Summary & Review

Most programming languages only support operations on larger units (e.g bytes or words)

C provides a set of low level operators

Left shift "<<" moves a bit along by one, moving a zero in.
(Another way of thinking of it is as multiplying the value by 2)

Right shift ">>" does the opposite:

And (&), Or (|), and X-Or (^) let us set or clear individual bits by using a mask.

If we AND a bit patterns with a mask, where a 1 is in the mask the value remains the same; where a 0 is the bit is set to 0.

AND is therefore useful for CLEARING a particular bit.

If we OR a bit pattern with a mask, where a 1 is in the mask the value is set to 1, where a 0 is the bit remains the same.

OR is useful for SETTING a particular bit.

X-OR is less commonly useful.

Consider the program on the next slide:

```
#include <stdio.h>
void genbits(char *p,unsigned char val);

int main(int argc, char *argv[])
{
  char bits[9];
  unsigned char i=0xff;
  int j;

  for (j=0;j<8;j++)
    {
      genbits(bits,i);
      printf("%3d %s %02X\n",i,bits,i);
      i = i >> 1;
    }
}

/**********************************************
 ** Name:   genbits                          **
 ** Author: IRJ                              **
 ** Date:   26-SEP-2001                      **
 ** Desc:   function to produce an 8 char    **
 **         ASCII string of the binary of val**
 **********************************************/
void genbits(char *p, unsigned char val)
{
  int i;
  unsigned char mask;

  mask = 0x01;           /* init mask   */
  *(p+8) = '\0';         /* insert null */
  for(i=0;i<8;i++)       /* foreach bit */
    {
      if (val & mask)    /* if its 1    */
        *(p+7-i) = '1';
      else
        *(p+7-i) = '0';
      mask = mask << 1; /* move mask to next
                            bit             */
    }
}
```

This program will print:

```
255 11111111 FF
127 01111111 7F
 63 00111111 3F
 31 00011111 1F
 15 00001111 0F
  7 00000111 07
  3 00000011 03
  1 00000001 01
```

Decimal then Binary then Hexadecimal.

Since each hexadecimal digit represents 4 bits, it is a convenient way of representing bit patterns in our programs.

Combining operators

All of the operators can be combined with =

```
expressionA  operator=  expressionB
```

means the same as:

```
expressionA = (expressionA) operator (expressionB)
```

Be careful:

```
x = 3;
y = 4;

x *= y + 2;
```

What is the new value of x? Remember if you're not sure, why do you expect a reader to be sure!

```
x = (x * y) + 2;
```

is unambiguous!

Relational Operators

>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	equality
!=	inequality (not equal)

Logical Operators

&&	And
||	Or
!	Not

Expressions connected by these operators are evaluated left to right, and evaluation stops once the truth of the expression is known.

This leads to side-effects!

Consider the following horrible code:

(*Taken from K&R p41*.)

```
i < lim1 && (c = getchar()) != '\n' && c != EOF
```

- The order matters!
- Not everything will be executed!

Boolean (or not) Conditions

C takes an unusual approach to evaluating conditions.

Zero = False
Non-Zero = True

However the value of an expression which is TRUE will be 1.

! negates an expression. Zero to One, or Non-Zero to Zero.

This allows side-effect laden conditions, **NOT A GOOD IDEA** (but like all bad ideas, occasionally useful ☺).

Precedence and Associativity of Operators in C

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>() [] -> .</td><td>left to right</td></tr>
<tr><td>! ~ ++ -- + - * & (type) sizeof</td><td>right to left</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ - (binary)</td><td>left to right</td></tr>
<tr><td><< >></td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>== !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>?:</td><td>right to left</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>right to left</td></tr>
<tr><td>,</td><td>left to right</td></tr>
</table>

Unary +, - and * have higher precedence than the binary forms.
taken from: ***Kernighan & Ritchie***, p. 53

What is the value of: 3 * 4 + 1?

What is the value of: 0x70 | 0x0a == 10?

Advice: Bracket wildly, do not overcompound statements.

Use brackets to make your meaning clear and ensure the accuracy of your expressions.

C deliberately does not specify the order in which operands are evaluated.

Any code that depends on the order in which operands are evaluated will fail on some computers (or compilers)

Consider:

```
int myarray[10];
int x=7;

myarray[x] = x++;
```

Also, multiple function calls are undefined.

Control Flow

Like most programming languages C has various constructs for control flow.

Royce (chapter 2) discusses JSP like program design.
READ THIS CHAPTER, AND WORK THROUGH THE EXERCISES!

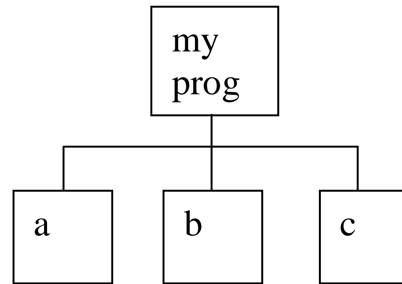
Jackson (M.A. Jackson, "*Principles of Program Design*", 1976) believes than ANY program can be built just using:

- Sequence (A then B then C)
- Selection (if A then X else Y)
- Iteration (repeat A zero or more times)

This does very rarely lead to convoluted code, but is worth remembering in principle.

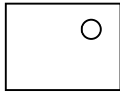
Sequence

```
x = 2;  /* a */
y = 3;  /* b */
z = 4;  /* c */
```



Selection

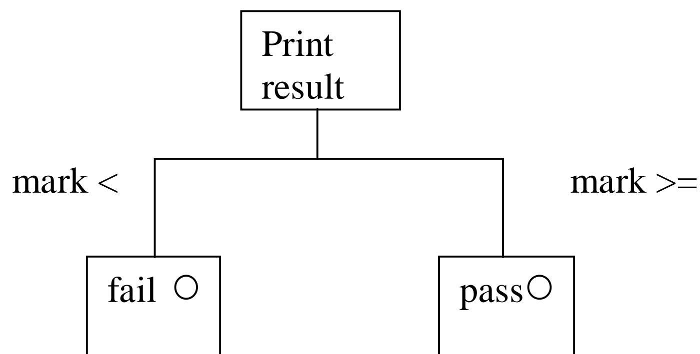
C offers a selection of selection constructs! Diagrammatically we would represent them all identically: The choices being represented as boxes with a circle in the top right corner:



Consider:

```
if (mark < 40)
  printf("Oh no! You've failed\n");
else
  printf("Congrats! You've passed\n");
```

We could draw this as:



If (you can keep your head while all around…)

```
if (condition)  statement;


if (condition) statement; else statement;
```

BEWARE:

```
if (x=0)
  y=4;
```

is legal C, x will be assigned the value 0, the if will evaluate to false, and y will never be altered.

Remember! in C Boolean expressions are numerically evaluated

false = ZERO
true = NOT ZERO

Therefore:

```
if (x-2) statement;
```

is legal. Subtract 2 from x and if the result is not zero perform statement.

```
if ((x-2) != 0) statement;
```

says the same, but is more readable!

Also, an else closes the innermost if. If this isn't what you want use braces to build a compound statement.

```
if (a > b)
  {
     if (n == 4)
      x++;
  }
else
  x--;
```

Gcc hackers may wish to turn parenthesis checking on!

Other selections

Switch

```
switch (expression)
```

```
  {
    case const-expression: statements;
    default: statements;
  }
```

Switch statements can be very useful, their design in C is a little tricky however.

Execution begins at the statement following the case that *expression* evaluates to and continues to the end of the switch. Normally you will want to place a *break* statement at the end of each case, to terminate the switch expression after the first choice has been made.

The Ternary operator

Beloved by those who like to write impenetrable C, it can be occasionally useful.

```
expression1 ? expression2 : expression3
```

expression1 is evaluated, if true then expression2 if false expression3

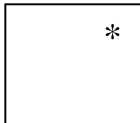
This is the same as:

```
if (expression1)
  expression2;
else
  expression3;
```

Iteration

Correctly, we iterate zero or more times. Some constructs implement one or more repetition however. In other words, the statements are performed at least once.

Diagrammatically iteration is shown thus (a box with an asterisk in the top right corner:



Repetition

C provides a set of looping or repetition operators:

While and Do-while

```
while (condition) statement;
```

While loops are true iterations.

```
do
  statement
while (condition);
```

While loops are only executed if condition is true, whilst do-while loops are executed at least once.

For

For loops in C can be simple counted loops but are far more powerful, and often abused!

```
for (init-expr; test-expr; repeat-expr)
    statement;
```

init-expr is performed at the start of the loop. The loop is repeated while test-expr is true, with repeat-expr being executed for each iteration.

Used sensibly:

```
for (i=0; i<10; i++)
  x[i] = 0;
```

Used differently:

```
char x[10];
for (i=0; i < 10; x[i++]=0);
```

The above example is reasonably sensible, it is easy to write impenetrable code like this however!

Compound Statements

So far, we've mentioned statements or expressions, often we want to perform several in an selection or repetition. In those cases we group statements with braces:

i.e.

```
  {
    statement;
    statement;
    statement;
  }
```

Using "Jackson-like" structure diagrams

We want to write a program to process a ***Portable Pixel Map*** (PPM) file.

First stop – the manual!

`man ppm`

would give us something close to the following:

(my comments like this)

The PPM Manual Page

NAME

ppm portable pixmap file format

DESCRIPTION

The portable pixmap format is a lowest common denominator color image file format.

(i.e. It's simple!)

It should be noted that this format is egregiously inefficient. It is highly redundant, while containing a lot of information that the human eye can't even discern. Furthermore, the format allows very little information about the image besides basic color, which means you may have to couple a file in this format with other independent information to get any decent use out of it. However, it is very easy to write and analyze programs to process this format, and that is the point.

Simple to process ☺, but inefficient & limited

It should also be noted that files often conform to this format in every respect except the precise semantics of the sample values. These files are useful because of the way PPM is used as an intermediary format. They are informally called PPM files, but to be absolutely precise, you should indicate the variation from true PPM. For example, "PPM using the red, green, and blue colors that the scanner in question uses."

(i.e. RGB values are not absolute, but device relative)

The format definition is as follows.

A PPM file consists of a sequence of one or more PPM images. There are no data, delimiters, or padding before, after, or between images. A common subformat, and the only one defined before July 2000, has exactly one image. Most tools to process PPM files ignore any data after the first image.

(before 2000, 1 image per file, now multiple are allowed)

Each PPM image consists of the following: *(Image Structure)*

- A "magic number" for identifying the file type. A ppm file's magic number is the two characters "P6".
- Whitespace (blanks, TABs, CRs, LFs).
- A width, formatted as ASCII characters in decimal.
- Whitespace.
- A height, again in ASCII decimal.
- Whitespace.
- The maximum color value (Maxval), again in ASCII decimal. Must be less than 65536.

(If less than 256 8 bits per colour value, else 16 bits)

- Newline or other single whitespace character.
- A raster of Width * Height pixels, proceeding through the image in normal English reading order. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in pure binary by either 1 or 2 bytes. If the Maxval is less than 256, it is 1 byte.
  Otherwise, it is 2 bytes. The most significant byte is first.

(Big-endian versus little endian! See later)

- In the raster, the sample values are proportional to the intensity of the CIE Rec. 709 red, green, and blue in the pixel. A value of Maxval for all three samples represents CIE D65 white and the most intense color in the color universe of which the image is part (the color universe is all the colors in all images to which this image might be compared).
- Characters from a "#" to the next endofline, before the maxval line, are comments and are ignored.

(Comments start with # end with newline)

Note that you can use **pnmdepth** To convert between the format with 1 byte per sample and the one with 2 bytes per sample.

There is actually another version of the PPM format that is fairly rare: "plain" PPM format. The format above, which generally considered the normal one, is known as the "raw" PPM format. See **pbm**(5) for some commentary on how plain and raw formats relate to one another.

(Plain files are enormous, but how much bigger typically than raw?)

The difference in the plain format is:

- There is exactly one image in a file.
- The magic number is P3 instead of P6.
- Each sample in the raster is represented as an ASCII decimal number (of arbitrary size).
- Each sample in the raster has white space before and after it. There must be at least one character of white space between any two samples, but there is no maximum. There is no particular separation of one pixel from another just the required separation between the
  blue sample of one pixel from the red sample of the next pixel.
- No line should be longer than 70 characters.

Here is an example of a small pixmap in this format:

```
P3
# feep.ppm
4 4
15
0 0 0 0 0 0 0 0 0  0 15 0 15 0 0 0
0 15 7 0 0 0 0 0 0 0 0 0 15 0 15
0 0 0 0 0 0
0 15 7 0 0 0
0 0 0 0 0 0
```

Programs that read this format should be as lenient as possible, accepting anything that looks remotely like a pixmap.

COMPATIBILITY

Before April 2000, a raw format PBM file could not have a maxval greater than 255. Hence, it could not have more than one byte per sample. Old programs may depend on this.

(Old stuff may depend on 256 colours or less, 1 image per file)

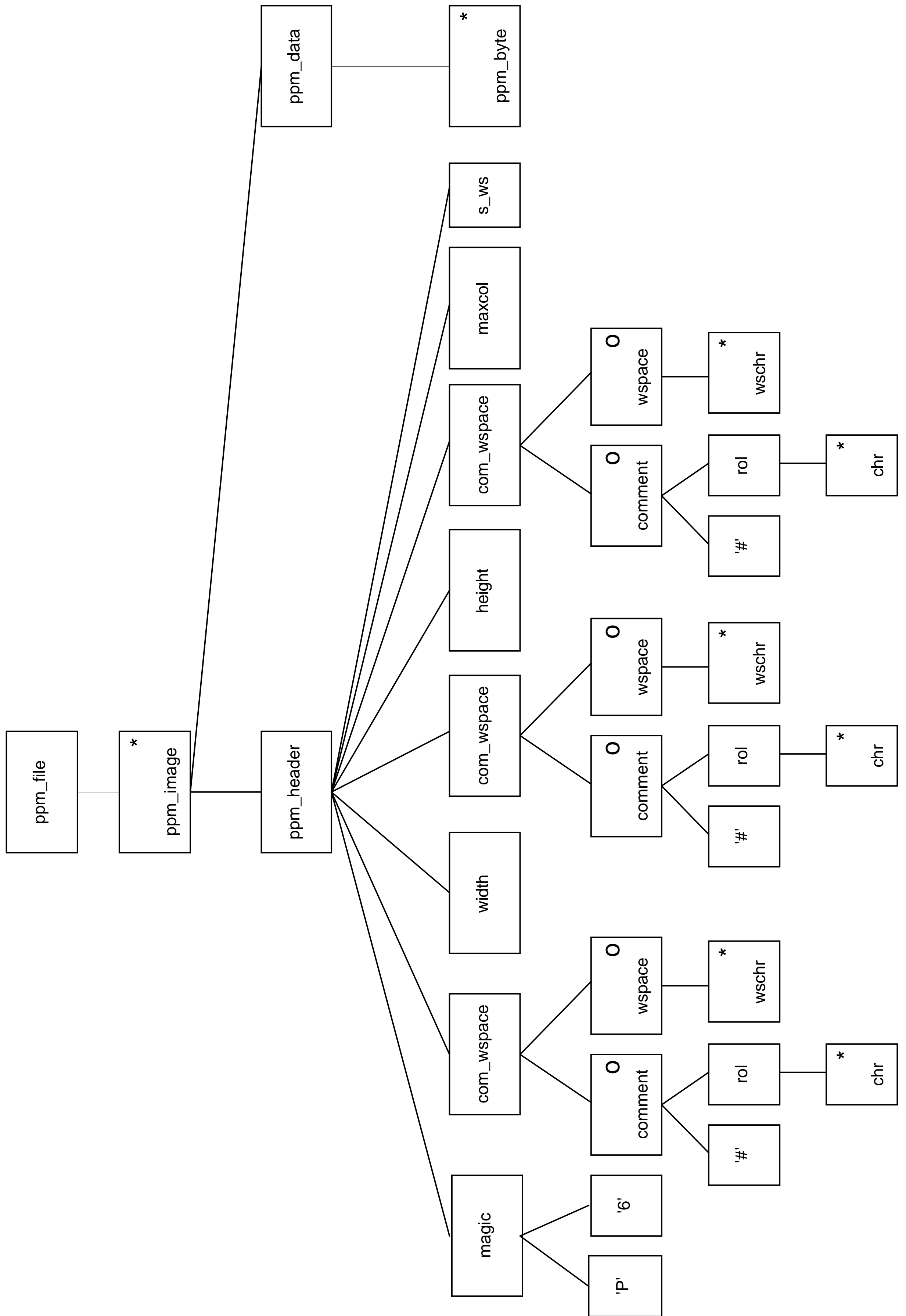
SEE ALSO

giftopnm(1), gouldtoppm(1), ilbmtoppm(1), imgtoppm(1), pgmtoppm(1), pi1toppm(1), picttoppm(1), pjtoppm(1), rgb3toppm(1), sldtoppm(1), spctoppm(1), sputoppm(1), xpmtoppm(1), yuvtoppm(1), ppmtoacad(1), ppmtogif(1), ppmtopcx(1), ppmtopgm(1), ppmtopi1(1), ppmtopict(1), mtvtoppm(1), pcxtoppm(1), qrttoppm(1), rawtoppm(1), tgatoppm(1), ximtoppm(1), ppmtoicr(1), ppmtoilbm(1), ppmtopj(1), ppmtopuzz(1), ppmtorgb3(1), ppmtosixel(1), ppmtotga(1), ppmtouil(1), ppmtoxpm(1), ppmtoyuv(1), ppmdither(1), ppmforge(1), ppmhist(1), ppmmake(1), ppmpat(1), ppmquant(1), ppmquantall(1), ppmrelief(1), pnm(5), pgm(5), pbm(5)

AUTHOR

Copyright (C) 1989, 1991 by Jef Poskanzer.

So, what would our structure chart look like?


ppm_file
ppm_image
ppm_header
ppm_data
ppm_byte
magic
'P'
'6'
com_wspace
comment
'#'
rol
chr
wspace
wschr
width
com_wspace
comment
'#'
rol
chr
wspace
wschr
height
com_wspace
comment
'#'
rol
chr
wspace
wschr
maxcol
s_ws

Pointers & Arrays

Pointers cause **EVERYBODY** problems at some time or another.

A good tutorial on pointers is available on the module web site.

Pointers in C, are closely related with arrays.

Arrays

To declare an array of char:

`char x[10]` for a one dimensional array

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]

Remember: element 0 is always the first element in an array!

`char y[3][4]` for a two dimensional array:

y[0][0]	y[0][1]	y[0][2]	y[0][3]
y[1][0]	y[1][1]	y[1][2]	y[1][3]
y[2][0]	y[2][1]	y[2][2]	y[2][3]

`char z[2][3][4]` for three dimensions etc.

Arrays are stored in row order, and really are just one dimensional arrays abstracted. As we move through memory the rightmost subscript will vary the quickest.

(SEGV) Segmentation Error! Core Dumped

Is an error message you'll regularly see (just after your program crashes ☺) when you start using arrays and pointers in anger.

One of the biggest dangers in C, is the lack of bounds checking. If I want I could write

```
x[42] = 'x';
```

or

```
x[-569] = 'x';
```

If you're using x as we've defined it above, both the above will compile. If the subscripts are constant expressions the compiler may be kind enough to warn you, otherwise you're on your own!.

But a good lint may help!

A side-effect of this lack of bounds checking is that we can pass arrays to functions WITHOUT specifying the left most subscript.

So

`void foo(int x[])` is totally legal…

…as is `void bar(int y[][4])`

Real C Programmers use pointers (or arrays of pointers ☺)

Pointers

to declare a char pointer:

```
char *cp;
```

We read this as

"*cp is a char"

or

"cp is a pointer to char"

The pointer declaration only allocates space in memory to hold the pointer (cp), NOT any data (*cp).

To refer to the pointer we use cp;

To refer to the pointed-at data we use *cp

We can say, `cp = x` or `cp = &x[0]`
(both have the same meaning)

& is the "address of" operator.

Arrays in C are really only a shorthand of describing where in memory some data is.

Pointers work in exactly the same way.

if I have:

```
int x[10];
int *y;

y=x;
```

Then x[3] or *(y+3) refer to the same address!

Be careful!

`*y+3` adds 3 to the *value* pointed at by y

`*y++` increments *the pointer* y

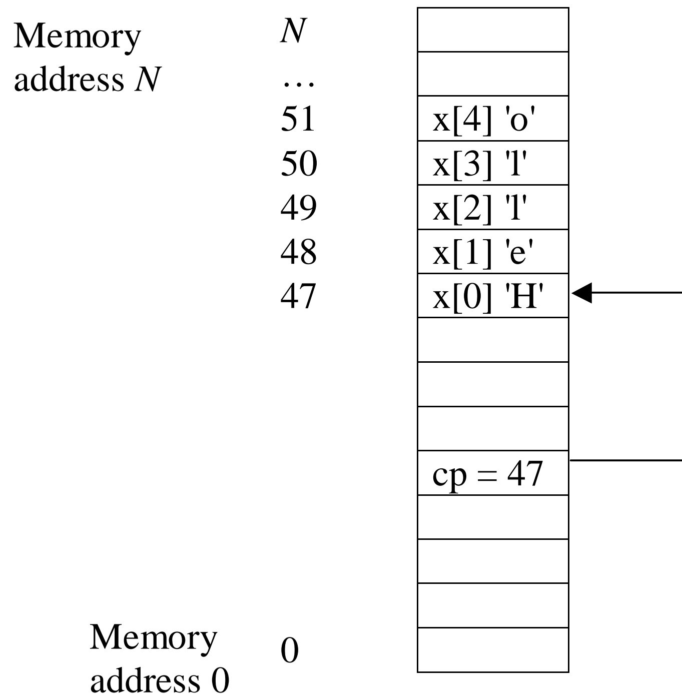
When we pass an array to a function in C:

```
foo(int x[]);
```

we are really passing a pointer!

```
foo(int *x);
```

makes that point clear! (and is identical)



Pointers are vital in C

Without pointers we would not be able to modify parameters passed to functions.

In Pascal you can specify a parameter as *Var* to avoid needing pointers for parameter passing . In technical terms you can pass parameters by value or by reference.

By value, means the function is given a copy of the current value of a variable.

By reference, means the function is told where in memory the variable is stored.

In C we can only pass parameters by value but can achieve the same effect as passing by reference by passing a pointer to the variable.

Imagine a function to update a bank balance. We want to add or subtract a value from the balance and return the new balance, but we want to set