boxes

'H'	'e'	'l'	'l'	'o'	*?*	*?*	*?*	*?*	*?*

where *?* indicates a box which has been reserved for us but for which have defined no value.

The `%s` inside the format string of the printf statement says to start with the box to which `cc` points, which happens to have an `'H'` in it, and print characters until a `'\0'` is found. Now some characters print and some do not, so what happened to us is the following. After we printed the `'o'`, we printed some undetermined number of non–printing characters until we ran into an `'8'`. This can of occurred no earlier than `cc[5]` but might be anywhere after that. It might be well past `cc[9]`, the last box reserved by our declaration. Then we printed another undetermined number of non–printing characters until we ran into an `'\0'`.

Notice the **size** of the declaration is irrelevant to this problem. Once the `%s` starts, it goes until it finds a `'\0'` regardless of how many of those boxes you actually reserved.

To replace the given declaration with `char cc[4];` is definitely an error since then you have only reserved 4 spaces but you have used 5 since you do `cc[4]=’o’;`

One fix is to add the statement `cc[5]=’\0’`. This puts a zero at the end of the string. This is the “best” fix since now `cc` contains a genuine C string and if expand your program and use this string elsewhere it should work correctly.

Other possible fixes are

1. Replace the printf statement with `printf("%.5s\n",cc);` which gives you 5 characters or until the `’\0’`, which ever comes first.
2. Replace the printf statement with `printf("%5.5s\n",cc);` which gives you 5 characters or until the `’\0’`, which ever comes first and always uses 5 spaces, padding the leading part with spaces if the `’\0’` comes before all 5 characters have been used.
3. Replace the printf statement with `printf("Hello\n");`. This answers the last question, if somewhat inelegantly, but of course says nothing to the point on the first one.

3.

x	1	0	1	1	0	0	1	0
y	0	1	1	0	0	1	1	0
&	0	0	1	0	0	0	1	0

x	1	0	1	1	0	0	1	0
y	0	1	1	0	0	1	1	0
|	1	1	1	1	0	1	1	0

x	1	0	1	1	0	0	1	0
y	0	1	1	0	0	1	1	0
^	1	1	0	1	0	1	0	0

x	1	0	1	1	0	0	1	0
~	0	1	0	0	1	1	0	1

4. The first problem is to find a way to determine if an integer such as `4` or `100` does or does not divide `x`. There are several possible solutions. The simplest is `x%4==0`, `x%4!=0`, `x%100==0`, etc. Another possibility is `x==4*(x/4)`, etc. Two other possibilities are to use the library functions `modf` or `fmod`. Both of these functions return doubles. Using `fmod` one can write `fmod(x,4)==0`, etc. If you want to use `modf` you need to declare a double, say `double xx`. Then `modf(x/4.0,&xx)==0`, etc. The statement `modf(x/4,&xx)==0`, does not work, because first we compute `x/4` as an INTEGER, so `modf(x/4,&xx)` is always `0`. If you want to use either `fmod` or `modf` you must remember to include math.h: `#include <math.h>`

With this problem out of the way, we can write the following code

```
char leap( short x);
char leap( short x) {
if( x%4==0 && x%100!=0 || x%400==0 ) return 1;
else return 0;
}
```

It would be better to write the test as

```
( ( (x%4==0) && (x%100!=0) ) || (x%400==0) )
```

unless you are absolutely sure of the precedence of the various operators. Of course you can replace the `%` tests with any of the alternative tests discussed above and there are other logically equivalent expressions which may be used. Here is one alternative

```
char leap( short x);
char leap( short x) {
if( x%400==0) return 1;
else if(x%100==0) return 0;
else if( x%4==0 ) return 1;
else return 0;
}
```

Note that the order in which you do the tests is important in this version.

5. After the first for loop, `zz[0]=0`, `zz[1]=1`, `zz[2]=2`, `zz[3]=3`, etc. In the second for loop, each `zz[ix]` is computed by taking the current value and adding the current value of `ix` to it, so we get `zz[0]=0+0=0`, `zz[1]=1+1=2`, `zz[2]=2+2=4`, `zz[3]=3+3=6`, etc. The last for loop is the most confusing to figure out: to compute `zz[2]` first notice that `zz[2]=zz[9-7]` so `zz[2]=zz[2]-7=4-7=-3` and `zz[3]=zz[9-6]=zz[3]-6=6-6=0`.

Now consider the format string for the printf statement: `"%3d\t%2d\n"`: this will then print a space, then a `'-'`, then a `'3'`, then a tab, then a space, then a `'0'` and finally a newline.